OPERATING SYSTEMS

CPU Scheduling

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Contents

Multiple-Processor Scheduling

- Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, *load sharing* becomes possible—but scheduling problems become correspondingly more complex.
  - Many possibilities have been tried; and as we saw with single processor CPU scheduling, there is no one best solution.
- Several concerns in multiprocessor scheduling
  - We concentrate on systems in which the processors are identical—homogeneous (同质) —in terms of their functionality.
    - We can then use any available processor to run any process in the queue.
  - However, even with homogeneous multiprocessors, there are sometimes limitations on scheduling.
    - Consider a system with an I/O device attached to a private bus of one processor. Processes that wish to use that device must be scheduled to run on that processor.

Multiple-Processor Scheduling

- Approaches to Multiple-Processor Scheduling
  - Asymmetric multiprocessing
    - A single processor (the master server) has all scheduling decisions, I/O processing, and other system activities handled. The other processors execute only user code.
    - This approach is simple because only one processor accesses the system data structures, reducing the need for data sharing.
  - Symmetric multiprocessing (SMP)
    - All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
    - If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.
      - We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.
    - Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X.

Processor Affinity

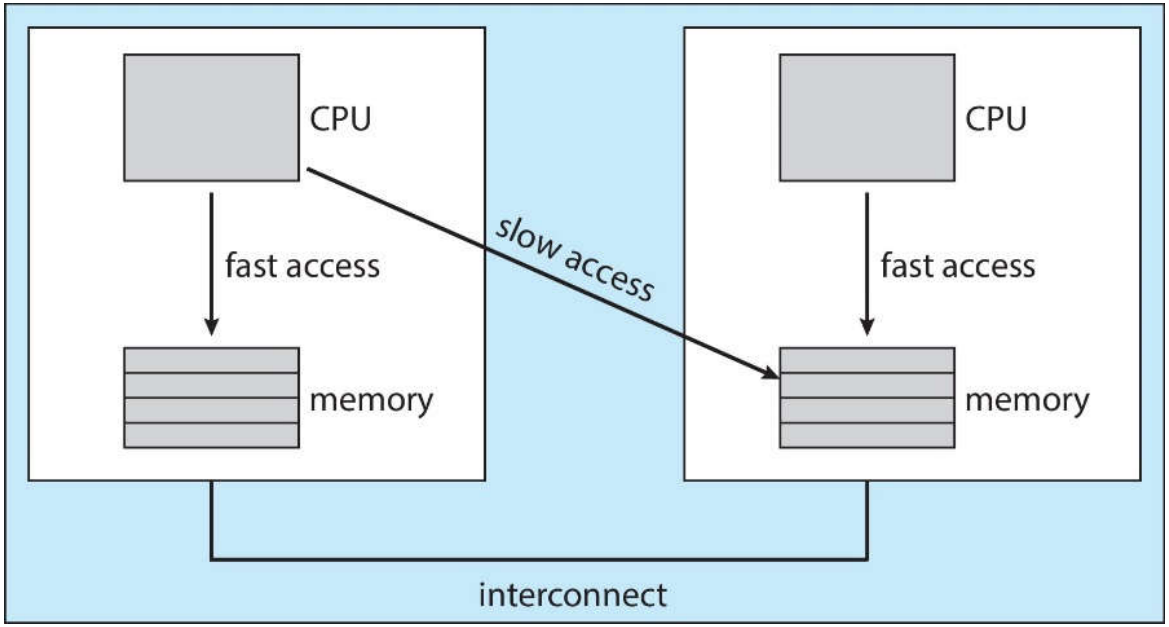
- Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory.
- Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as *processor affinity* (处理器亲和性).
  - Processor Affinity means a process has an affinity for the processor on which it is currently running.

Processor Affinity

- Soft Affinity (软亲和性)
  - The operating system will attempt to keep a process running on the same processor, but it is possible for a process to migrate between processors.
- Hard Affinity (硬亲和性)
  - Some operating systems provide system calls, allowing a process to specify a subset of processors on which it may run.
- Many systems provide both soft and hard affinity.
  - Linux implements soft affinity, and also supports hard affinity by providing the system call sched_setaffinity().

Processor Affinity

- The main-memory architecture of a system can affect processor affinity issues.
  - For example, in an architecture featuring non-uniform memory access (NUMA, 非一致内存访问) a CPU has faster access to some parts of main memory than to other parts. The CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system.



Load Balancing

- *Load Balancing* (负载均衡) attempts to keep the workload evenly distributed across all processors in an SMP system to fully utilize the benefits of having more than one processor.
- It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute (每个处理器拥有各自私有的可运行进程队列).
  - In most contemporary operating systems supporting SMP, each processor does have a private queue of eligible processes.
- On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

Load Balancing

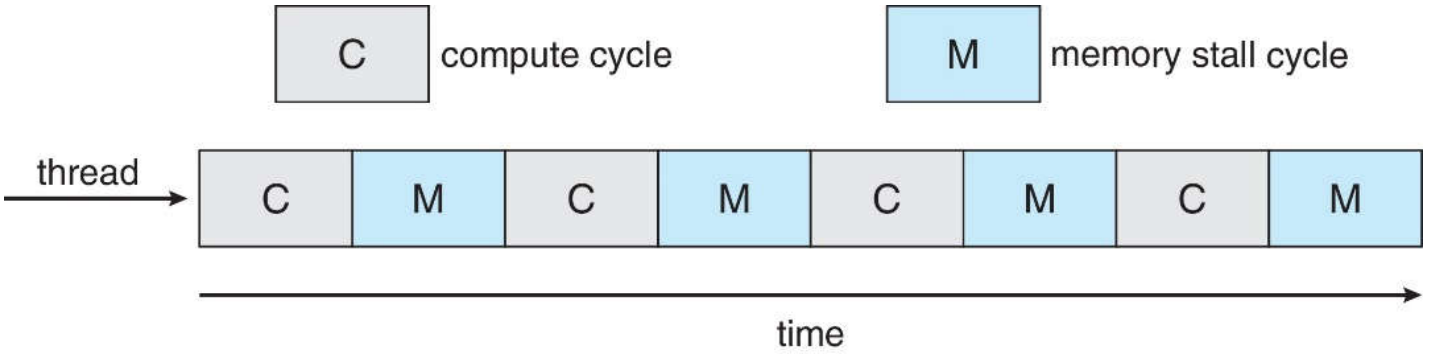
- There are two general approaches to load balancing
  - push migration
    - a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving processes from overloaded to idle or less-busy processors.
  - pull migration
    - pull migration occurs when an idle processor pulls a waiting task from a busy processor.
- Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.
  - For example, the Linux scheduler and the ULE scheduler available for FreeBSD systems implement both techniques.
- Load balancing often counteracts the benefits of processor affinity.
  - As is often the case in systems engineering, there is no absolute rule concerning what policy is best.

