DCS216 Operating Systems

Lecture 17
Scheduling (2)

Apr 24th, 2024

Instructor: Xiaoxi Zhang

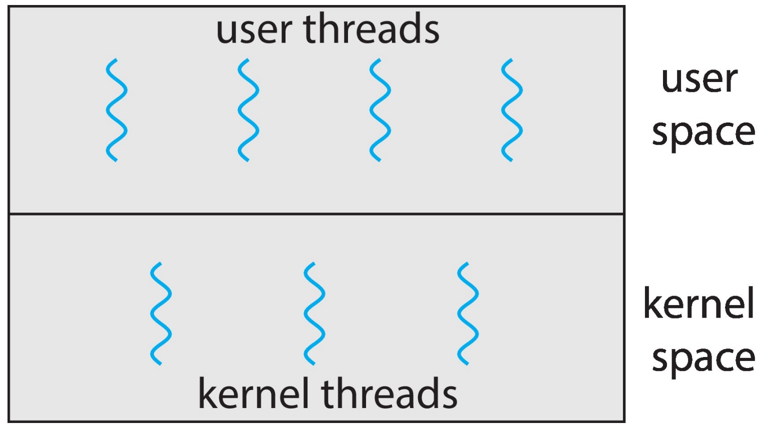
Sun Yat-sen University

Content

- Content
  - Thread Scheduling
  - Multi-Processor Scheduling
    - Multicore Processors
    - Load Balancing
    - Processor Affinity
    - Heterogeneous Multiprocessing
  - Real-Time CPU Scheduling
    - Minimizing Latency
    - Priority-Based Scheduling
    - Rate-Monotonic Scheduling
    - Earliest-Deadline-First Scheduling (**EDF**)
    - Proportional Share Scheduling
    - POSIX Real-Time Scheduling
  - Operating System Examples
  - Algorithm Evaluation

Thread Scheduling

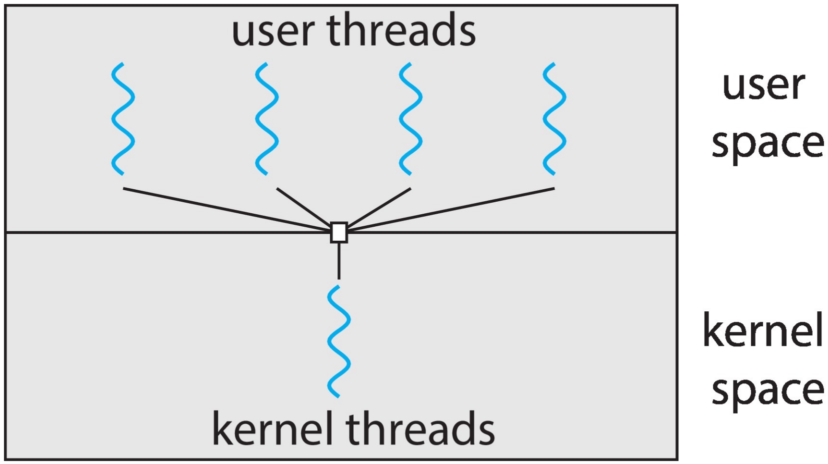
- **User-Level** Threads vs. **Kernel-Level** Threads

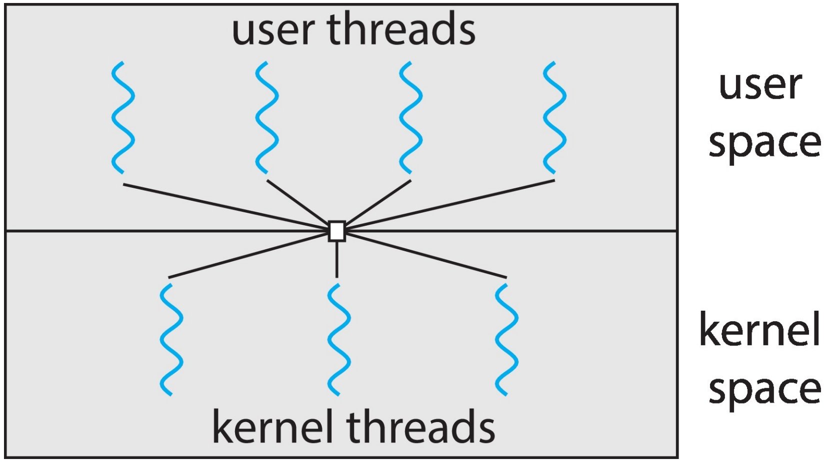


- On modern OSes, it is the **Kernel-Level Threads** (**NOT Processes**) that are being scheduled by the OS.
  - The kernel is unaware of user-level threads, which are managed by a thread library
- To run on a CPU, **User-Level Threads** must ultimately be mapped to an associated **Kernel-Level Thread** (e.g., via a **LWP, Lightweight Process**)

Thread Scheduling

- On systems implementing **Many-to-One** and **Many-to-Many** models, thread library schedule ULTs to run on an available **LWP**.

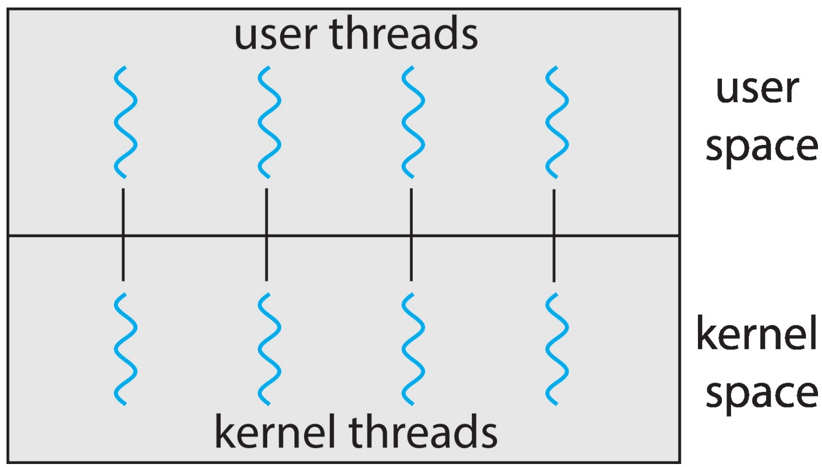




- This scheme is known as **Process-Contention Scope (进程竞争域)**, since competition for the CPU takes place among threads in the same process.
- Kernel threads scheduled onto available CPU –**System Contention Scope (系统竞争域)**: competition among all KLTs in the system.

Thread Scheduling

- Kernel threads scheduled onto available CPU –**System Contention Scope (系统竞争域)**: competition among all KLTs in the system.
  - Systems with One-to-One model, such as Linux, schedule threads using only **SCS**.



Thread Scheduling

- **PThread Scheduling**
  - Pthread API allows specifying **PCS** or **SCS** during thread creation:
    - `PTHREAD_SCOPE_PROCESS`
    - `PTHREAD_SCOPE_SYSTEM`
  - Functions for getting and setting the contention scope policy:
    - `pthread_attr_getscope(pthread_attr_t *attr, int scope);`
    - `pthread_attr_setscope(pthread_attr_t *attr, int *scope);`

Thread Scheduling

PThread Scheduling

```c
/* pthread_scope.c */
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#define NUM_THREADS 5

void *threadfun(void *arg) {
    printf("tid: %d\n", gettid());
    pthread_exit(0);
}
int main(int argc, char *argv[]) {
    int scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* Get default attr */
    pthread_attr_init(&attr);
    /* Inquire current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) {
        fprintf(stderr, "Unable to get sched scope\n");
    } else {
        if (scope == PTHREAD_SCOPE_PROCESS) {
            printf("PTHREAD_SCOPE_PROCESS\n");
        } else if (scope == PTHREAD_SCOPE_SYSTEM) {
            printf("PTHREAD_SCOPE_SYSTEM\n");
        } else {
            fprintf(stderr, "Illegal scope value.\n");
        }
    }
```

```c
    /* Set sched algo to PCS */
    if (pthread_attr_setscope(&attr,
                PTHREAD_SCOPE_PROCESS) != 0) {
        fprintf(stderr, "Failed to set PCS\n");
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&tid[i], &attr, threadfun, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}
```

```
$ ./pthread_scope
PTHREAD_SCOPE_SYSTEM
```

PThread Scheduling

```c
/* pthread_scope.c */
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#define NUM_THREADS 5

void *threadfun(void *arg) {
    printf("tid: %d\n", gettid());
    pthread_exit(0);
}
int main(int argc, char *argv[]) {
    int scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* Get default attr */
    pthread_attr_init(&attr);
    /* Inquire current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) {
        fprintf(stderr, "Unable to get sched scope\n");
    } else {
        if (scope == PTHREAD_SCOPE_PROCESS) {
            printf("PTHREAD_SCOPE_PROCESS\n");
        } else if (scope == PTHREAD_SCOPE_SYSTEM) {
            printf("PTHREAD_SCOPE_SYSTEM\n");
        } else {
            fprintf(stderr, "Illegal scope value.\n");
        }
    }
```

```c
    /* Set sched algo to PCS */
    if (pthread_attr_setscope(&attr,
                PTHREAD_SCOPE_PROCESS) != 0) {
        fprintf(stderr, "Failed to set PCS\n");
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&tid[i], &attr, threadfun, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}
```

```
$ ./pthread_scope
PTHREAD_SCOPE_SYSTEM
Failed to set PCS
```

Linux and macOS systems allow only PTHREAD_SCOPE_SYSTEM.

PThread Scheduling

```c
/* pthread_scope.c */
#define _GNU_SOURCE
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#define NUM_THREADS 5

void *threadfun(void *arg) {
    printf("tid: %d\n", gettid());
    pthread_exit(0);
}
int main(int argc, char *argv[]) {
    int scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* Get default attr */
    pthread_attr_init(&attr);
    /* Inquire current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) {
        fprintf(stderr, "Unable to get sched scope\n");
    } else {
        if (scope == PTHREAD_SCOPE_PROCESS) {
            printf("PTHREAD_SCOPE_PROCESS\n");
        } else if (scope == PTHREAD_SCOPE_SYSTEM) {
            printf("PTHREAD_SCOPE_SYSTEM\n");
        } else {
            fprintf(stderr, "Illegal scope value.\n");
        }
    }
```

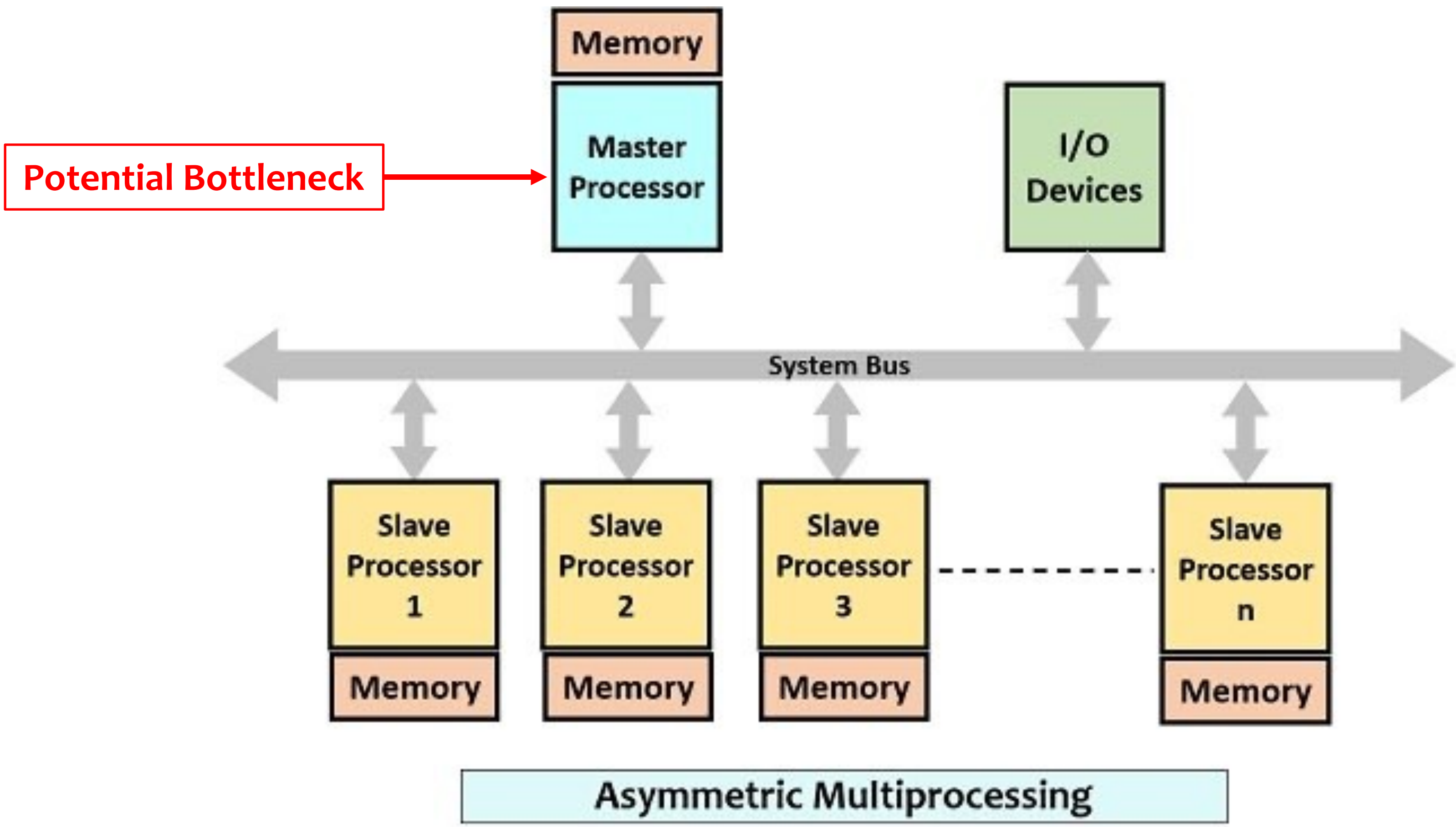
```c
    /* Set sched algo to PCS */
    if (pthread_attr_setscope(&attr,
                PTHREAD_SCOPE_PROCESS) != 0) {
        fprintf(stderr, "Failed to set PCS\n");
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&tid[i], &attr, threadfun, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}
```

```
$ ./pthread_scope
PTHREAD_SCOPE_SYSTEM
Failed to set PCS
tid: 16770
tid: 16768
tid: 16771
tid: 16772
tid: 16769
```

