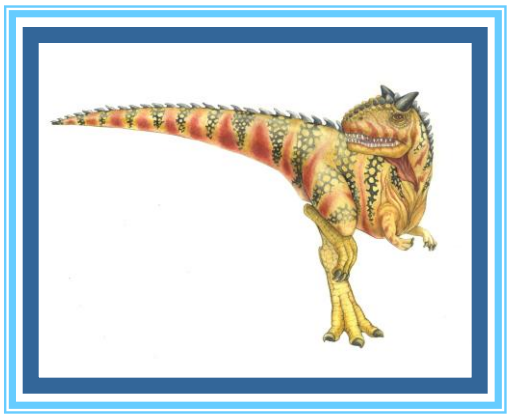
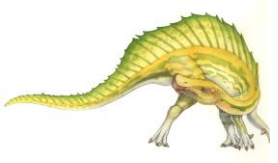
Chapter 5: CPU Scheduling





Outline

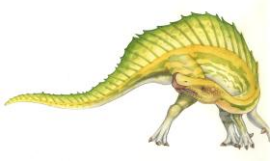
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation





Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms

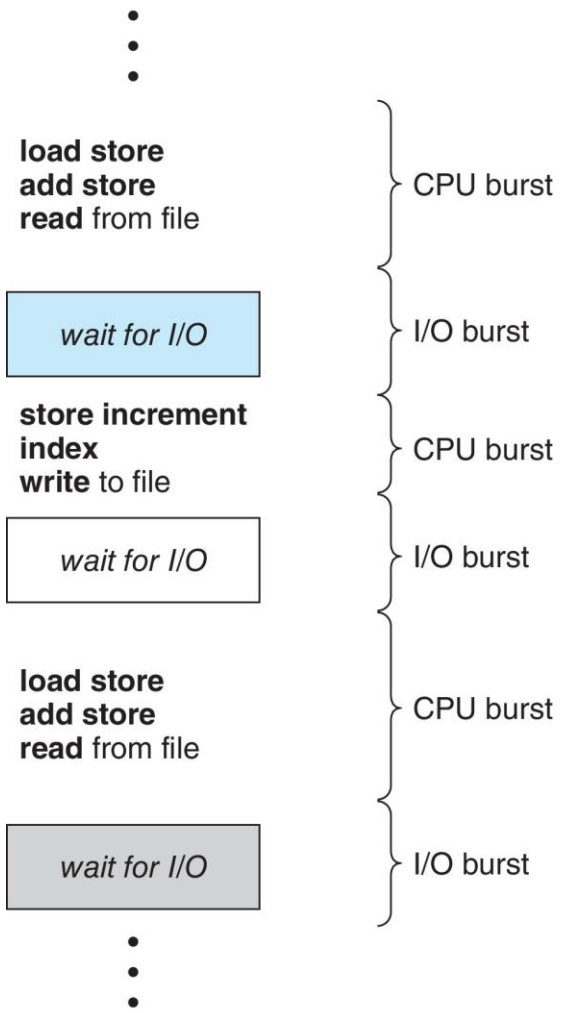


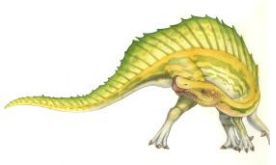
Troy University Resources



Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



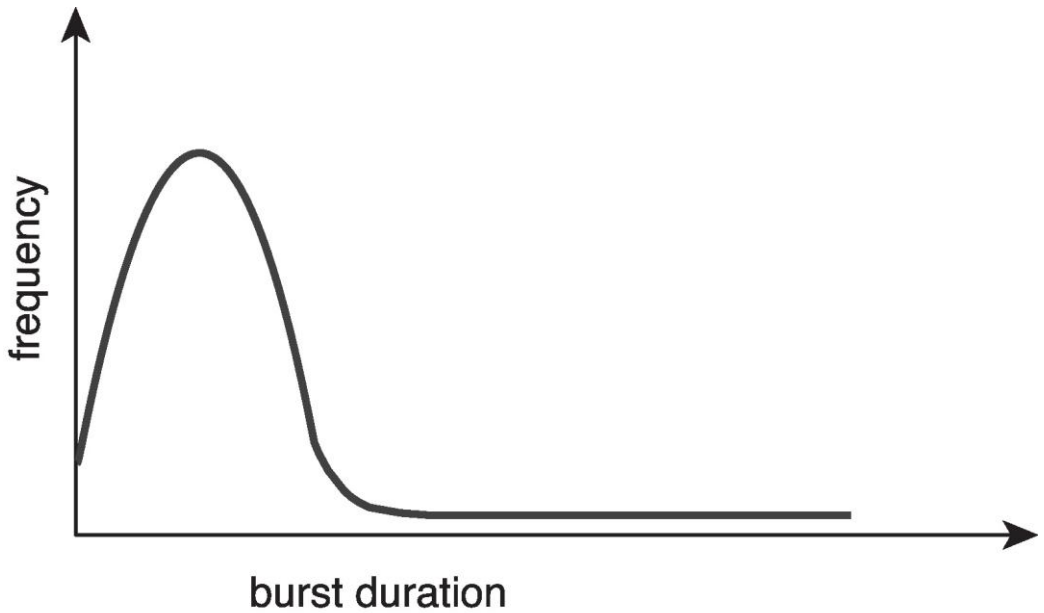




Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts



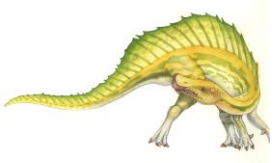


Troy University Resources



CPU Scheduler

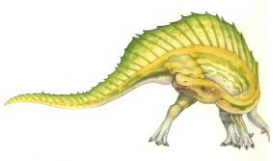
- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.





Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

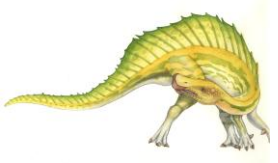


Troy University Resources



Preemptive Scheduling and Race Conditions

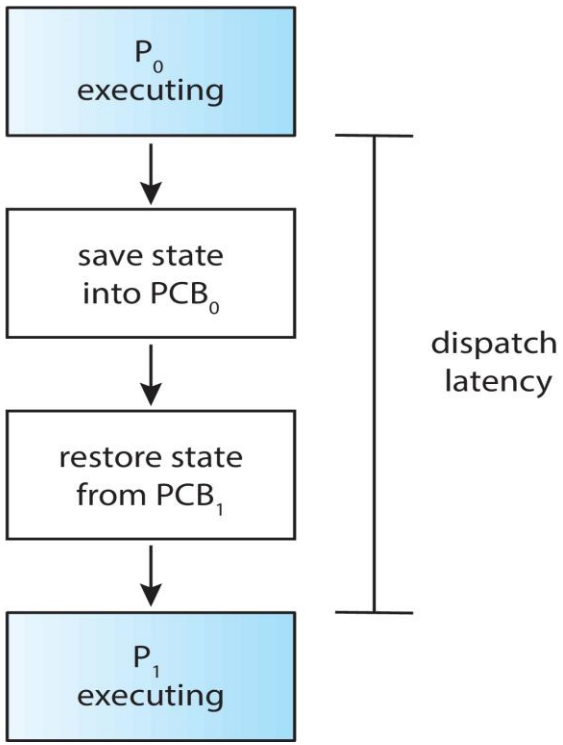
- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- This issue will be explored in detail in Chapter 6.





Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



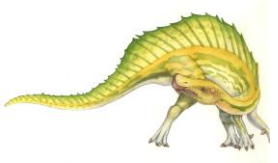


Troy University Resources



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time: elapsed time from arrival or submission to completion, including waiting and I/O**
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced.





Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time



Troy University Resources



First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17





FCFS Scheduling (Cont.)

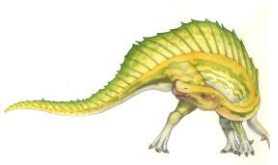
Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

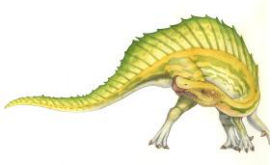


Troy University Resources



Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- For known bursts with all jobs available together, nonpreemptive SJF minimizes average waiting time (ignoring switching overhead)
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

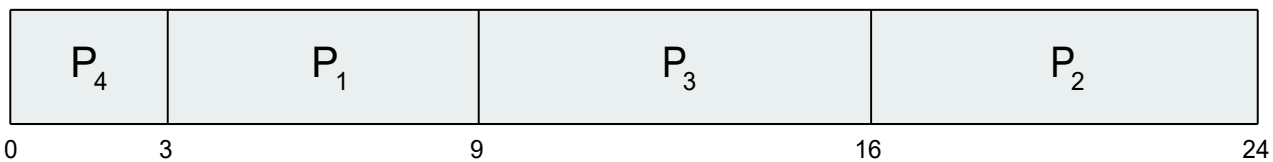




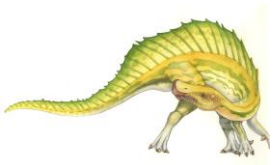
Example of SJF

Process	CPU burst
P1	6
P2	8
P3	7
P4	3

All processes arrive at time 0. CPU-switch overhead is ignored.