Multicore Processors

- A *multicore processor* places multiple processor cores on the same chip (physically). Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor.
- SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.
- Memory Stall (存储器停顿)
  - when a processor accesses memory, it may spend a significant amount of time waiting for the data to become available. This situation is called a *memory stall*.
  - Memory stall may occur for various reasons, such as a cache miss (accessing data that are not in cache memory).

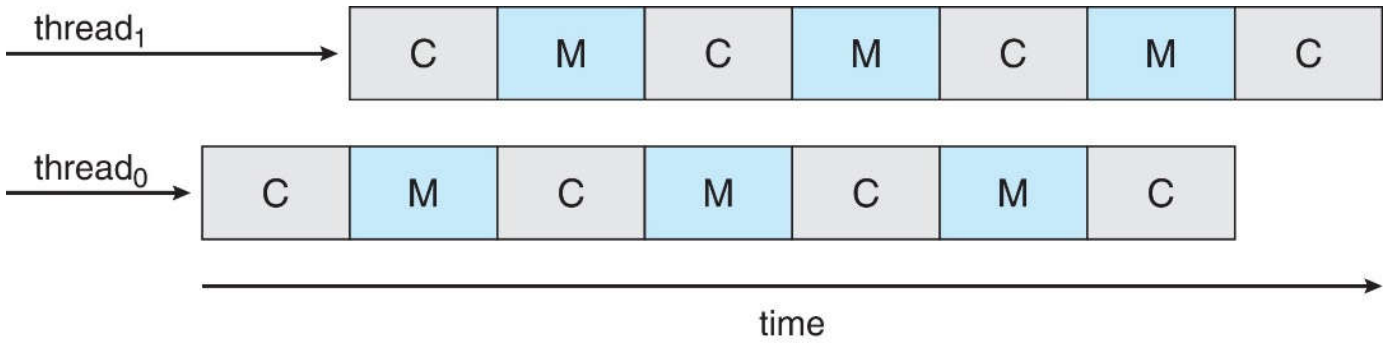
Multicore Processors

- Illustration of a memory stall.
  - The processor can spend up to 50 percent of its time waiting for data to become available from memory.



Multicore Processors

- To remedy this situation, many recent hardware designs have implemented multithreaded processor cores in which two (or more) *hardware threads* are assigned to each core. That way, if one thread stalls while waiting for memory, the core can switch to another thread.
- Illustration of a dual-threaded processor core
  - on which the execution of thread 0 and the execution of thread 1 are interleaved.



Multicore Processors

- From an operating-system perspective, each hardware thread appears as a logical processor that is available to run a software thread. Thus, on a dual-threaded, dual-core system, four logical processors are presented to the operating system.
  - But the UltraSPARC T3 CPU has sixteen cores per chip and eight hardware threads per core. From the perspective of the operating system, there appear to be 128 logical processors.
- Processing Core Multithreading
  - There are two ways to multithread a processing core: *coarse-grained* (粗粒度) and *fine-grained* (细粒度).

Multicore Processors

- Coarse-grained multithreading
  - A thread executes on a processor until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the processor must switch to another thread to begin execution.
  - The cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.
- Fine-grained (or interleaved) multithreading
  - Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity (粒度) —typically at the boundary of an instruction cycle.
  - The architectural design of fine-grained systems includes logic for thread switching.
  - The cost of switching between threads is small.

Multicore Processors

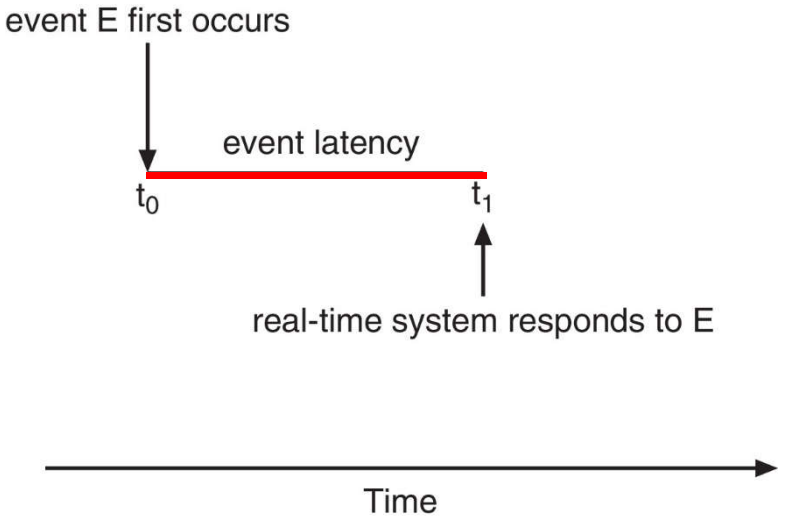
- Scheduling Levels of Multicore Processor Multithreading
  - There are two different levels of such scheduling: scheduling decisions and hardware thread selections
  - Scheduling decisions are made by the operating system as it chooses which software thread to run on each hardware thread (logical processor). For this level of scheduling, the operating system may choose any scheduling algorithm we have discussed before in the previous lecture.
  - In second level, each core decides which hardware thread to run.
    - The UltraSPARC T3 uses a simple Round-Robin algorithm to schedule the eight hardware threads to each core.
    - The Intel Itanium (dual-core processor with two hardware threads per core) assigned to each hardware thread a dynamic *urgency* value ranging from 0 (the lowest) to 7. The Itanium identifies five different events that may trigger a thread switch. When one of these events occurs, the thread-switching logic selects the thread with the highest urgency value to execute on the processor core.

Real-Time CPU Scheduling

- *Soft real-time systems* guarantee that a critical real-time process will be given preference over noncritical processes, with no guarantee as to when a critical real-time process will be scheduled.
- *Hard real-time systems* have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.
- We explore several issues related to process scheduling in both soft and hard real-time operating systems including:
  - Minimizing Latency
  - Priority-Based Scheduling
  - Rate-Monotonic Scheduling (RMS, 单调速率调度)
  - Earliest-Deadline-First Scheduling (EDFS,最早截止期优先调度)
  - Proportional Share Scheduling.

Minimizing Latency

- Real-time system is event-driven. When an event occurs, the system must respond to and service it as quickly as possible.
  - Software event
    - e.g., a timer expires.
  - Hardware event
    - e.g., a remote-controlled vehicle detects that it is approaching an obstruction.
- Event Latency
  - *Event latency* is the amount of time that elapses from when an event occurs to when it is serviced.

