CS 60038: Advances in Operating Systems Design

Kernel Data Structures

Process Scheduling

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

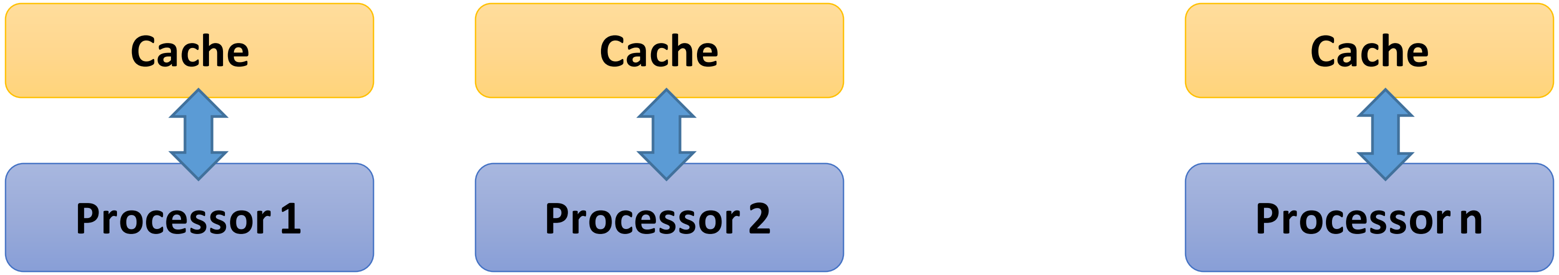
Linux Scheduling

- Linux is a multi-tasking Operating System
  - The system can execute multiple processes simultaneously

Linux Scheduling

- Linux is a multi-tasking Operating System
  - The system can execute multiple processes simultaneously

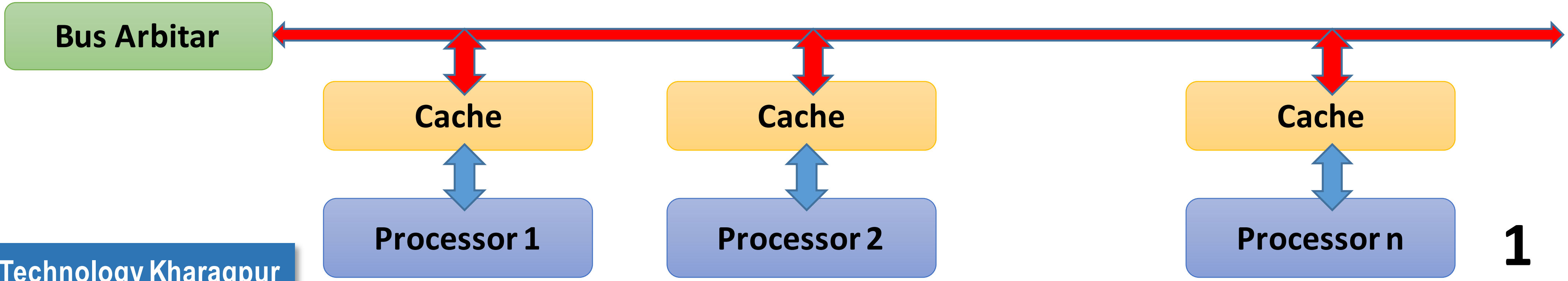
- **Symmetric Multiprocessing (SMP):**
  - Processing of programs by multiple processors that share a common operating system and memory
  - Linux kernel was designed to support SMP



Linux Scheduling

- Linux is a multi-tasking Operating System
  - The system can execute multiple processes simultaneously

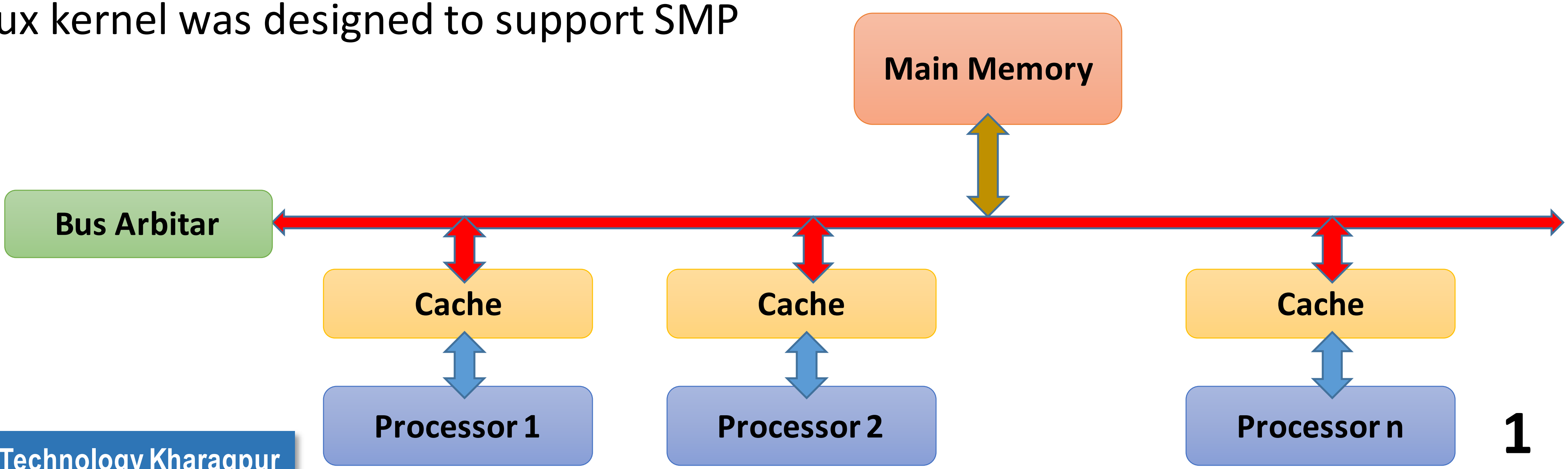
- **Symmetric Multiprocessing (SMP):**
  - Processing of programs by multiple processors that share a common operating system and memory
  - Linux kernel was designed to support SMP