Multi-Processor Scheduling

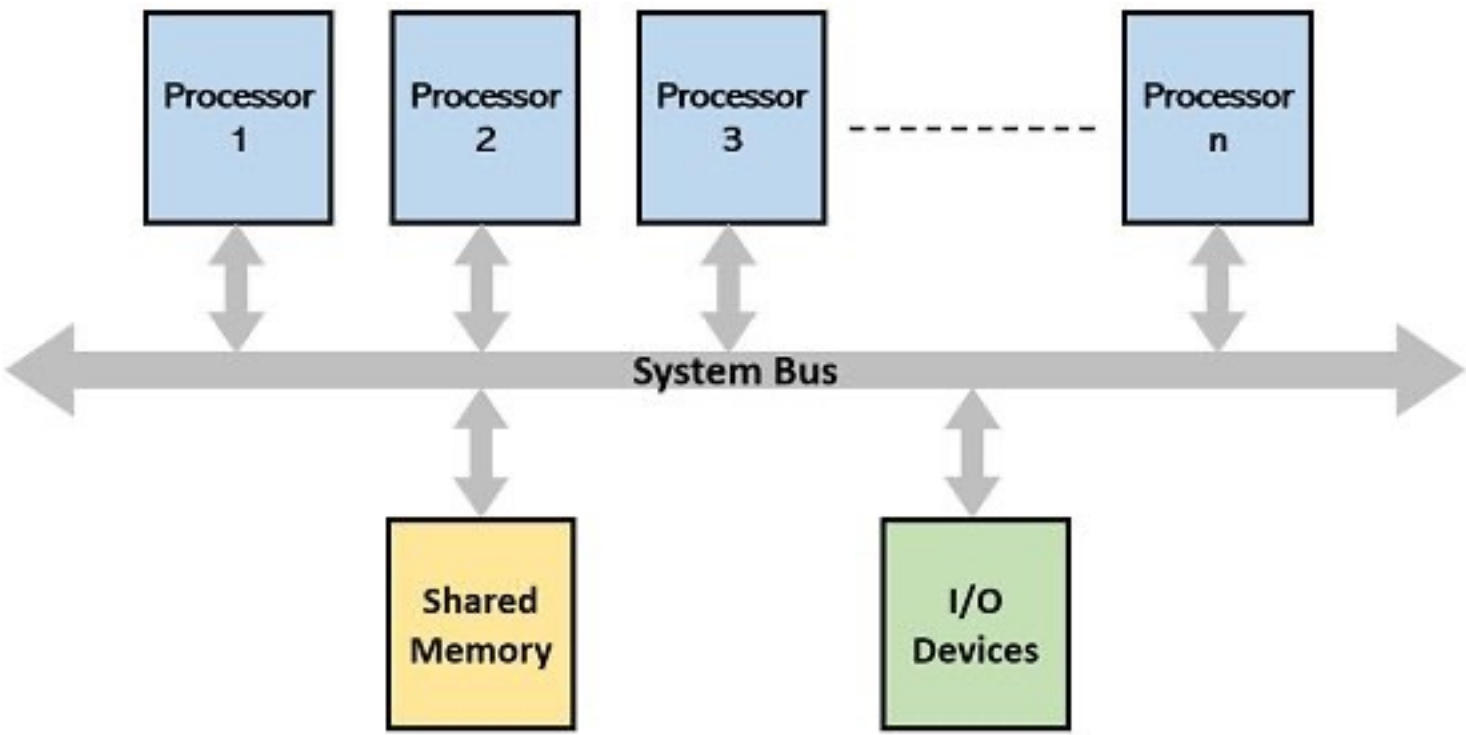
- So far, we have focused on scheduling on a single processing core.
- CPU scheduling more complex when multiple CPUs are available.
  - Do FCFS, RR, SJF, SRTF,... work on multi-processor systems as well?
- **Multiprocessor** may be any of the following architectures:
  - Multicore CPUs (多核CPU)
  - Multithreaded Cores (多**线程**核)
  - NUMA systems (**N**on-**U**niform **M**emory **A**ccess, 非均匀访存系统)
  - Heterogeneous Multiprocessing (异构多处理)

- **Approaches to Multiple-Processor Scheduling**
  - **A**symmetric **M**ulti**p**rocessing (**AMP**)
    - All scheduling decisions handled by **a single (master) processor.**
    - Work offloaded to **slave processors.**



Approaches to Multiple-Processor Scheduling

- **S**ymmetric **M**ulti**p**rocessing (**SMP**)
  - Each processor is self-scheduling.



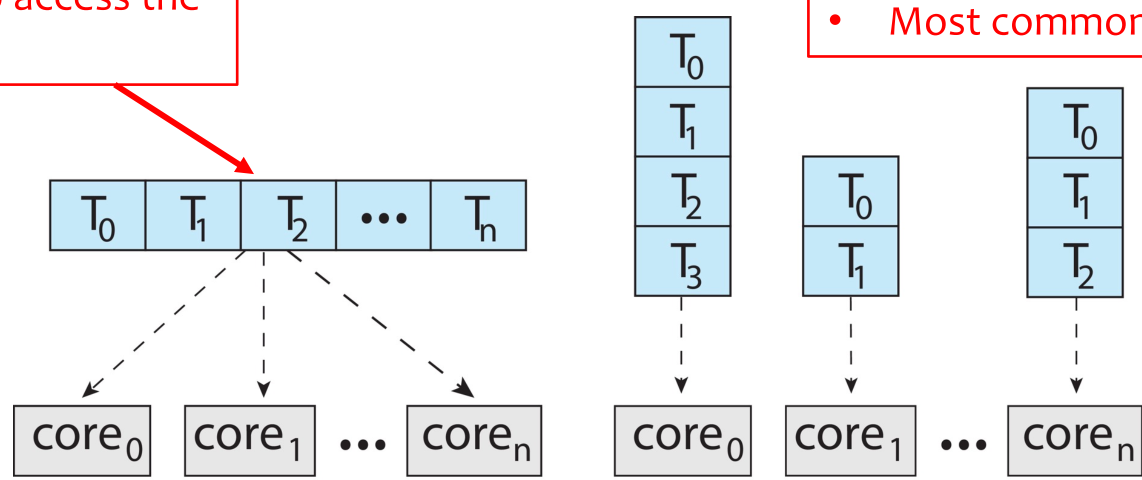
Symmetric Multiprocessing

Approaches to Multiple-Processor Scheduling

- **S**ymmetric **M**ulti**p**rocessing (**SMP**)
  - Each processor is self-scheduling.
  - Two possible strategies for organizing the threads to be scheduled:
    a) All threads in a **common** ready queue.
    b) Each processor has its own **private** queue of threads.

Possible race condition on shared queue; must use some form of locking to access the shared queue.

- No race condition among queues;
- Better performance;
- Communication overhead;
- Most common approach in SMP.

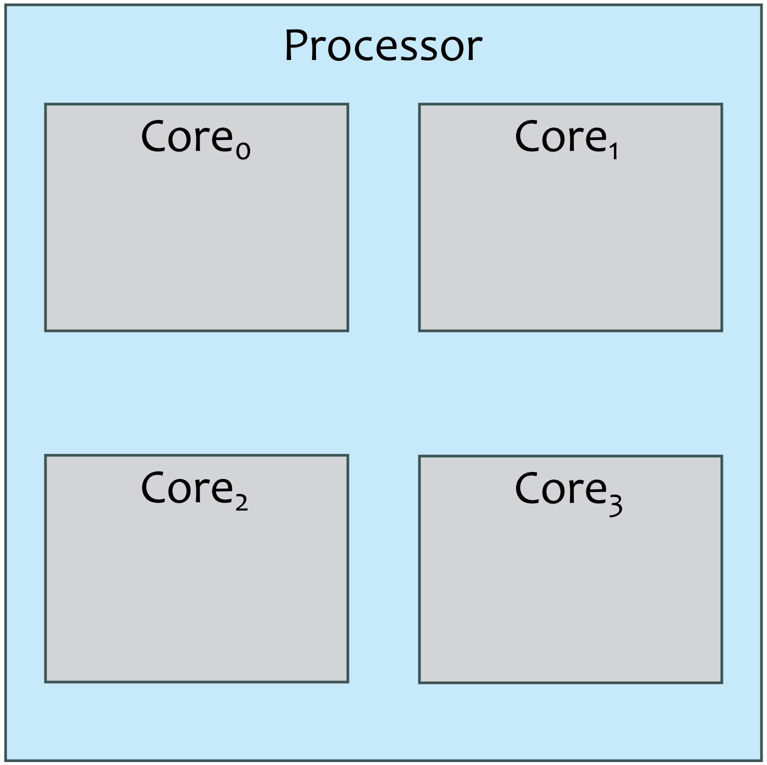


common ready queue
(a)

per-core run queues
(b)

Multi-Processor Scheduling

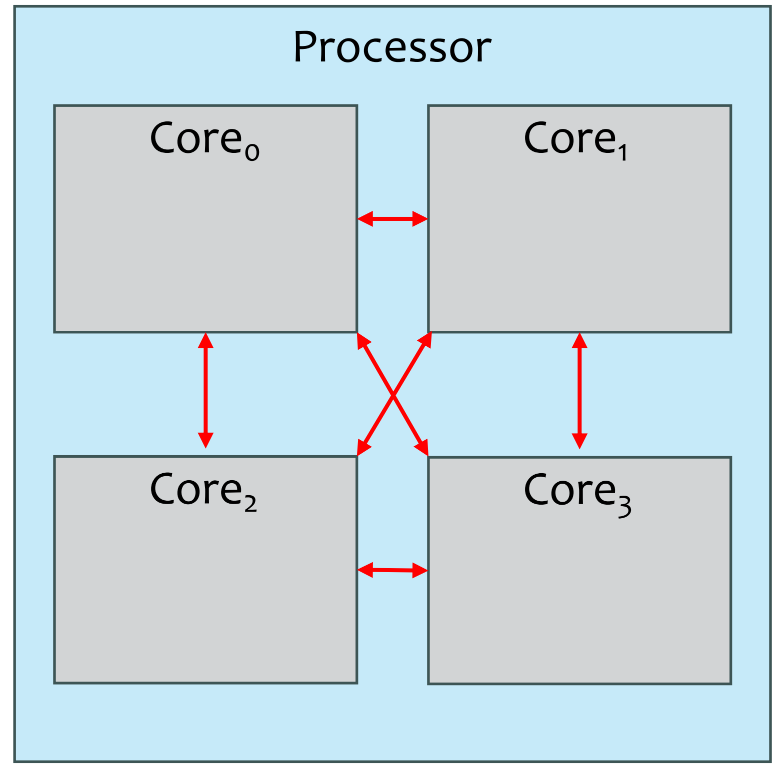
- Multicore CPUs
  - Multiple Cores on the same physical processor chip.

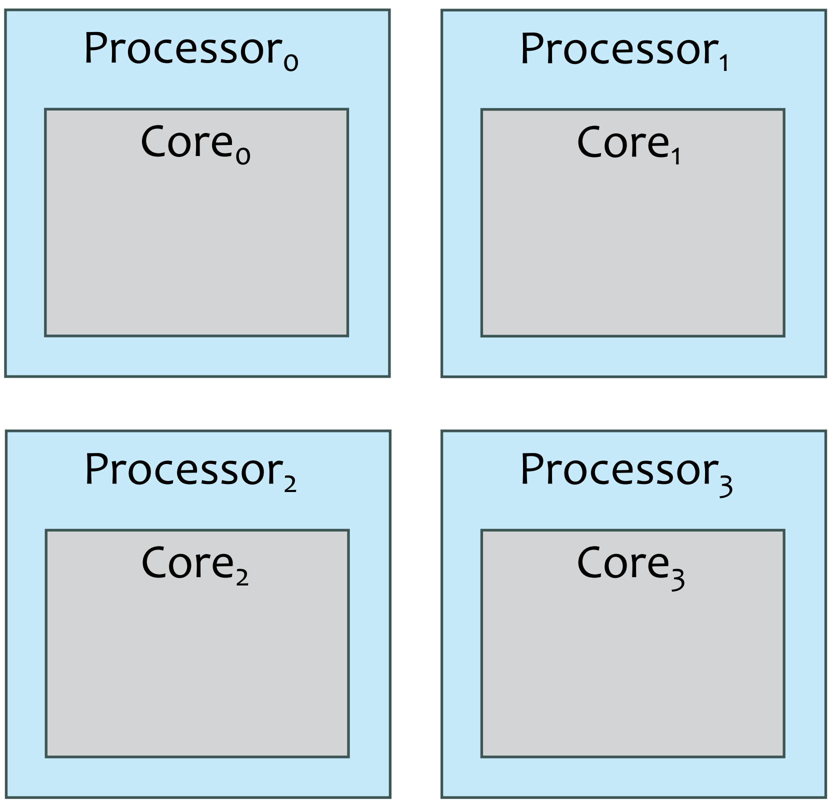


Multi-Processor Scheduling

Multi-Processor Scheduling

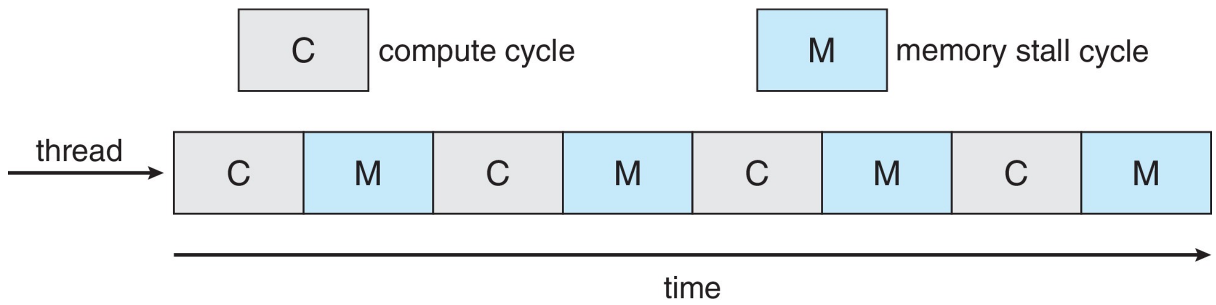
- Multicore CPUs
  - vs. Multiple Physical CPUs each with a single processor core.





A Multicore CPU with 4 cores is generally **faster** and more **energy efficient** than 4 single-core CPUs. (*due to better interconnect between cores on the same physical chip*)

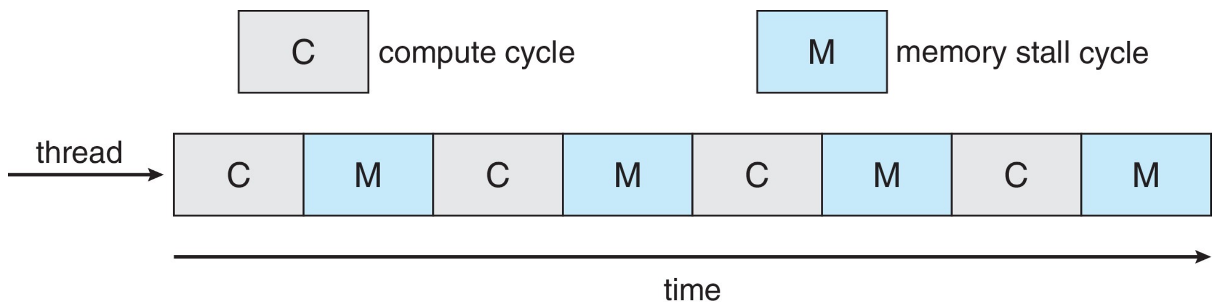
- **Memory Stall (内存停顿)**
  - When a processor access memory, it spends a significant amount of time waiting for the data to become available.
  - This situation, known as **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory.



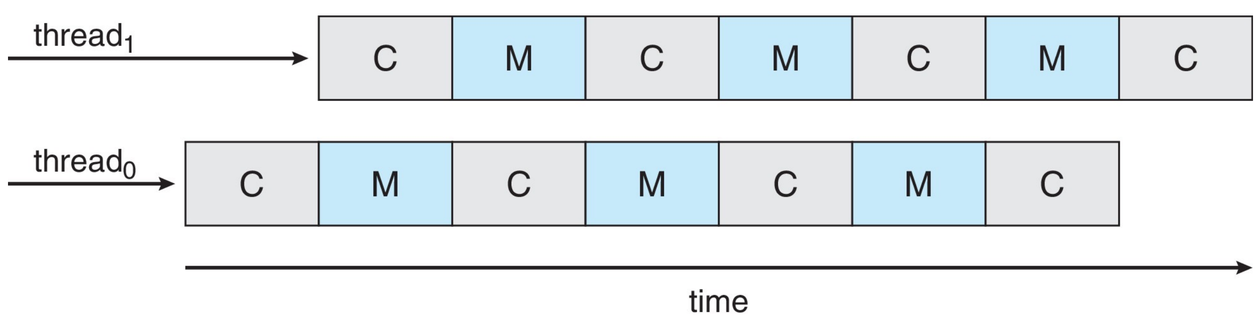
Multi-Processor Scheduling

Memory Stall (内存停顿)

- When a processor access memory, it spends a significant amount of time waiting for the data to become available.
- This situation, known as **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory.