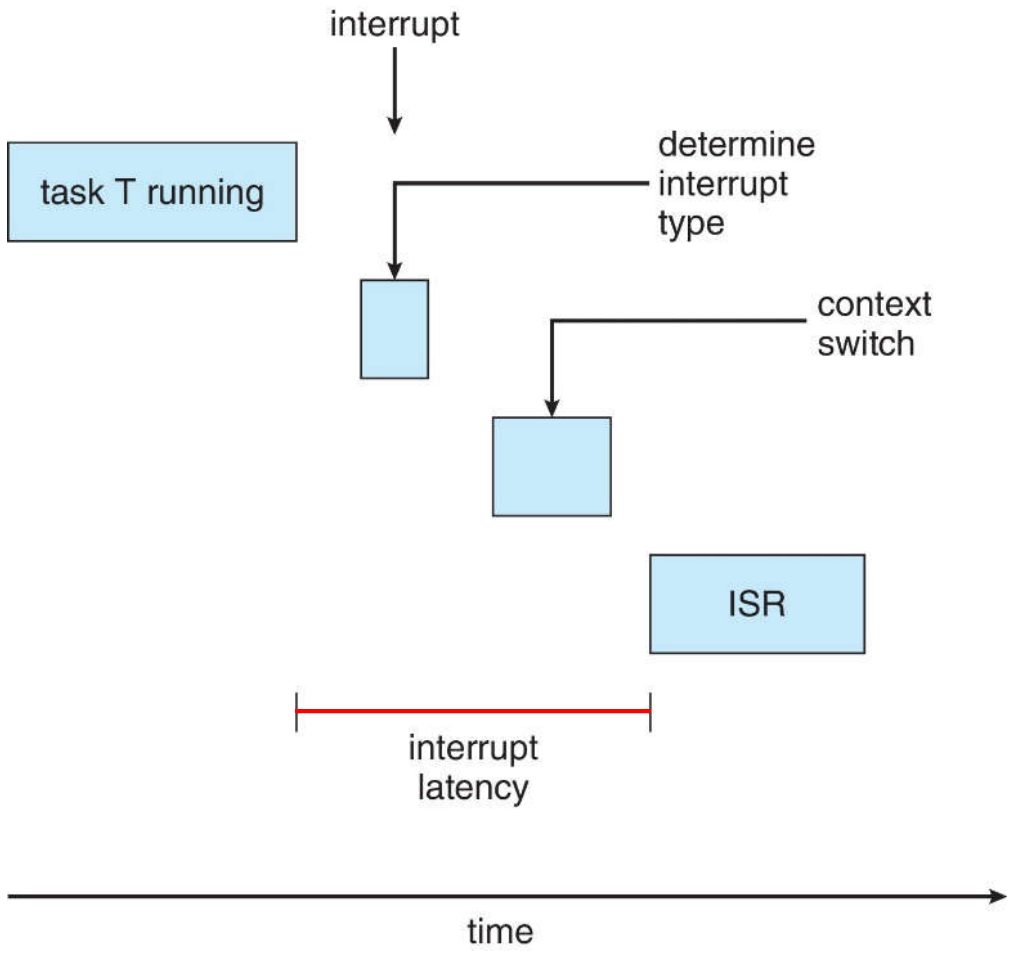


Minimizing Latency

- Two types of latencies affect the performance of real-time systems:
  - Interrupt latency
  - Dispatch latency
- Interrupt Latency
  - *Interrupt latency* refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.
    - When an interrupt occurs, the interrupt latency is the total time required by OS to perform these tasks:
      - first complete the instruction it is executing
      - determine the type of interrupt that occurred
      - save the state of the current process
      - service the interrupt by using the specific interrupt service routine (ISR)
    - For hard real-time systems, interrupt latency must be bounded to meet the strict requirements
    - The amount of time interrupts may be disabled while kernel data structures are being updated is important.

Minimizing Latency

- Interrupt Latency.

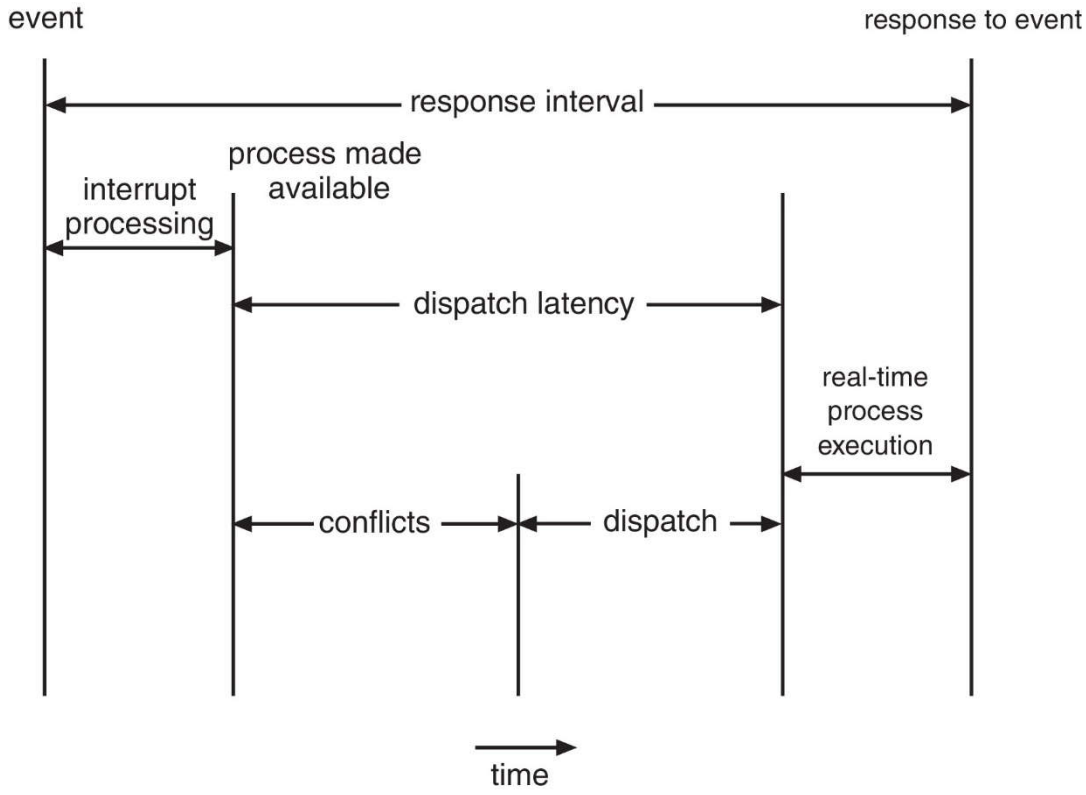


Minimizing Latency

- Dispatch Latency
  - *Dispatch latency* refers to the amount of time required for the scheduling dispatcher to stop one process and start another.
    - Providing real-time tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well.
    - The most effective technique for keeping dispatch latency low is to provide preemptive kernels.
  - The makeup of dispatch latency illustrates in next slide. The *conflict phase* of dispatch latency has two components:
    - Preemption of any process running in the kernel
    - Release by low-priority processes of resources needed by a high-priority process.
  - E.g., in Solaris, the dispatch latency with preemption disabled is over a hundred milliseconds. With preemption enabled, it is reduced to less than a millisecond.

Minimizing Latency

- Dispatch Latency



- Two components in the conflict phase of dispatch latency:
  - Preemption of any process running in the kernel
  - Release by low-priority processes of resources needed by a high-priority process.

