Real-World Scheduling
Linux Scheduling

• Policies
  – SCHED_FIFO
    - “real time”
    - infinite time quantum
  – SCHED_RR
    - “real time”
    - adjustable time quantum
  – SCHED_OTHER
    - “normal” scheduler
    - parameterized allocation of processor time
Linux Scheduler Evolution

- **Old scheduler**
  - very simple
  - poor scaling
- **O(1) scheduler**
  - introduced in 2.5
  - less simple
  - better scaling
- **Completely fair scheduler (CFS)**
  - even better
  - simpler in concept
  - much less so in implementation
  - based on stride scheduling
Old Scheduler

- Four per-process scheduling variables
  - *policy*: which one
  - *rt_priority*: real-time priority
    - 0 for SCHED_OTHER
    - 1 – 99 for others
  - *priority*: time-slice parameter ("nice" value)
  - *counter*: records processor consumption
Old Scheduler: Time Slicing

• Clock “ticks” HZ times per second
  – interrupt/tick
• Per-process *counter*
  – current process’s is decremented by one each tick
  – time slice over when counter reaches 0
Old Scheduler: Throughput

• Scheduling cycle
  – length, in “ticks,” is sum of priorities
  – each process gets priority ticks/cycle
    - counter set to priority
    - cycle over when counters for runnable processes are all 0
  – sleeping processes get “boost” at wakeup
    - at beginning of each cycle, for each process:

\[
\text{counter} = \text{counter}/2 + \text{priority}
\]
Old Scheduler: Who’s Next?

• Run queue searched beginning to end
  – new arrivals go to front
  – SCHED_RR processes go to end at completion of time slices

• Next running process is first process with highest “goodness”
  – 1000 + rt_priority for SCHED_FIFO and SCHED_RR processes
  – counter for SCHED_OTHER processes
Old Scheduler: Problems

- $O(n)$ execution
- Poor interactive performance with heavy loads
- SMP contention for run-queue lock
- SMP affinity
  - cache “footprint”
O(1) Scheduler

• All concerns of old scheduler plus:
  – efficient, scalable execution
  – identify and favor interactive processes
  – good SMP performance
    - minimal lock overhead
    - processor affinity
O(1) Scheduler: Data Structures

Processor 0

struct runqueue

active

expired

bitmap

bitmap

Processor 1

struct runqueue

active

expired

bitmap

bitmap
O(1) Scheduler: Queues

- Two queues per processor
  - active: processes with remaining time slice
  - expired: processes with no more time slice
  - each queue is an array of lists of processes of the same priority
    - bitmap indicates which priorities have processes
  - processors scheduled from private queues
    - infrequent lock contention
    - good affinity
"nice" value

O(1) Scheduler: Priorities
O(1) Scheduler: Actions

• Process switch
  – pick best priority from active queue
    - if empty, switch active and expired
  – new process’s time slice is function of its priority

• Wake up
  – priority is boosted or dropped depending on sleep time
  – interactive processes are defined as those who priority is above a certain threshold

• Time-slice expiration
  – interactive processes rejoin active queue
  - unless processes have been on expired queue too long
O(1) Scheduler: Load Balancing

- Processors with empty queues steal from busiest processor
  - checked every millisecond
- Processors with relatively small queues also steal from busiest processor
  - checked every 250 milliseconds
Scheduling in Windows

- Handling “normal” interactive and compute-bound threads
- Real-time threads
- Multiple processors
Priorities

31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0

Real Time

High

Above Normal

Normal

Below Normal

Low
Uniprocessor Windows

Runnable threads
Improving Real Time

- Multimedia applications need 80% of processor time
- Make sure normal applications get at least 20%
- How?
- Windows solution: MMCSS
  - multimedia class scheduler service
  - dynamically manage multimedia threads
    - run at real-time priority 80% of time
    - run at normal priority 20% of time
Which Processor?

- Newly created thread assigned *ideal processor*
  - randomly chosen
- May also set *affinity mask*
  - may be scheduled only on processors in mask
- Scheduling decision:
  - if idle processors available
    - first preference: ideal processor
    - second preference: most recent processor
  - otherwise
    - joins run queue of ideal processor
Some Details …

- **Waiting** → **Deferred ready**
  - Unwait
  - Change affinity
  - Select processor

- **Deferred ready** → **Ready**
  - Preempt
  - Schedule
  - Select processor

- **Ready** → **Standby**
  - Schedule
  - Preempt

- **Standby** → **Running**
  - Preempt
  - Schedule
  - Switch

- **Running** → **Standby**
  - Preempt

- **Running** → **Waiting**
  - Wait
Scheduling Concerns

• Hyperthreads
  – two instruction streams sharing same functional units and same L1 cache
• How long does cache footprint matter?
  – what cache parameters are important?
• When is it a good idea to put a thread on:
  – a different core?
  – a different NUMA node?
Hyperthreads

- Hyperthread
- Hyperthread
- Functional Units
- L1 Cache
- Core
Cores

- Hyperthread
- Functional Units
- L1 Cache

L2 Cache

- Hyperthread
- Functional Units
- L1 Cache
NUMA Nodes
Quiz 1

We have a system comprised of two NUMA nodes, each with four cores, each with two hyperthreads. The first node has four threads of different processes running on it; the other node is completely idle. One of the threads on the first node calls `pthread_create`.

a) The new thread should be assigned to an idle hyperthread on the first node

b) The new thread should be assigned to a hyperthread on the other node

c) The new thread should be assigned to a hyperthread on the same node and some other thread should move to the other node
Quiz 2

We have a system comprised of two NUMA nodes, each with four cores, each with two hyperthreads. The first node has four threads of different processes running on it; the other node is completely idle. One of the threads on the first node calls \textit{fork}.

a) The new thread should be assigned to an idle hyperthread on the first node

b) The new thread should be assigned to a hyperthread on the other node

c) The new thread should be assigned to a hyperthread on the same node and some other thread should move to the other node
We have a system comprised of two NUMA nodes, each with four cores, each with two hyperthreads. The first node has four threads of different processes running on it; the other node is completely idle. One of the threads on the first node calls `exec`.

a) The thread should stay where it is  
b) The thread should be assigned to a hyperthread on the other node  
c) The thread should stay where it is and some other thread should move to the other node