Average waiting time = (3 + 16 + 9 + 0) / 4 = 7



Troy University Resources



Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define:

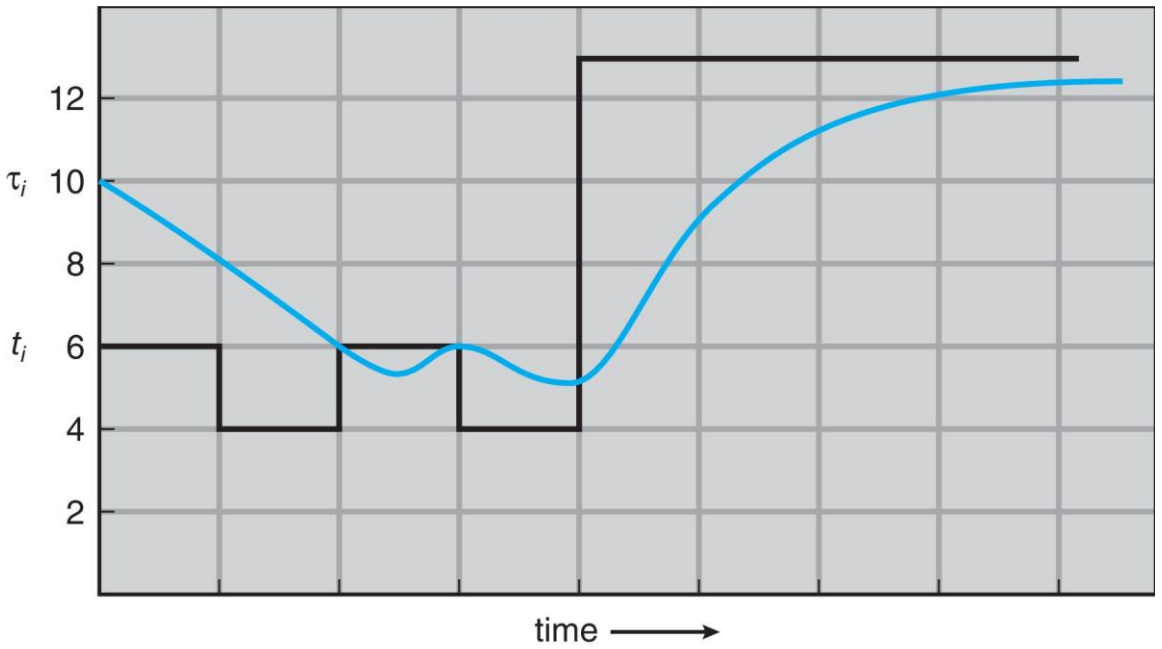
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

- Commonly, α set to ½

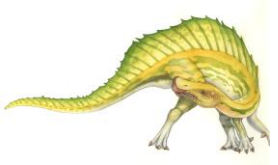




Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13 ...
"guess" (τ_i)	10	8	6	6	5	9	11	12 ...



Troy University Resources



Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

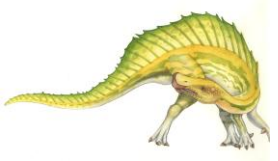
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successor predecessor term has less weight than its predecessor





Shortest Remaining Time First Scheduling

- Shortest Remaining Time First (SRTF) is the preemptive form of SJF.
- At each relevant scheduling event, compare the remaining CPU-burst times of the ready tasks.
- A newly ready task preempts the running task when its remaining burst is shorter.
- With known burst lengths on one CPU and negligible switching overhead, SRTF minimizes average completion time for this model.
- Practical schedulers must estimate burst lengths and account for preemption cost.



Troy University Resources

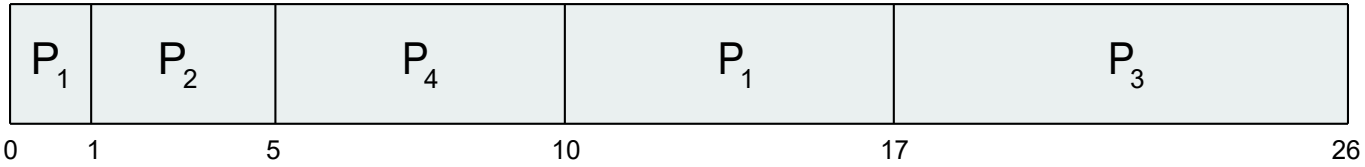


Example of Shortest Remaining Time First

Different arrival times; ignore preemption and switching overhead.

Process	Arrival time	CPU burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Preemptive SJF schedule:



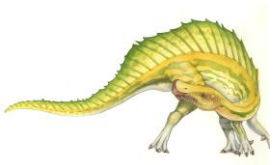
Average waiting time = (9 + 0 + 15 + 2) / 4 = 6.5





Round Robin (RR)

- Each ready task receives up to a time quantum q. On expiration, a still-runnable task joins the back of the ready queue.
- For n continuously runnable tasks on one CPU, ignoring overhead, each gets about 1/n of CPU time and waits at most (n - 1)q between turns.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO (FCFS)
  - *A smaller q improves responsiveness but increases switching overhead*
- Note that q must be large with respect to context switch, otherwise overhead is too high



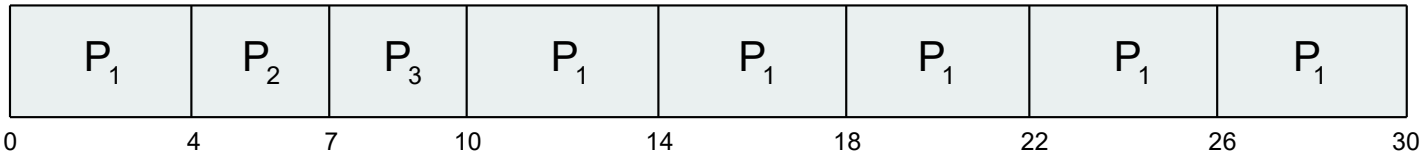
Troy University Resources



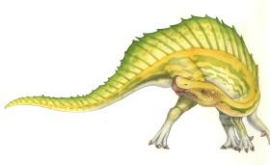
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

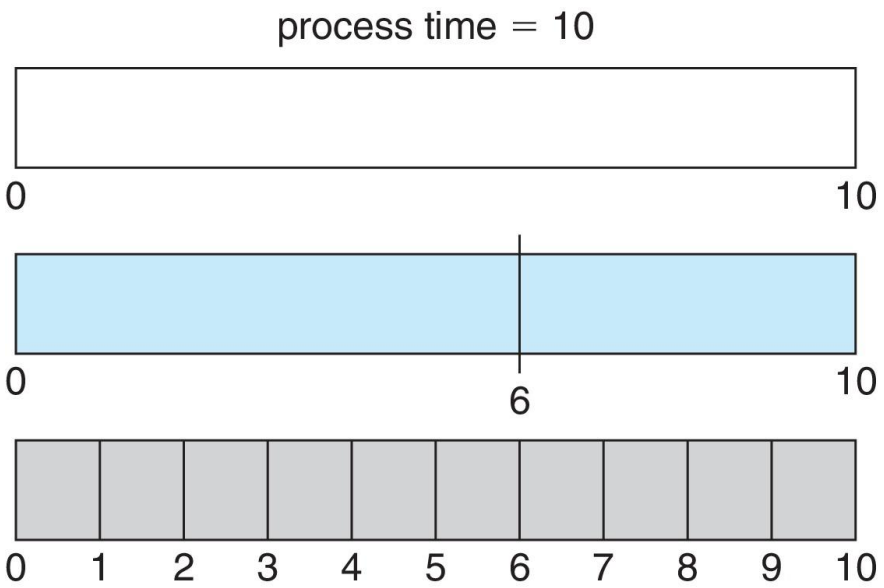


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - The example quantum is illustrative. Real values depend on the scheduler and workload.
  - Context-switch cost depends on hardware, OS state and cache effects





Time Quantum and Context Switch Time



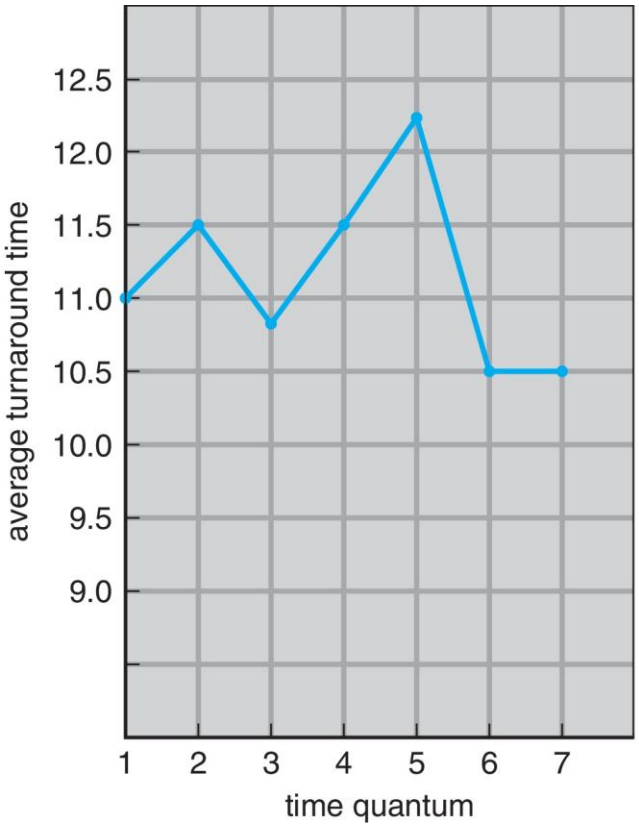
quantum	context switches
12	0
6	1
1	9



Troy University Resources



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

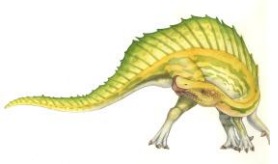
A common textbook heuristic chooses q longer than most CPU bursts. It is not a universal requirement.





Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process



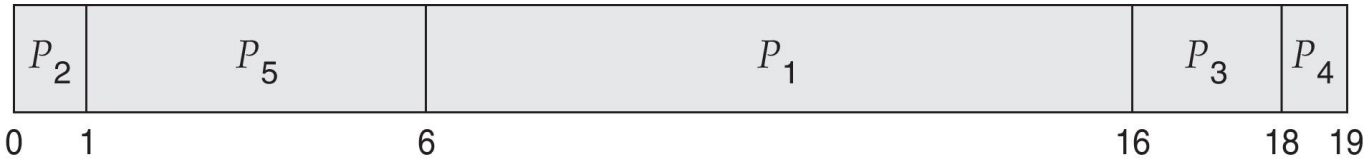
Troy University Resources



Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2



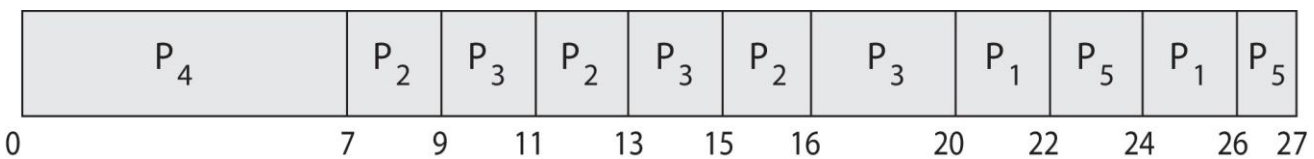


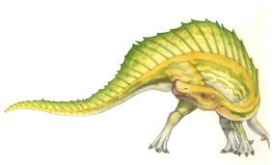
Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Gantt Chart with time quantum = 2



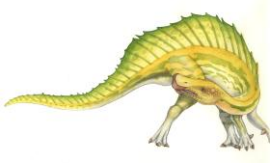


Troy University Resources



Multilevel Queue

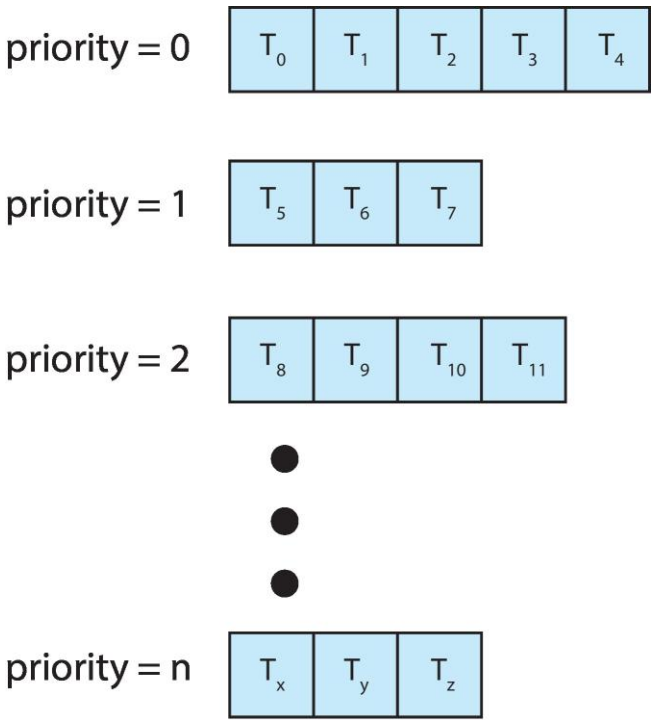
- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

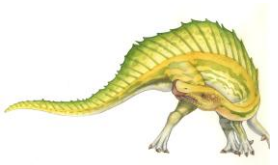




Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



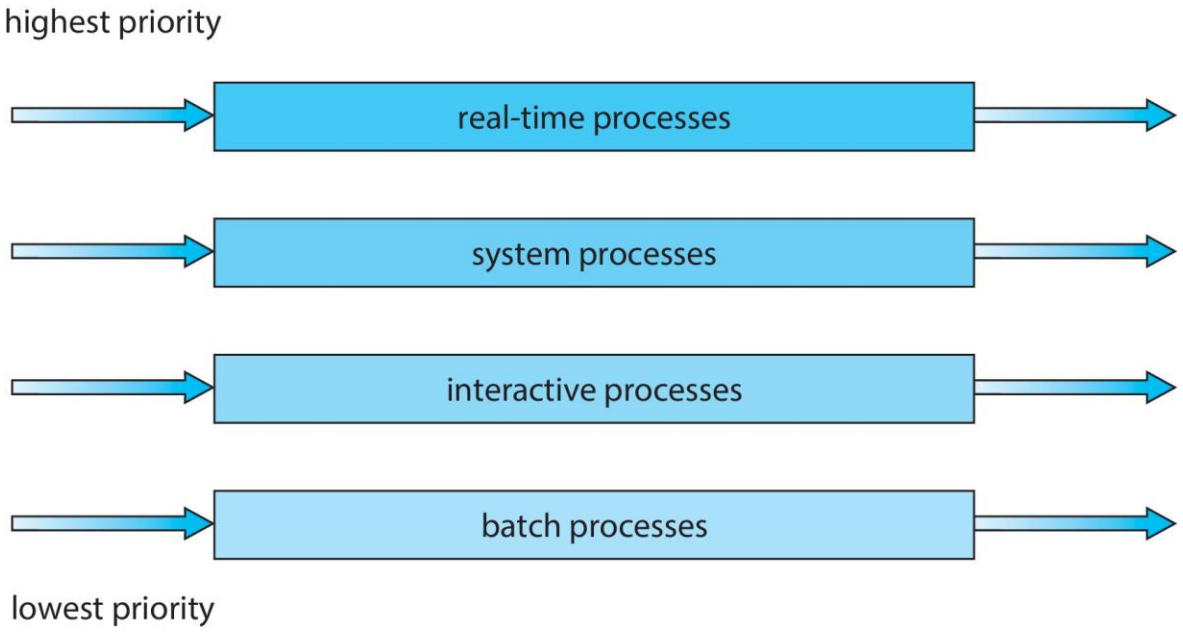


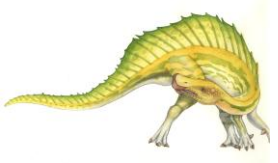
Troy University Resources



Multilevel Queue

- Prioritization based upon process type

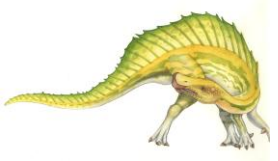






Scheduling Within and Between Queues

- In a multilevel queue model, tasks usually remain in their assigned class.
- Different queues may use different algorithms, for example RR for interactive work and FCFS for batch work.
- Fixed queue priority always serves the highest nonempty queue and can starve lower queues.
- A time-share policy can instead reserve a fraction of CPU service for each queue.
- For example, an illustrative 80/20 allocation shares service between foreground and batch queues.
- Multilevel feedback queues add movement between queues.

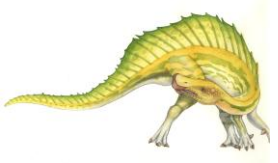


Troy University Resources



Multilevel Feedback Queue

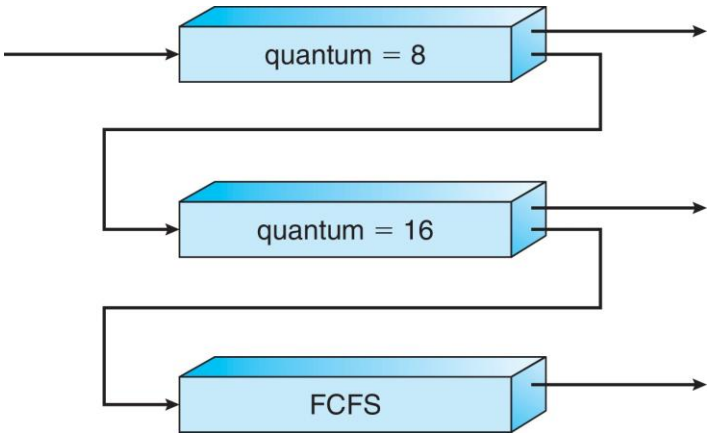
- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

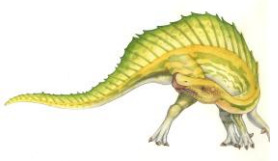




Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



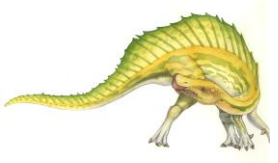


Troy University Resources



Thread Scheduling

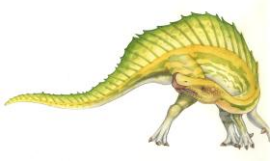
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system





Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM



Troy University Resources



Pthread Scheduling Scope: Setup

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM_THREADS 5

static void check(int e) {
    if (e) {
        fprintf(stderr, "%s\n", strerror(e));
        exit(EXIT_FAILURE);
    }
}
static void *runner(void *unused) {
    (void)unused;
    return NULL;
}
int main(void) {
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    int scope;
    check(pthread_attr_init(&attr));
```

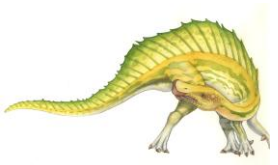




Pthread Scheduling Scope: Execution

```
    check(pthread_attr_getscope(&attr, &scope));
    printf("scope = %s\n",
        scope == PTHREAD_SCOPE_SYSTEM
        ? "SYSTEM" : "PROCESS");
    check(pthread_attr_setscope(&attr,
                                PTHREAD_SCOPE_SYSTEM));
    for (int i = 0; i < NUM_THREADS; ++i)
        check(pthread_create(&tid[i], &attr,
                             runner, NULL));
    check(pthread_attr_destroy(&attr));
    for (int i = 0; i < NUM_THREADS; ++i)
        check(pthread_join(tid[i], NULL));
    return 0;
}
```

This continues the previous slide. On Linux, PTHREAD_SCOPE_SYSTEM is supported. PROCESS scope is unsupported.

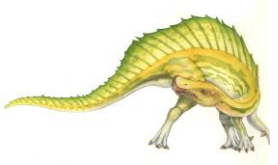


Troy University Resources



Multiple-Processor Scheduling

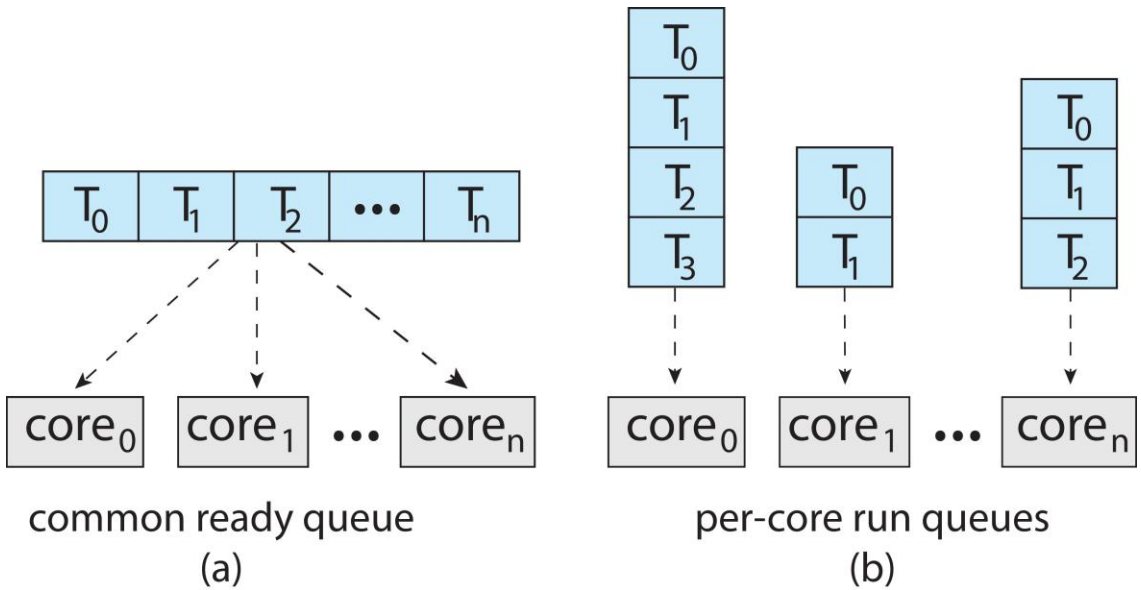
- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

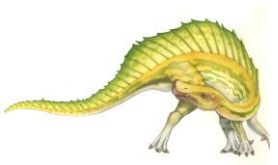




Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



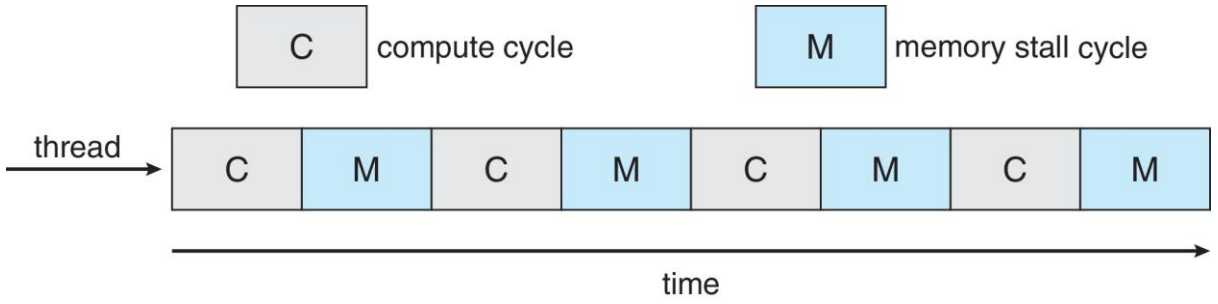


Troy University Resources



Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

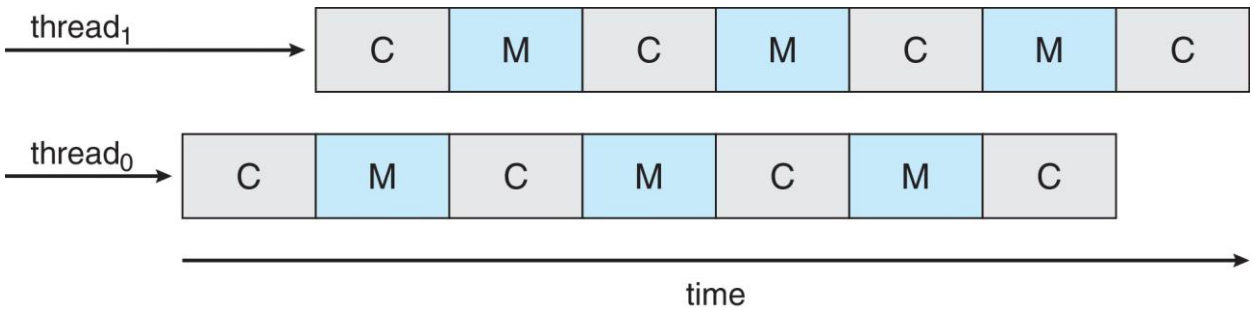


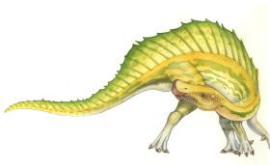




Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!



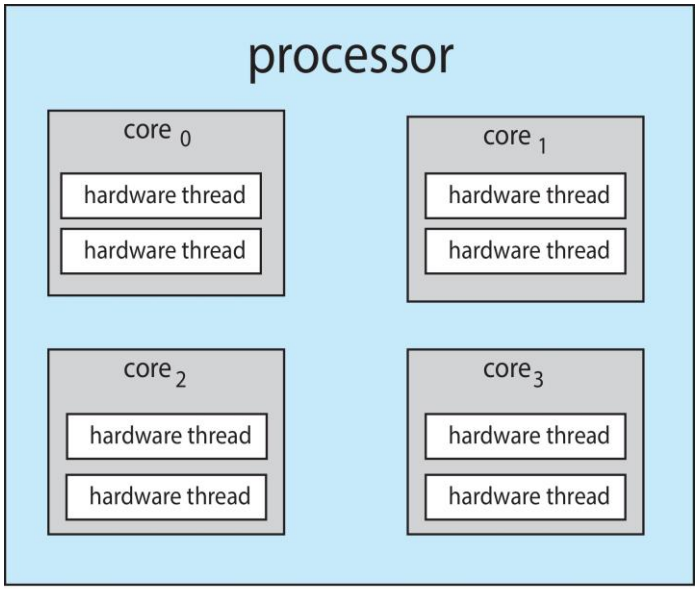


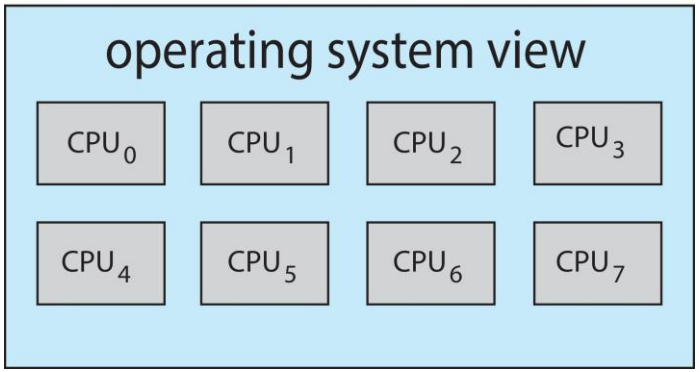