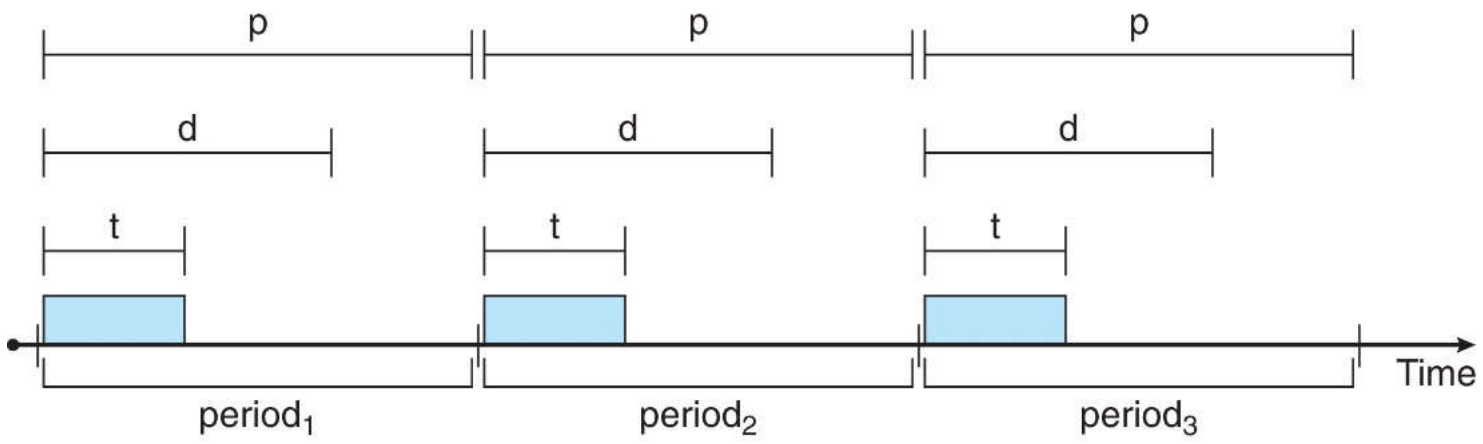
Priority-Based Scheduling

- The scheduler for a real-time operating system must support a priority-based algorithm with preemption.
  - The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU.
- Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features.
  - In the remainder of this section, we cover scheduling algorithms appropriate for hard real-time systems.
- Linux, Windows, and Solaris assign real-time processes the highest scheduling priority.
  - Windows has 32 different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes.
  - Solaris and Linux have similar prioritization schemes.

Rate-Monotonic Scheduling

- Periodic Process
  - A process is *periodic* if it requires the CPU at constant intervals (periods). Once a periodic process has acquired the CPU, it has a fixed processing time *t*, a deadline *d* by which it must be serviced by the CPU, a period *p*, and the *rate* of a periodic task *1/p*. The CPU utilization of the process may be measured as *t/p*.
  - The relationship of *t*, *d* and *p* can be expressed as

$$0 \leq t \leq d \leq p.$$



Rate-Monotonic Scheduling

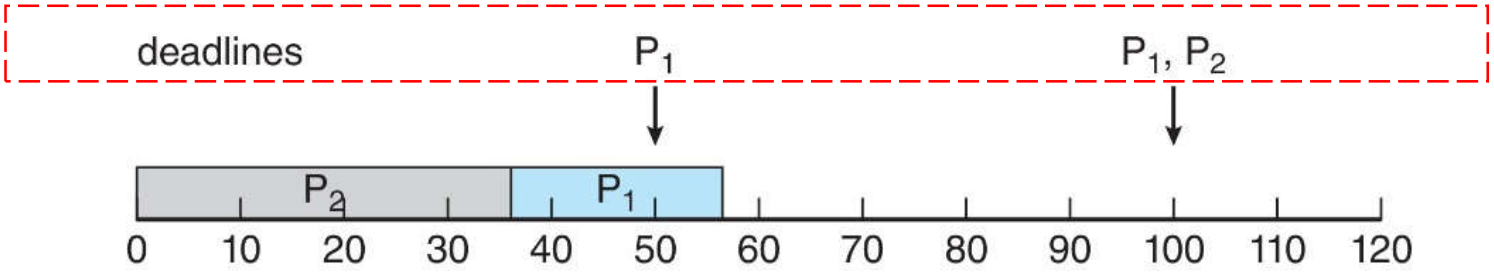
- Admission Control
  - A process may have to announce its deadline requirements to the scheduler. Using an *admission-control* algorithm, the scheduler does one of two things:
    - admits the process, guaranteeing that the process will complete on time.
    - rejects the request as impossible.

Rate-Monotonic Scheduling

- The *Rate-Monotonic Scheduling* (RMS, 单调速率调度) algorithm schedules periodic tasks using a static priority policy with preemption.
- Upon entering the system, each periodic task is assigned a priority based on its rate (i.e., inversely on its period).
  - The shorter the period (the larger the rate), the higher the priority. That is to assign a higher priority to tasks that require the CPU more often.
- Furthermore, rate-monotonic scheduling assumes that the 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.

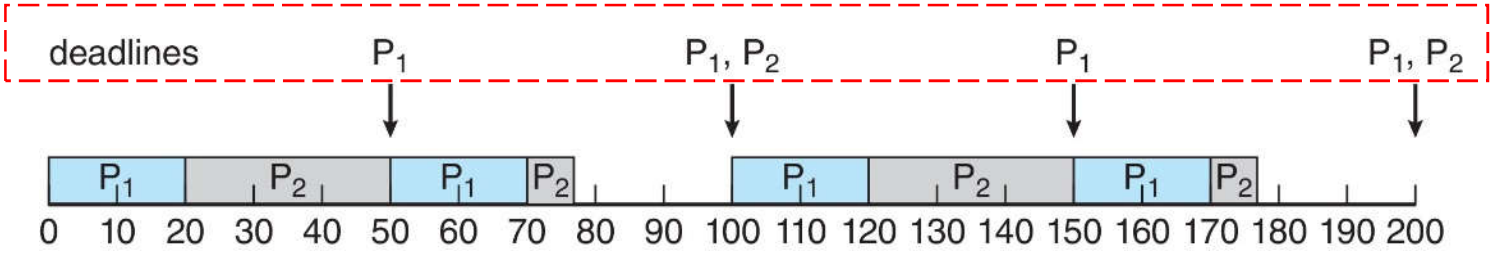
Rate-Monotonic Scheduling

- Example 1.
  - Let P_1 and P_2 be periodic processes. $p_1 = 50$, $t_1 = 20$, $p_2 = 100$, and $t_2 = 35$. The deadline for each process requires that it complete its CPU burst by the start of its next period ($d_i = p_i$).
  - The combined CPU utilization of the two processes is
    $(20/50)+(35/100) = 0.75$.
    and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 25 percent available time.
  - Suppose we assign P_2 a higher priority than P_1. The execution of P_1 and P_2 in this situation is shown below.
  - 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 at time 50, so the scheduler has caused P_1 to miss its deadline.