Linux Scheduling

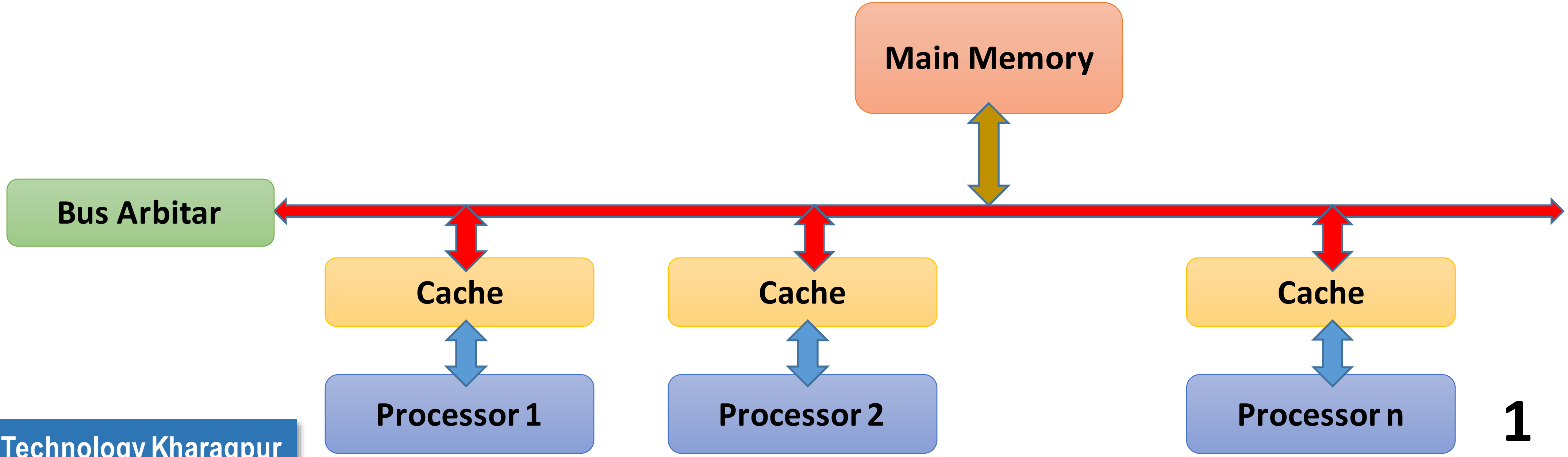
- Linux is a multi-tasking Operating System
  - The system can execute multiple processes simultaneously

- **Symmetric Multiprocessing (SMP):**
  - Processing of programs by multiple processors that share a common operating system and memory
  - Linux kernel was designed to support SMP

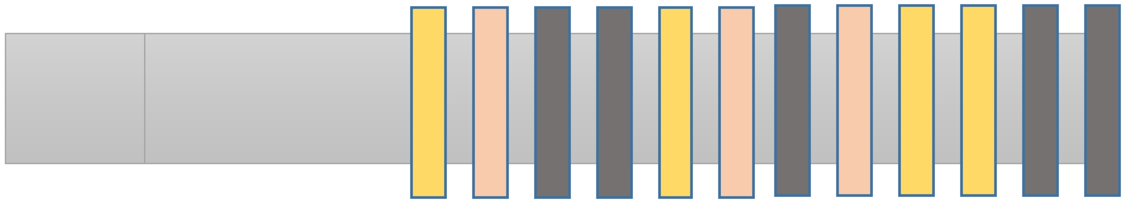


Linux Scheduling

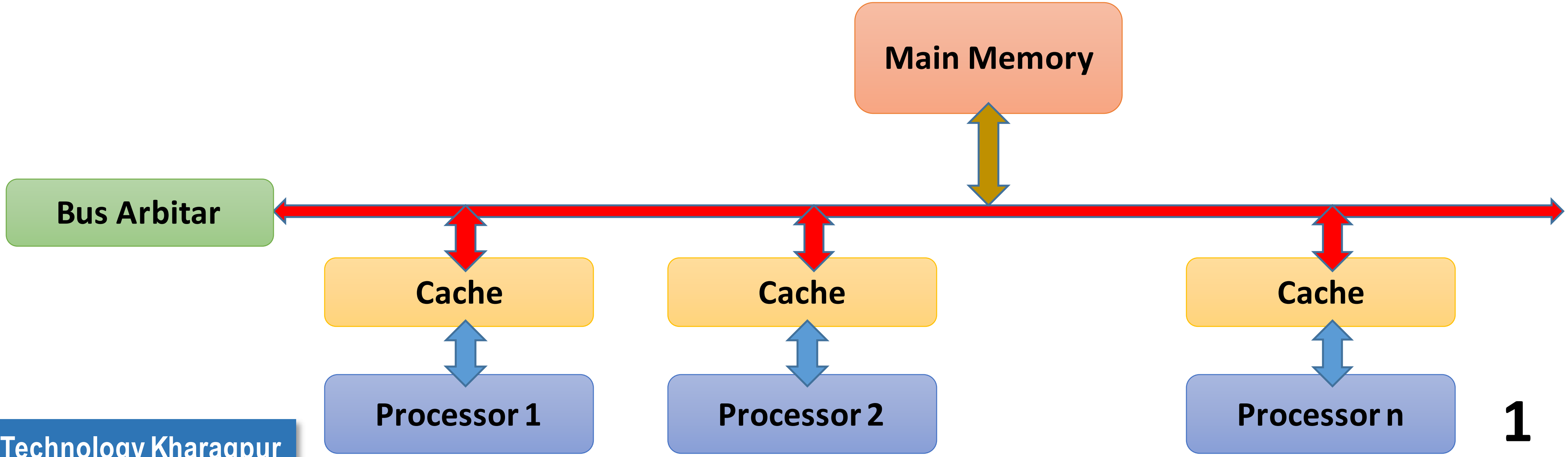
How do you assign processes to processors?



