Operating Systems Lab

CSCE 000/3402

Lab Lecture 3: Scheduling

Karim Sobh

kmsobh@aucegypt.edu

Introduction to Scheduling

- Scheduling algorithms were simple in **old batch systems**; just run the next job on the tape.
- More complex scheduling algorithms are needed by **multiprogramming timesharing systems**:
    - **Multiple users** waiting for a service.
    - **CPU** time is a **scarce** resource.
    - **User** perceived **performance** and **satisfaction** is an important measure.
- **Personal Computers:**
    - Most of the time there is only one active process.
    - Computers have gotten so much faster and CPU is rarely a scarce resource.
    - Limited to the rate at which the user can present input.
    - Scheduling does not matter much on simple PCs.
- **Networked Servers:**
    - Multiple processes often do compete for the CPU.
    - Scheduling matters again.

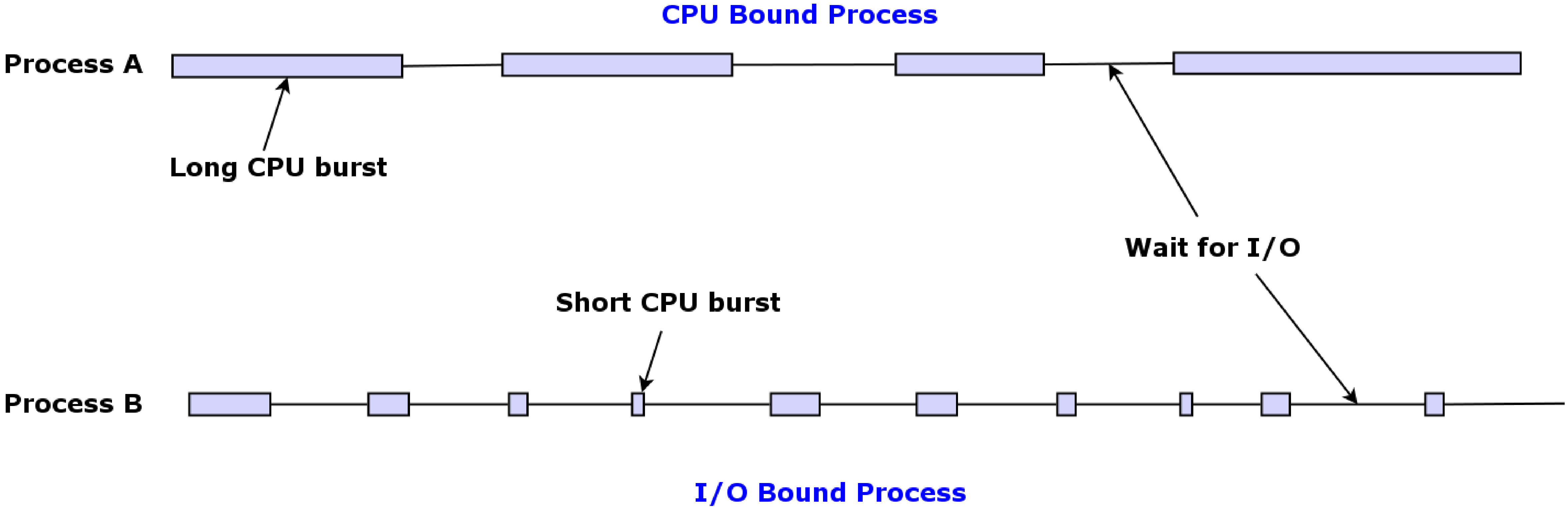
Scheduling Decision Factors

- Essentially, a scheduler decision will result in sharing the CPU among different processes.
- The sharing is implemented through **context switching**; switching from one running process to another.
- **Two** important **factors** affect the scheduler decision:
  - **Picking** up the **right process** to run.
  - **Making efficient use** of the **CPU** because of the expensive process of switching.

What Happens in a Context Switch?

