CPU Scheduling (Part II)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Motivation

Reminder

- CPU scheduling is the basis of multiprogrammed OSs.

- By switching the CPU among processes, the OS makes the computer more productive.

Basic Concepts

- In a single-processor system, only one process can run at a time.
- Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

Scheduling Criteria

- CPU utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

Process Scheduling Algorithms

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Thread Scheduling

Thread Scheduling (1/2)

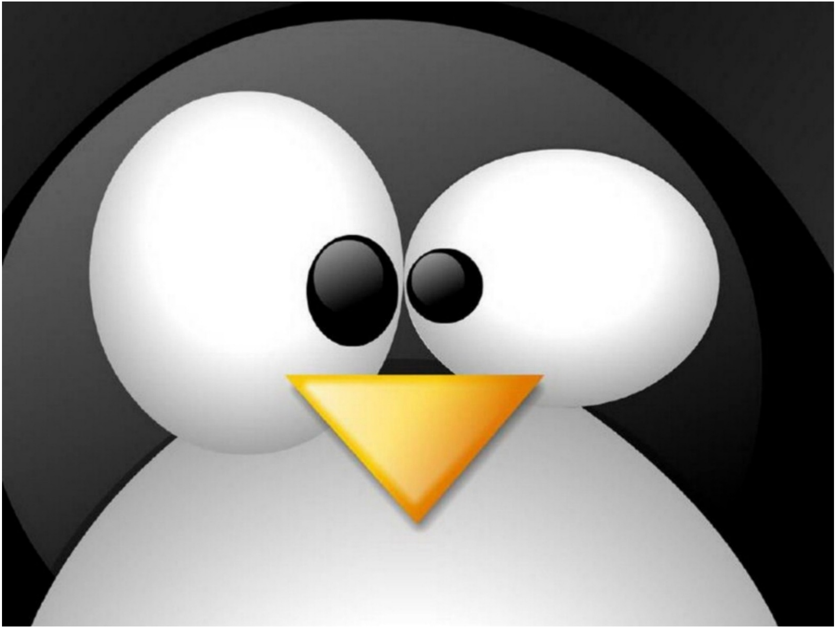
- Distinction between user-level and kernel-level threads.
- When threads supported by the OS, threads scheduled, not processes.

Thread Scheduling (2/2)

- Process-Contention Scope (PCS)
  - In many-to-one and many-to-many models
  - Scheduling competition is within the process.

Thread Scheduling (2/2)

- Process-Contention Scope (PCS)
  - In many-to-one and many-to-many models
  - Scheduling competition is within the process.
- System-Contention Scope (SCS)
  - In one-to-one model.
  - Scheduling 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.

Contention Scope

- pthread_attr_setscope and pthread_attr_getscope set/get contention scope attribute in thread attributes object.

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

Pthread Scheduling API (1/2)

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

int main(int argc, char *argv[]) {
  int i, scope;
  pthread_t tid[NUM THREADS];
  pthread_attr_t attr;

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

  /* first inquire on the current scope */
  if (pthread_attr_getscope(&attr, &scope) != 0)
    fprintf(stderr, "Unable to get scheduling scope\n");
  else {
    if (scope == PTHREAD_SCOPE_PROCESS)
      printf("PTHREAD_SCOPE_PROCESS");
    else if (scope == PTHREAD_SCOPE_SYSTEM)
      printf("PTHREAD_SCOPE_SYSTEM");
    else
      fprintf(stderr, "Illegal scope value.\n");
  }
```

Pthread Scheduling API (2/2)