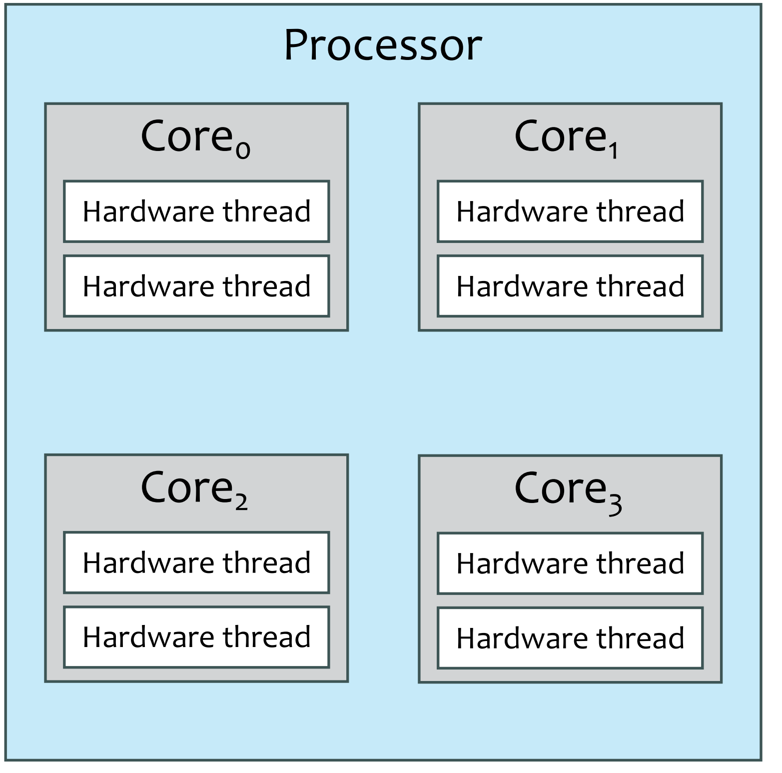


- Recent hardware designs implement **multi**threaded processor cores
  - two (or more) hardware threads assigned in each **core**.
    - If one thread memory stalls, the core can switch to another thread.



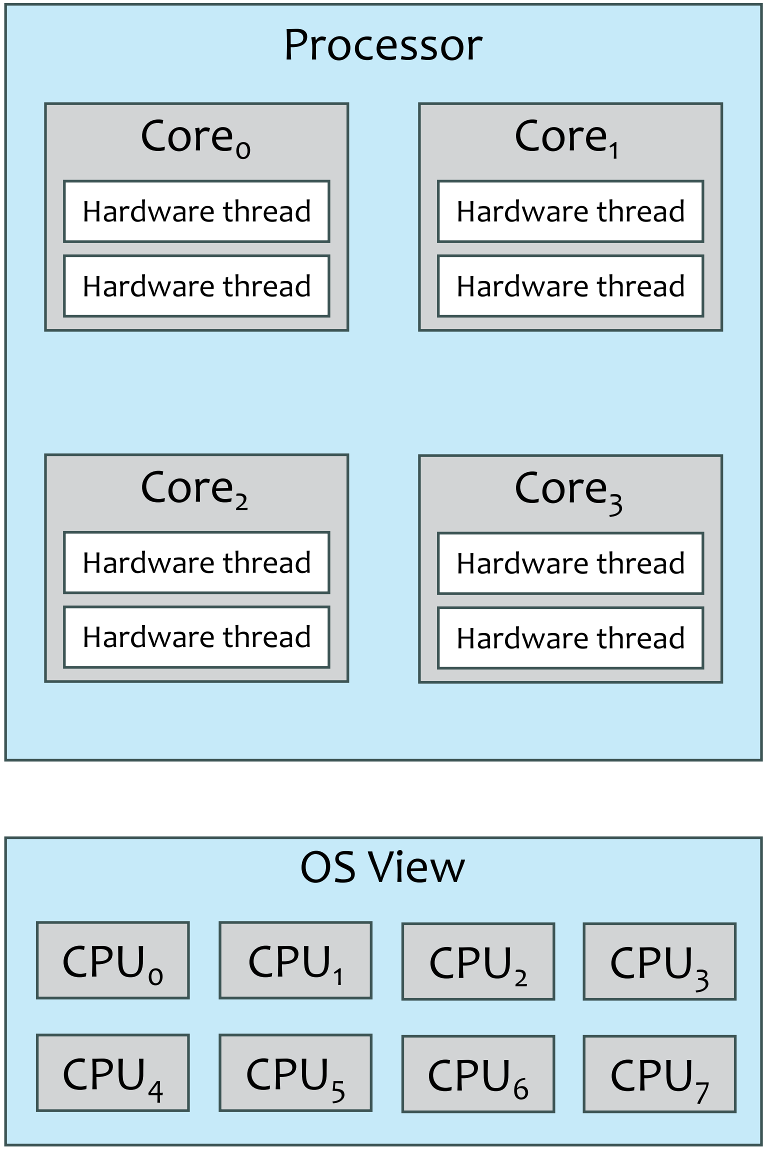
Multi-Processor Scheduling

- Multithreaded Cores
  - Chip multithreading (**CMT**)
    - (芯片多线程)
  - Intel calls it **Hyper-Threading**
    - (超线程)
  - also known as **S**imultaneous **M**ulti**t**hreading (**SMT**)
    - (同时多线程)



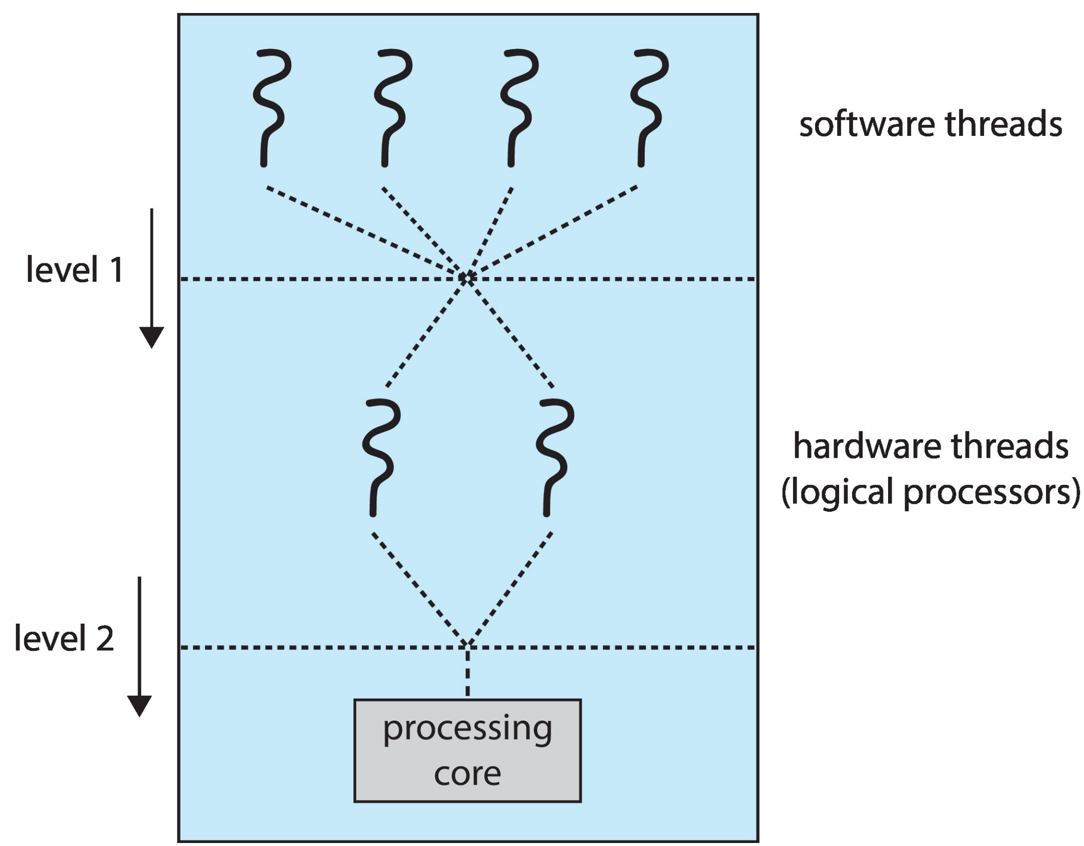
Multi-Processor Scheduling

- Example:
  - One Physical CPU Package
  - 4 Cores
  - 2 Hardware Threads per Core
  - OS View: 8 logical CPUs



Multi-Processor Scheduling

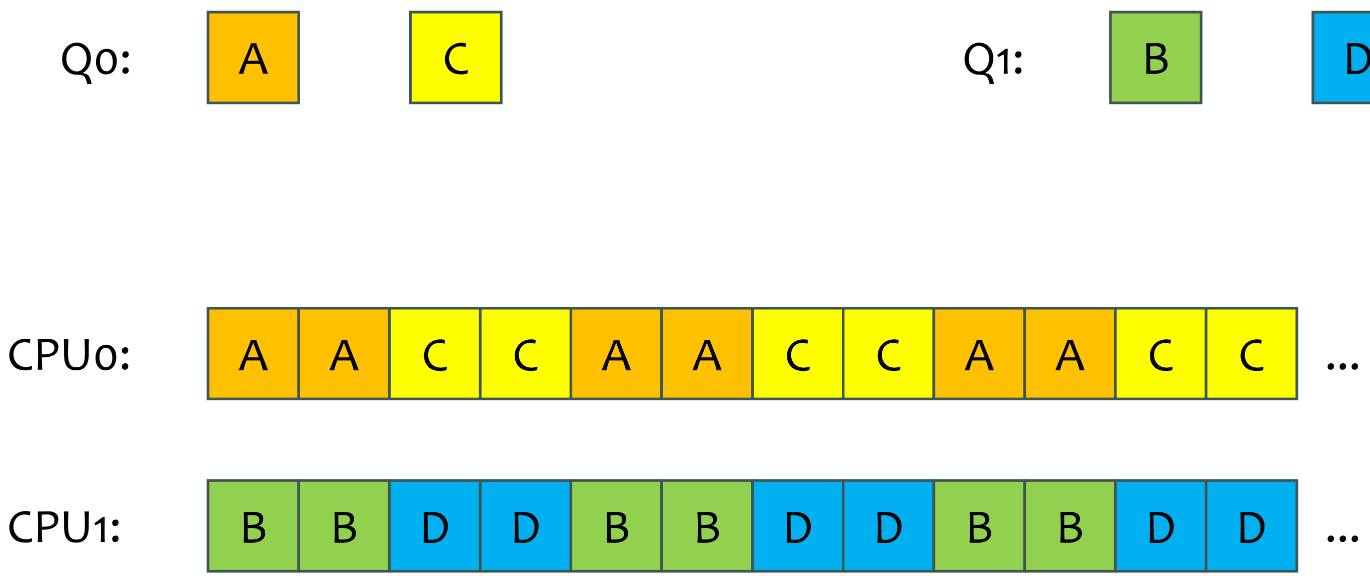
- Two levels of **scheduling:**
  - The OS deciding which software thread to run on a **logical** CPU
  - How each core decides which hardware thread to run on the **physical** core.



Multi-Processor Scheduling

Load Balancing

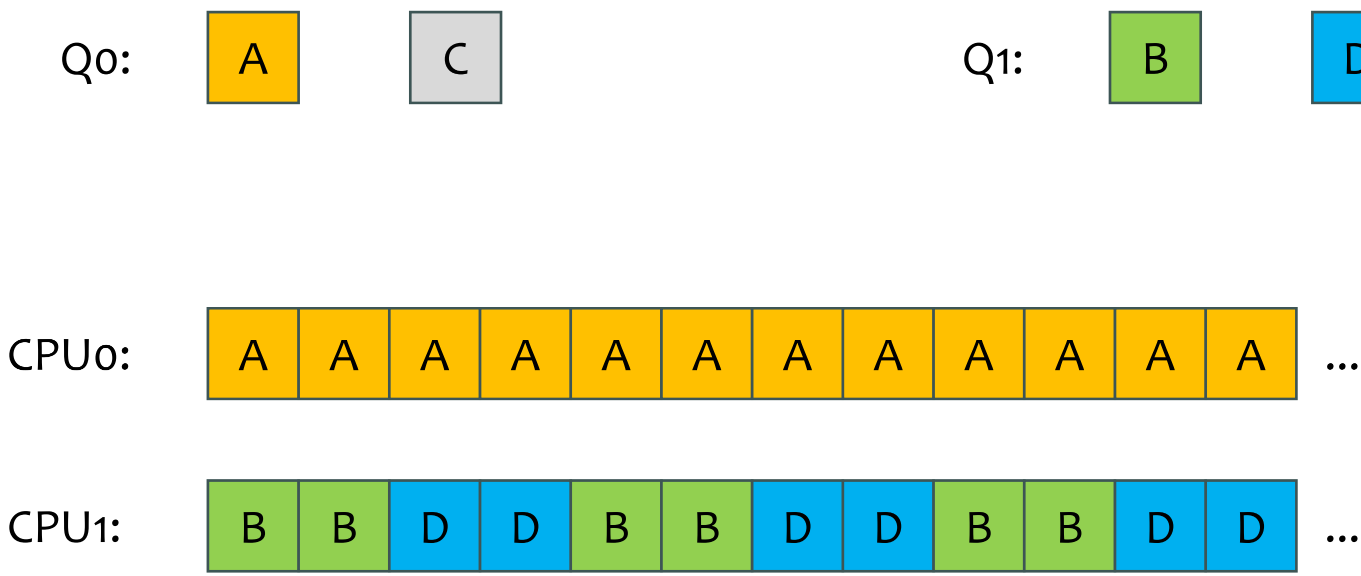
- Example:
  - Each CPU has its own queue of threads. (**A,C** on CPU0; **B,D** on CPU1)



Multi-Processor Scheduling

Load Balancing

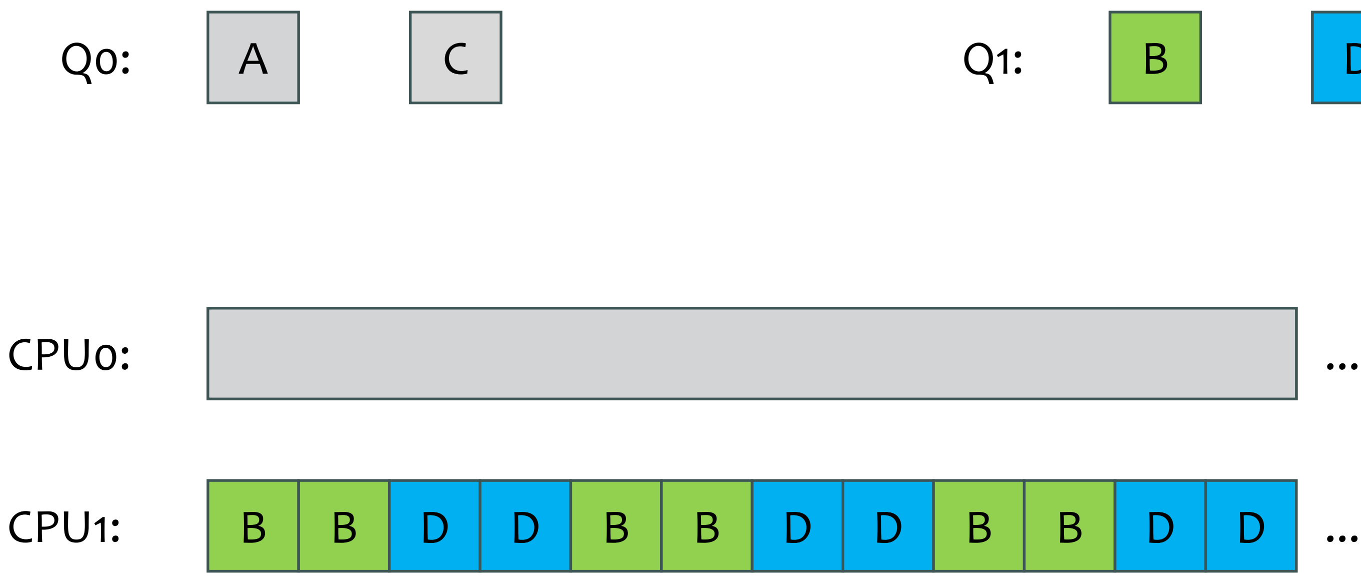
- Example:
  - Each CPU has its own queue of threads. (**A,C** on CPU0; **B,D** on CPU1)
  - **C** has completed ⇒ **A** owns 100% of CPU0; **B,D** owns 50% of CPU1.