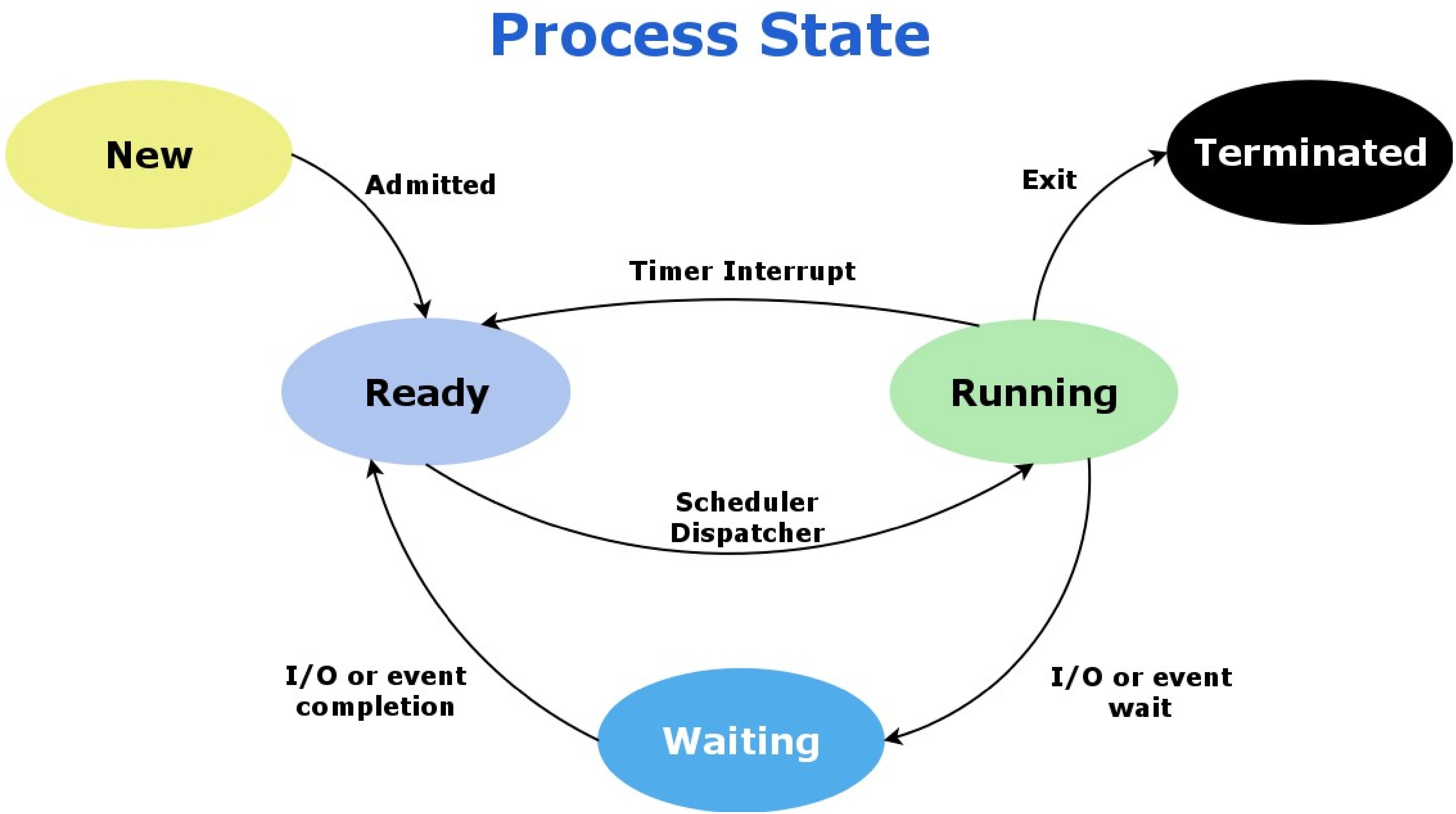
- A switch from user mode to kernel mode must occur.
- The state of the current process must be saved;
  - Storing registers in the process table so they can be reloaded later.
  - Might need to store the memory map as well.
- The new process must be then selected by running the scheduler algorithm.
- MMU must be reloaded with the new process memory map.
- The new process must be started.
- The process switch may invalidate the memory cache and the memory map **T**able **L**ook-aside **B**uffer TLB.
- Performing too many process switches per second waste a considerable amount of the CPU time; consumed in context switching.