```
  /* set the scheduling algorithm to PCS or SCS */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* 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);
}
```

Multi-Processor Scheduling

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.

- Asymmetric multiprocessing
  - Only one processor does all scheduling decisions, I/O processing, and other system activities.
  - The other processors execute only user code.

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.

- Asymmetric multiprocessing
  - Only one processor does all scheduling decisions, I/O processing, and other system activities.
  - The other processors execute only user code.

- Symmetric multiprocessing (SMP)
  - Each processor is self-scheduling
  - All processes in common ready queue, or each has its own private queue of ready processes.
  - Currently, the most common.

Processor Affinity

- What happens to cache memory when a process has been running on a specific processor, and then, the process migrates to another processor?

Processor Affinity

- What happens to cache memory when a process has been running on a specific processor, and then, the process migrates to another processor?

- Invalidating and repopulating caches is costly.

Processor Affinity

- What happens to cache memory when a process has been running on a specific processor, and then, the process migrates to another processor?

- Invalidating and repopulating caches is costly.

- Processor affinity: keep a process running on the same processor.

Processor Affinity

- What happens to cache memory when a process has been running on a specific processor, and then, the process migrates to another processor?

- Invalidating and repopulating caches is costly.

- Processor affinity: keep a process running on the same processor.
  - Soft affinity: the OS attempts to keep a process on a single processor, but it is possible for a process to migrate between processors.

Processor Affinity

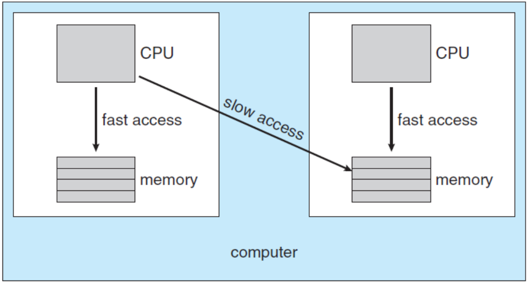
- What happens to cache memory when a process has been running on a specific processor, and then, the process migrates to another processor?

- Invalidating and repopulating caches is costly.

- Processor affinity: keep a process running on the same processor.
  - Soft affinity: the OS attempts to keep a process on a single processor, but it is possible for a process to migrate between processors.
  - Hard affinity: allowing a process to specify a subset of processors on which it may run.

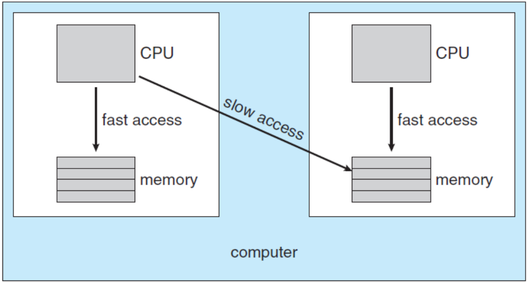
NUMA and CPU Scheduling

- Non-Uniform Memory Access (NUMA): a CPU has faster access to some parts of main memory than to other parts.



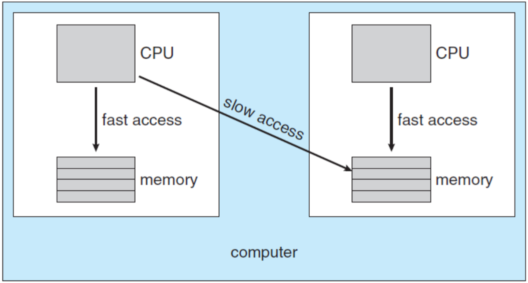
NUMA and CPU Scheduling

- Non-Uniform Memory Access (NUMA): a CPU has faster access to some parts of main memory than to other parts.
- Systems containing combined CPU and memory boards.



NUMA and CPU Scheduling

- Non-Uniform Memory Access (NUMA): a CPU has faster access to some parts of main memory than to other parts.
- Systems containing combined CPU and memory boards.
- A process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides.



Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency.

Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency.
- Load balancing attempts to keep workload evenly distributed.

Load Balancing

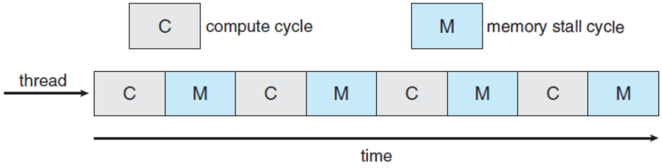
- If SMP, need to keep all CPUs loaded for efficiency.
- Load balancing attempts to keep workload evenly distributed.
- Push migration: periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs.

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.

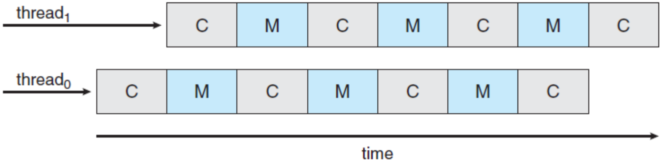
Multicore Processors

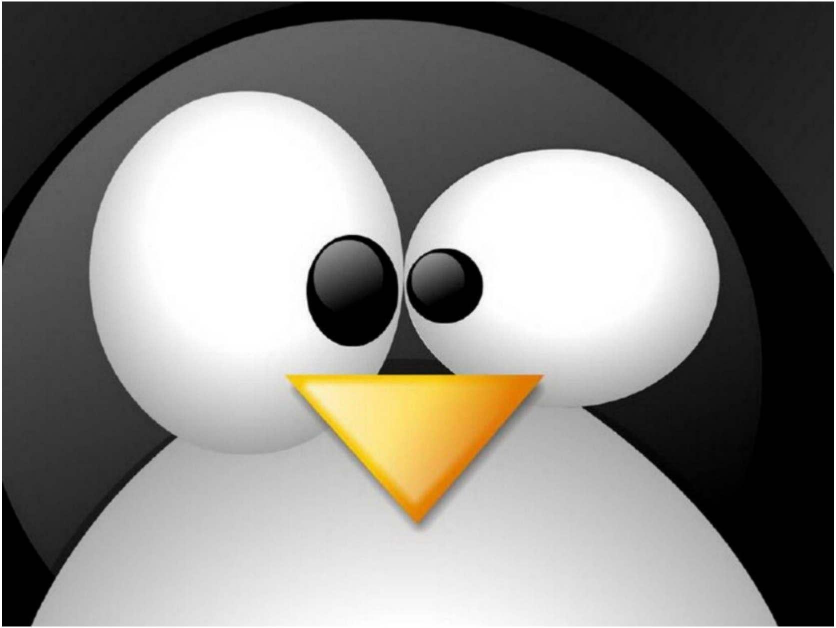
- Place multiple processor cores on same physical chip.
- Faster and consumes less power.
- Memory stall: when a processor accesses memory, it spends a significant amount of time waiting for the data to become available.



Multithreaded Multicore System

- Multiple threads per core also growing.
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens.





CPU Affinity

- sched_setaffinity() and sched_getaffinity() sets/gets the CPU affinity of the process specified by pid.

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t len, cpu_set_t *set);

int sched_getaffinity(pid_t pid, size_t len, cpu_set_t *set);
```

