Histogram of CPU Burst Cycles
Alternating Sequence of CPU and IO

- load store
- add store
- read from file
- wait for I/O
- store increment
- index
- write to file
- wait for I/O
- load store
- add store
- read from file
- wait for I/O
- CPU burst
- I/O burst
- CPU burst
- I/O burst
- CPU burst
- I/O burst
Processor Scheduling

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

- 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

- Scheduling under 1 and 4 is non-preemptive.
- All other scheduling is preemptive.
Scheduling Criteria

- CPU utilization - Keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- 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, not output (for time-sharing environment)
Classic Scheduling Algorithms

- Non preemptive scheduling
  - FIFO
  - Shortest job first
- Preemptive scheduling
  - Round robin
  - Earliest deadline first
Real-Time Systems

- Systems whose correctness depends on their temporal aspects as well as their functional aspects

- Performance measure
  - Timeliness on timing constraints (deadlines)
  - Speed/average case performance are less significant

- Key property
  - Predictability on timing constraints

- Hard and soft real-time systems

Adopted from CSE480 slides by Insup Lee at UPenn
Real-Time System Examples

- Real-time monitoring systems
- Industrial robots
- Multimedia processing
- On-line transaction systems
- Military weapon control systems
- All sensor-actuator systems
  - Stimulus-response model
Real-Time Workload

- Job (unit of work)
  - a computation, a file read, a message transmission, etc

- Attributes
  - Resources required to make progress
  - Timing parameters

![Diagram showing released time, execution time, relative deadline, and absolute deadline.](attachment:diagram.png)
Real-Time Task

- Task: a sequence of similar jobs
  - Periodic task \((p,e)\)
    - Its jobs repeat regularly
    - Period \(p\) = inter-release time \((0 < p)\)
    - Execution time \(e\) = maximum execution time \((0 < e < p)\)
    - Utilization \(U = \frac{e}{p}\)
Schedulability

- Property indicating whether a real-time system (a set of real-time tasks) can meet their deadlines
Real-Time Scheduling

- Determines the order of real-time task executions
- Static-priority scheduling
- Dynamic-priority scheduling
RM (Rate Monotonic)

- Optimal static-priority scheduling
- It assigns priority according to period
- A task with a shorter period has a higher priority
- Executes a job with the shortest period

\[ T_1(4,1) \]
\[ T_2(5,2) \]
\[ T_3(7,2) \]
RM (Rate Monotonic)

- Executes a job with the shortest period

- $T_1(4,1)$
- $T_2(5,2)$
- $T_3(7,2)$
EDF (Earliest Deadline First)

- Optimal dynamic priority scheduling
- A task with a shorter deadline has a higher priority
- Executes a job with the earliest deadline
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline

Diagram:
- T₁(4,1)
- T₂(5,2)
- T₃(7,2)
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline

Diagram:
- $T_1(4,1)$
- $T_2(5,2)$
- $T_3(7,2)$

Timeline from 0 to 15 units of time.
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline

**Diagram:**

- $T_1(4,1)$
- $T_2(5,2)$
- $T_3(7,2)$
EDF (Earliest Deadline First)

- Optimal scheduling algorithm
- If there is a schedule for a set of real-time tasks, EDF can schedule it
RM vs. EDF

- Rate Monotonic
  - Simpler implementation, even in systems without explicit support for timing constraints (periods, deadlines)
  - Predictability for the highest priority tasks

- EDF
  - Full processor utilization
  - Misbehavior during overload conditions

For more details: Buttazzo, “Rate monotonic vs. EDF: Judgement Day”, EMSOFT 2003
Linux Scheduler
Overview

- Scheduling implementation is in charge of ...
  - Context switching
  - Task selection

- Linux implements scheduler in `kernel/sched.c`
  - `schedule()` is core function

- History
  - 2.4 ~ 2.6: Epoch Scheduler (a.k.a. O(n) scheduler)
  - 2.6.8.1 ~ 2.6.23: O(1) Scheduler
  - 2.6.23 ~ Currently: Completely Fair Scheduler
Scheduler Invocation

- Linux scheduler is invoked in two ways
  - Direct invocation
  - Lazy invocation
- Direct invocation
  - When the current process is going to block
- Lazy invocation
  - When the current process has used up its quantum
  - When a process calls `sched_yield()`
  - Lazy invocation used the `need_resched` flag of process descriptor and will cause `schedule()` to be called later
Scheduling Classes

- POSIX standard defines two real-time schedulers:
  - FIFO (first-in-first-out)
  - RR (round robin)

- Linux implements these schedulers for real-time class scheduling mode.