Process Behavior



- **Bursts** of CPU usage **alternate** with periods of waiting for I/O.
- **CPU-bound process:** relatively spend more time in CPU processing operations than performing I/O.
- **I/O-bound process:** initiate many kernel I/O system calls and relatively spend more time waiting for I/O operations.

Process States



When to Schedule?

- **When a new process is created;** whether to run the parent process or the child process.
- **When a process exists;** the process can no longer run so another process must be chosen.
- **When a process blocks** for I/O, semaphore, or for any other reason.
- **When an I/O interrupt occurs**; a process waiting for I/O completion might need to be started.
- **If the timer interrupt is enabled**, that fires a number of time per second, scheduling decisions can be made at each timer interrupt

Preemptive vs. Nonpreemptive Scheduling

- **Nonpreemptive Scheduling:**
  - Scheduling algorithm picks a process to run.
  - The process acquires the CPU until:
    - It blocks (wait for I/O or for another process), or
    - Voluntarily releases the CPU.
  - No scheduling decisions are made during timer interrupts.
    - Even if the timer interrupt is enabled the same process that was running before the timer interrupt will resume.
- **Preemptive Scheduling:**
  - Scheduling algorithm picks a process and lets it run.
  - The process runs for a maximum of some fixed time (quantum).
  - It gets suspended by the scheduler if still running by the end of its quantum and no I/O requests or blocking was initiated.

Categories of Scheduling Algorithms

- **Batch:**
  - Still in widespread use in business world, e.g. rolling up payroll, inventory, accounts receivable, ...etc.
  - No users impatiently are waiting at their terminals for quick response.
  - Reduces process switches and thus improves performance.
  - Fairly general and often applicable to many situations.
- **Interactive:**
  - Interactivity is essential to keep processes from hogging the CPU.
  - Used to avoid starvation.
  - Usually used in online servers environments.
  - Designed to serve multiple users.
- **Realtime:**
  - Designed to meet deadlines.
  - Preemption might conflict with the system targets.

Scheduling Goals

- **All Systems:**
  - **Fairness:** give each process a fair share of the CPU time.
  - **Policy Enforcement:** ensure that stated policy is carried out.
  - **Balance:** keeping all parts of the system busy.
- **Batch Systems:**
  - **Throughput:** maximize jobs per unit time.
  - **Turnaround time:** minimize time users waiting for jobs.
  - **CPU utilization:** keep the CPU busy all the time.
- **Interactive Systems:**
  - **Response time:** respond to requests quickly.
  - **Proportionality:** meet users' expectations.
- **Real-time Systems:**
  - **Meeting deadlines:** respond to requests quickly.
  - **Predictability:** avoid quality degradation.

Performance Metrics

- **Throughput:** number of jobs per time unit.
- **Turnaround time:** average time from the moment a batch job is submitted until the moment it is completed.
  - Measures how long the average user has to wait.
- **Response time (Most important):** time between issuing a command and getting the result.
  - Applies for interactive requests within a running task.
- **Proportionality:** being close to users expectations about the execution time of a task.