Multi-Processor Scheduling

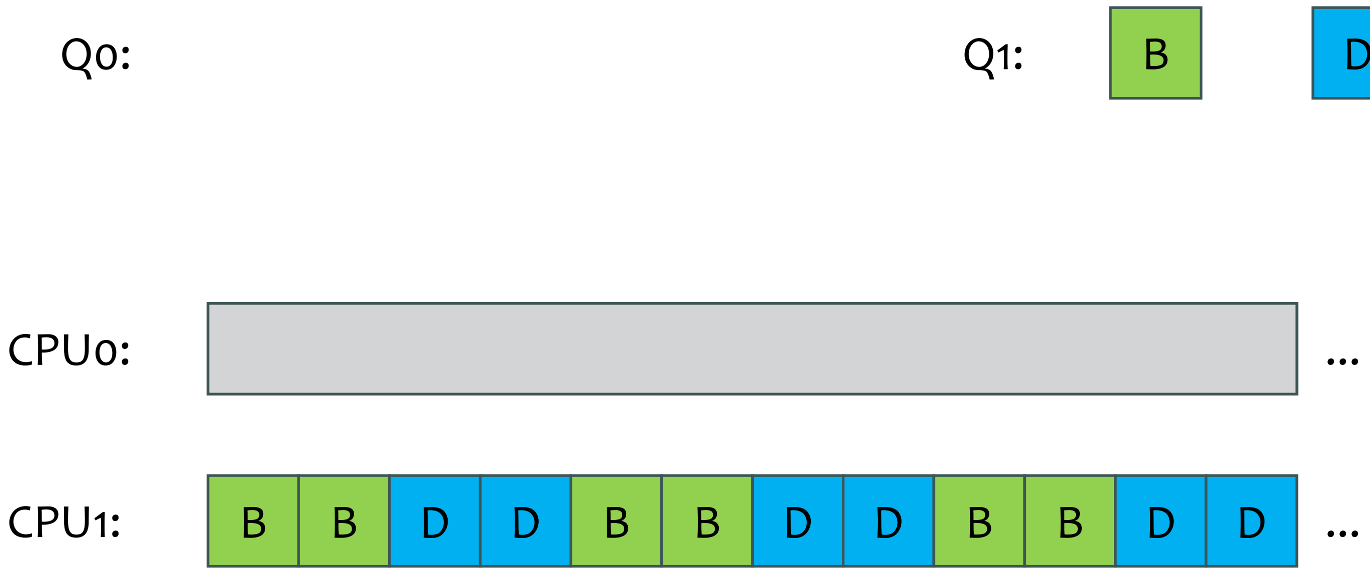
Load Balancing

- Example:
  - Each CPU has its own queue of threads. (**A,C** on CPU0; **B,D** on CPU1)
  - **C** has completed ⇒ **A** owns 100% of CPU0; **B,D** owns 50% of CPU1.
  - **A** has also completed ⇒ CPU0 idle!!; **B,D** owns 50% of CPU1.



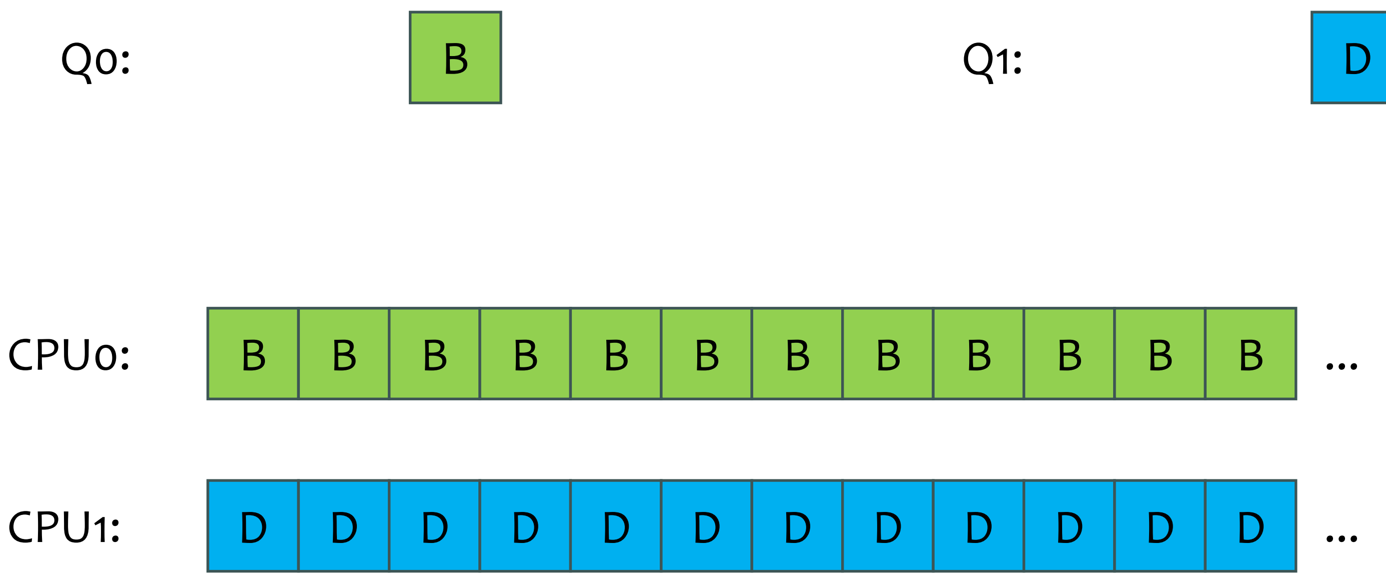
Load Balancing

- Example:
  - Each CPU has its own queue of threads. (**A,C** on CPU0; **B,D** on CPU1)
  - **C** has completed ⇒ **A** owns 100% of CPU0; **B,D** owns 50% of CPU1.
  - **A** has also completed ⇒ CPU0 idle!!; **B,D** owns 50% of CPU1.
  - The obvious solution is to migrate one thread from Q1 to Q0:
    - **Load Balancing**



Load Balancing

- Example:
  - Each CPU has its own queue of threads. (**A,C** on CPU0; **B,D** on CPU1)
  - **C** has completed ⇒ **A** owns 100% of CPU0; **B,D** owns 50% of CPU1.
  - **A** has also completed ⇒ CPU0 idle!!; **B,D** owns 50% of CPU1.
  - The obvious solution is to migrate one thread from Q1 to Q0:
    - **Load Balancing** ⇒ fair share of CPUs among threads

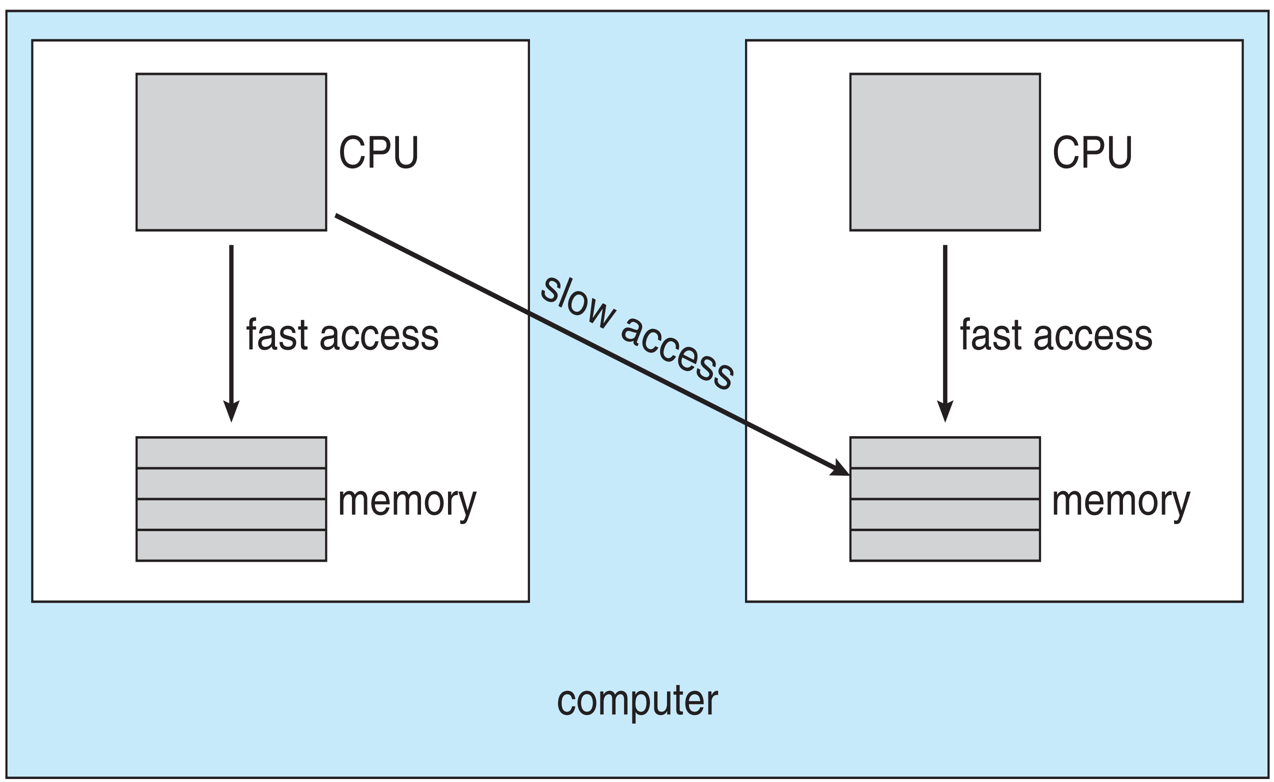


Multi-Processor Scheduling

- **Load Balancing**
  - With SMP, we need to keep all CPUs loaded for efficiency
  - Load balancing attempts to keep workload evenly distributed
  - Push migration
    - a specific task periodically checks the load on each processor
    - and if it finds an imbalance
    - evenly distributes the load by movign (pushing) tasks from overloaded to idle or less-busy processors.
  - Pull migration
    - an idle processor pull a waiting task from a busy processor

- **Processor Affinity**
  - When a thread has been running on one processor, the cache of that processor stores the memory accesses by that thread.
  - We refer to this as a thread having **affinity** (亲和性) for that processor (i.e., "**processor affinity**")
    - It's best to maintain such affinity for a specific thread
  - Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache
  - **Soft Affinity** – the OS attempts to keep a thread running on the same processor, but no guarantees
  - **Hard Affinity** – allows a process to specify a set of processors it may run on.

- **NUMA (Non-Uniform Memory Access)**
  - If the OS is NUMA-aware, then a thread that has been scheduled onto a particular CPU can be allocated memory closest to where the CPU resides, thus providing the thread the fastest possible memory access.



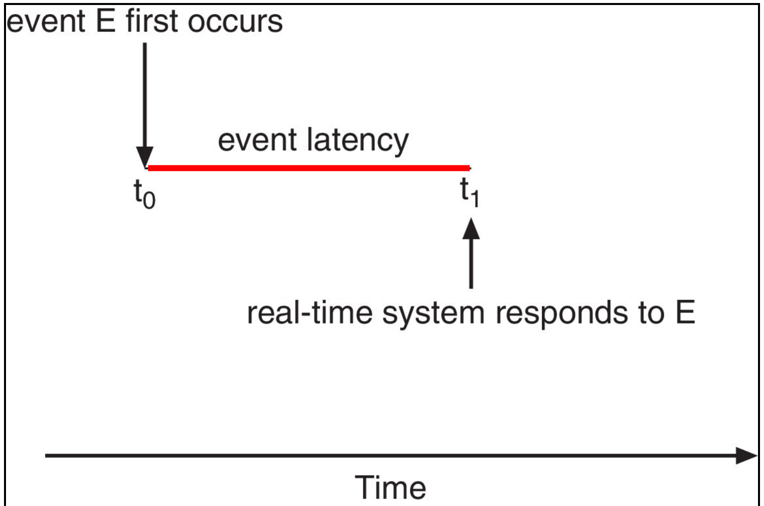
Real-Time CPU Scheduling

- **Soft Real-Time Systems**
  - Guarantee that a critical real-time process will be given preference over noncritical processes.
  - **No guarantee** as to when a critical real-time process will be scheduled.
- **Hard Real-Time Systems**
  - Stricter requirements.
  - A task must be serviced by its deadline.
  - Service after the deadline has expired $\Rightarrow$ no service at all.

Real-Time CPU Scheduling

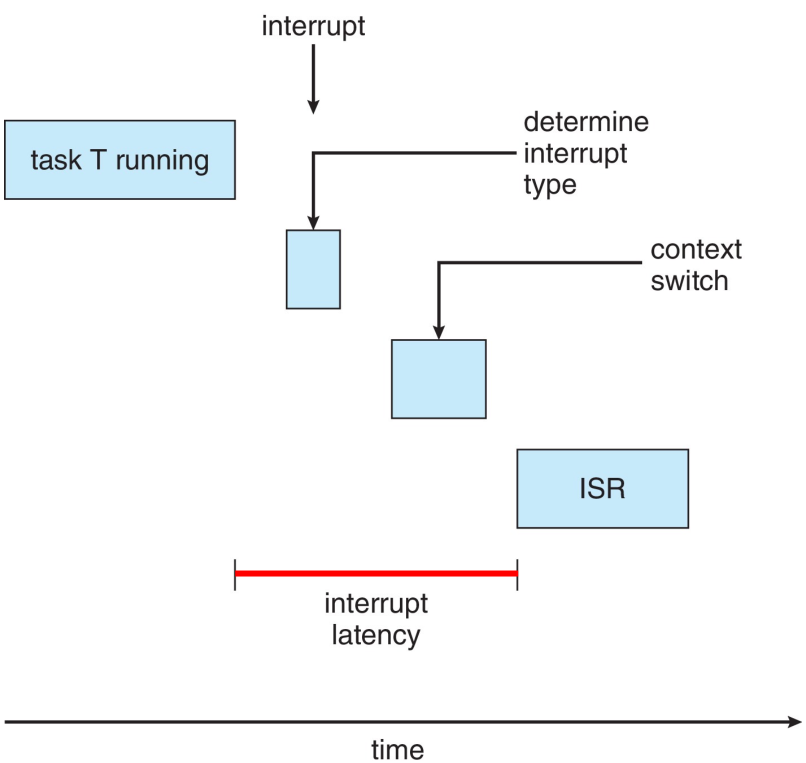
Minimizing Latency

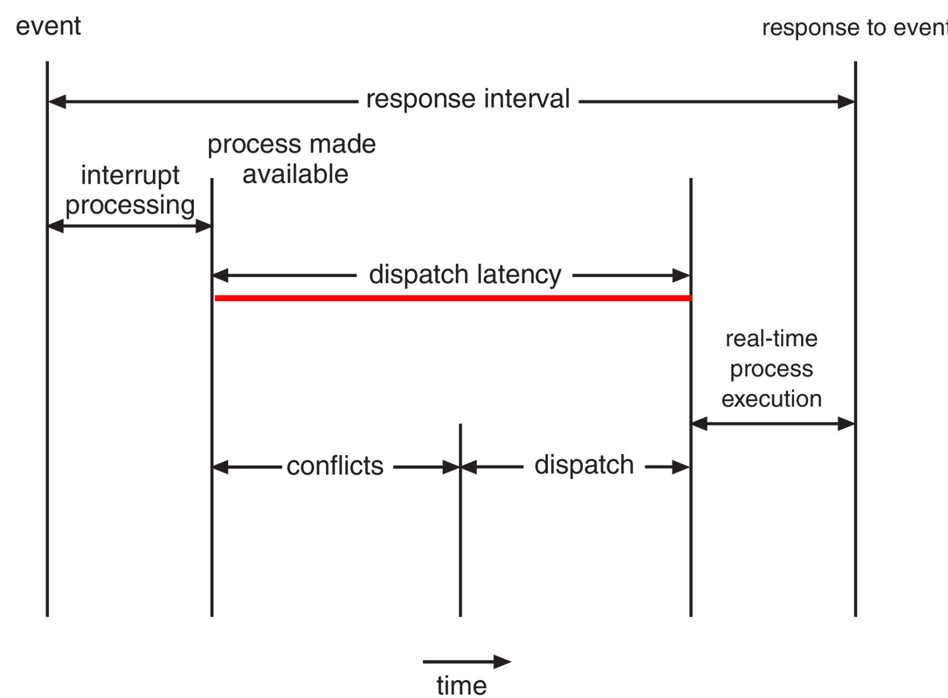
- Real-Time systems are event-driven.
  - When an event occurs, the system must respond to and service it ASAP.
  - Software events
    - E.g., a timer expires
  - Hardware events
    - E.g., a remote-controlled vehicle detects it is approaching an obstruction.
- Event Latency (事件延迟)
  - **Event latency** is the amount of time that elapses from when an event occurs to when it is serviced.