- When there are runnable tasks of real-time class, one of both real-time schedulers is accordingly used.

- Other normal priority processes are scheduled by fair-share scheduler.

- When system idles, idle task is scheduled.
Scheduling Decision Principle

Start with top scheduler class

Runnable task available?

Y

Pick next task of scheduler class

N

Pick next scheduler class
Epoch Scheduler

- Scheduler divides processor time into epochs
- Within each epoch, every task can execute up to its time slice
- If a task does not use all of its time slice, then the scheduler adds half of remaining time slice to allow it to execute longer in the next slice
- On beginning of each epoch, every task earns time slices based on its priority
- An epoch ends when all processes in the ready queue have used their quantum
Epoch Scheduler

- Scheduler chooses a runnable task with the longest remaining time slice
- To achieve responsiveness a task can schedule up to
O(1) Scheduler

- Use priority sorted array of lists
- O(1) insertion and selection time
- Nice values are converted to priorities
- There are two runqueues
  - Active
  - Expired

Realtime Priority 99
Realtime Priority 98
Realtime Priority 2
Realtime Priority 1
Nice -20
Nice -19

Nice +1
Nice 0
Nice -1

Nice +18
Nice +19
O(1) Scheduler

- Priority scheduler
  - If there exists a task in a high priority active runqueue, all tasks in low priority active runqueues will never be scheduled
  - To rapidly find non-empty highest priority queue, O(1) scheduler uses a priority queue bitmap
  - All tasks in the same priority queue will be executed in round robin manner

- A task will be moved to expired run queue with corresponding priority when it uses up its time slice
- Once all tasks in all active run queues are moved to expired run queues, expired and active arrays will be simply switched
O(1) Scheduler

- I/O bound task priority boost
  - Interactive tasks (which sleep for long time) get priority boost
  - Dynamic priority = static priority ±5
- SMP support
  - Per-CPU runqueues
  - Task migration for load balancing
Completely Fair Scheduler

- Red-black tree for task management keeps a virtual timeline of tasks to schedule
  - Scheduler decision takes $O(1)$
  - Reinsertion of a task is $O(\log(N))$

Completely Fair Scheduler

- Nanoseconds based accounting
  - Independent of HZ and jiffies
- Task with the longest wait time in RB tree is selected next
- Nice levels are not depending on timeslice
- Nice levels are multiplicative
- Interactive task sleep time is honored
Nice in CFS

- Difference in nice value by 1 is intended to provide 10% more (or less) CPU utilization
- A kernel cannot deal with floating point operations
- Nice value (priority) is translated to weight
  - Nice 0 (priority 120) translates to weight 1024
Nice in CFS

runtime weight vs. nice level

weight(n) = 1.25 * weight(n-1)
Virtual Clock

- Key in RB tree of CFS is virtual runtime a task used
  - \( vruntime - \text{min}_vrun\text{time} \) of RB tree
- \( vruntime \) increases as follows when task executes for \( delta\_exec \) time

\[
delta\_exec\_weighted = \delta\_exec \times \frac{\text{NICE}_0\_LOAD}{\text{curr-}\rightarrow\text{load}.\text{weight}}
\]
CGROUP

- Makes use of the hierarchical modular design
- Groups tasks per UID or user defined control groups
- Exact CPU time share for each group Fairness across users
- Can limit the total CPU time available for RT tasks
SMP Issues

- Run queues are CPU-specific
- In principle, a task is assigned to a dedicated core
  - Load balancing
  - Processor affinity of a task
- A task can be migrated to balance load
  - With great care (why?)