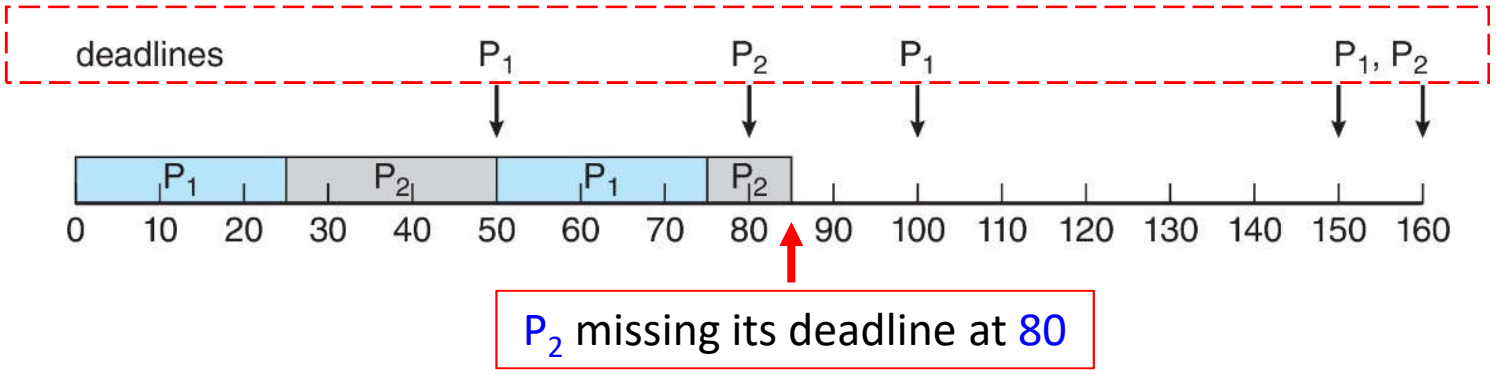
Rate-Monotonic Scheduling

- Example 1.
  - Now suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2 because the period of P_1 is shorter than that of P_2 ($50 = p_1 < p_2 = 100$). The execution of these processes in this situation is shown below.
  - P_1 starts first and completes its CPU burst at time 20, 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 (P_2 still has 5 remaining in its CPU burst). P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2. P_2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P_1 is scheduled again.



Rate-Monotonic Scheduling

- Example 2. Missing deadline with RMS.
  - Let P_1 and P_2 be periodic processes. $p_1 = 50$, $t_1 = 25$, $p_2 = 80$ and $t_2 = 35$. The deadline for each process is also at the start of its next period ($d_i = p_i$). Rate-monotonic scheduling would assign process P_1 a higher priority. The combined CPU utilization is
    $(25/50)+(35/80) = 0.94$.
  - The scheduling of processes P_1 and P_2 is shown below. Initially, P_1 runs until it completes its CPU burst at time 25. Process P_2 then begins running and runs until time 50, when it is preempted by P_1. At this point, P_2 still has 10 remaining in its CPU burst. Process P_1 runs until time 75; consequently, P_2 finishes its burst at time 85, missing the deadline for completion of its CPU burst at time 80.



Rate-Monotonic Scheduling

- 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 fully to maximize CPU resources. The worst-case CPU utilization for scheduling N processes is

$$N \times (2^{1/N} - 1).$$

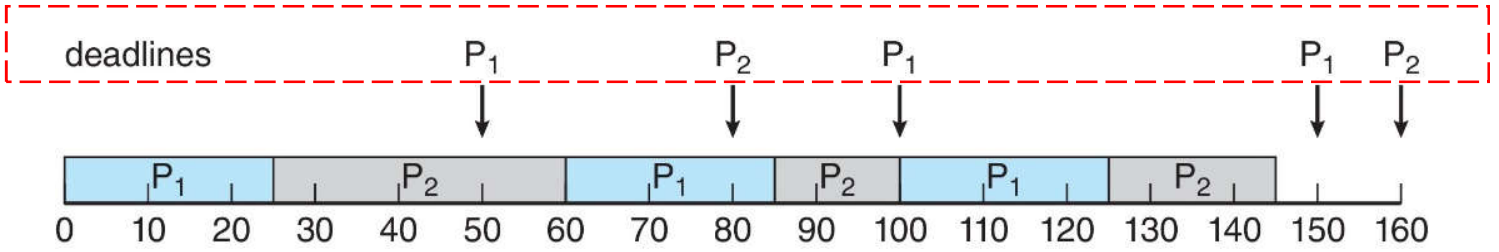
  - When N = 1, $N \times (2^{1/N} - 1) = 1$.
  - When N = 2, $N \times (2^{1/N} - 1) \approx 0.83$.
  - When $N \to \infty$, $\lim_{n\to\infty} (N \times (2^{1/N} - 1)) = \ln(2) \approx 0.69$.
- In example 1, combined CPU utilization for the two processes scheduled is 75 percent; therefore, the RMS algorithm is guaranteed to schedule them so that they can meet their deadlines.
- In example 2, combined CPU utilization is approximately 94 percent; therefore, RMS cannot guarantee that they can be scheduled so that they meet their deadlines.

Earliest-Deadline-First Scheduling

- *Earliest-Deadline-First* (EDF, 最早截止期优先) is a priority-based scheduling. It *dynamically* assigns priorities according to next deadlines. The earlier the next deadline, the higher the priority.
  - Under the EDF policy, 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.
- Note how this differs from rate-monotonic scheduling, where priorities are *fixed*.

Earliest-Deadline-First Scheduling

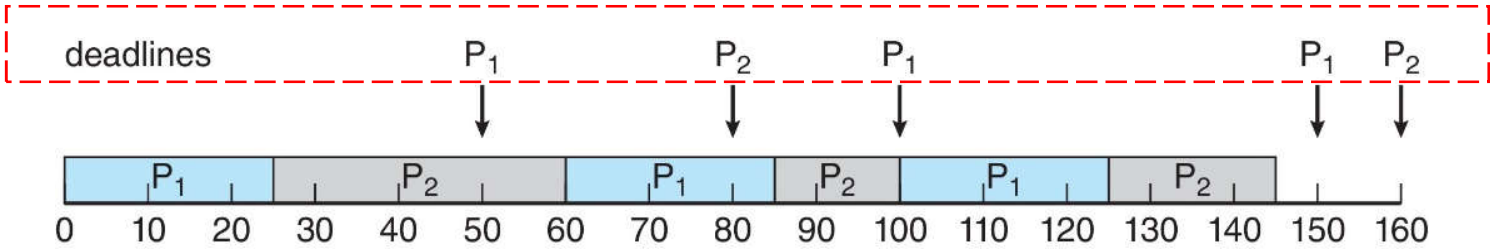
- Example 3.
  - Consider the processes P_1 and P_2 of Example 2. $p_1 = 50$, $t_1 = 25$, $p_2 = 80$, $t_2 = 35$ and each deadline at the start of its next period ($d_i = p_i$). This example was missing deadline with RMS.
  - The EDF scheduling of these processes is shown below.