Troy University Resources

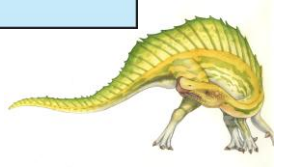


Multithreaded Multicore System

- **Hardware multithreading exposes multiple threads per core. Intel Hyper-Threading implements simultaneous multithreading (SMT).**
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



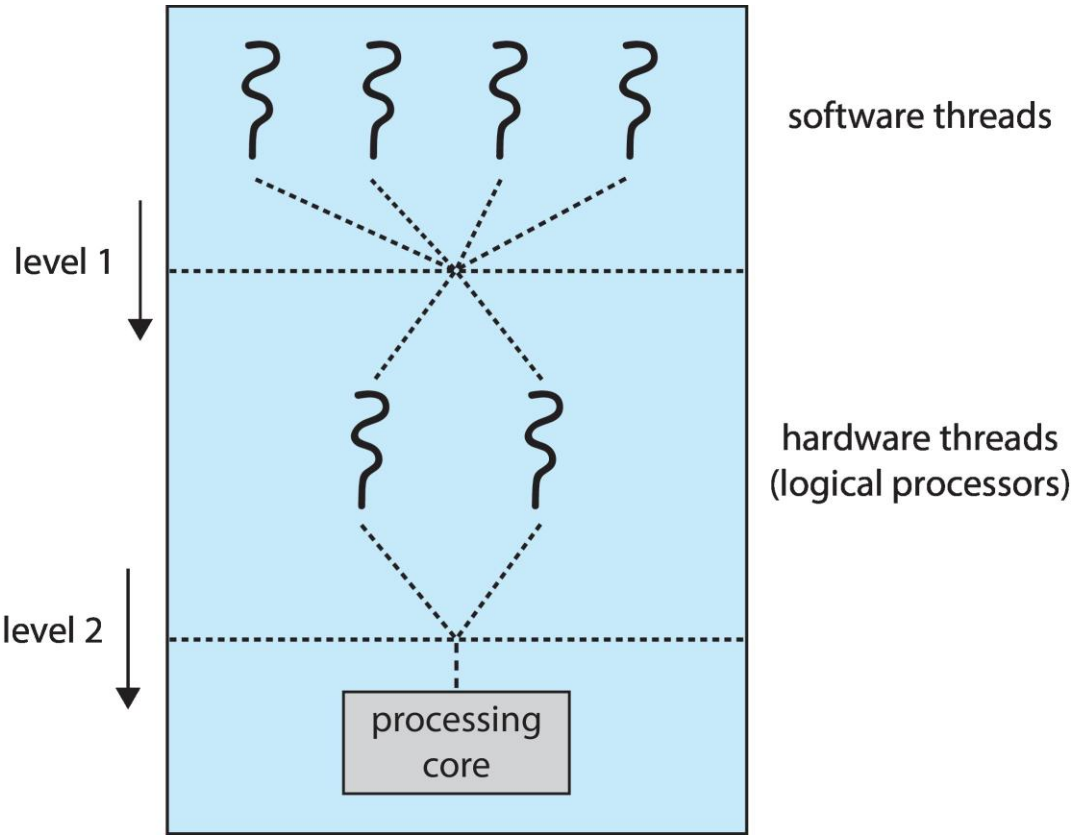


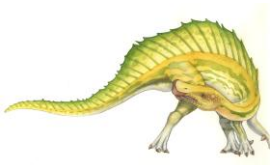




Multithreaded Multicore System

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

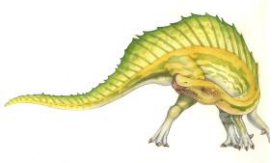






Multiple-Processor Scheduling – Load Balancing

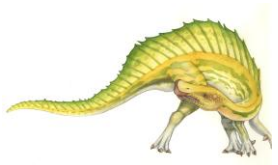
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- 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 of the processor it was moved off of.
- **Soft affinity** – the operating system 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.

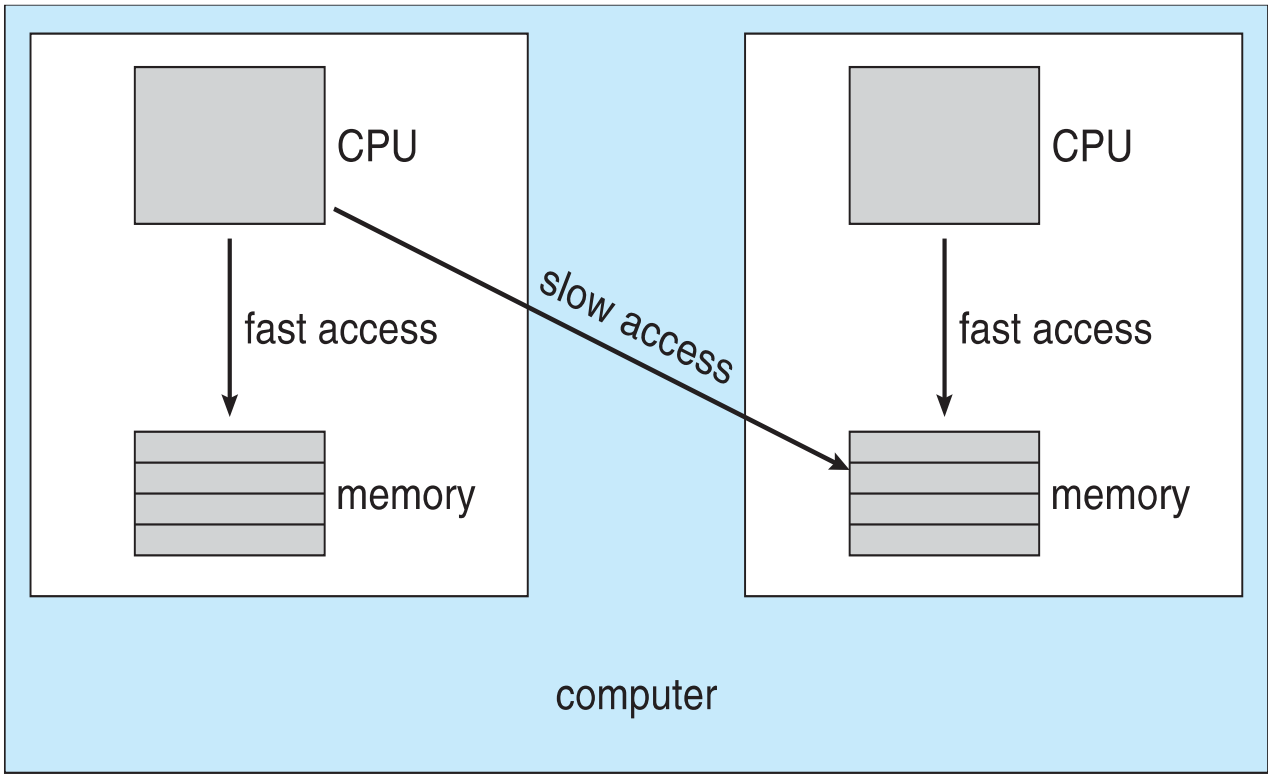


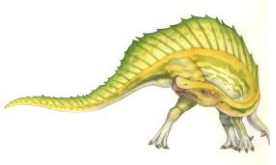
Troy University Resources



NUMA and CPU Scheduling

If the operating system is NUMA-aware, it will assign memory close to the CPU the thread is running on.

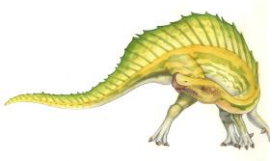






Virtualization and CPU Scheduling

- The hypervisor schedules virtual CPUs onto physical or logical CPUs.
- Each guest OS separately schedules its own tasks onto virtual CPUs.
- A task can be ready inside a guest while its virtual CPU waits for host CPU time.
- This double scheduling can increase latency and disrupt assumptions made by a guest scheduler.
- CPU overcommitment, affinity and virtual-clock support affect the resulting behavior.

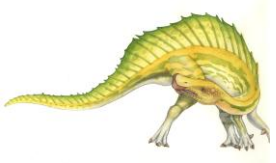


Troy University Resources



Real-Time CPU Scheduling

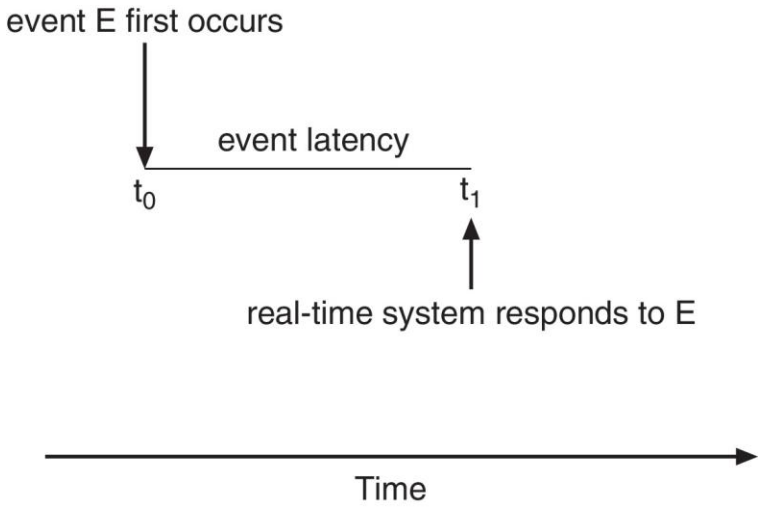
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**

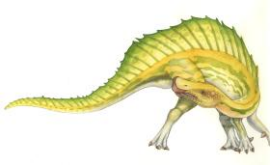




Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another