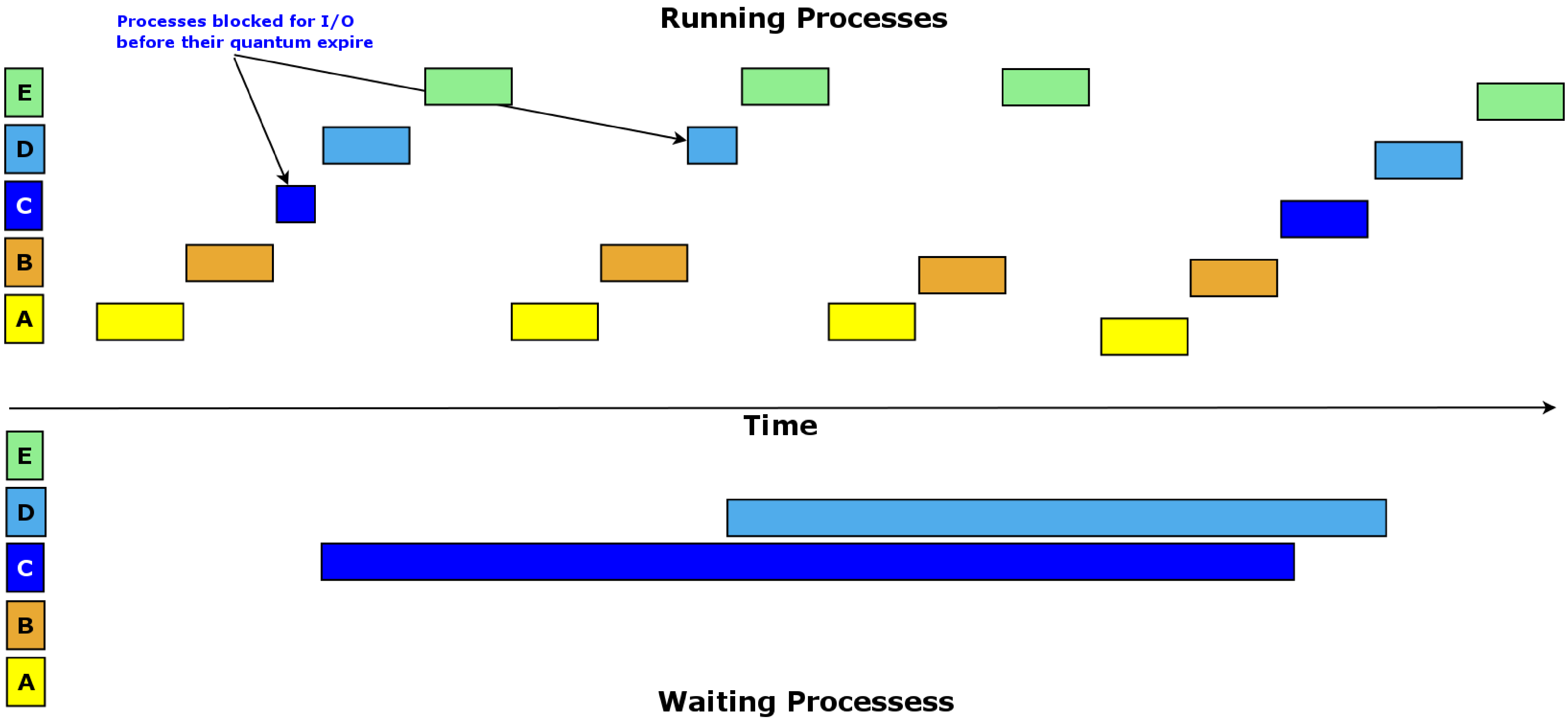
Scheduling in Interactive Systems

- We will study seven scheduling algorithms that are used in Interactive Systems:
  - Round-Robin Scheduling (RR).
  - Priority Scheduling.
  - Multiple Queues Scheduling.
  - Shortest Job Next.
  - Guaranteed Scheduling.
  - Lottery Scheduling.
  - Fair-Share Scheduling.

Round-Robin Scheduling (RR)

- **Interactive** scheduling algorithm, which is very **easy** to **implement**.
- Assigns a fixed time interval to each process during which it is allowed to run; called **quantum**.
- Performs **strict rotation** over processes in the ready queue.
- If a process is still running at the end of its quantum, the process having the CPU is preempted and the CPU is given to the next process in the ready queue.
- If the running process has blocked or finished execution before the quantum has elapsed, this results in a trap during which the CPU switching is done.
- The scheduler needs to maintain a list of runnable processes.
- When a process uses up its quantum, it gets put at the end of the list.
- **The question is how to decide on the quantum duration?**
  - A too short quantum will lead to higher number of switches and hence will lead to degraded performance.
  - A too long quantum will affect negatively the interactivity.
  - A quantum around **20-50 msec** is often a reasonable compromise.

Round-Robin Scheduling (RR)



