CS 3204
Operating Systems

Lecture 14

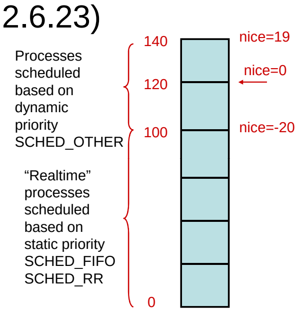
Godmar Back

CPU Scheduling

Part II

Case Study: 2.6 Linux Scheduler (pre 2.6.23)

- Variant of MLFQS
- 140 priorities
  - 0-99 “realtime”
  - 100-140 nonrealtime
- Dynamic priority computed from static priority (nice) plus “interactivity bonus”



Linux Scheduler (2)

- Instead of recomputation loop, recompute priority at end of each timeslice
  - dyn_prio = nice + interactivity bonus (-5…5)
- Interactivity bonus depends on sleep_avg
  - measures time a process was blocked
- 2 priority arrays (“active” & “expired”) in each runqueue (Linux calls ready queues “runqueue”)

Linux Scheduler (3)

```
struct prio_array {
        unsigned int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        struct list_head queue[MAX_PRIO];
};
typedef struct prio_array prio_array_t;

/* find the highest-priority ready thread */
 idx = sched_find_first_bit(array->bitmap);
 queue = array->queue + idx;
 next = list_entry(queue->next, task_t, run_list);
```

```
/* Per CPU runqueue */
struct runqueue {
 prio_array_t *active;
 prio_array_t *expired;
 prio_array_t arrays[2];
 …
}
```

- Finds highest-priority ready thread quickly
- Switching active & expired arrays at end of epoch is simple pointer swap (“O(1)” claim)

Linux Timeslice Computation

- Linux scales *static* priority to timeslice
  - Nice [-20 … 0 … 19] maps to [800ms … 100 ms … 5ms]
- Various tweaks:
  - “interactive processes” are reinserted into active array even after timeslice expires
    - Unless processes in expired array are starving
  - processes with long timeslices are round-robin’d with other of equal priority at sub-timeslice granularity

Proportional Share Scheduling

- Aka “Fair-Share” Scheduling
- None of algorithms discussed so far provide a direct way of assigning CPU shares
  - E.g., give 30% of CPU to process A, 70% to process B
- Proportional Share algorithms do by assigning “tickets” or “shares” to processes
  - Process get to use resource in proportion of their shares to total number of shares
- Lottery Scheduling, Weighted Fair Queuing/Stride Scheduling [Waldspurger 1995]

Lottery Scheduling

- Idea: number tickets between 1…N
  - every process gets p_i tickets according to importance
  - process 1 gets tickets [1… p_1-1]
  - process 2 gets tickets [p_1… p_1+p_2-1] and so on.
- Scheduling decision:
  - Hold a lottery and draw ticket, holder gets to run for next time slice
- Nondeterministic algorithm
- Q.: how to implement priority donation?

Weighted Fair Queuing

- Uses ‘per process’ virtual time
- Increments process’s virtual time by a “stride” after each quantum, which is defined as $(\text{process\_share})^{-1}$
- Choose process with lowest virtual finishing time
  - ‘virtual finishing time’ is virtual time + stride
- Also known as stride scheduling
- Linux now implements a variant of WFQ/Stride Scheduling as its “CFS” completely fair scheduler

WFQ Example (A=3, B=2, C=1)

Ready Queue is sorted by Virtual Finish Time
(Virtual Time at end of quantum if a process were scheduled)

Time	Task A	Task B	Task C	Ready Queue	Who Runs	
0	1/3	1/2	1	A (1/3) B (1/2) C (1)	A	One scheduling epoch. A ran 3 out of 6 quanta, B 2 out of 6, C 1 out of 6.
1	2/3	1/2	1	B (1/2) A (2/3) C (1)	B	
2	2/3	1	1	A (2/3) C(1) B(1)	A	
3	1	1	1	C(1) B(1) A(1)	C	
4	1	1	2	B(1) A(1) C(2)	B	This process will repeat, yielding proportional fairness.
5	1	3/2	2	A(1) B(3/2) C(2)	A	
6	4/3	3/2	2	A (4/3) B(3/2) C(2)		

WFQ (cont’d)

- WFQ requires a sorted ready queue
  - Linux now uses R/B tree
  - Higher complexity than O(1) linked lists, but appears manageable for real-world ready queue sizes
- Unblocked processes that reenter the ready queue are assigned a virtual time reflecting the value that their virtual time counter would have if they’d received CPU time proportionally
- Accommodating I/O bound processes still requires fudging
  - In strict WFQ, only way to improve latency is to set number of shares high – but this is disastrous if process is not truly I/O bound
  - Linux uses “sleeper fairness,” to identify when to boost virtual time; similar to the sleep average in old scheduler