Linux Scheduling



How do you assign processes to processors when the **different process has different priority** ?



Looking Back at the Process Descriptor

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

Task priority

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

Real-time task priority

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

Real-time task priority

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

Scheduling Class:

A generic structure to implement various scheduling algorithms

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

sched_class

```
struct sched_class {
    const struct sched_class *next;

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task) (struct rq *rq);
    bool (*yield_to_task) (struct rq *rq, struct task_struct *p,
                                                     bool preempt);
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p,
                                                               int flags);

    …
}
```

Looking Back at the Process Descriptor

Scheduling Entity:

Used for group scheduling

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

Scheduling Entity (for RT tasks):

Used for group scheduling for real-time tasks

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

Scheduling Policy:

SCHED_NORMAL
SCHED_BATCH
SCHED_IDLE
SCHED_FIFO or
SCHED_RR

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

SCHED_NORMAL

Scheduling policy for regular tasks

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

SCHED_BATCH

Scheduling policy for batch jobs – does not preempt often as regular tasks

Ex: Bulk database updates

Allow tasks to run longer and make better use of caches

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

SCHED_IDLE

Scheduling policy for very low priority processes, like background tasks

Objective is not to disturb the regular tasks

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

SCHED_FIFO
SCHED_RR

Scheduling policy for real time processes

Handled by real-time schedulers

`kernel/sched/rt.c`

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

A bitmask indicating a task's affinity towards a CPU

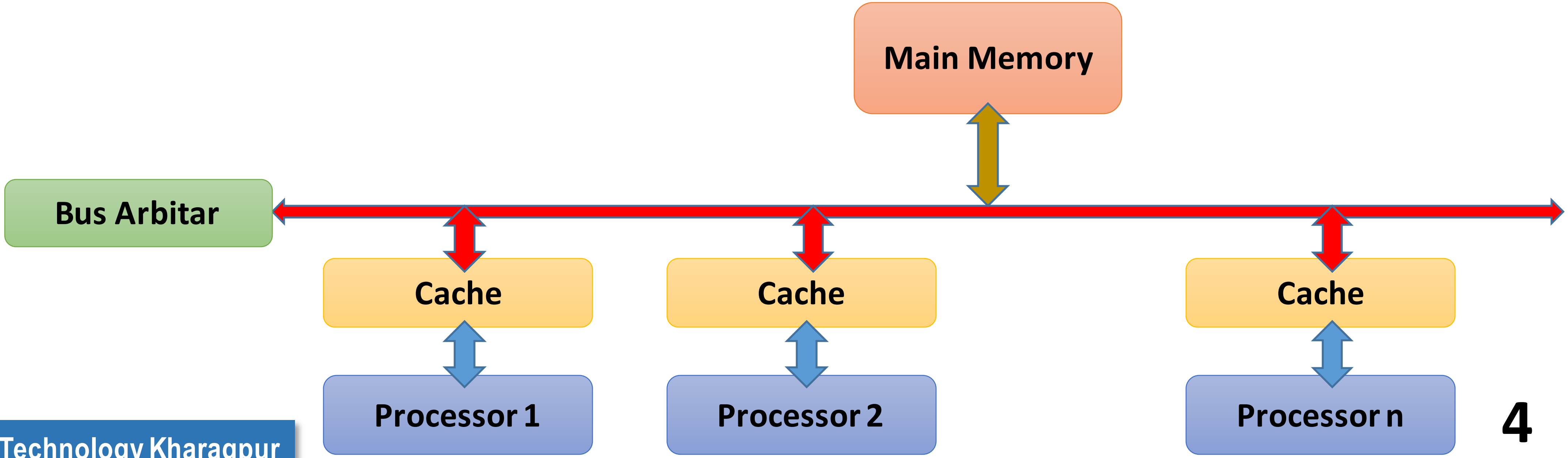
Better utilization of the CPU cache

```
struct task_struct {
    …
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    …
    unsigned int policy;
    cpumask_t cpus_allowed;
    …
};
```