CPU Affinity Macros

- CPU_ZERO() initializes set to be empty.
- CPU_SET() adds the CPU cpu to set.
- CPU_CLR() removes the CPU cpu from set.
- CPU_ISSET() returns true if the CPU cpu is a member of set.

```
#define _GNU_SOURCE
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);
int CPU_ISSET(int cpu, cpu_set_t *set);
```

CPU Affinity Macros

- The process identified by `pid` runs on any CPU other than the first CPU of a four-processor system.

```
cpu_set_t set;

CPU_ZERO(&set);
CPU_SET(1, &set);
CPU_SET(2, &set);
CPU_SET(3, &set);

sched_setaffinity(pid, sizeof(set), &set);
```

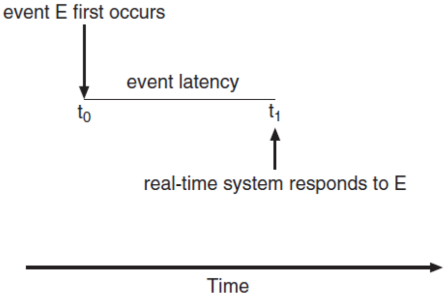
Real-Time CPU Scheduling

Minimizing Latency

- When an event occurs, the system must respond to and service it as quickly as possible.

Minimizing Latency

- When an event occurs, the system must respond to and service it as quickly as possible.
- Event latency: the amount of time that elapses from when an event occurs to when it is serviced.



Soft vs. Hard Real-Time

- Soft real-time
  - No guarantee as to when a critical real-time process will be scheduled.
  - They guarantee only that the process will be given preference over noncritical processes.