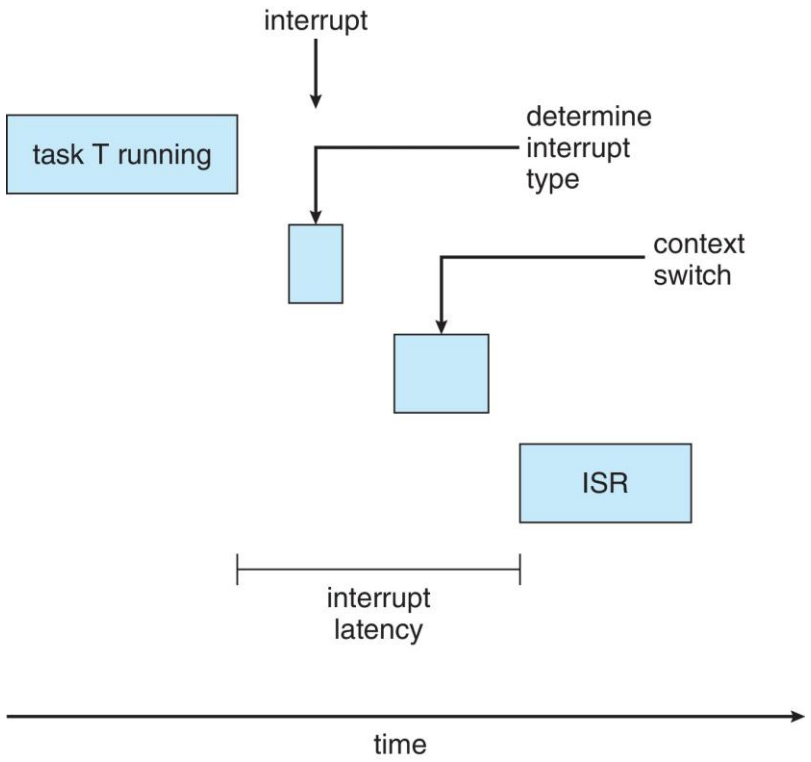


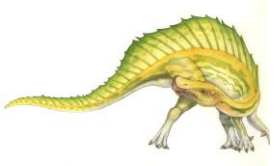


Troy University Resources



Interrupt Latency

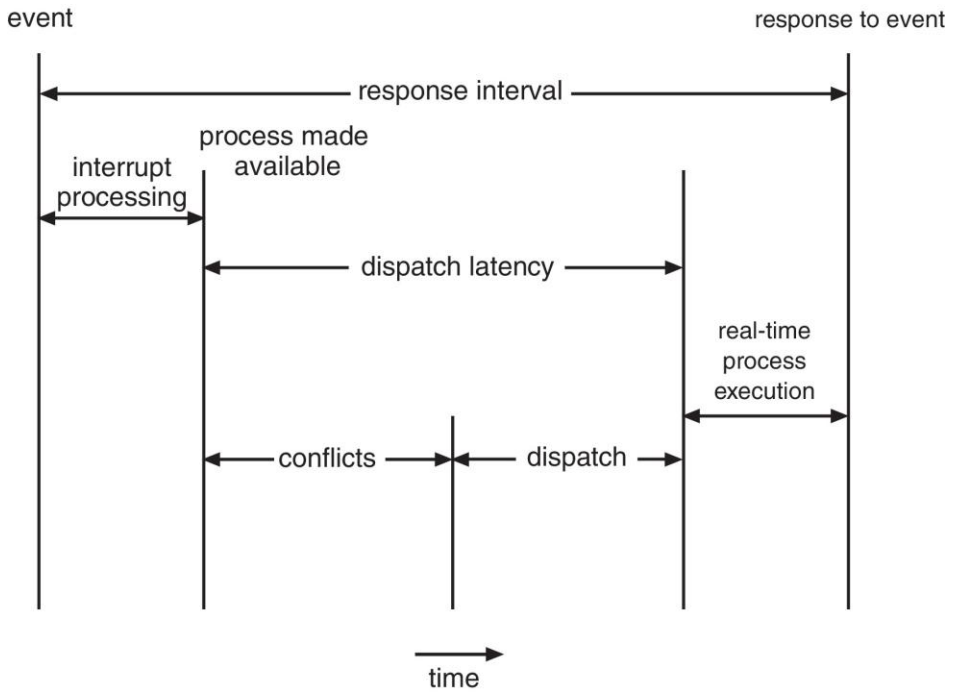






Dispatch Latency

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



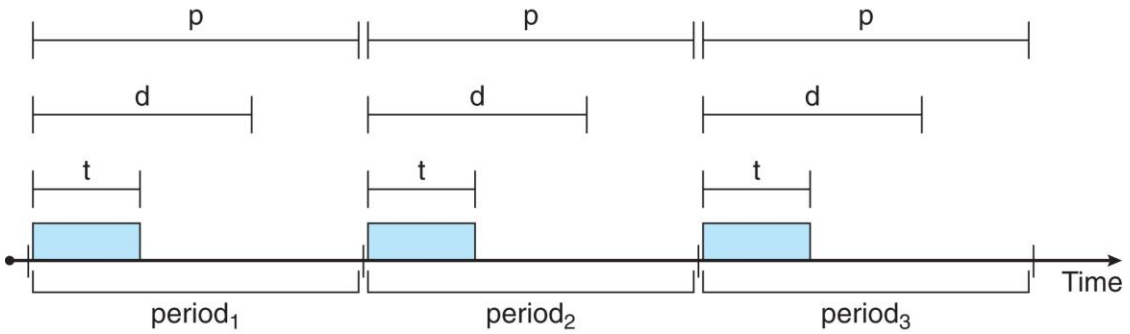


Troy University Resources



Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$

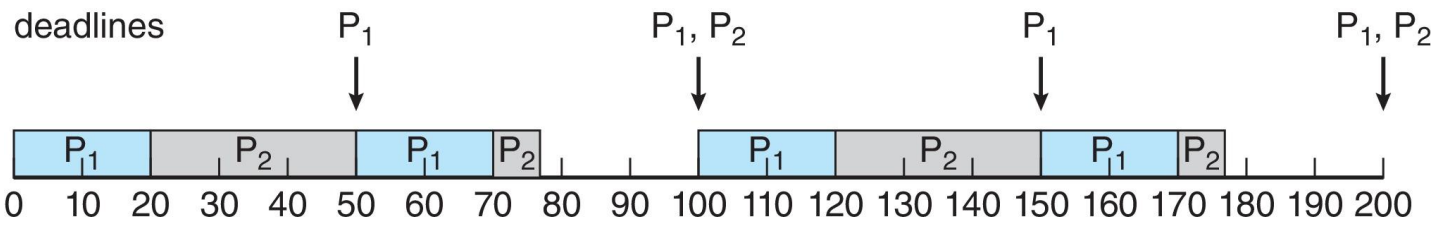




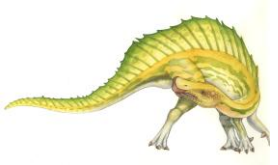


Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



Example: P1 has C = 20, T = D = 50. P2 has C = 35, T = D = 100.

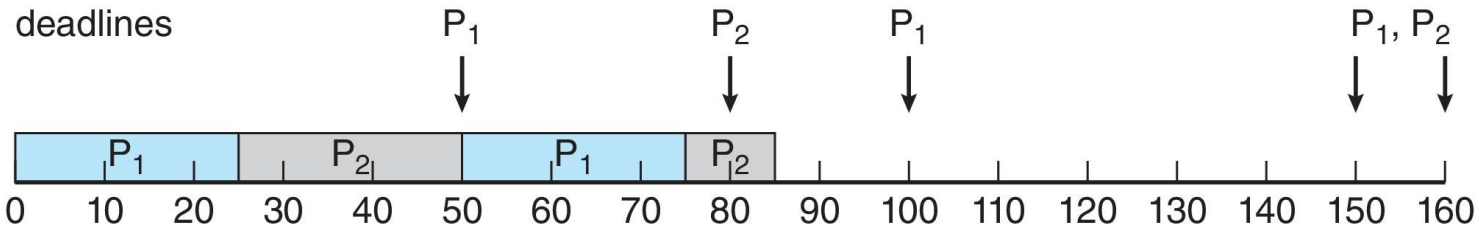


Troy University Resources



Missed Deadlines with Rate Monotonic Scheduling

- Process P_2 misses finishing its deadline at time 80



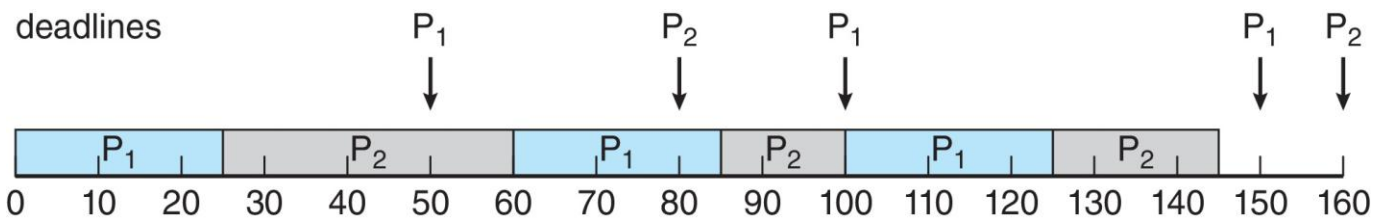
P1: C = 25, T = D = 50. P2: C = 35, T = D = 80. The first P2 job finishes at 85, after its deadline at 80.



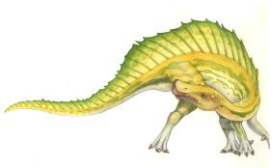


Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority



Same task set as the preceding slide. At time 50, P2 has deadline 80 and P1 has deadline 100, so EDF continues P2 until 60.

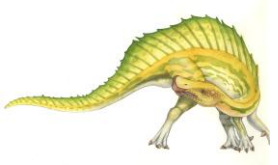


Troy University Resources



Proportional Share Scheduling

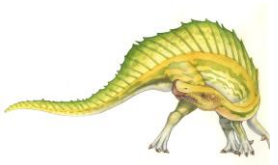
- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time





POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)
  2. pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)



Troy University Resources



POSIX Real-Time Thread: Setup

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void check(int e) {
    if (e) {
        fprintf(stderr, "%s\n", strerror(e));
        exit(EXIT_FAILURE);
    }
}
static void *runner(void *unused) {
    (void)unused;
    return NULL;
}
int main(void) {
    pthread_attr_t attr;
    pthread_t tid;
    check(pthread_attr_init(&attr));
```