Looking Back at the Process Descriptor

A bitmask indicating a task's affinity towards a CPU

Better utilization of the CPU cache – **remember the SMP architecture**

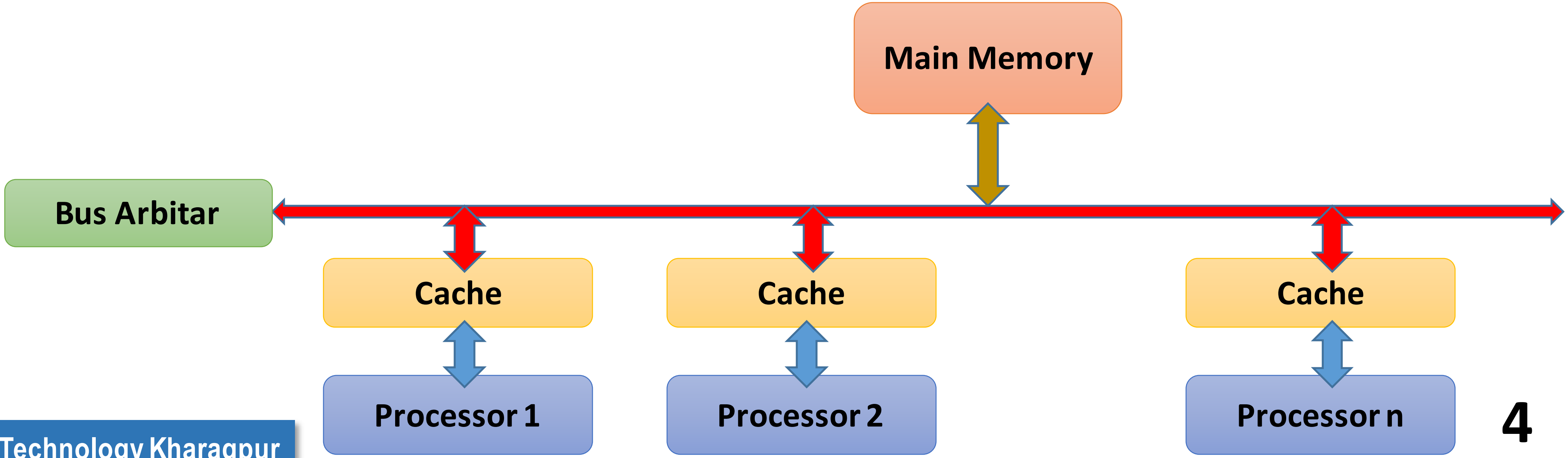


Looking Back at the Process Descriptor

A bitmask indicating a task's affinity towards a CPU

Better utilization of the CPU cache – **remember the SMP architecture**

Defines the **CPU Affinity** of a process !!



Process Priority

- Any UNIX-based system (hence, Linux) implements a priority-based scheduling
  - Assign some value to every task – indicates how important this task is compared to other tasks in the system

Process Priority

- Any UNIX-based system (hence, Linux) implements a priority-based scheduling
  - Assign some value to every task – indicates how important this task is compared to other tasks in the system

- Each task is assigned a nice value
  - An integer between –20 to 19 with default being 0

Process Priority

- Any UNIX-based system (hence, Linux) implements a priority-based scheduling
  - Assign some value to every task – indicates how important this task is compared to other tasks in the system

- Each task is assigned a nice value
  - An integer between –20 to 19 with default being 0
  - The higher the niceness – the lower the priority (**it is "nice" to other tasks**)
  - Use `<PS -Al>` (lowercase L) to check the niceness of the processes running in your system

Process Priority

- Use kernel system call `nice (int increment)` to change the niceness of a process

Process Priority

- Use kernel system call `nice (int increment)` to change the niceness of a process

- Change niceness from the user-space
  - `nice -n increment process_name`
  - `renice -n priority -p PID`

Real-time Tasks

- Execution should complete within a time boundary

- **Hard Real-time**
  - Strict time limits – the tasks must be completed within this time limit
  - By default, Linux does not support hard real-time processes

- **Soft Real-time**
  - The scheduler tries its best to maintain the time limit
  - However, the process can run a bit late depending on the current load

Process Priority (Kernel's Perspective)

- Kernel sees priorities in a different way than user
  - User may set the priority, but the kernel decides what to do with that priority
  - There are processes which are not under the control of any user-space program

Process Priority (Kernel's Perspective)

- Kernel sees priorities in a different way than user
  - User may set the priority, but the kernel decides what to do with that priority
  - There are processes which are not under the control of any user-space program

- Kernel uses a scale from 0 to 139 to represent the task priority values
  - 0 to 99 are reserved for real-time (soft) processes
  - 100 to 139 (mapped to nice –20 to +19) are for normal processes

Process Priority (Kernel's Perspective)

- Kernel sees priorities in a different way than user
  - User may set the priority, but the kernel decides what to do with that priority
  - There are processes which are not under the control of any user-space program
- Kernel uses a scale from 0 to 139 to represent the task priority values
  - 0 to 99 are reserved for real-time (soft) processes
  - 100 to 139 (mapped to nice –20 to +19) are for normal processes
- Kernel sets the priority of a task depending on different factors
  - Real-time vs normal tasks
  - `static_prio` set by the user-space program (through nice)
  - The scheduling policy being used

The `schedule()` Function

- Main entry point to the kernel task scheduler
  - Replace the currently running task with a new task – **Context Switch**

The `schedule()` Function

- Main entry point to the kernel task scheduler
  - Replace the currently running task with a new task – **Context Switch**
- **But, what is a scheduler?**
  - **A process?**
  - **A hardware module?**
  - **Something else?**

The `schedule()` Function

- Main entry point to the kernel task scheduler
  - Replace the currently running task with a new task – **Context Switch**

- **But, what is a scheduler?**
  - **A process?**
  - **A hardware module?**
  - **Something else?**

- **Who does schedule the scheduler?**

Let's try to have a simplistic view ...

P1

Let's try to have a simplistic view ...

P1

`fork()`

Let's try to have a simplistic view ...

Let's try to have a simplistic view ...

P1

`fork()`

P2

Runqueue

Let's try to have a simplistic view ...

P1

`fork()`

P2

Runqueue

Let's try to have a simplistic view ...

P1

fork()

P2

schedule()

Runqueue

Let's try to have a simplistic view ...

Runqueue

Runqueue gets updated, context switching is being done

Let's try to have a simplistic view ...

P1

P2

Runqueue

Let's try to have a simplistic view ...

`schedule()`

P1

P2

Runqueue

Let's try to have a simplistic view ...

`schedule()`

P1

P2

Runqueue

Let's try to have a simplistic view ...

P1

P2

`fork()`

Runqueue

Let's try to have a simplistic view ...