  - Process P_1 has a higher initial priority than P_2 because its first deadline $d_1 = 50 < d_2 = 80$.
  - Process P_2 begins running at the end of the CPU burst for P_1.
  - P_2 now has a higher priority than P_1 because its next deadline $d_2 = 80 < d_1 = 100$. EDF scheduling allows process P_2 to continue running. Thus, both P_1 and P_2 meet their first deadlines.
    - Recall that, however, RMS allows P_1 to preempt P_2 at the beginning of its next period at time 50.

Earliest-Deadline-First Scheduling

- Example 3.
  - Consider the processes P_1 and P_2 of Example 2. $p_1 = 50$, $t_1 = 25$, $p_2 = 80$, $t_2 = 35$ and each deadline at the start of its next period ($d_i = p_i$). This example was missing deadline with RMS.
  - The EDF scheduling of these processes is shown below.



  - Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. And P_2 begins running at this point.
  - At the point of time 100, P_2 is preempted by P_1 because at this point, $d_1 = 150 < d_2 = 160$.
  - 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

- Unlike RMS, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst.
  - The only requirement is that a process announce its deadline to the scheduler when it becomes runnable.
- The appeal (魅力) of EDF scheduling is that it is theoretically optimal.
  - Theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent.
- 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

- Proportional share schedulers operate by allocating T shares of time among all n applications. Let the i^{th} application receive N_i shares of time, then $T < \Sigma N_i$ (i = 1 to n). The i^{th} application will have N_i /T of the total processor time.
- Example 4.
  - Assume that a total of T = 100 shares is to be divided among three processes, A, B, and C. N_A = 50, N_B = 15, N_C = 20. A proportional share scheduler ensures that A will have 50 percent of total processor time, B will have 15 percent, and C will have 20 percent.
  - Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available.
    - As 50 + 15 + 20 = 85 shares of the total of 100 shares have been allocated, if a new process D requested 30 shares, the admission controller would deny D entry into the system.

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 a 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_getsched_policy(pthread_attr_t *attr, int *policy)
Pthread_attr_setsched_policy(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
    (1) a pointer to an integer that is set to the current scheduling policy for the pthread_attr_getsched_policy() function or
    (2) 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.
- A POSIX Pthread real-time scheduling program using this API is illustrated in next slide. This program first determines the current scheduling policy and then sets the scheduling algorithm to SCHED_FIFO.

POSIX Real-Time Scheduling

- Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (1)

```
  /* compiling with -lpthread option */
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

void *runner(void*);

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    struct sched_param param;
    pthread_attr_t attr;

      /* get the default attributes */
    pthread_attr_init(&attr);
```

POSIX Real-Time Scheduling

Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (2)

```
  /* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    printf("Unable to get policy.\n");
else {
    printf("The current scheduling policy is ");
    if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
}
  /* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get priority from SCHED_OTHER.\n");
else
    printf("current sched_priority = %d\n", param.sched_priority);
printf("priority_min of OTHER is %d, max is %d\n",
sched_get_priority_min(SCHED_OTHER), sched_get_priority_max(SCHED_OTHER));
param.sched_priority = 10; /* set the priority to 10 */
if (pthread_attr_setschedparam(&attr, &param) != 0)
    printf("Unable to set prority to 10.\n");
  /* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get prioty from SCHED_RR.\n");
else
    printf("The new sched_priority = %d\n", param.sched_priority);
```

POSIX Real-Time Scheduling

Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (3)

```
  /* set the scheduling policy to RR */
if (pthread_attr_setschedpolicy(&attr, SCHED_RR) != 0)
    printf("Unable to set policy to SCHED_RR.\n");

  /* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    printf("Unable to get policy.\n");
else {
    printf("The new scheduling policy is ");
    if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
}

  /* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get prioty from SCHED_RR.\n");
else
    printf("current sched_priority = %d\n", param.sched_priority);
printf("priority_min of RR is %d, max is %d\n",
sched_get_priority_min(SCHED_RR), sched_get_priority_max(SCHED_RR));
```

POSIX Real-Time Scheduling

- Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (4)

```
  /* set the priority to 10 */
param.sched_priority = 10;
if (pthread_attr_setschedparam(&attr, &param) != 0)
    printf("Unable to set prority.\n");

  /* get the current priority */
if (pthread_attr_getschedparam(&attr, &param) != 0)
    printf("Unable to get prioty from SCHED_RR.\n");
else
    printf("The new sched_priority = %d\n", param.sched_priority);

  /* set the scheduling policy to FIFO */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    printf("Unable to set policy to SCHED_FIFO.\n");

  /* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    printf("Unable to get policy.\n");
else {
    printf("The new scheduling policy is ");
    if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR) printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
}
```

POSIX Real-Time Scheduling

Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (5)

```
    /* get the current priority */
    if (pthread_attr_getschedparam(&attr, &param) != 0)
        printf("Unable to get prioty from SCHED_FIFO.\n");
    else
        printf("current sched_priority = %d\n", param.sched_priority);
    printf("priority_min of FIFO is %d, max is %d\n",
sched_get_priority_min(SCHED_FIFO), sched_get_priority_max(SCHED_FIFO));

    /* set the priority to 50 */
    param.sched_priority = 50;
    if (pthread_attr_setschedparam(&attr, &param) != 0)
        printf("Unable to set prority.\n");

    /* get the current priority */
    if (pthread_attr_getschedparam(&attr, &param) != 0)
        printf("Unable to get prioty from SCHED_FIFO\n");
    else
        printf("The new sched_priority = %d\n", param.sched_priority);
```

POSIX Real-Time Scheduling

- Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (6)

```
      /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, &runner, NULL);

      /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

  /* Each thread will begin control in this function */
void *runner(void *param)
{
      /* do some work ... */
    pthread_exit(0);
}
```

POSIX Real-Time Scheduling

- Alg.20-1-pthread-scheduling-1.c: Scheduling Policies and Priorities (6)

```
    /* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, &runner, NULL);
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.20-1-pthread-scheduling-1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
The current scheduling policy is SCHED_OTHER
current sched_priority = 0
priority_min of OTHER is 0, max is 0
Unable to set prority to 10.
The new sched_priority = 0
The new scheduling policy is SCHED_RR
current sched_priority = 0
priority_min of RR is 1, max is 99
The new sched_priority = 10
The new scheduling policy is SCHED_FIFO
current sched_priority = 10
priority_min of FIFO is 1, max is 99
The new sched_priority = 20
isscgy@ubuntu:/mnt/os-2020$
```

Linux Scheduling

- Prior to kernel Version 2.5
  - not designed with SMP systems and not adequately support systems with multiple processors.
  - resulted in poor performance for systems with a large number of runnable processes.
- Kernel 2.5
  - include a $O(1)$ scheduling algorithm
    - that ran in constant time regardless of the number of tasks in the system.
  - provide increased support for SMP systems, including processor affinity and load balancing between processors.
  - led to poor response times for the interactive processes.
- Kernel 2.6
  - in release 2.6.23 of the kernel, the *Completely Fair Scheduler* (CFS) became the default Linux scheduling algorithm.