Real-Time CPU Scheduling

- **Minimizing Latency**
  - Two types of latencies affect the performance of RT systems:
    - **Interrupt** Latency
    - **Dispatch** Latency



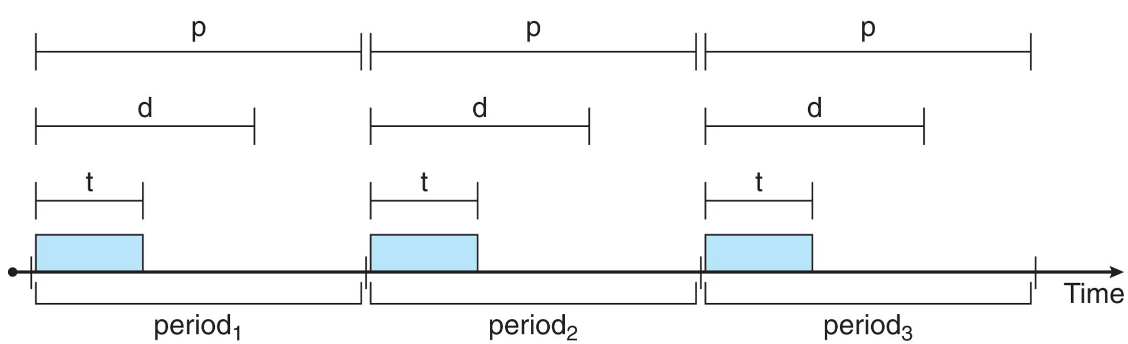


Real-Time CPU Scheduling

- **Priority-Based Scheduling**
  - The scheduler for a RT system MUST support a **priority**-based algorithm with **preemption**.
    - The most important feature of a Real-Time OS is to respond immediately to a Real-Time process as soon as that process requires CPU.
    - But only guarantees **soft** Real-Time.
  - For **hard** Real-Time, it must also provide ability to meet deadlines.

Priority-Based Scheduling

- The scheduler for a RT system MUST support a **priority**-based algorithm with **preemption**.
  - The most important feature of a Real-Time OS is to respond immediately to a Real-Time process as soon as that process requires CPU.
  - But only guarantees **soft** Real-Time.
- For **hard** Real-Time, it must also provide ability to meet deadlines.
- **Periodic Process**: a process is periodic (周期性的) if it requires CPU at constant intervals (periods).
  - Has processing time t, deadline d, period p.
  - $0 \le t \le d \le p$
  - Rate of periodic task is $^1/_p$



Real-Time CPU Scheduling

- **Priority-Based Scheduling**
  - **Admission Control**
    - A process may have to announce its deadline requirements to the scheduler. Using an admission-control (准入控制) algorithm, the scheduler does one of two things:
      - admits the process, guaranteeing that the process will complete on time
      - rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

Rate Monotonic Scheduling (单调速率调度)

- The Rate Monotonic Scheduling (RMS) algorithm schedules periodic tasks using a static priority policy with preemption.
- A priority is assigned based on the inverse of its period
  - Shorter periods $p \Rightarrow$ higher priority
  - Longer periods $p \Rightarrow$ lower priority
- Furthermore, RMS assumes that processing time of a periodic process is the same for each CPU burst.
  - Assume that every time a process acquires the CPU, the duration of its CPU burst is the same.

Rate Monotonic Scheduling (单调速率调度)

- Example 1:
  - Let P_1 and P_2 be periodic processes.
    - $P_1 = \{p_1 = 50,\ t_1 = 20,\ d_1 = p_1\}$
    - $P_2 = \{p_2 = 100,\ t_2 = 35,\ d_2 = p_2\}$
  - The combined CPU utilization of the two processes is:
    - $\left(\frac{20}{50}\right) + \left(\frac{35}{100}\right) = 0.75$
    - therefore, it should seem logical that the two processes can be scheduled, and still leave the CPU with 25% idle time.
  - Suppose we assign P_2 a higher priority than P_1:

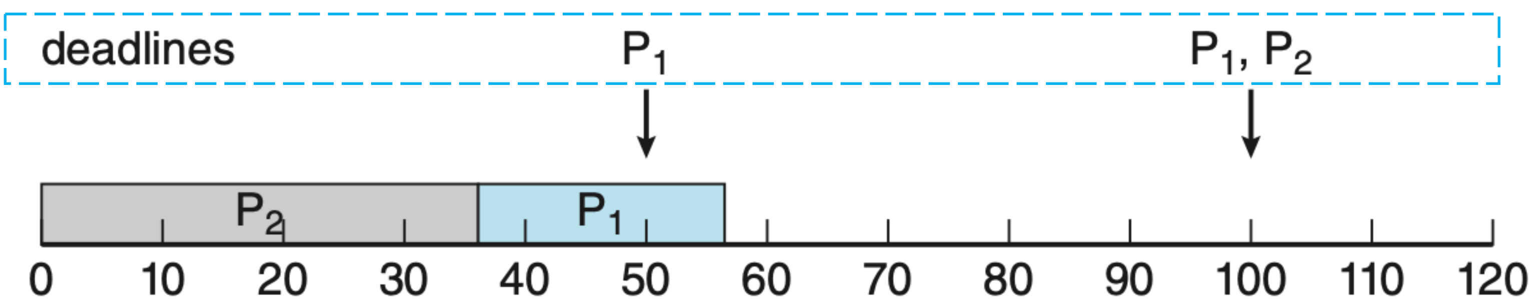


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

Rate Monotonic Scheduling (单调速率调度)

- Example 1:
  - Suppose we assign P_2 a higher priority than P_1:
  - P_2 starts execution first and completes at time 35.
  - At this point, P_1 starts
    - it completes its CPU burst at time 55
  - However, the first deadline for P_1 was time 50
    - so the scheduler has caused P_1 to miss its deadline.

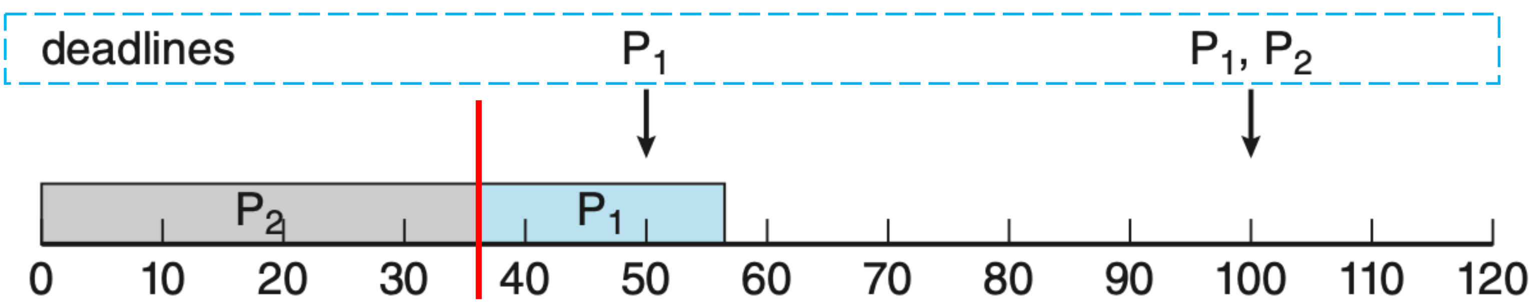


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

Rate Monotonic Scheduling (单调速率调度)

- Example 1 (**RMS**):
  - Suppose we assign P_1 a higher priority than P_1:
    - $p_1 < p_2 \quad \Longrightarrow \quad Priority(P_1) > Priority(P_2)$
  - P_1 starts first and completes its CPU burst at time 20
    - thereby meeting its first deadline

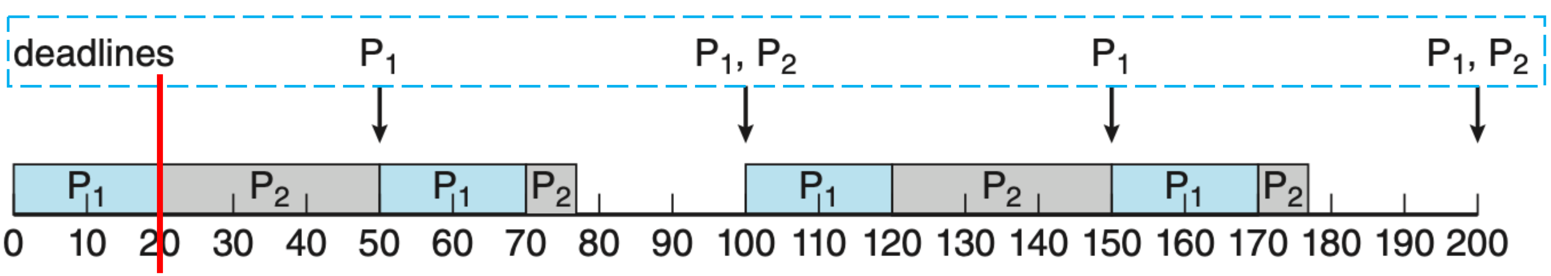


Figure 5.22 Rate-monotonic scheduling.

Rate Monotonic Scheduling (单调速率调度)

- Example 1 (**RMS**):
  - Suppose we assign P_1 a higher priority than P_1:
    - $p_1 < p_2 \quad \Longrightarrow \quad Priority(P_1) > Priority(P_2)$
  - P_1 starts first and completes its CPU burst at time 20
    - thereby meeting its first deadline
  - P_2 starts running at this point and runs until time 50
    - at this time, it is preempted by P_1, although 5ms remaining for P_2

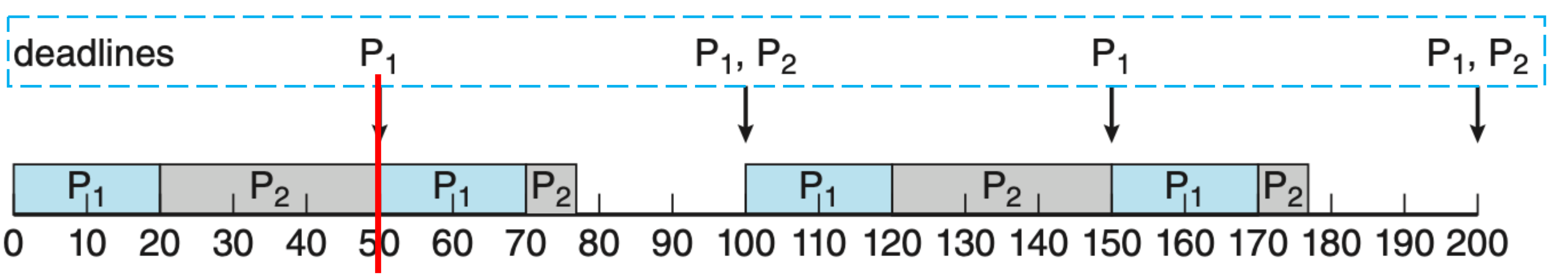


Figure 5.22 Rate-monotonic scheduling.

Rate Monotonic Scheduling (单调速率调度)

- Example 1 (**RMS**):
  - Suppose we assign P_1 a higher priority than P_1:
    - $p_1 < p_2 \quad \Longrightarrow \quad Priority(P_1) > Priority(P_2)$
  - P_1 starts first and completes its CPU burst at time 20
    - thereby meeting its first deadline
  - P_2 starts running at this point and runs until time 50
    - at this time, it is preempted by P_1, although 5ms remaining for P_2
  - P_1 completes its CPU burst at time 70

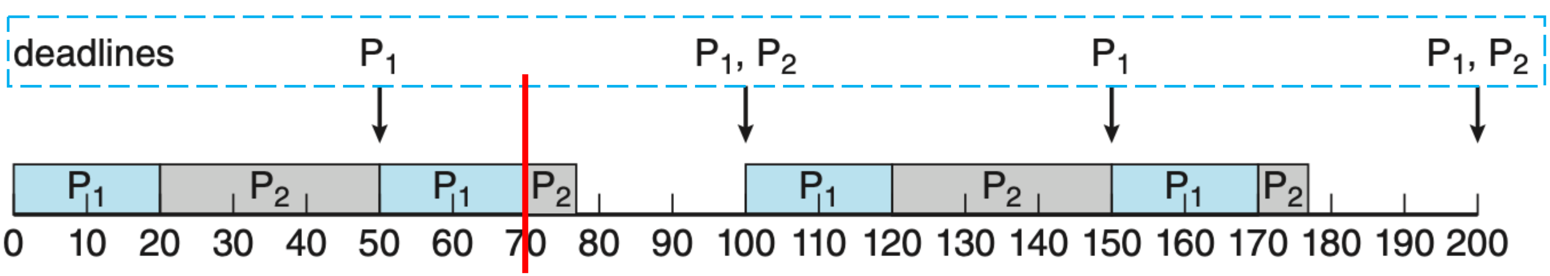


Figure 5.22 Rate-monotonic scheduling.

Rate Monotonic Scheduling (单调速率调度)

- Example 1 (**RMS**):
  - Suppose we assign P_1 a higher priority than P_1:
    - $p_1 < p_2 \quad \Longrightarrow \quad Priority(P_1) > Priority(P_2)$
  - P_1 starts first and completes its CPU burst at time 20
    - thereby meeting its first deadline
  - P_2 starts running at this point and runs until time 50
    - at this time, it is preempted by P_1, although 5ms remaining for P_2
  - P_1 completes its CPU burst at time 70
  - P_2 resumes and completes at time 75, meeting its first deadline.

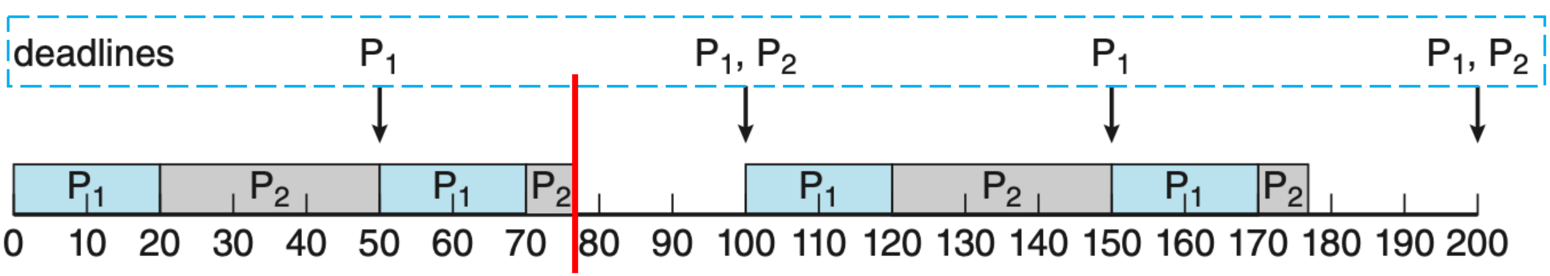


Figure 5.22 Rate-monotonic scheduling.

Rate Monotonic Scheduling (单调速率调度)

- Example 1 (**RMS**):
  - Suppose we assign P_1 a higher priority than P_1:
    - $p_1 < p_2 \quad \Longrightarrow \quad Priority(P_1) > Priority(P_2)$
  - P_1 starts first and completes its CPU burst at time 20
    - thereby meeting its first deadline
  - P_2 starts running at this point and runs until time 50
    - at this time, it is preempted by P_1, although 5ms remaining for P_2
  - P_1 completes its CPU burst at time 70
  - P_2 resumes and completes at time 75, meeting its first deadline.
  - The system is idle until time 100, when P_1 is scheduled again.