P1

P2

`fork()`

P3 (User)

Runqueue

Let's try to have a simplistic view ...

P1

P2

`fork()`

P3 (User)

Runqueue

Let's try to have a simplistic view ...

P1

P2

`schedule()`

P3 (User)

Runqueue

Let's try to have a simplistic view ...

P1

P2

`schedule()`

P3 (User)

Runqueue

Let's try to have a simplistic view ...

`schedule()`

P1

P2

P3 (User)

Runqueue

Let's try to have a simplistic view ...

`schedule()`

P1

P2

P3 (User)

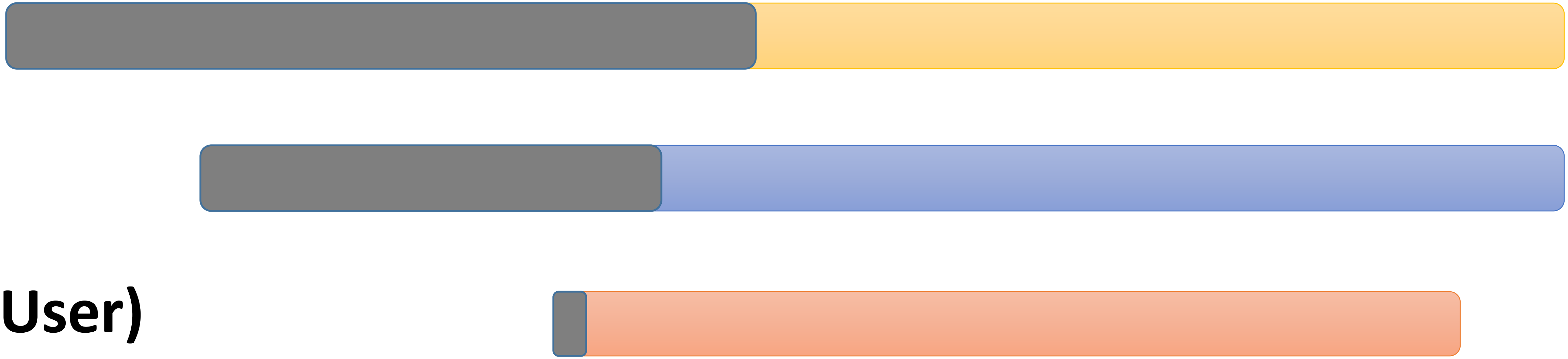
Runqueue

Let's try to have a simplistic view ...

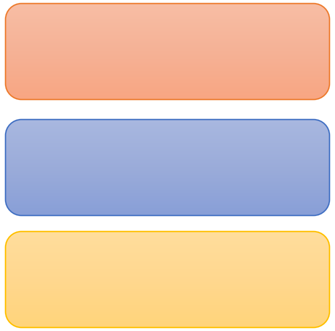
P1

P2

P3 (User)



Runqueue



Let's try to have a simplistic view ...

P1

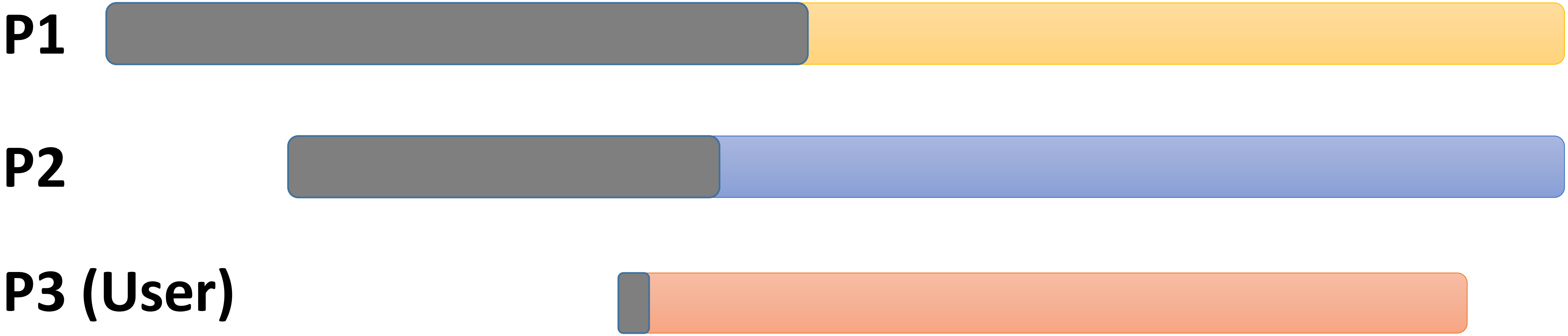
P2

P3 (User)

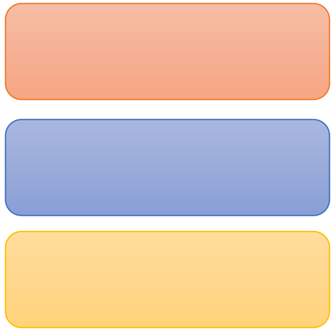
Runqueue

How do you control the execution of a user process?

Let's try to have a simplistic view ...



Runqueue



How do you control the execution of a user process?

Hardware to rescue !!!

Let's try to have a simplistic view ...