Example: Linux Scheduling

Scheduling Class

- Scheduling in the Linux system is based on scheduling classes.
  - Each class is assigned a specific priority.
- Different scheduling classes has its specific scheduling algorithms
  - E.g., the scheduling criteria for a Linux server may be different from those for a mobile device running Linux.
- The scheduler selects the highest-priority task belonging to the highest-priority scheduling class.
- Standard Linux kernels implement two scheduling classes:
  (1) a default scheduling class using the CFS scheduling algorithm.
  (2) a real-time scheduling class.
- New scheduling classes can, of course, be added.

Completely Fair Scheduler

- Nice Value
  - Completely Fair Scheduler (CFS) assigns a proportion of CPU processing time to each process (task). This proportion is calculated based on the *nice value* (善意値？) assigned to each task.
    - The term *nice* comes from the idea that if a task increases its nice value, it is *being nice* to other tasks in the system by lowering its relative priority.
  - Nice values, default of 0, range from -20 to +19, where a lower nice value indicates a higher relative priority.
    - Thus tasks with lower nice values will receive a higher proportion of CPU processing time.
  - CFS doesn't use discrete values of time slices and instead identifies a *targeted latency*, which is an interval of time during which every runnable task should run at least once. Proportions of CPU time are allocated from the value of targeted latency. In addition to having default and minimum values, targeted latency can increase if the number of active tasks in the system grows beyond a certain threshold.

Example: Linux Scheduling

Completely Fair Scheduler

- Virtual Run Time
  - CFS doesn't directly assign priorities. Rather, it records how long each task has run by maintaining the *virtual run time* of each task using the per-task variable vruntime.
  - The virtual run time is associated with a *decay factor* (衰减因子) based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks.
  - For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time.
    - E.g., if a task with default priority runs for 200 milliseconds, its vruntime will also be 200 milliseconds.
  - For a lower-priority task, its virtual run time will be higher than its actual run time. Similarly, the vruntime of a higher-priority task will be less than its actual run time.

Completely Fair Scheduler

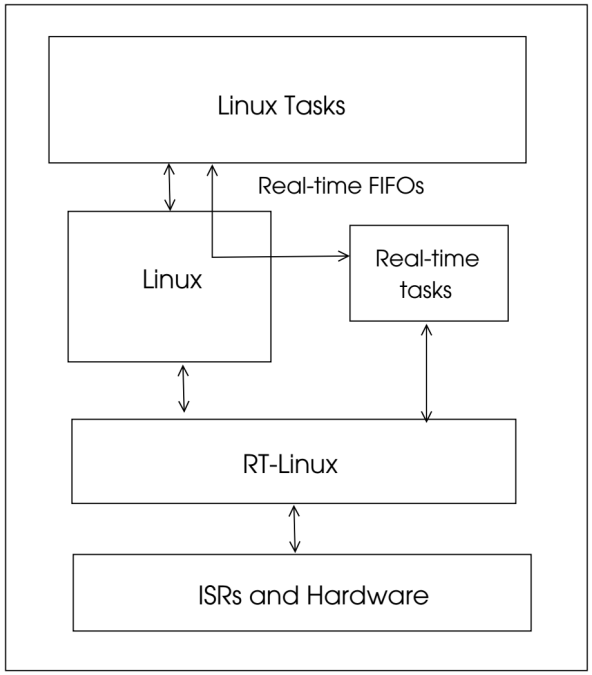
- Virtual Run Time
  - Assume that two tasks have the same nice values. One task is I/O-bound and the other is CPU-bound. Then the value of vruntime will eventually be lower for the I/O-bound task than for the CPU-bound task, giving the I/O-bound task higher priority than the CPU-bound task. At that point, if the CPU-bound task is executing when the I/O-bound task becomes eligible to run, the I/O-bound task will preempt the CPU-bound task.

Completely Fair Scheduler

- Virtual Run Time
  - Linux also implements real-time scheduling using the POSIX standard as described. Any task scheduled using either the SCHED_FIFO or the SCHED_RR real-time policy runs at a higher priority than normal (non-real-time) tasks.
  - Linux uses two separate priority ranges, one for real-time tasks and a second for normal tasks.
    - Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned from 100 to 139.
  - These two ranges map into a global priority scheme wherein numerically lower values indicate higher relative priorities. Normal tasks are assigned a priority based on their nice values, where a value of -20 maps to priority 100 and a nice value of +19 maps to 139.

Real-time Linux (RTLinux)

- RTLinux (1997) was developed as an extension to the Linux operating system with added real-time capabilities to make it predictable. The main sources of unpredictability in the Linux operating system is the scheduler which is optimized for best throughput rather than being predictable; and interrupt handling and virtual memory management.
- RTLinux is constructed as a small real-time kernel that runs under Linux and as such, it has a higher priority than the Linux kernel as depicted.



Real-time Linux (RTLinux)

- Interrupts are first handed to the RTLinux kernel and the Linux kernel is preempted when a real-time task becomes available.
- When an interrupt occurs, it is first handled by the ISR of the RTLinux which activates a real-time task resulting in the scheduler to be called.
  - Passing of interrupts between the RTLinux and Linux is handled differently by different versions of RTLinux.
- The Linux operating system handles device initialization, any blocking dynamic resource allocation and installs the components of the RTLinux.
- The scheduler and the real-time FIFOs are two core modules of RTLinux and rate monotonic and earliest deadline first scheduling policies are provided by RTLinux.
- The application interface includes system calls for interrupt and task management.

Algorithm Evaluation

- Selecting a CPU-scheduling algorithm for a particular system can be difficult. The answer depends on too many factors to give any ...
  - the system workload 工作负载 (extremely variable).
  - hardware support for the dispatcher.
  - relative weighting of performance criteria (response time, CPU utilization, throughput ...).
    - Maximizing CPU utilization under the constraint that the maximum response time is 1 second.
    - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time.
  - The evaluation method used (each has its limitations ...).
- Evaluation methods may include
  - Deterministic modeling
  - Queuing models
  - Simulations
  - Implementation

Deterministic Modeling

- One major class of evaluation methods is *analytic evaluation*. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload.
- *Deterministic modeling* (确定性建模) is one type of analytic evaluation. It takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Assume the workload shown below. All five processes arrive at time 0, in the order given, with the CPU burst time given in milliseconds. Consider the FCFS, SJF, and RR (quantum = 10ms) scheduling algorithms for this set of processes.
  - Which algorithm would give the minimum average waiting time?