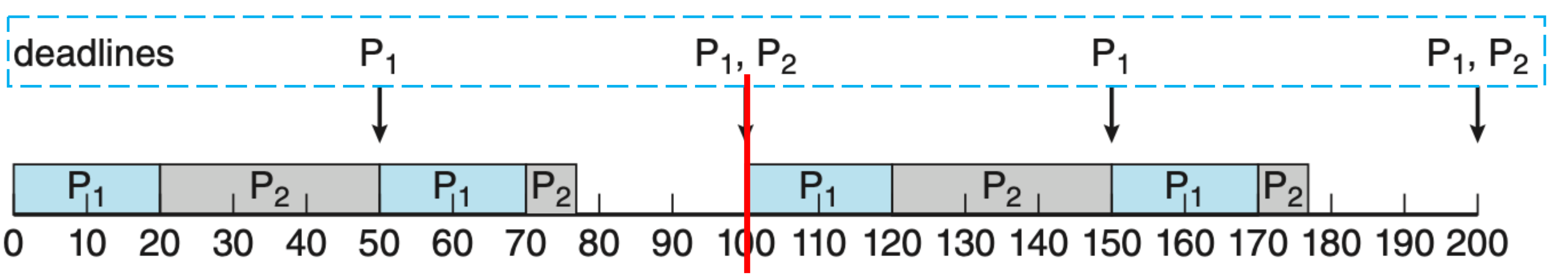


Figure 5.22 Rate-monotonic scheduling.

Rate Monotonic Scheduling (单调速率调度)

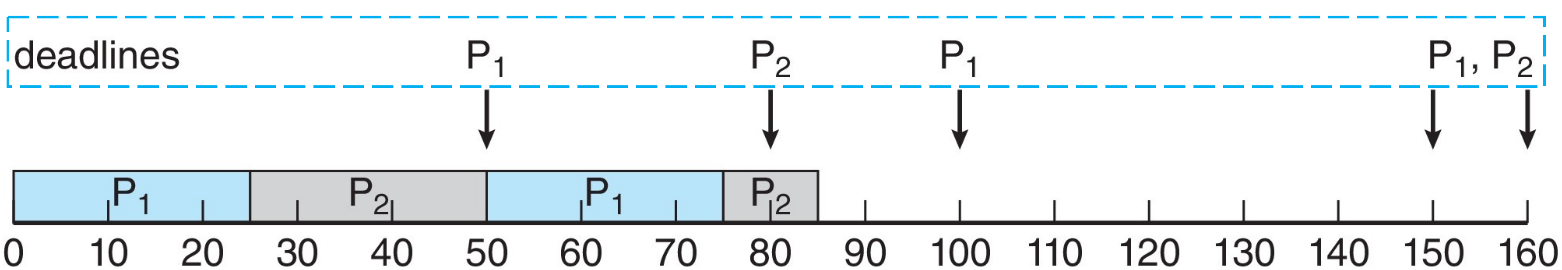
- Example 2 (missing deadline with **RMS**):

$$Priority(P_1) > Priority(P_2)$$

  - Let P_1 and P_2 be periodic processes.
    - $P_1 = \{p_1 = 50,\ t_1 = 25,\ d_1 = p_1\}$
    - $P_2 = \{p_2 = 80,\ t_2 = 35,\ d_2 = p_2\}$
  - The combined CPU utilization of the two processes is:
    - $\left(\frac{25}{50}\right) + \left(\frac{35}{80}\right) = \mathbf{0.94}$
    - therefore, it should seem logical that the two processes can be scheduled, and still leave the CPU with 6% idle time.

Rate Monotonic Scheduling (单调速率调度)

- Example 2 (missing deadline with **RMS**): $Priority(P_1) > Priority(P_2)$
  - Initially, P_1 runs until it completes its CPU burst at time 25
  - P_2 then runs until time 50
    - then P_2 is preempted by P_1
  - At this point, P_2 still has 10 left in its CPU burst. P_1 runs until time 75
  - Consequently, P_2 finishes its burst at time 85
    - missing the deadline of time 80.



Rate Monotonic Scheduling (单调速率调度)

- Rate-Monotonic Scheduling is considered **optimal**.
  - If a set of processes cannot be scheduled by RMS, it cannot be scheduled by any other algorithm that assigns **static** priorities.
- **RMS** has a limitation: CPU utilization is bounded, and it is not always possible to fully maximize CPU resources. The **worst-case CPU utilization** for scheduling N processes is $N(2^{1/N}-1)$
  - When $N=1,\ N(2^{1/N}-1)=1$
  - When $N=2,\ N(2^{1/N}-1)\approx 0.83$
  - When $N\to\infty,\ \lim_{n\to\infty} N(2^{1/N}-1)=\ln 2\approx 0.69$
- In Example 1, combined CPU utilization for the two processes scheduled is **0.75** (< 0.83); therefore, **RMS** is guaranteed to schedule them so that they can meet their deadlines.
- In Example 2, combined CPU utilization is **0.94** (> 0.83); so **RMS** cannot guarantee they can be scheduled to meet their deadlines.

Rate Monotonic Scheduling (单调速率调度)

- **Theorem.** (The schedulability test for RMS, *Liu, C. L.; Layland, J., 1973*)
  A set of n periodic hard real-time tasks $T = \{\tau_1, \tau_2, \dots, \tau_n\}$, with C_i as the Worst Case Time and T_i as the period of task τ_i, will meet their deadlines independent of their start times if,

$$\boldsymbol{U} = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

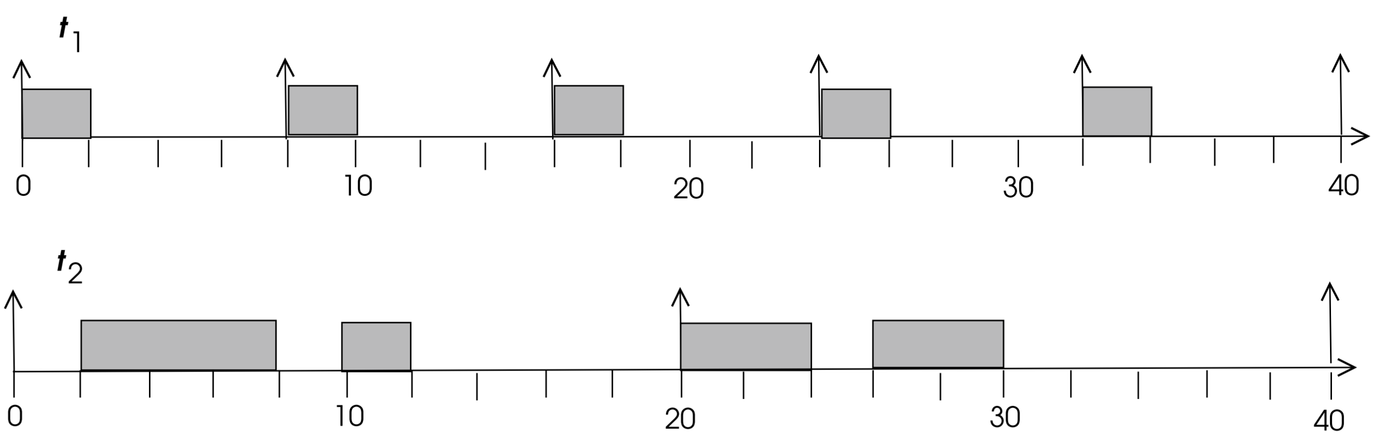
- Utilization factor $\boldsymbol{U}$ converges to $\ln 2 \approx 0.69$ as $n > 10$. We can therefore test whether a task set can be scheduled using this inequality. Note that this test is sufficient but not a necessary condition. In other words, there may be a task set that has a total utilization larger than 0.69 but tasks in this set may still meet their deadlines under RMS.

Rate Monotonic Scheduling (单调速率调度)

- Example 3
  - Consider the task set $\tau_1(2,8)$ and $\tau_2(8,20)$ in $\tau_i(C_i, T_i)$ form for RMS. We can see that these two tasks can be scheduled since

$$\left(\frac{2}{8}\right) + \left(\frac{8}{20}\right) = 0.65 < 2(2^{\frac{1}{2}} - 1) \approx 0.83$$

  - Hence, this task set can be admitted to the system (with RMS). The scheduling of these tasks using RMS is depicted below.
  - Since LCM of their periods is 40, the scheduler will repeat itself every 40 time units.



Rate Monotonic Scheduling (单调速率调度)

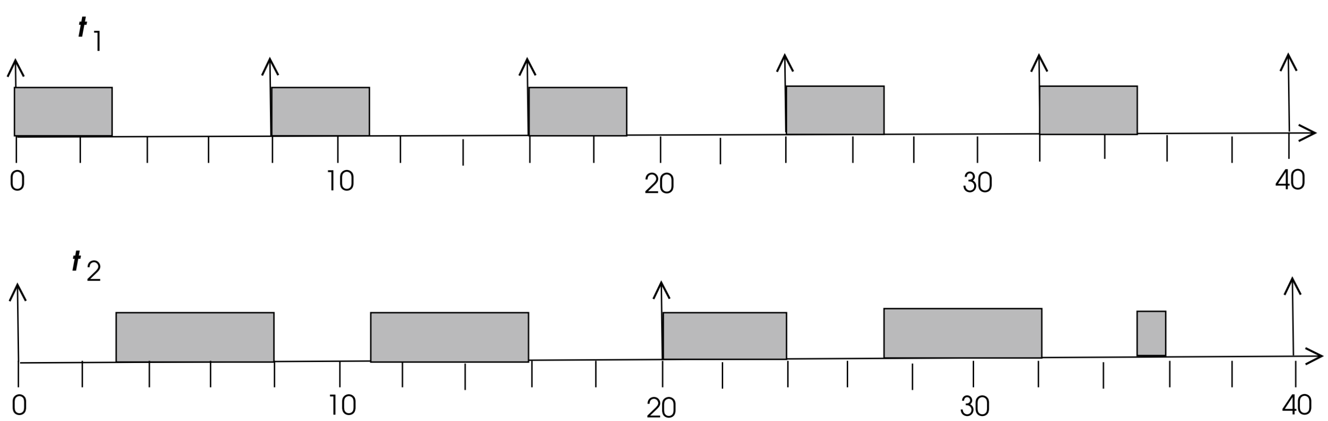
- Example 4
  - Let us now increase the computation time of tasks:

$$\tau_1(2,8) \Rightarrow \tau_1(3,8)$$
$$\tau_2(8,20) \Rightarrow \tau_2(10,20)$$

  - Apply the schedulability test:

$$\left(\frac{3}{8}\right) + \left(\frac{10}{20}\right) = 0.875 > 2(2^{\frac{1}{2}} - 1) \approx 0.83$$

  - It means that RMS does not guarantee to have a feasible schedule for this task set. However, we can still schedule these tasks to meet their deadlines: (no longer **static priority**)

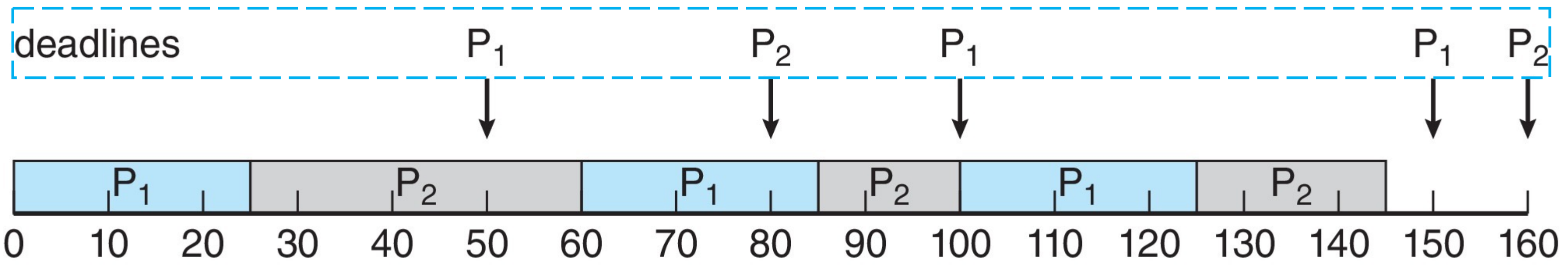


Earliest-Deadline-First Scheduling (EDF)

- Earliest-Deadline-First (最早截止期限优先) is a priority-based scheduling.
  - It dynamically assigns priorities according to next deadlines.
  - (The earlier the next deadline, the higher the priority)
  - with **EDF**, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process.
- Difference between **EDF** and **RMS**:
  - RMS $\Rightarrow$ Static (Fixed) Priority
  - EDF $\Rightarrow$ Dynamic Priority

Earliest-Deadline-First Scheduling (EDF)

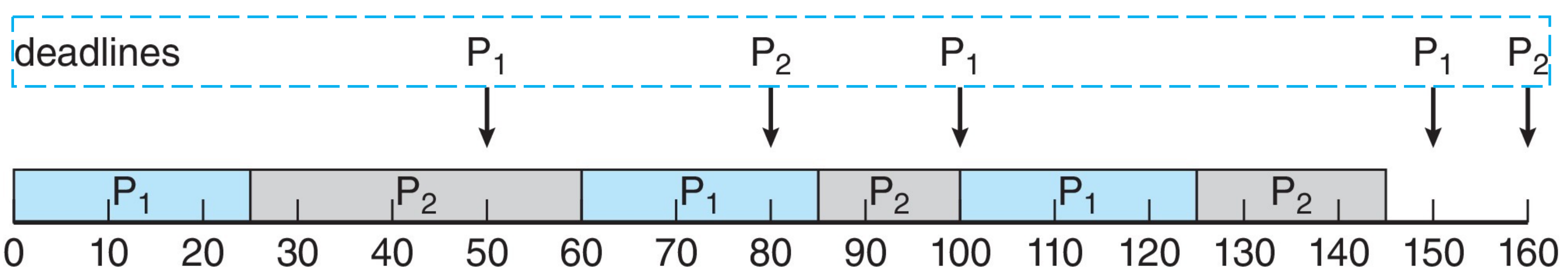
- Example 5 (same as Example 3, missing deadlines with **RMS**)
  - Let P_1 and P_2 be periodic processes.
    - $P_1 = \{p_1 = 50,\ t_1 = 25,\ d_1 = p_1\}$
    - $P_2 = \{p_2 = 80,\ t_2 = 35,\ d_2 = p_2\}$
  - With EDF, P_1 has a higher initial priority than P_2 $(d_1 = 50 < d_2 = 80)$.
  - P_2 begins running at the end of the CPU burst of P_1 (time 25)
  - P_2 now has a higher priority than (*the next*) P_1 $(d_2 = 80 < d_1 = 100)$.
    - EDF allows P_2 to continue running
    - Both P_1 and P_2 meet their deadlines.
    - *(Recall that **RMS** allows P_1 to preempt P_2 at time 50).*



Real-Time CPU Scheduling

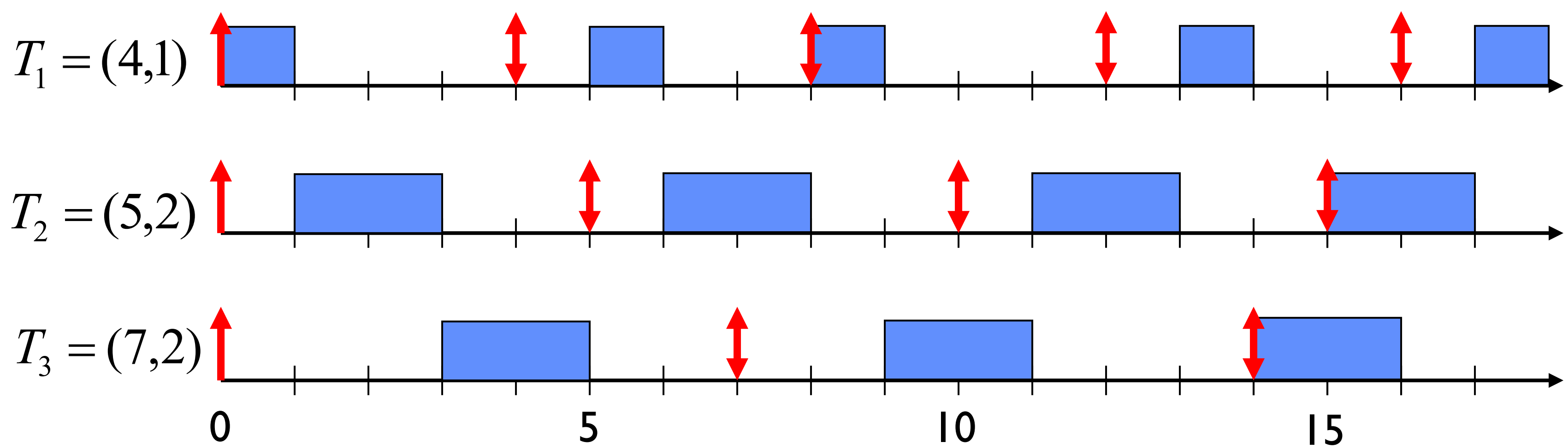
Earliest-Deadline-First Scheduling (EDF)

- Example 5 (same as Example 3, missing deadlines with **RMS**)
  - P_1 runs again at time 60 and complets its 2nd CPU burst at time 85
    - also meeting its deadline at time 100
  - At this point (time 85), P_2 begins running again.
  - At time 100, P_2 is preempted by P_1 because $(d_1 = 150 < d_2 = 160)$.
  - At time 125, P_1 completes its CPU burst and P_2 resumes execution, finishing at time 145 and meeting its deadline as well.
  - The system is idle until time 150, when P_1 is scheduled to run its next period.