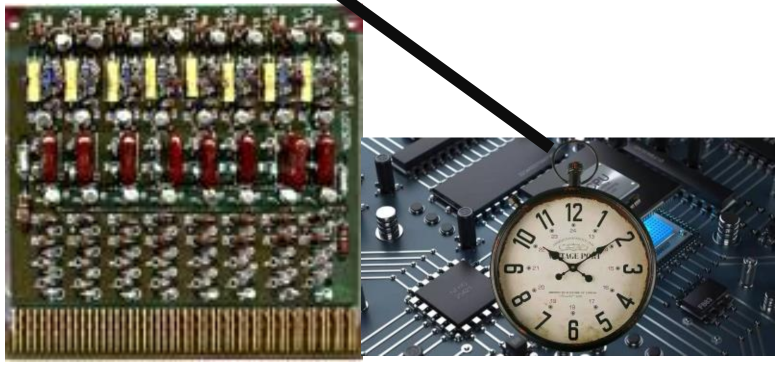
P1

P2

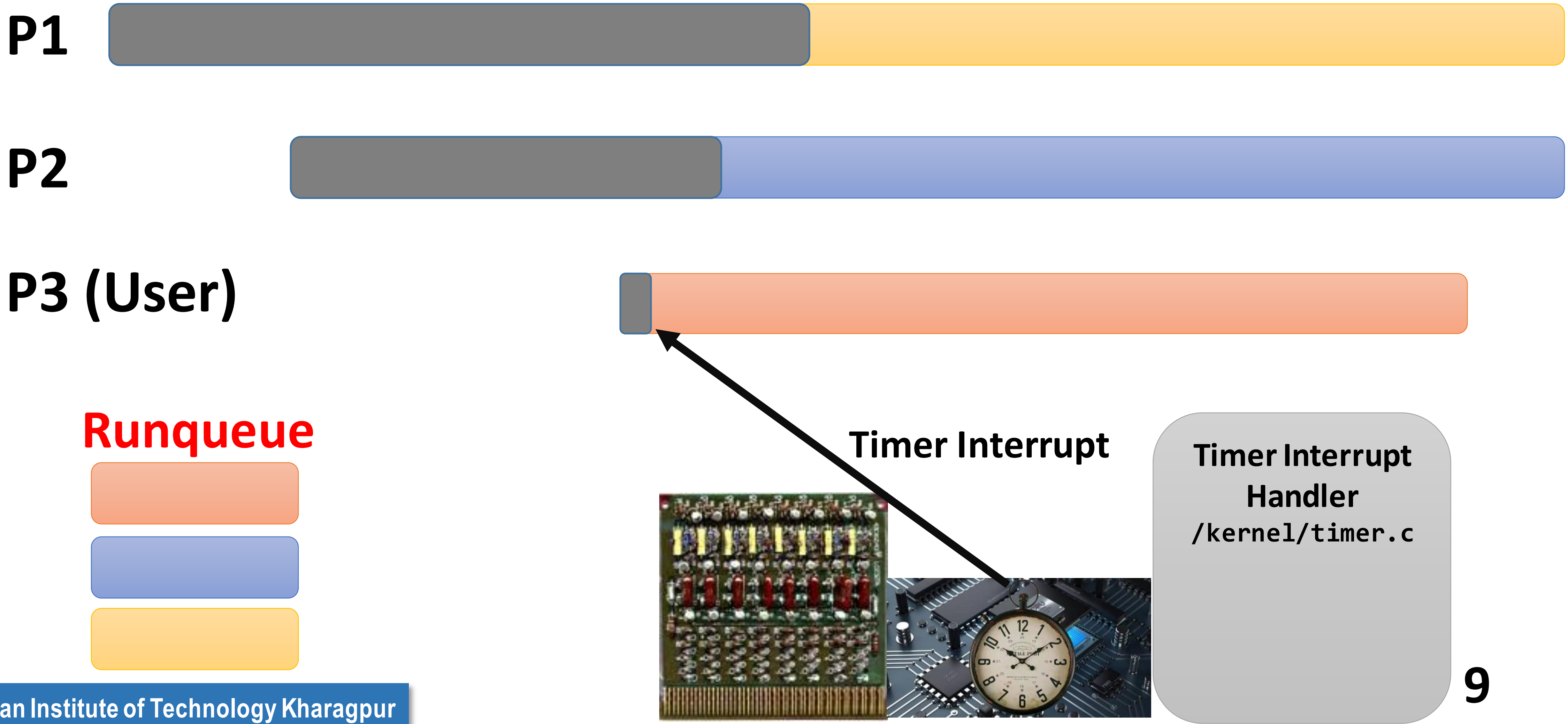
P3 (User)

Runqueue

Timer Interrupt



Let's try to have a simplistic view ...

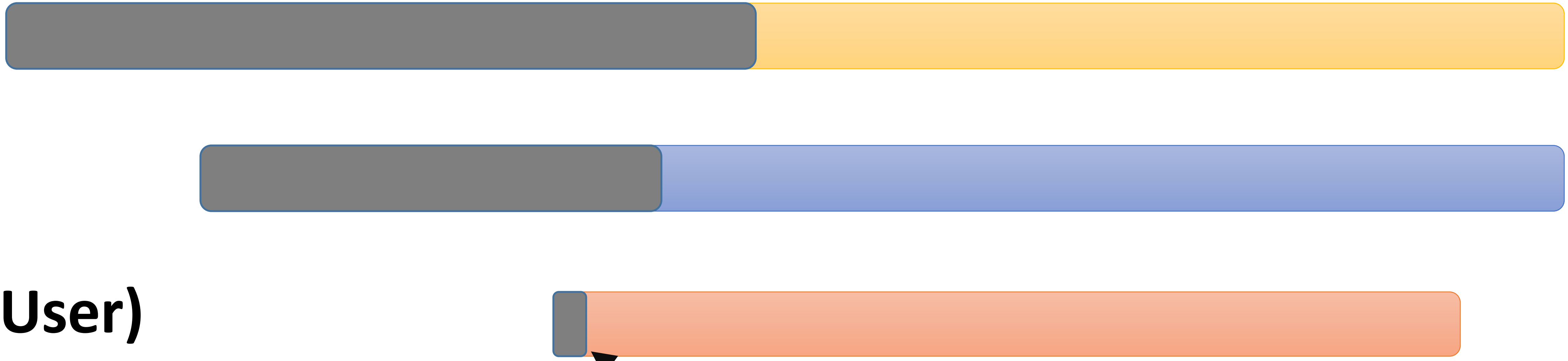


Let's try to have a simplistic view ...