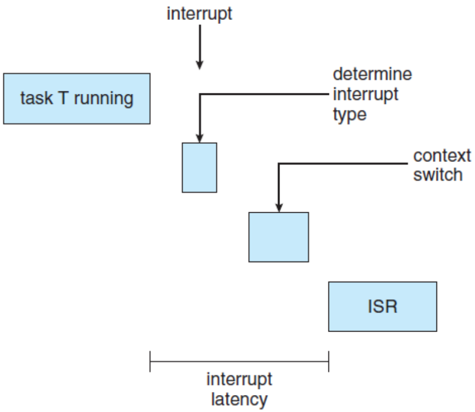
Soft vs. Hard Real-Time

- Soft real-time
  - No guarantee as to when a critical real-time process will be scheduled.
  - They guarantee only that the process will be given preference over noncritical processes.

- Hard real-time
  - The task must be serviced by its deadline.

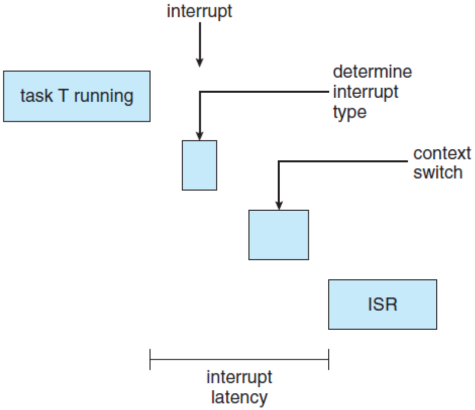
Real-Time CPU Scheduling (1/2)

- Two types of latencies affect performance:
  1. Interrupt latency: time from arrival of interrupt to start of routine that services interrupt.



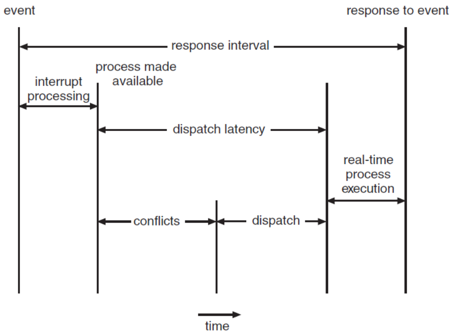
Real-Time CPU Scheduling (1/2)

- 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.



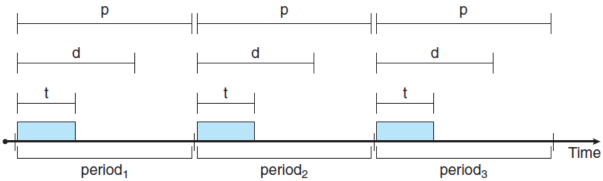
Real-Time CPU Scheduling (2/2)

- 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.



Periodic Processes

- Periodic processes require CPU at constant intervals.
  - Processing time t, deadline d, period p
  - $0 \leq t \leq d \leq p$
  - Rate of periodic task is $\frac{1}{p}$



Priority-Based Scheduling

Priority-Based Scheduling

- For real-time scheduling, scheduler must support preemptive and priority-based scheduling.

Priority-Based Scheduling

- For real-time scheduling, scheduler must support preemptive and priority-based scheduling.

- Only guarantees soft real-time.

Priority-Based Scheduling

- For real-time scheduling, scheduler must support preemptive and priority-based scheduling.
- Only guarantees soft real-time.
- For hard real-time must also provide ability to meet deadlines.

Rate-Monotonic Scheduling

Rate-Monotonic Scheduling

- A priority is assigned based on the inverse of its period.

Rate-Monotonic Scheduling

- A priority is assigned based on the inverse of its period.
- Shorter periods = higher priority
- Longer periods = lower priority

Rate-Monotonic Scheduling

- A priority is assigned based on the inverse of its period.
- Shorter periods = higher priority
- Longer periods = lower priority
- Assign a higher priority to tasks that require the CPU more often.

Rate-Monotonic Example 1 (1/4)

- Two processes: P_1 and P_2
- $p_1 = 50$ and $p_2 = 100$
- $t_1 = 20$ and $t_2 = 35$
- $d_1 = d_2 =$ complete its CPU burst by the start of its next period

Rate-Monotonic Example 1 (2/4)

- Is it possible to schedule these tasks so that each meets its deadlines?