Earliest-Deadline-First Scheduling (EDF)

- **Preemptive** priority-based dynamic scheduling
- The scheduler always schedules the active task with the closest absolute deadline.



Earliest-Deadline-First Scheduling (EDF)

- Unlike **RMS**, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per CPU burst.
  - The only requirement is that a process announces its deadline to the scheduler when it becomes runnable.
  - It is useful for aperiodic tasks (非周期任务) and sporadic tasks (偶发任务) scheduling
- **EDF** is theoretically optimal on **preemptive uniprocessors.**
  - If a collection of independent processes can be scheduled (by any algorithm) in a way that ensures all the processes complete by their deadline, then **EDF** will schedule them so that each process can meet its deadline requirements.

Earliest-Deadline-First Scheduling (EDF)

- The Schedulability Test for **EDF**
  - with scheduling periodic processes that have deadlines equal to their periods ($d_i = p_i$), EDF has a utilization bound of 100%. Thus, the schedulability test for EDF is

$$\boldsymbol{U} = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

  - EDF can guarantee that all deadlines are met provided that the total CPU utilization is not more than 100%.
    - In contrast to fixed priority scheduling (e.g., **RMS**), EDF can guarantee all the deadlines be met in heavy-loaded systems.
  - In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

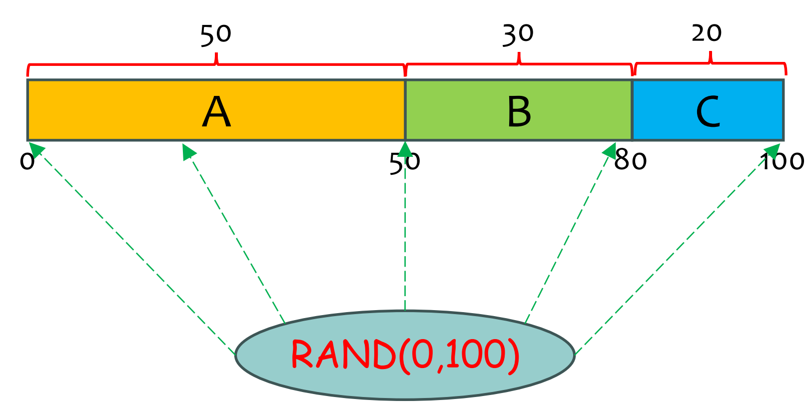
Proportional Share Scheduling (比例分享调度)

- **Proportional Share** Schedulers (a.k.a., **Fair-Share** Scheduler, **Tickets** Scheduler) operate by allocating T shares among all n processes.
  - Let the i^{th} process receive N_i shares of time ($T > \sum_{i=1}^{n} N_i$)
  - The i^{th} process will have N_i/T of the total CPU time.
- **Example 6**
  - Assume that a total of $T = 100$ shares is to be divided among three processes, A, B, and C. $N_A = 50$, $N_B = 30$, $N_C = 20$.
  - A proportional share scheduler ensures that
    - A will have 50% of total processor time.
    - B will have 30% of total processor time.
    - C will have 20% of total processor time.
- **PSS** must work in conjunction with an admission-control policy to guarantee that a process receives its allocated share of time. An admission-control policy will admit a client requesting a particular number of shares **only if sufficient shares are available.**

Proportional Share Scheduling (比例分享调度)

```c
/* tickets.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct {
    int pid;
    int tickets;
} Process;

void execute_process(int pid) {
    printf("Process %c is running.\n", 'A'+pid-1);
}

int pick_next(Process processes[], int n) {
    int total_tickets = 0;
    // Calculate the total number of tickets
    for (int i = 0; i < n; i++) {
        total_tickets += processes[i].tickets;
    }

    int ticket_draw = rand() % total_tickets;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += processes[i].tickets;
        if (ticket_draw < sum) {
            return processes[i].pid;
        }
    }
    return -1;
}
```

```c
int main() {
    srand(time(NULL));

    // Create an array of processes
    Process processes[3] = {
        {1, 50}, // Process A with 50 tickets
        {2, 30}, // Process B with 30 tickets
        {3, 20}  // Process C with 20 tickets
    };

    // Simulate scheduling 20 times
    for (int i = 0; i < 20; i++) {
        int pid = pick_next(processes, 3);
        execute_process(pid);
    }

    return 0;
}
```



Proportional Share Scheduling (比例分享调度)

```c
/* tickets.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct {
    int pid;
    int tickets;
} Process;

void execute_process(int pid) {
    printf("Process %c is running.\n", 'A'+pid-1);
}

int pick_next(Process processes[], int n) {
    int total_tickets = 0;
    // Calculate the total number of tickets
    for (int i = 0; i < n; i++) {
        total_tickets += processes[i].tickets;
    }

    int ticket_draw = rand() % total_tickets;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += processes[i].tickets;
        if (ticket_draw < sum) {
            return processes[i].pid;
        }
    }
    return -1;
}
```

```c
int main() {
    srand(time(NULL));

    // Create an array of processes
```

```
$ ./tickets
Process A is running.
Process B is running.
Process C is running.
Process B is running.
Process B is running.
Process A is running.
Process A is running.
Process C is running.
Process C is running.
Process B is running.
Process A is running.
Process B is running.
Process C is running.
Process A is running.
Process B is running.
Process B is running.
Process B is running.
Process C is running.
Process C is running.
Process B is running.
...
```

$Share(A) \approx 50\%$

$Share(B) \approx 30\%$

$Share(C) \approx 20\%$

POSIX Real-Time Scheduling

- POSIX.1b is the POSIX standard extensions for Real-Time computing. It defines two scheduling classes for Real-Time threads:
  - `SCHED_FIFO`
  - `SCHED_RR`
- `SCHED_FIFO` schedules threads according to a FCFS policy using a FIFO queue. There is no time slicing among threads of equal priority. The highest-priority Real-Time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks.
- `SCHED_RR` uses an RR policy. It is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority.
- `SCHED_OTHER` is an additional scheduling class provided by POSIX, but its implementation is undefined and system specific; it may behave differently on different systems.