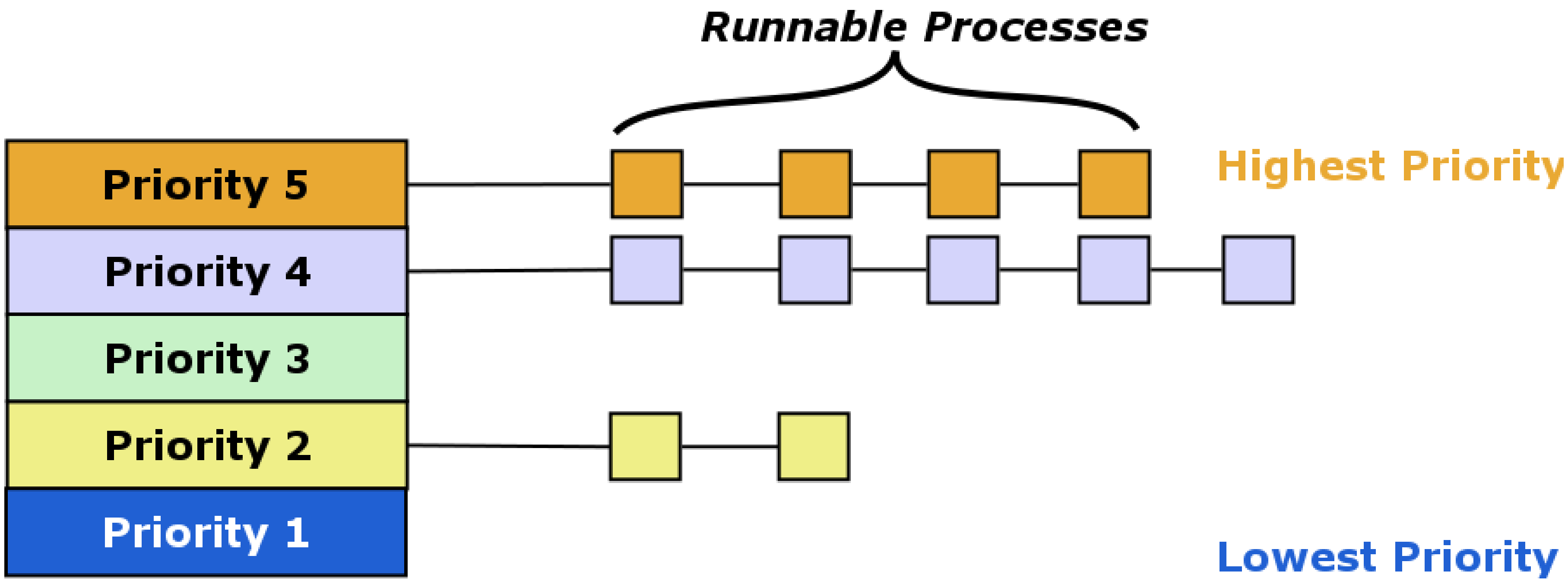
Priority Scheduling

- In Round-Robin scheduling, all processes are assumed to be equally important.
- In some environments, processes do not have the same level of importance.
- The need to take external factors into account leads to **priority scheduling**.
- Each process is assigned a priority.
- The scheduler picks up the next highest priority runnable process.
- Priorities must be changed regularly to avoid starvation.
- Priorities can be assigned statically or dynamically based on system goals.
    - More on this when we study FreeBSD ULE scheduler.

Multiple Queues

- Combine Priority Scheduling with Round-Robin.
- Group processes into priority classes.
- Use **priority scheduling** among the classes.
- Use **round-robin** within each class.
- Processes can move between different priority classes upon changing their priorities.
- Priorities need to be adjusted occasionally to avoid starvation.

Multiple Queues



- As long as there are runnable processes in priority class 5, just run each one for one quantum in round-robin fashion.
- Never bother with lower priority classes.
- When priority class 5 gets empty start running processes in class 4 round-robin.
- When class 4 is empty, start running processes in class 2 round-robin, and so on.
- If priorities are not adjusted occasionally, lower priority classes may all starve to death.

Scheduling in FreeBSD

- The FreeBSD scheduler has a well-defined set of kernel-application interfaces (kernel APIs), that allows it to support different schedulers.
- Since FreeBSD 5.0, the kernel had two schedulers.
  - The traditional 4.4BSD scheduler; still maintained but no longer the default.
  - The ULE scheduler; the new one which we will study here as a case study.
    - It is located in /*sys*/kern/sched_ule.c
    - The name is not an acronym.
    - If you remove the underscore in the file name, the rational of its name becomes apparent.

FreeBSD Process Types

- **Realtime:** very responsive.
- **Interactive:** terminal-like which are dominated by user interaction.
- **Timeshare:** batch-like which requires a lot of CPU, and run for long durations.