POSIX Real-Time Thread: Policy

```
    struct sched_param sp = {0};
    sp.sched_priority = sched_get_priority_min(SCHED_FIFO);
    if (sp.sched_priority == -1) { perror("priority"); return 1; }
    check(pthread_attr_setinheritsched(&attr,
                                       PTHREAD_EXPLICIT_SCHED));
    check(pthread_attr_setschedpolicy(&attr, SCHED_FIFO));
    check(pthread_attr_setschedparam(&attr, &sp));
    check(pthread_create(&tid, &attr, runner, NULL));
    check(pthread_attr_destroy(&attr));
    check(pthread_join(tid, NULL));
    return 0;
}
```

PTHREAD_EXPLICIT_SCHED makes the requested attributes take effect.
Creating a FIFO thread may fail with EPERM without suitable permission.

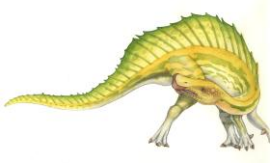


Troy University Resources



Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling





Historical Linux O(1) Scheduler

- Historical example: the O(1) scheduler developed during Linux 2.5 and used in early Linux 2.6.
- Normal tasks used internal priorities 100–139, derived from nice values -20 to +19.
- Lower internal priority numbers indicated higher priority.
- Normal runnable tasks occupied per-CPU active or expired priority arrays.
- After a task used its time slice, it moved to the expired array. When the active array emptied, the arrays exchanged roles.
- O(1) referred to task selection using bounded priority arrays. It did not mean every scheduling operation had constant cost.

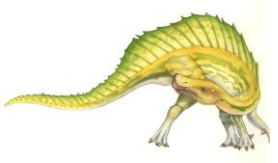


Troy University Resources



Linux Completely Fair Scheduler (CFS)

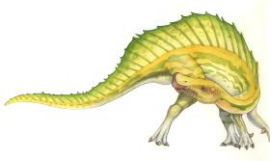
- CFS entered mainline Linux in version 2.6.23 for fair scheduling.
- The scheduler uses scheduling classes with different precedence.
- Fair tasks receive weighted proportions of CPU time rather than fixed quanta determined only by priority.
- Real-time policies and deadline scheduling use other scheduling classes.
- The following CFS slides describe the historical design. Linux began moving fair-task selection to EEVDF in version 6.6.





CFS Virtual Runtime

- Nice values range from -20 to +19. Lower nice values give greater scheduling weight.
- Each task tracks a weighted virtual runtime, vruntime.
- A task with lower weight accumulates virtual runtime faster for the same actual CPU time.
- At the reference weight, virtual runtime advances approximately with actual runtime.
- Classic CFS orders runnable entities in a red-black tree and normally selects the leftmost entity, subject to scheduling rules.
- Granularity settings reduce excessive switching. Virtual runtime is accounting, not a decaying priority.

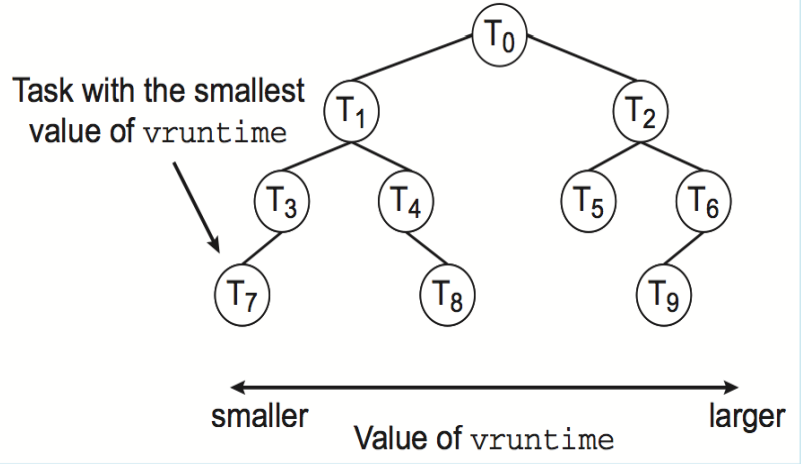


Troy University Resources

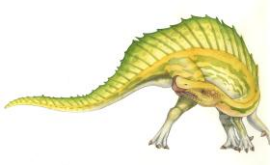


CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



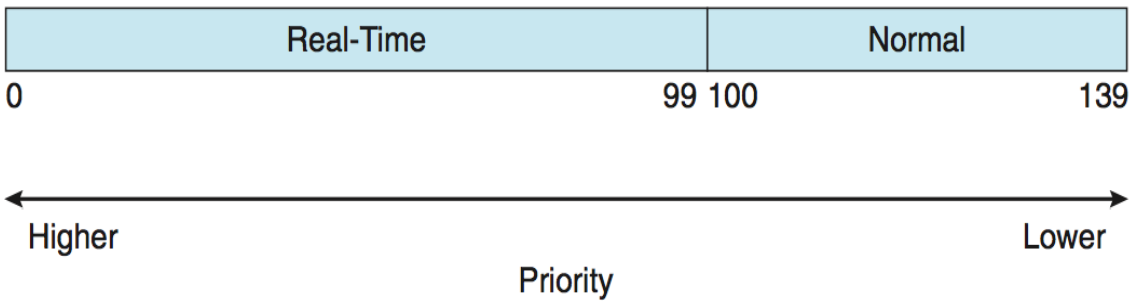
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.





Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



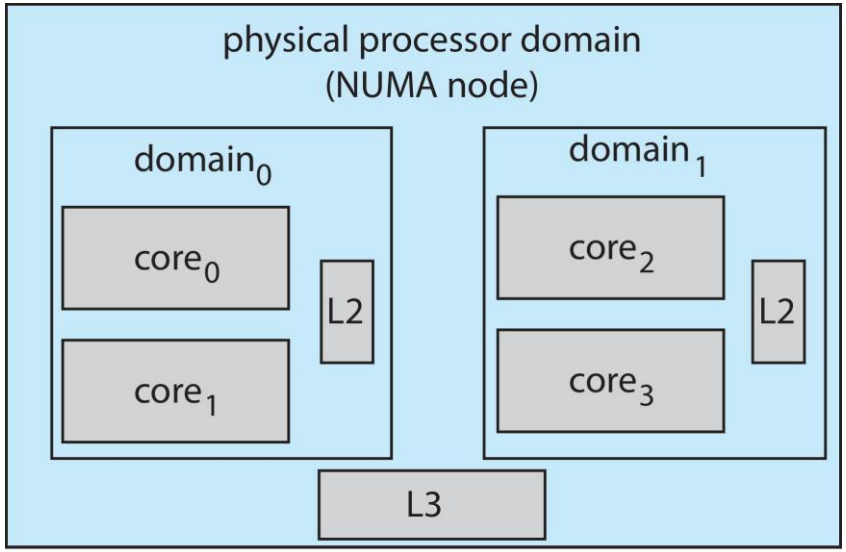


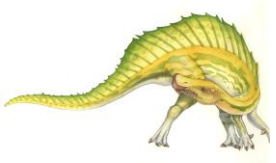
Troy University Resources



Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e., cache memory.) Goal is to keep threads from migrating between domains.