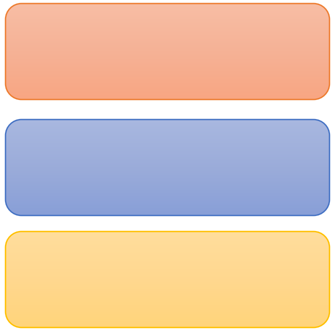
P1

P2

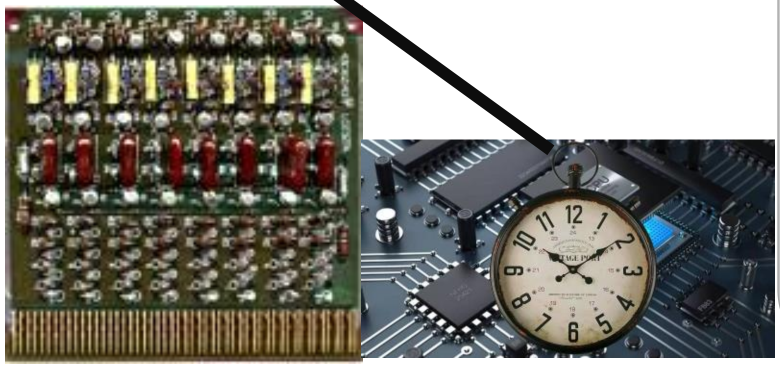
P3 (User)



Runqueue



Timer Interrupt



Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`

Let's try to have a simplistic view ...

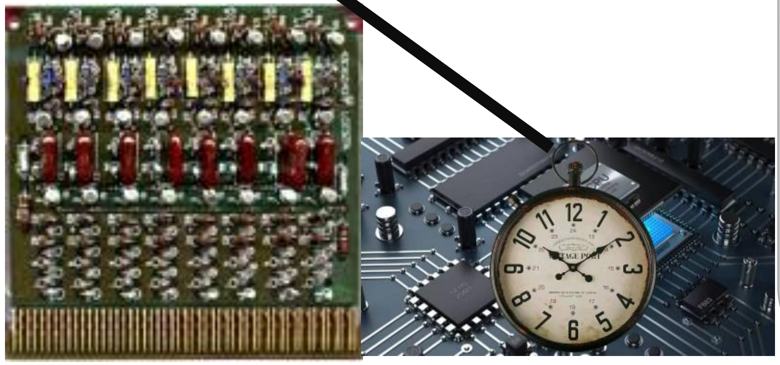
P1

P2

P3 (User)

Runqueue

Timer Interrupt



Timer Interrupt Handler
/kernel/timer.c

update_process_times()
scheduler_tick()

Let's try to have a simplistic view ...

P1

P2

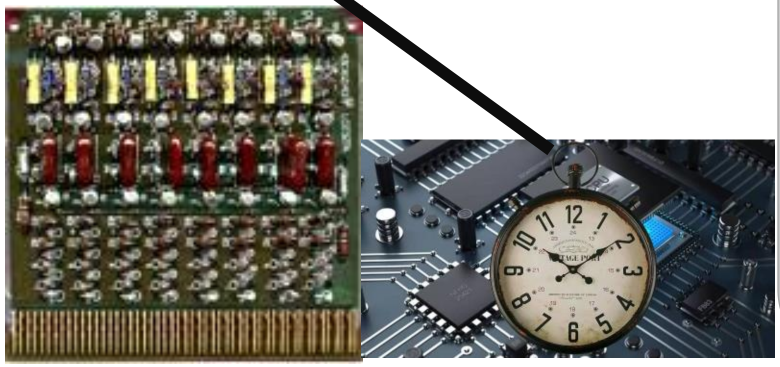
P3 (User)

Runqueue

`scheduler_tick()`

Check the current runqueue

Timer Interrupt



Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`
`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

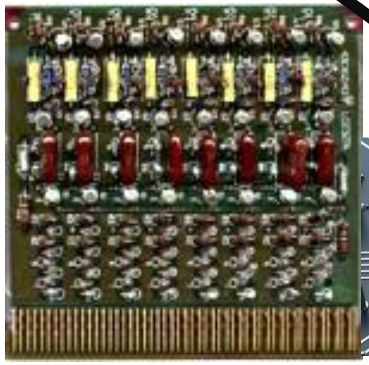
P3 (User)

Runqueue

`scheduler_tick()`

Timer expires or high priority process waiting – Pre-empt

Timer Interrupt



Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`
`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

P3 (User)

Timer Interrupt

Runqueue

`scheduler_tick()`

Timer expires or high priority process waiting – Pre-empt

Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`
`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

P3 (User)

Runqueue

`scheduler_tick()`

Timer expires or high priority process waiting – Pre-empt

Timer Interrupt

Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`
`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

P3 (User)

Timer Interrupt

Runqueue

`scheduler_tick()`

Timer expires or high priority process waiting – Pre-empt

Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`
`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

P3 (User)