Rate-Monotonic Example 1 (2/4)

- Is it possible to schedule these tasks so that each meets its deadlines?
- Measure the CPU utilization: $\frac{t_i}{p_i}$

Rate-Monotonic Example 1 (2/4)

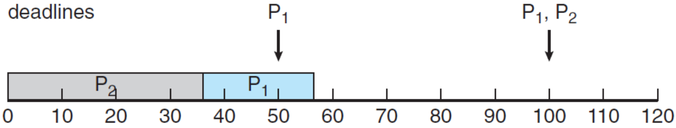
- Is it possible to schedule these tasks so that each meets its deadlines?
- Measure the CPU utilization: $\frac{t_i}{p_i}$
- $\frac{t_1}{p_1} = \frac{20}{50} = 0.4$
- $\frac{t_2}{p_2} = \frac{35}{100} = 0.35$

Rate-Monotonic Example 1 (2/4)

- Is it possible to schedule these tasks so that each meets its deadlines?
- Measure the CPU utilization: $\frac{t_i}{p_i}$
- $\frac{t_1}{p_1} = \frac{20}{50} = 0.4$
- $\frac{t_2}{p_2} = \frac{35}{100} = 0.35$
- $40\% + 35\% < 100\%$

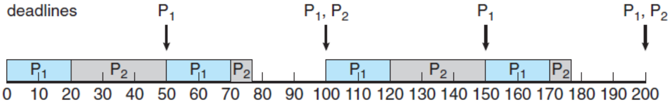
Rate-Monotonic Example 1 (3/4)

- Suppose we assign P_2 a higher priority than P_1.
  - P_1 misses its deadline.



Rate-Monotonic Example 1 (4/4)

- Suppose we assign P_1 a higher priority than P_2.



Rate-Monotonic Example 2 (1/3)

- Two processes: P_1 and P_2
- $p_1 = 50$ and $p_2 = 80$
- $t_1 = 25$ and $t_2 = 35$
- $d_1 = d_2 =$ complete its CPU burst by the start of its next period

Rate-Monotonic Example 2 (2/3)

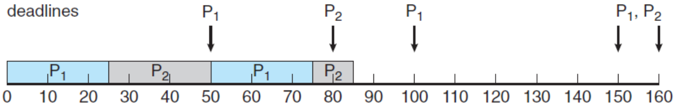
- Is it possible to schedule these tasks so that each meets its deadlines?

Rate-Monotonic Example 2 (2/3)

- Is it possible to schedule these tasks so that each meets its deadlines?
- $\frac{t_1}{p_1} = \frac{25}{50} = 0.5$
- $\frac{t_2}{p_2} = \frac{35}{80} = 0.44$
- $50\% + 44\% < 100\%$

Rate-Monotonic Example 2 (3/3)

- Suppose we assign P_1 a higher priority than P_2.
  - P_2 misses its deadline.



Earliest-Deadline-First (EDF) Scheduling

Earliest Deadline First Scheduling

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

EDF Example (1/3)

- Two processes: P_1 and P_2
- $p_1 = 50$ and $p_2 = 80$
- $t_1 = 25$ and $t_2 = 35$
- $d_1 = d_2 =$ complete its CPU burst by the start of its next period

EDF Example (2/3)

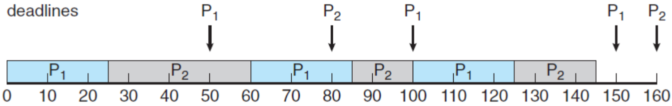
- Is it possible to schedule these tasks so that each meets its deadlines?

EDF Example (2/3)

- Is it possible to schedule these tasks so that each meets its deadlines?
- $\frac{t_1}{p_1} = \frac{25}{50} = 0.5$
- $\frac{t_2}{p_2} = \frac{35}{80} = 0.44$
- $50\% + 44\% < 100\%$

EDF Example (3/3)

- Suppose we assign P_1 a higher priority than P_2.