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Modeling

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- For the FCFS algorithm, we would execute the processes as

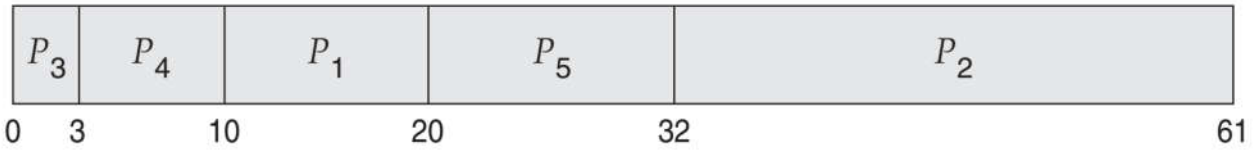
  - The waiting time is 0ms for process P_1, 10ms for P_2, 39ms for P_3, 42ms for P_4, and 49ms for P_5. Thus, the average waiting time is

```
(0 + 10 + 39 + 42 + 49)/5 = 28ms.
```

Deterministic Modeling

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- With non-preemptive SJF scheduling, we execute the processes as



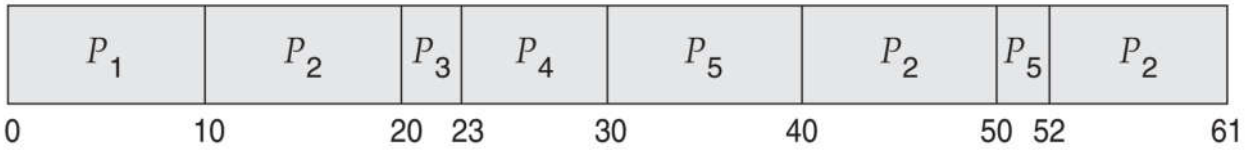
  - The waiting time is 10ms for process P_1, 32ms for P_2, 0ms for P_3, 3ms for P_4, and 20ms for P_5. Thus, the average waiting time is

```
(10 + 32 + 0 + 3 + 20)/5 = 13ms.
```

Deterministic Modeling

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- With the RR algorithm, we execute the processes as



  - The waiting time is 0ms for P_1, 32ms for P_2, 20ms for P_3, 23ms for P_4, and 40ms for P_5. Thus, the average waiting time is

$$(0 + 32 + 20 + 23 + 40)/5 = 23\text{ms}.$$

Deterministic Modeling

- In this case, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.
- Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms.
  - However, it requires exact numbers for input, and its answers apply only to those cases.
- The main uses of deterministic modeling are in describing scheduling algorithms and providing examples.
  - In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately.
  - For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

Queueing Models

- On many systems, what can be determined is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated.
- It is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms from these two distributions:
  - The distribution of the probability of a particular CPU burst, commonly exponential and described by its mean.
  - The distribution of times when processes arrive in the system (the arrival-time distribution).
- Queueing-network Analysis
  - The computer system is described as a network of servers with its ready queue, as is the I/O system with its device queues.
  - Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called *queueing-network analysis.*

Queueing Models

- Queueing-network Analysis
  - Let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

    $$n = \lambda \times W.$$

  - This equation, known as *Little's formula* (利特尔公式/利特尔法则), is particularly useful because it is valid for any scheduling algorithm and arrival distribution.
    - *Little*'s formula (*John Little*, 1954) is one of the most well-known and most useful conservation laws in queueing theory and stochastic systems. It states that the time average number of units in system equals the arrival rate of units times the average time-in-system per unit .

Queueing Models

- Queueing-network Analysis
  - E.g., let $\lambda = 7$ (that 7 processes arrive every second on average), and $n = 14$ (that there are normally 14 processes in the queue), then by *Little*'s formula we have $W = n / \lambda = 2$ (the average waiting time per process as 2 seconds).
- Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations.
  - The classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with.
  - Arrival and service distributions are defined in mathematically tractable—but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate.
- Queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

Simulations

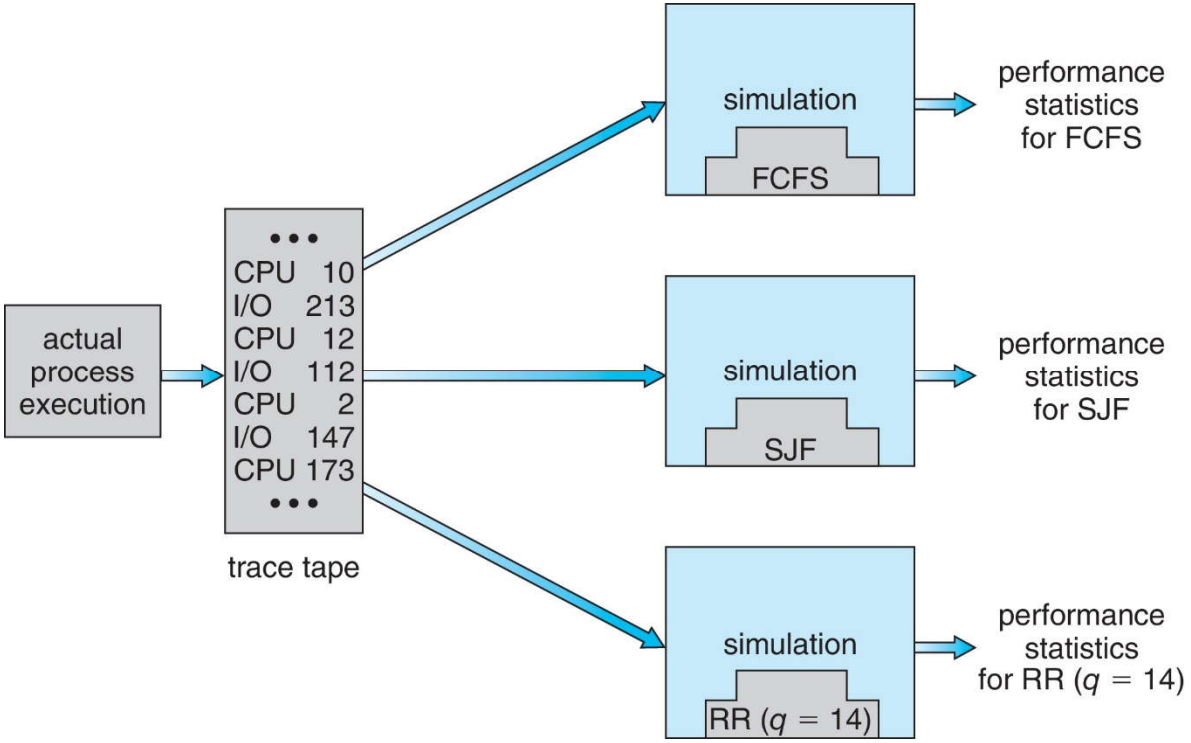
- Computer System Simulator
  - Software data structures represent the major components of the computer system.
  - System clock is represented by a variable.
    - As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler.
  - As the simulation executes, statistics that indicate algorithm performance are gathered and printed.
- Data to Drive the Simulation
  - The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to *probability distributions*.
  - The distributions can be defined mathematically such as uniform (均匀分布), exponential (指数分布), Poisson (泊松分布).
  - Also the distributions can be defined empirically (基于经验), and we need to take measurements of the actual system under study.

Simulations

- Trace Tape
  - A distribution-driven simulation may be inaccurate because the frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence.
  - A trace tape is used to monitor the real system and recording the sequence of actual events. And then the trace tape is used to drive the simulation.
  - Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations

- Evaluation of CPU schedulers by simulation



- Simulations can be expensive, often requiring hours of computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.