Timer Interrupt

Runqueue

`scheduler_tick()`

<u>Need for premption</u>

Call `schedule()`

Timer Interrupt Handler
`/kernel/timer.c`

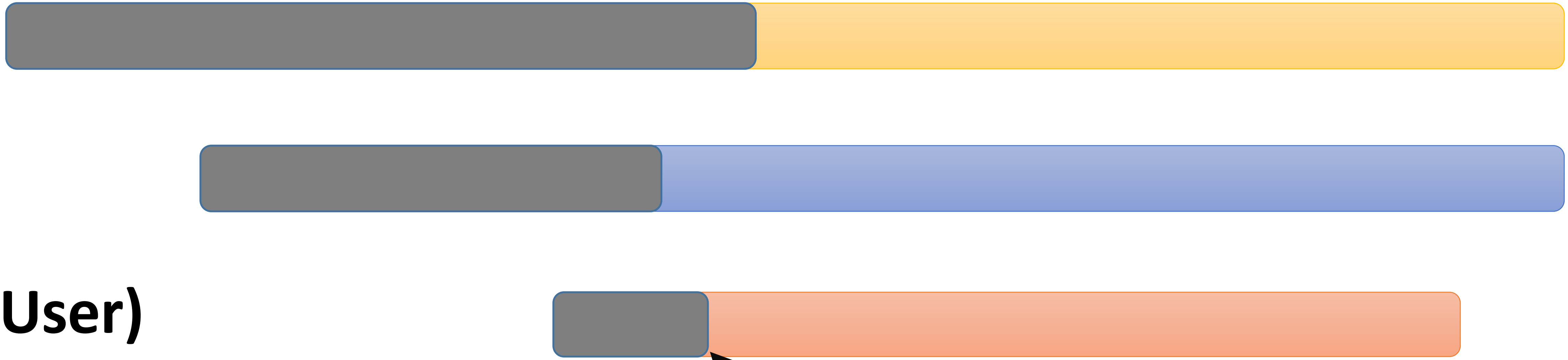
`update_process_times()`
`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

P3 (User)



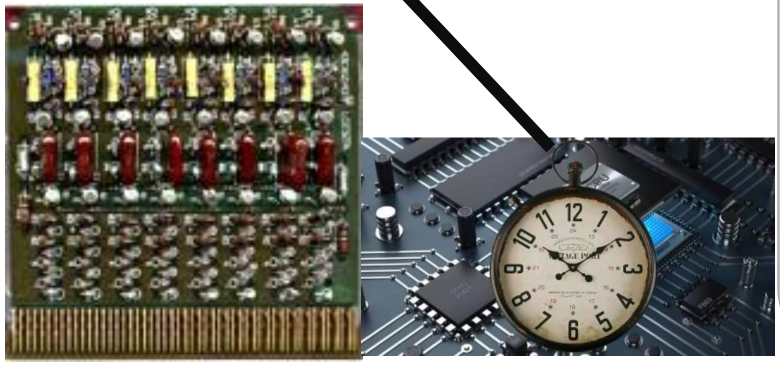
Timer Interrupt

Runqueue

scheduler_tick()

<u>Need for premption</u>

Call schedule()



Timer Interrupt Handler
/kernel/timer.c

update_process_times()
scheduler_tick()

Let's try to have a simplistic view ...

P1

P2

P3 (User)

The timer is triggered through a hardware clock, so it is applied on all the processes running – the currently running process will be preempted to run the Timer Interrupt Handler

Runqueue

`scheduler_tick()`

<u>Need for premption</u>

Call `schedule()`

Timer Interrup

Timer Interrupt Handler

`/kernel/timer.c`

`update_process_times()`

`scheduler_tick()`

Let's try to have a simplistic view ...

P1

P2

P3 (User)

A (kernel) process can directly call the `schedule()` function or it is called through the function `scheduler_tick()`

Timer Interrupt

Runqueue

`scheduler_tick()`

Need for premption

Call `schedule()`

Timer Interrupt Handler
`/kernel/timer.c`

`update_process_times()`
`scheduler_tick()`

Invoking the Scheduler

- The main scheduler function `schedule()` is called from many different places in the kernel and for various occasions.
  - The invocations can be direct or lazy
  - Lazy invocation does not call the function by its name – gives the kernel a hint that the scheduler needs to be called soon

Invoking the Scheduler

- The main scheduler function `schedule()` is called from many different places in the kernel and for various occasions.
  - The invocations can be direct or lazy
  - Lazy invocation does not call the function by its name – gives the kernel a hint that the scheduler needs to be called soon

- **Periodic Scheduler**
  - `scheduler_tick()` is called on every timer interrupt – frequency is set during the Kernel compilation time (default 1000)
  - Tells the kernel whether and when the process needs to be scheduled next depending on the possibility of preemption
  - In case an interrupt recommends that this process needs to be preempted, then the same is done during the execution of `scheduler_tick()`

Invoking the Scheduler

- **Currently running task enters the sleep state**
  - The task voluntarily gives up the CPU, waits for certain event to happen
  - The calling task adds itself to a *wait-queue* – sets itself as **TASK_INTERRUPTABLE** (can be interrupted when the event occurs) or **TASK_UNINTERRUPTABLE** (does not respond to an interrupt, periodically checks itself whether the event has occurred).
  - Call `schedule()` right before it goes to sleep

Invoking the Scheduler

- **Currently running task enters the sleep state**
  - The task voluntarily gives up the CPU, waits for certain event to happen
  - The calling task adds itself to a *wait-queue* – sets itself as **TASK_INTERRUPTABLE** (can be interrupted when the event occurs) or **TASK_UNINTERRUPTABLE** (does not respond to an interrupt, periodically checks itself whether the event has occurred).
  - Call `schedule()` right before it goes to sleep

- **Sleeping task wakes up**
  - `wake_up()` function is executed in the corresponding wait queue
  - The task is set to runnable and put back in the runqueue
  - If the task has higher priority than other tasks in the runqueue, `TIF_NEED_RESCHED` flag is set – **Lazy invocation** (kernel often checks for this flag)

Happy Learning