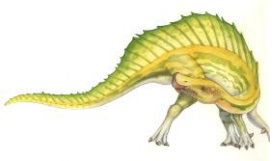






Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

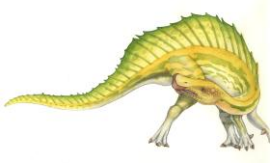


Troy University Resources



Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base





Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground preference may include a longer time quantum. This differs from multiplying a thread's numeric priority.
- Historical example: Windows 7 introduced UMS. Windows 11 does not support UMS.
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

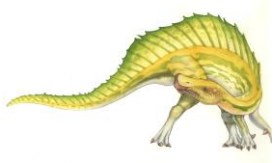


Troy University Resources



Windows Priorities

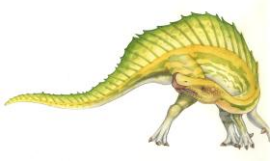
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1





Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin



Troy University Resources



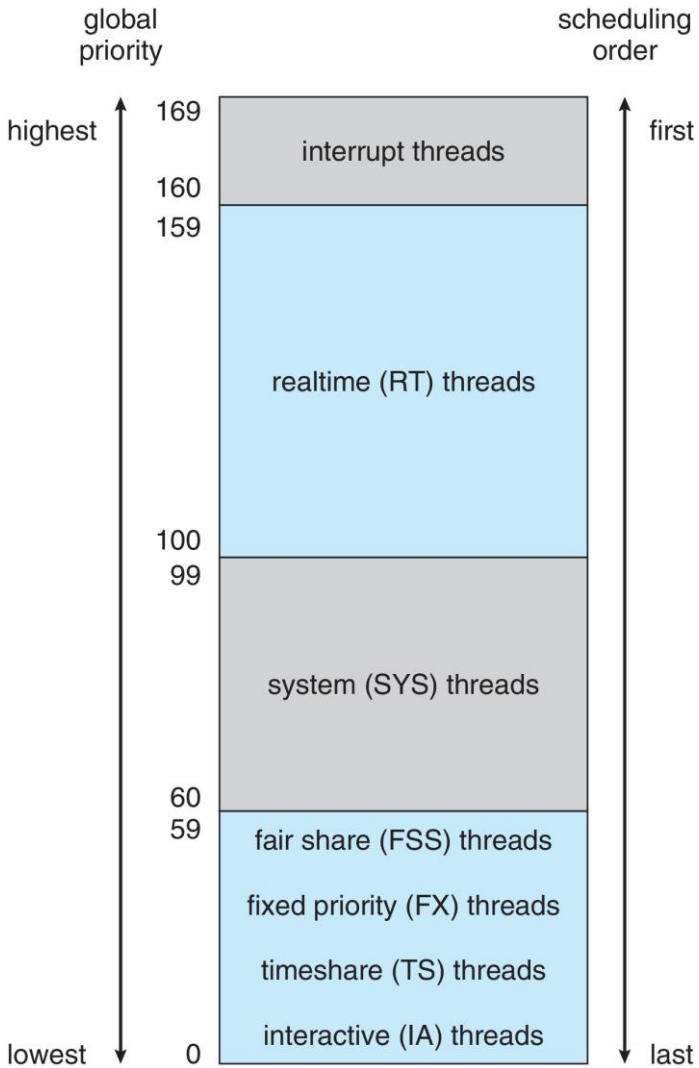
Solaris Dispatch Table

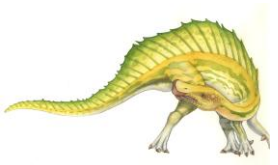
priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59





Solaris Scheduling



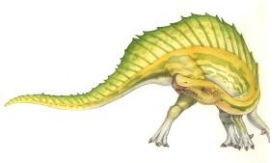


Troy University Resources



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR





Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

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



Troy University Resources



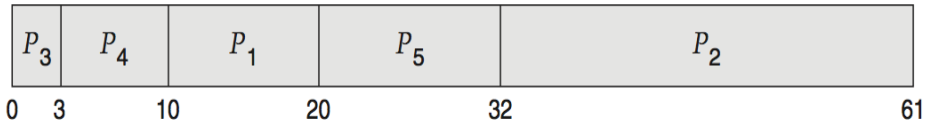
Deterministic Evaluation

All jobs arrive at time 0. Ignore switching overhead.

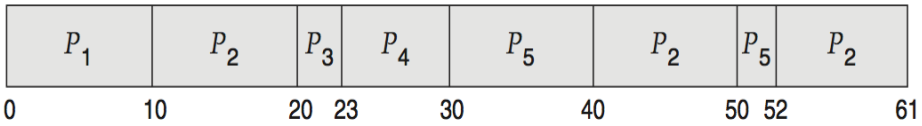
FCFS: average waiting time = 28 ms



Nonpreemptive SJF: average waiting time = 13 ms



RR with q = 10 ms: average waiting time = 23 ms







Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc.



Troy University Resources

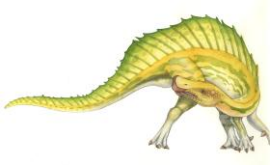


Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

  $n = \lambda \times W$

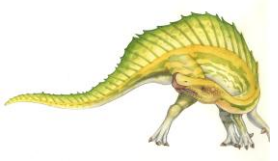
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds





Simulations

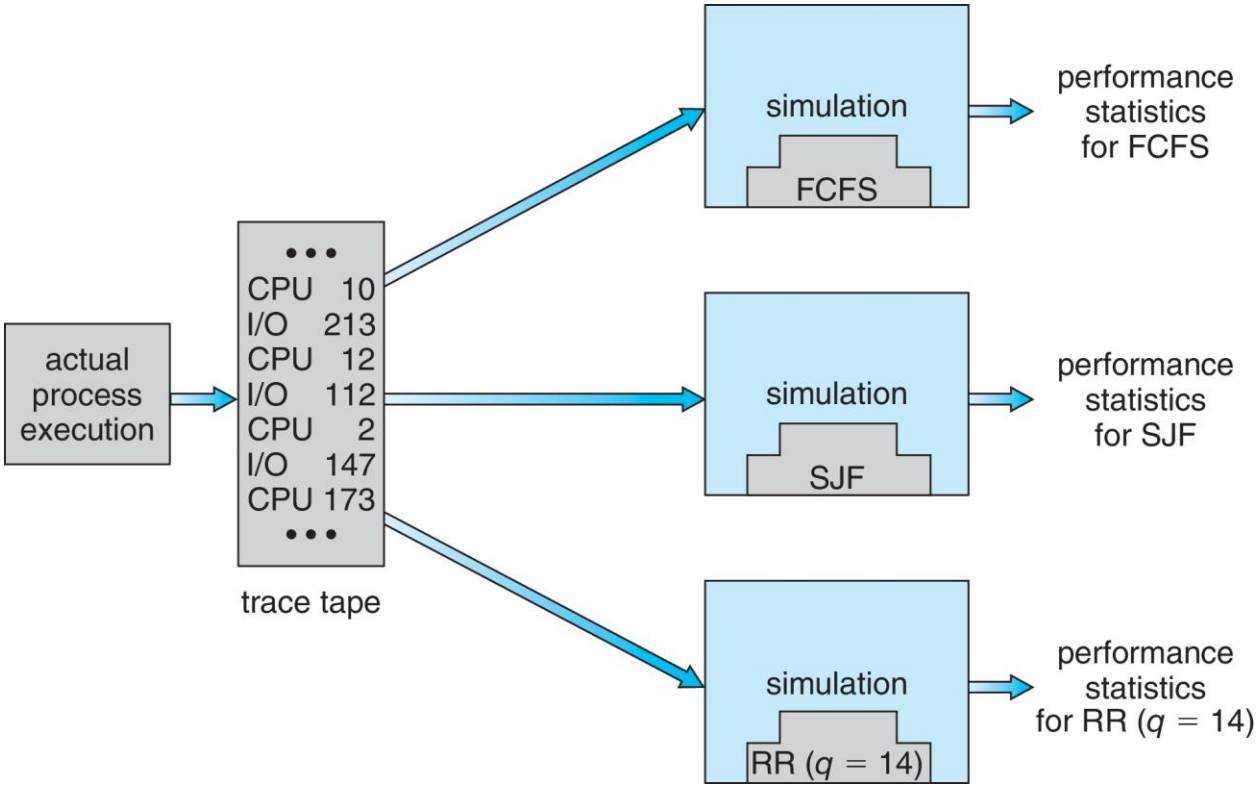
- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems



Troy University Resources



Evaluation of CPU Schedulers by Simulation

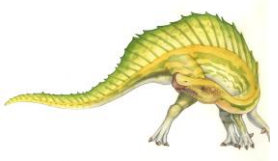






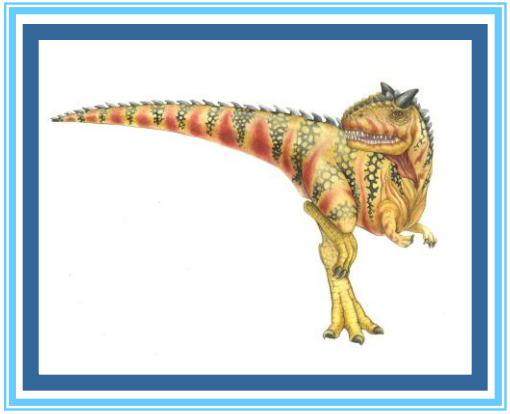
Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary



Troy University Resources

End of Chapter 5



Silberschatz, Galvin and Gagne ©2018