Linux SMP Load Balancing

- Runqueue is per CPU
- Periodically, lengths of runqueues on different CPU is compared
  - Processes are migrated to balance load
- Aside: Migrating requires locks on both runqueues

```
static void double_rq_lock(
   runqueue_t *rq1,
   runqueue_t *rq2)
{
   if (rq1 == rq2) {
     spin_lock(&rq1->lock);
   } else {
     if (rq1 < rq2) {
       spin_lock(&rq1->lock);
       spin_lock(&rq2->lock);
     } else {
       spin_lock(&rq2->lock);
       spin_lock(&rq1->lock);
     }
   }
}
```

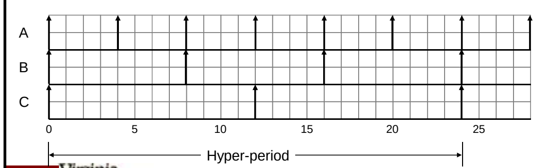
Real-time Scheduling

- Real-time systems must observe not only execution time, but a deadline as well
  - Jobs must finish by deadline
  - But turn-around time is usually less important
- Common scenario are recurring jobs
  - E.g., need 3 ms every 10 ms (here, 10ms is the recurrence period T, 3 ms is the cost C)
- Possible strategies
  - RMA (Rate Monotonic)
    - Map periods to priorities, fixed, static
  - EDF (Earliest Deadline First)
    - Always run what's due next, dynamic

EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

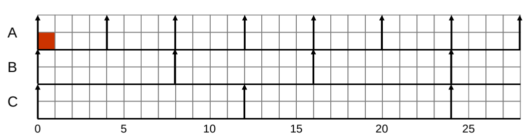
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

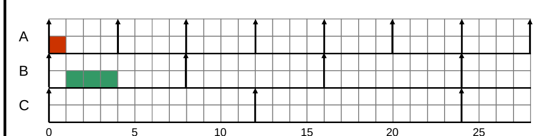
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

Assume deadline equals period (T).

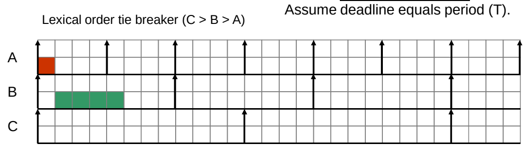


EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

Lexical order tie breaker (C > B > A)

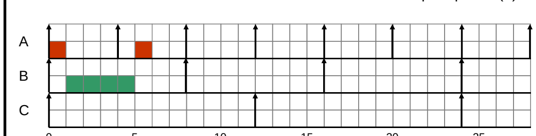
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

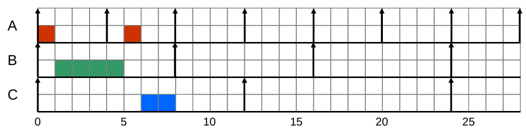
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

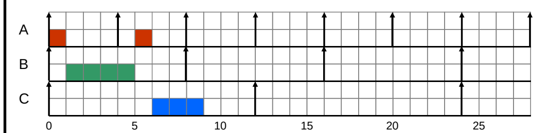
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

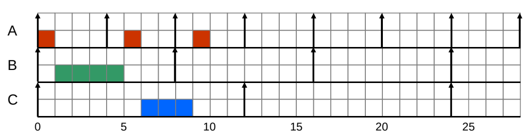
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

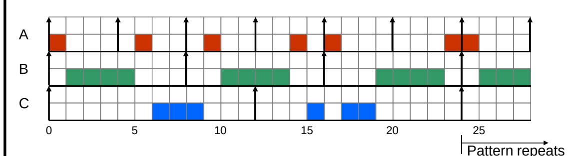
Assume deadline equals period (T).



EDF – Example

Task	T	C
A	4	1
B	8	4
C	12	3

Assume deadline equals period (T).



EDF Properties

- Feasibility test: $$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$
- $U = 100\%$ in example
- Bound theoretical
- Sufficient and necessary
- Optimal

Scheduling Summary

- OS must schedule all resources in a system
  - CPU, Disk, Network, etc.
- CPU Scheduling affects indirectly scheduling of other devices
- Goals for general purpose schedulers:
  - Minimizing latency (avg. completion or waiting time)
  - Maximing throughput
  - Provide fairness
- In Practice: some theory, lots of tweaking