FreeBSD Scheduler Objectives

- Realtime, and interactive must start execution after waking up from sleep and as soon as they are ready.
- Minimize the time taken by the scheduler to select a thread.
- Manage CPU affinity in an optimum way.
- Give a fair CPU slice, quantum, to every thread.
- Be able to classify threads into different type categories based on their execution pattern; realtime, interactive, or timeshare.

FreeBSD Scheduler

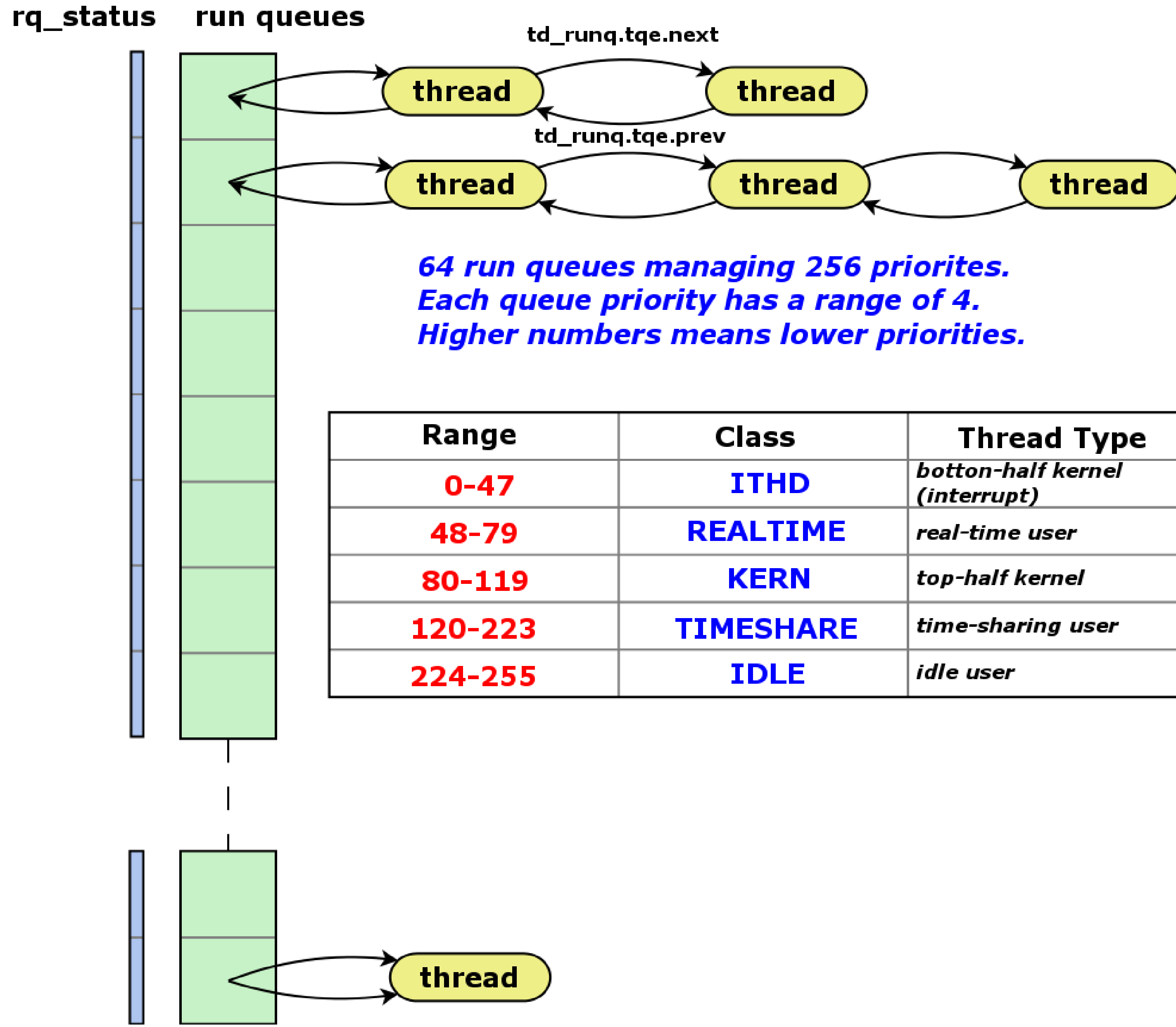
- Based on a multi-level feedback priority queues.
- Supports CPU affinity and has a constant execution time.
- Can identify interactive tasks and give them higher priority.
- FreeBSD is a two-level scheduler.
  - **Low-level scheduler:**
    - Runs frequently on every timer interrupt.
    - Picks the next thread to be run.
    - Very simple and fast.
  - **High-level scheduler:**
    - Runs less often.
    - Sets threads priorities.
    - Load-balance threads among processors.