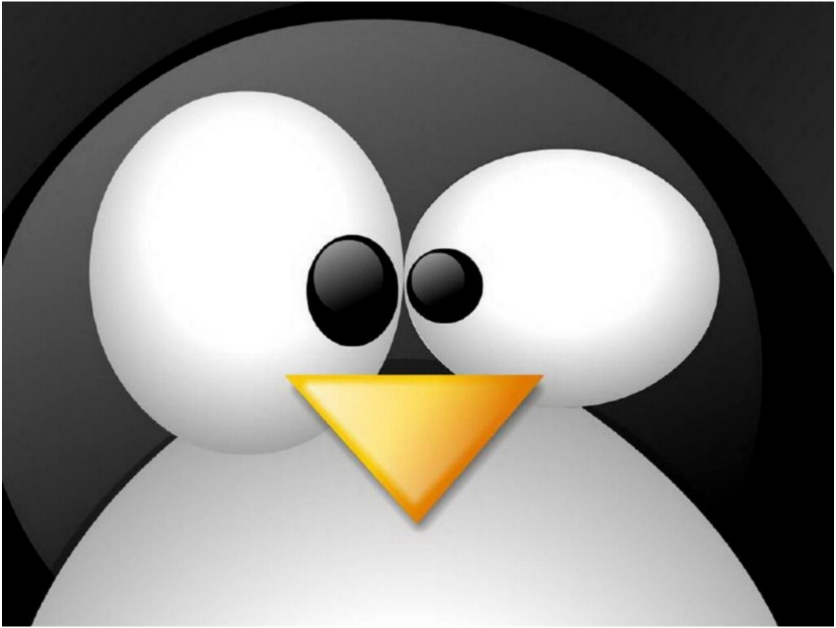
Proportional Share Scheduling

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 $\frac{N}{T}$ of the total processor time.

Proportional Share Example

- Three processes: P_1, P_2, and P_3
- $T = 100$
- P_1 is assigned 50, P_2 is assigned 15, and P_3 is assigned 20.
- P_1 will have 50% of total processor time, P_2 will have 15%, and P_3 will have 20%.



POSIX Real-Time Scheduling

- API provides functions for managing real-time threads/processes:

POSIX Real-Time Scheduling

- API provides functions for managing real-time threads/processes:
- Defines two scheduling classes for real-time threads:
  1. `SCHED_FIFO`: scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads/processes of equal priority.
  2. `SCHED_RR`: similar to `SCHED_FIFO` except time-slicing occurs for threads/processes of equal priority.

Setting the Linux Scheduling Policy

- `sched_getscheduler()` and `sched_setscheduler()` manipulate the scheduling policy.

```
#include <sched.h>

struct sched_param {
/* ... */
int sched_priority;
/* ... */
};

int sched_getscheduler(pid_t pid);

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);
```

Get Priority

```
int policy;

/* get our scheduling policy */
policy = sched_getscheduler(0);

switch(policy) {
  case SCHED_OTHER:
    printf("Policy is normal\n");
    break;
  case SCHED_RR:
    printf("Policy is round-robin\n");
    break;
  case SCHED_FIFO:
    printf("Policy is first-in, first-out\n");
    break;
  case -1:
    perror("sched_getscheduler");
    break;
  default:
  fprintf(stderr, "Unknown policy!\n");
}
```

Set Priority

```
struct sched_param sp = { .sched_priority = 1 };

int ret;
ret = sched_setscheduler(0, SCHED_RR, &sp);

if (ret == -1) {
  perror("sched_setscheduler");
  return 1;
}
```

Summary

Summary

- Thread scheduling: PCS and SCS

Summary

- Thread scheduling: PCS and SCS
- Multi-processor scheduling: SMP, processor affinity, load balancing

Summary

- Thread scheduling: PCS and SCS
- Multi-processor scheduling: SMP, processor affinity, load balancing
- Real-time scheduling: soft and hard real times

Summary

- Thread scheduling: PCS and SCS
- Multi-processor scheduling: SMP, processor affinity, load balancing
- Real-time scheduling: soft and hard real times
- Real-time scheduling algorithms
  - Priority-based
  - Rate-monotonic
  - Earliest-deadline-first
  - Proportional share scheduling

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.