POSIX Real-Time Scheduling

- The POSIX API specifies the following two functions for getting and setting the scheduling policy:

```
pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

- The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either
  - a pointer to an integer that is set to the current scheduling policy (for the `pthread_attr_getsched_policy` function)
  - an integer value {`SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`} (for the `pthread_attr_setsched_policy` function)
- Both functions return nonzero values if an error occurs.

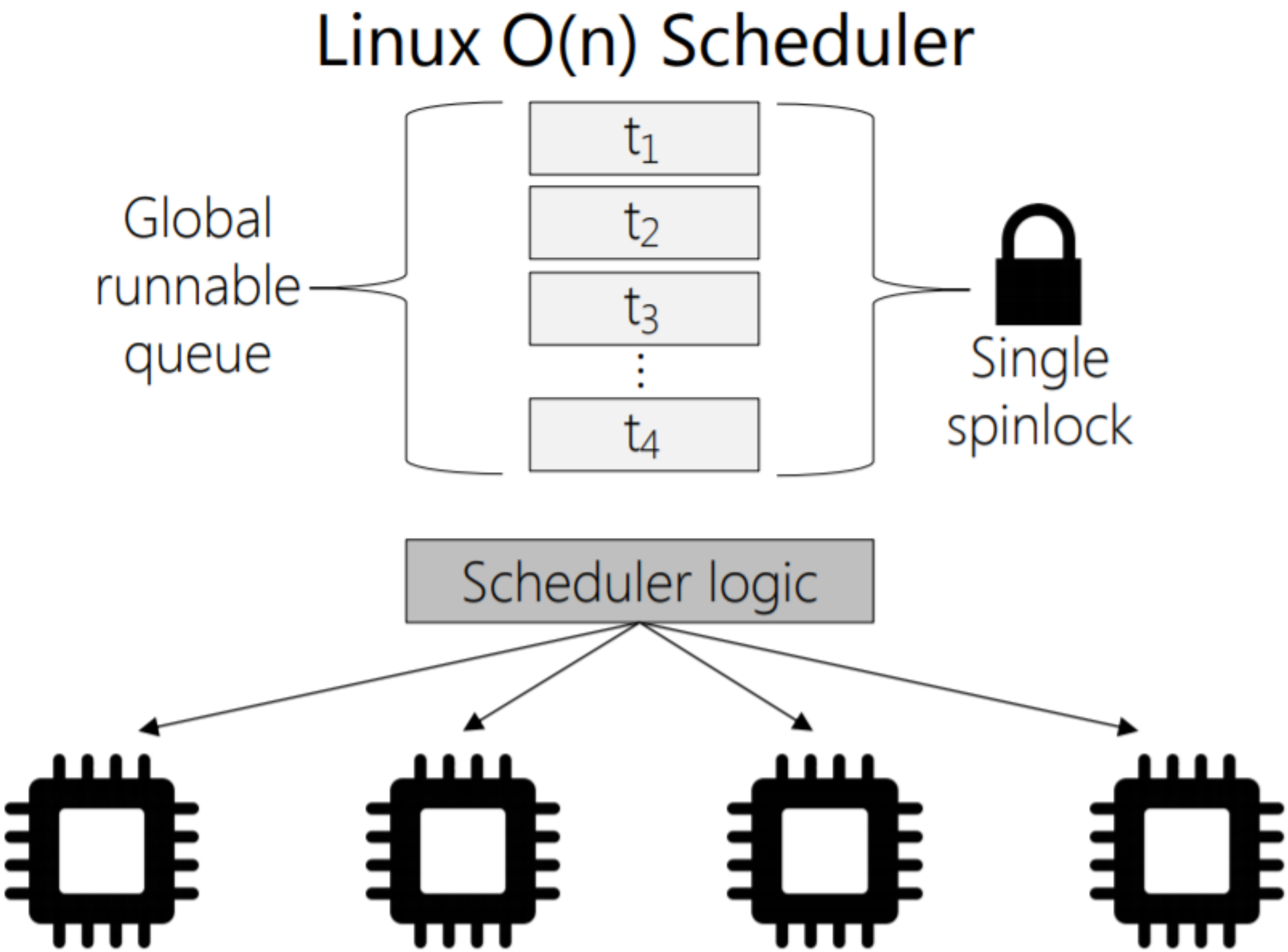
Linux Scheduling

- Prior to kernel version 2.6
  - ran variation of standard UNIX scheduling algorithms
  - not designed with SMP or multi-processor systems in mind
  - poor performance with a large number of runnable processes.
- Version 2.6.0 ~ 2.6.22
  - overhauled to include O(1) scheduler
    - preemptive, priority-based
    - runs in constant time regardless of # of tasks in the system.
    - support for SMP and multi-processors (load balancing, CPU affinity)
    - Poor response time for interactive processes.
- Version 2.6.23+
  - **Completely Fair Scheduler** (**CFS**) became the default Linux scheduling algorithm.
- Version 6.6+
  - **EEVDF** (Earliest Eligible Virtual Deadline First) Scheduler replaced **CFS**.

Linux Scheduler Evolution

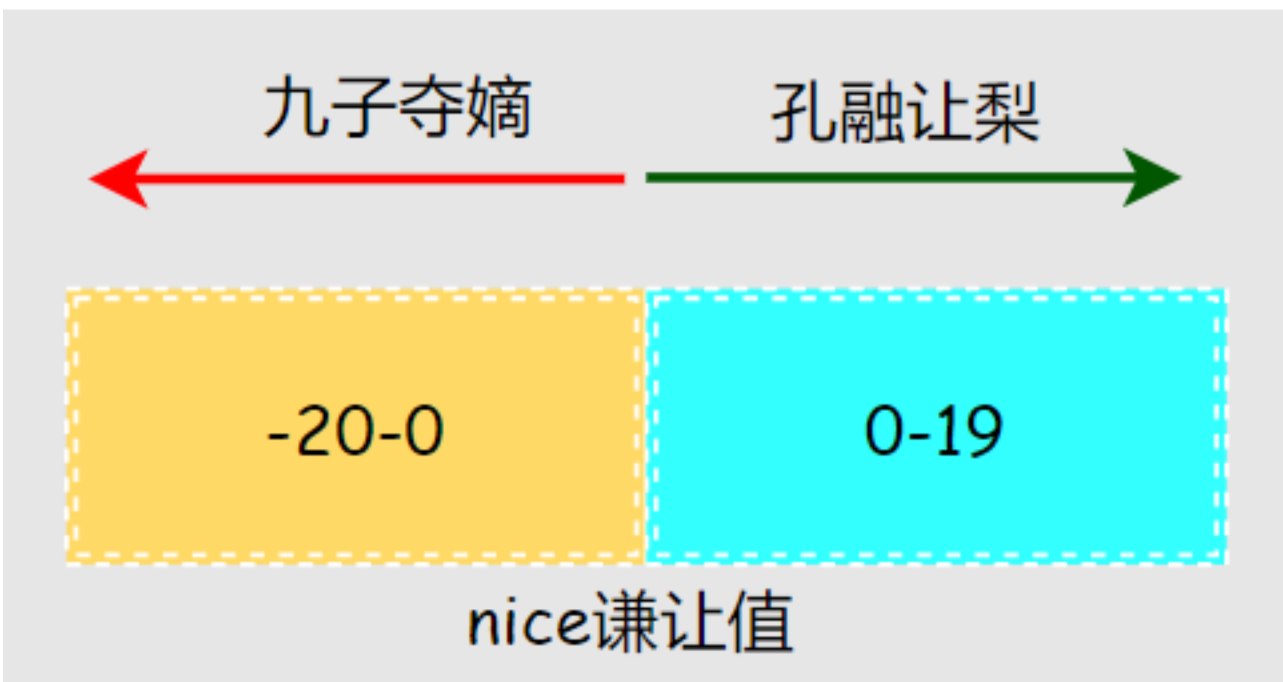
O(N) Scheduler (< Linux 2.6) → O(1) Scheduler (Linux 2.6) → CFS Scheduler O(logN) (2.6.23+)

Linux Scheduling



Nice Value

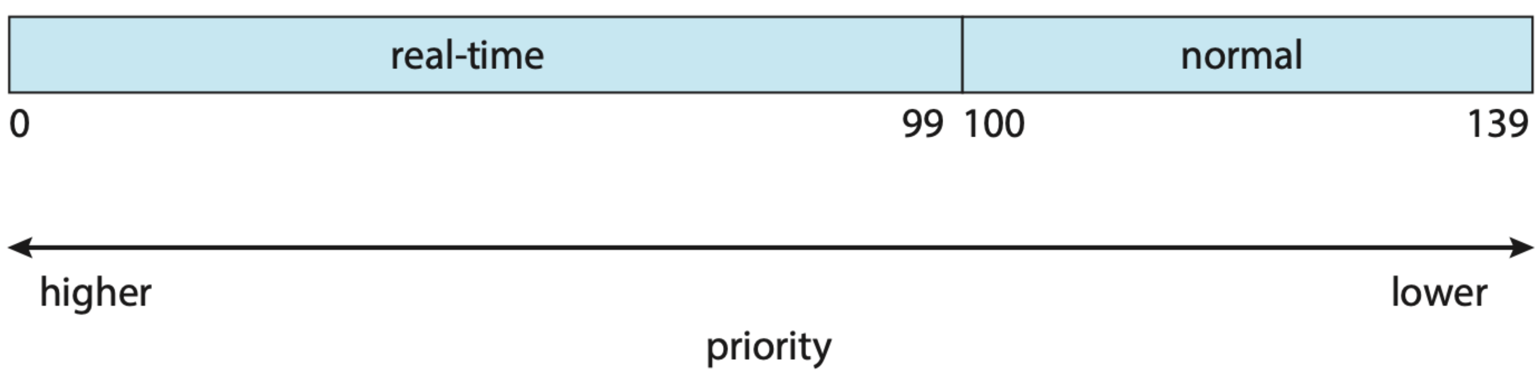
- Every normal process has a **nice value** (谦让值?善意值?)
  - default `0`, range from `-20` to `+19`.
  - "Be nice!"
    - being nice to other processes ⇒ let others run first.
    - a lower nice value ⇒ higher priority
    - a higher nice value ⇒ lower priority



Linux Scheduling

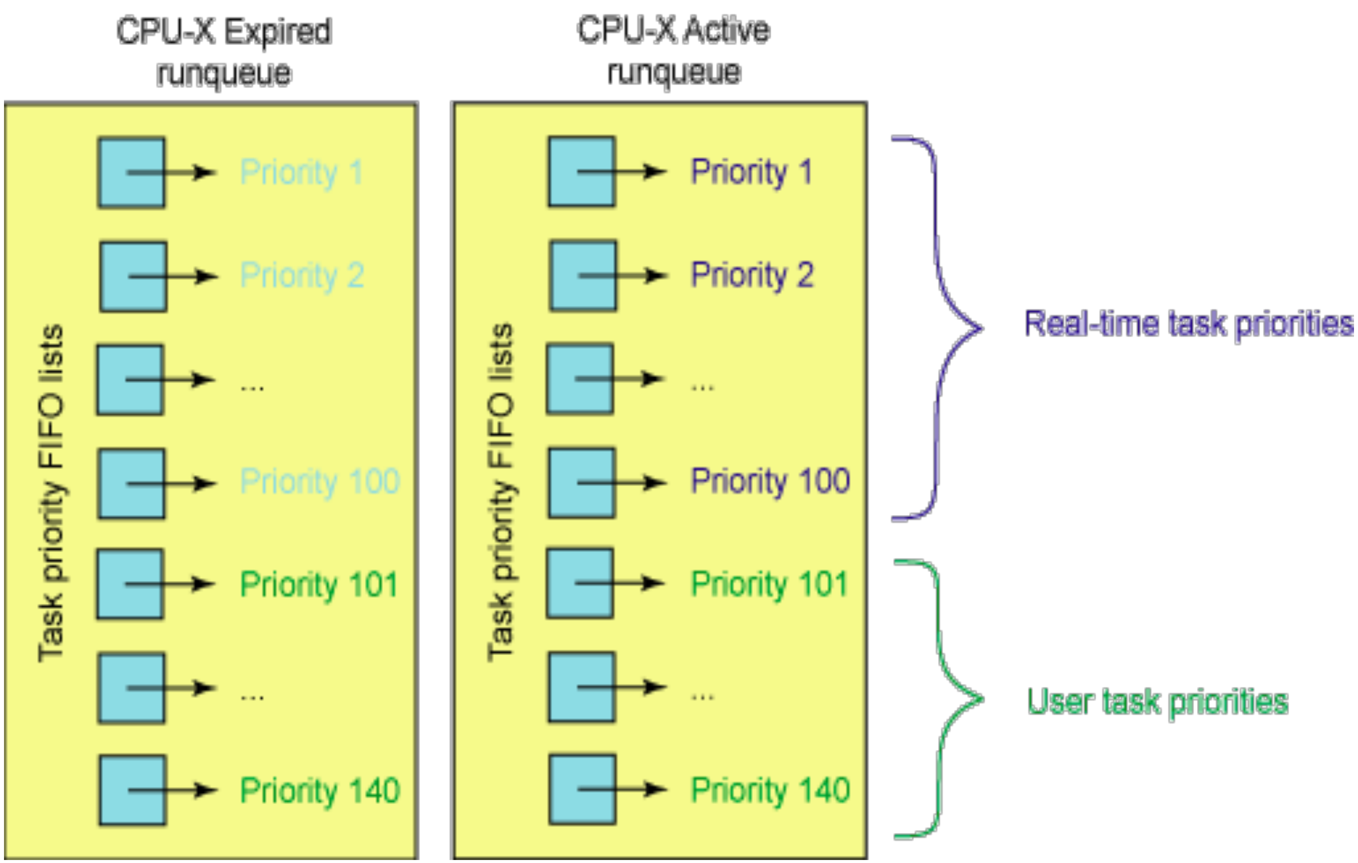
Priority

- Linux uses two separate priority ranges
  - **Real-Time/Kernel** Tasks (0 ~ 99)
  - **Normal** Tasks (100 ~ 139)
    - Priority = 120 + [Nice Value]



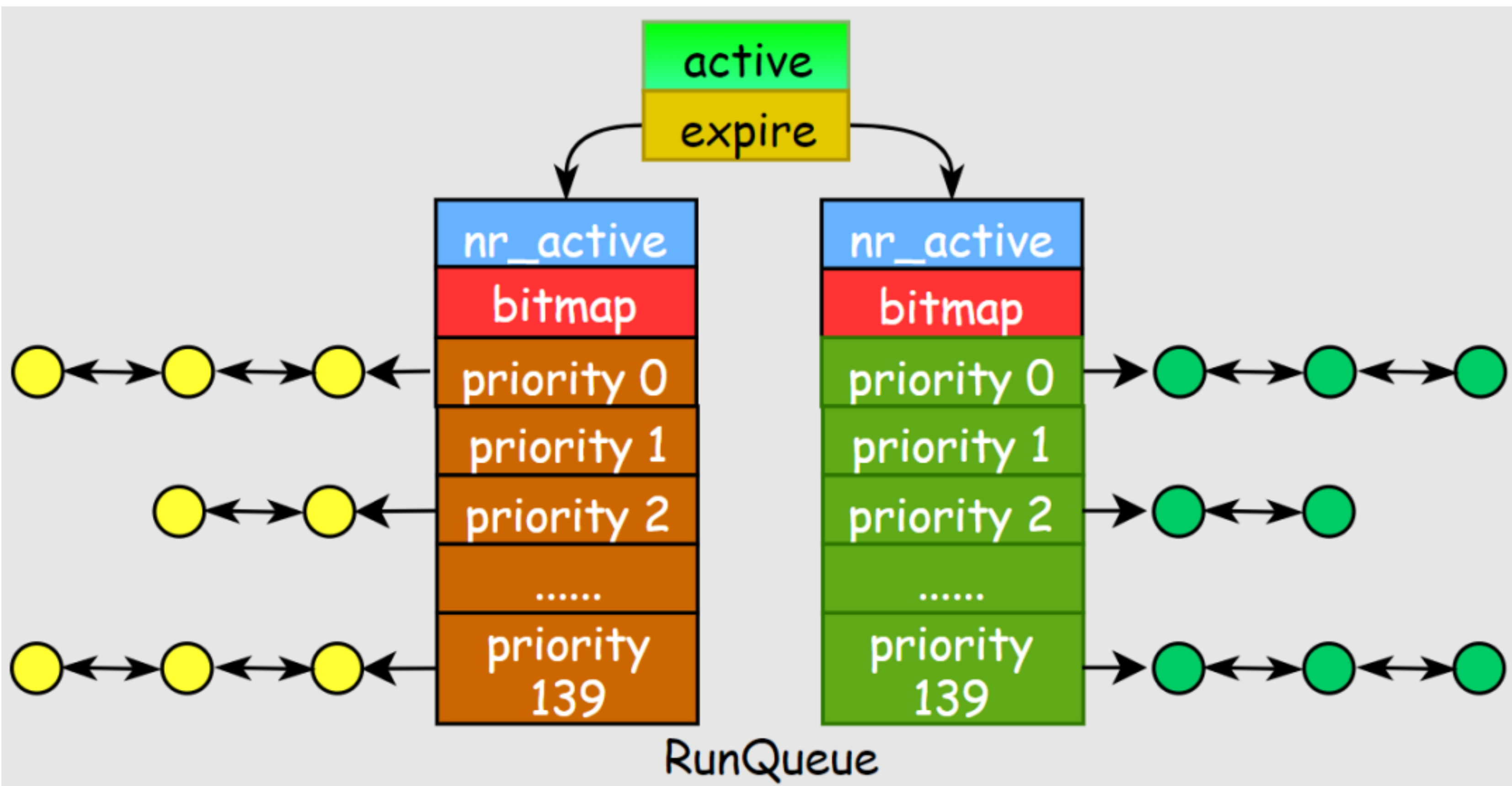
O(1) Scheduler

- All algorithms O(1)
  - ~~iterate the entire task list: O(N)~~
  - keeps track of runnable tasks in two run queues for each priority level
    - dequeue(), enqueue() $\Rightarrow$ O(1)



O(1) Scheduler

- All algorithms O(1)
  - ~~iterate the entire task list: O(N)~~
  - keeps track of runnable tasks in two run queues for each priority level
    - dequeue(), enqueue() $\Rightarrow$ O(1)



Scheduling Class

- Scheduling in Linux is based on **scheduling class**.
  - Each class is assigned a specific priority.
- Different scheduling classes $\Rightarrow$ different scheduling algorithms
- By default, Linux implements two scheduling classes:
  - A default scheduling class using the CFS scheduling algorithm.
  - A Real-Time scheduling class.
- New scheduling classes can be added, of course.

Completely Fair Scheduler (CFS)

Time Slice

- Compute effective time slice for each process:

$$time\_slice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} \times sched\_latency$$

- time_slice of current process is decreased on each tick of the system
- When time_slice reaches 0, the current process is **preempted.**

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Completely Fair Scheduler (CFS)

Virtual Run Time

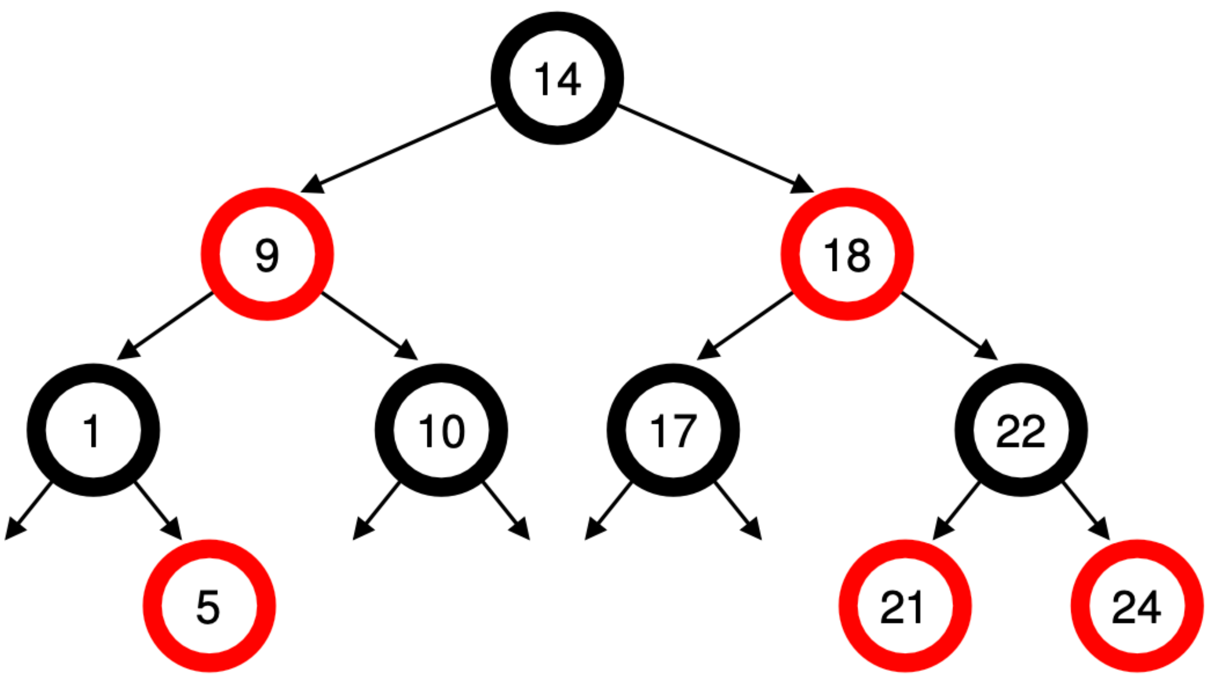
- CFS maintains per-task **virtual run time** in variable `vruntime`.
  - it records how long each task has run
- As each task runs, it accumulates `vruntime`.
- `update_curr()` is called periodically by the system timer

```
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;
    ...
    delta_exec = now - curr->exec_start;
    curr->exec_start = now;
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);
    ...
}
```

Completely Fair Scheduler (CFS)

Process Selection

- `pick_next()`: When a scheduling decision occurs, CFS will pick the task with the ***lowest*** `vruntime` to run next.
- CFS uses Red-Black Tree to manage the runnable processes
  - finding the process with smallest `vruntime`: $O(\ln N)$



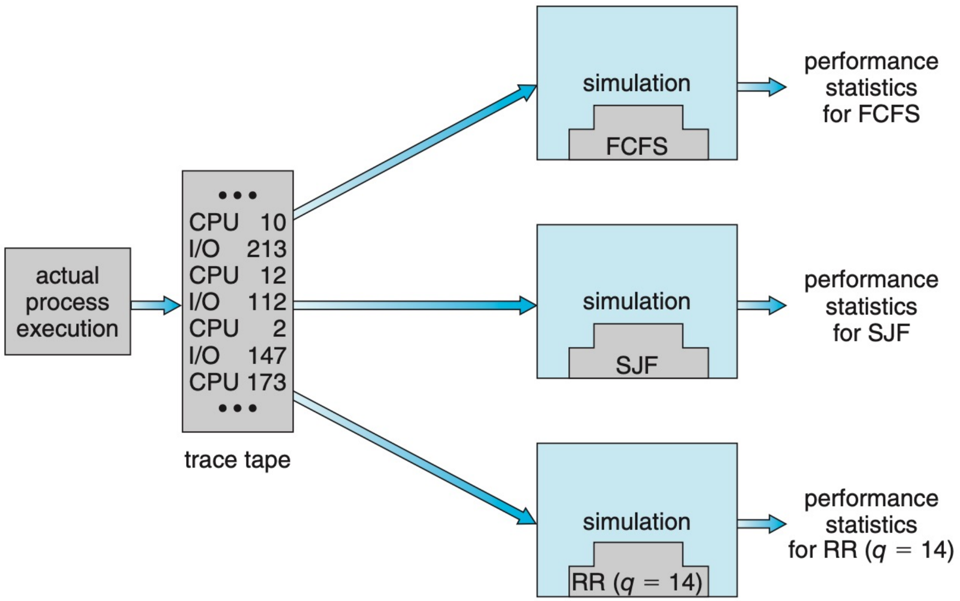
Red-Black Tree

```
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;
    /*
     * Find the right place in the rbtree:
     */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;


    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Red-Black Tree

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;


        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }


    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

- **How to Evaluate a Scheduling Algorithm?**
  - **Deterministic Modeling**
    - takes a predetermined workload and compute the performance of each algorithm for that workload
  - **Queueing Models**
    - Mathematical approach for handling stochastic workloads
  - **Implementation/Simulation**
    - Build system that allows actual algorithms to be run against actual data
    - Most flexible/general



Summary

Choosing the Right Scheduler

I Care About	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	Round Robin
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness (Wait Time to Get CPU)	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority Scheduling/MLFQ

Thank you!