High-level Scheduler Components

- Several Queues.
- Interactivity Scorer.
- Priority Calculator.
- CPU usage estimator.
- Slice Calculator.
- Two CPU load-balancing algorithm.

Scheduler Queues – Run Queues

- The low-level scheduler invokes runq_choose():
  - Ensures that lock is acquired.
  - Locates nonempty run queue by checking the rq_status bitmap.
  - If all queues are empty go to a pre-existing idle-loop thread.
  - Else remove the first thread from the selected queue.
  - If the queue becomes empty clear the rq_status bit.
  - When the thread consumes its quantum insert it at the end of it run queue.
- Other functions are runq_add() and runq_remove().



64 run queues managing 256 priorites.
Each queue priority has a range of 4.
Higher numbers means lower priorities.

Range	Class	Thread Type
0-47	ITHD	*botton-half kernel (interrupt)*
48-79	REALTIME	*real-time user*
80-119	KERN	*top-half kernel*
120-223	TIMESHARE	*time-sharing user*
224-255	IDLE	*idle user*

Scheduler Queues – Other Queues

- All ready to run threads are managed by run queues.
- As soon as a thread blocks for any reason it should be put on another set of queues that the low-level scheduler does not pick processes from.
- The thread will remain there until the reason behind the block is resolved.
- **Two extra queues are provided for this purpose:**
  - **Turnstile:** stores thread blocked on short-term locks.
  - **Sleepqueue:** stores threads blocked on medium and long term queue.

Interactivity Scorer

- Interactivity of a thread is determined through calculating the ratio of its voluntary sleep time versus its run time.
- The calculated interactivity is compared to a fixed threshold.
- Based on the comparison the thread is either placed in realtime queues, timeshare queues, or idle queues.
- The scaling factor is the maximum interactivity score divided by 2.

$$interactivity = \begin{cases} \dfrac{scaling\_factor}{sleep/runtime} & \text{for } sleep > runtime \\[2ex] \dfrac{scaling\_factor}{runtime/sleep} + scaling\_factor & \text{for } sleep \le runtime \end{cases}$$

CPU Usage Estimator and Priority Calculation

- Priority is used to indicate the order of threads in the run queues.
- Priority is adjusted every 40 ms.
- The CPU usage estimation is calculated as the sum of the number of ticks occurred during a thread is running; ***estcpu***.
- The value decays when processes sleep; improves priority of the processes.
- ***nice*** is a thread parameter that can be set by a user to lower the priority of a thread.
- ***estcpu***, ***priority range***, and ***nice*** are used to update the priority of a thread.

Slice Calculator

- The slice size is not fixed for all threads.
- Slices size is calculated as a function of the ***nice*** value.
- The nice values of the threads in the run queue are stored with each thread.
- The minimum nice value (the least nice process) is identified by the scheduler.
- The scheduler allows only threads within 20 of the least nice threads to obtain a slice; we call that the nice window.
- The rest of the threads get slices of size 0; when they are selected to be run their slice is reevaluated.
- The threads within the nice window are given a slice value such that → ***slice = 1/(thread nice – least nice)***

CPU Load Balancing

- The ULE is designed to work on multi-processor systems.
- The main idea is to schedule a thread on the same CPU it was last scheduled on.
- Moving threads between processors will flush the caches and will affect the page tables dramatically through flushing TLBs (More on this when we study memory).
- This needs to be done when its is really needed and of a benefit.
- Two load balancing mechanisms are used by the ULE scheduler.
  - **Pull mechanism:** an idle CPU steals a thread from a non-idle CPU.
  - **Push mechanism:** a periodic task evaluates the current load situation and apply thread migration, whenever needed, from overloaded CPUs to relatively less loaded ones.