Chapter 5

Process - Scheduling

Scheduler Organization
Scheduling Strategies and Methods

Policy versus Mechanism

• Separate what is allowed to be done with how it is done
  – a process knows which of its children threads are important and need priority

• Inter-process communication (signals, messages)
  – Realization – shared memory, mailbox, rendezvous, RPC

• Scheduling algorithm parameterized
  – mechanism in the kernel

• Parameters filled in by user processes
  – policy set by user process
Process Scheduling

- CPU resource allocation
- "scheduler" = allocator
- Policy is determined by a "scheduling strategy"
  - e.g.: FCFS, LCFS, Priority, “charm”, … (heuristics)?
  - Implemented by an algorithm/routine
- Scheduler routine rearranges the waiting list of "ready" processes (threads)
- Dispatcher routine (low-level) assigns CPU to the top of the ready list (RL)

Master / Shared Scheduler Organization

![Diagram showing Master/Shared Scheduler Organization]

- Centralized master scheduler (autonomous process)
- Shared scheduler (embedded procedure)
Process and Thread Scheduling

process p

PC1
th1
stack 1

PC2
th2
stack 2

shared data

RL

process q

PC1
th1
stack 1

PC2
th2
stack 2

shared data

RL

shared data

PCn
thn
stack n

PC1
th1
stack 1

PC2
th2
stack 2

shared data

RL

shared data

PCn
thn
stack n

Process Scheduling

Example process scheduling

Processor performs `receive`

Fig 3-7: Scheduling and dispatching in an object-oriented system
Possible Trigger Events

The scheduler can be invoked when any of the following events occur:

- A new thread is created.
- A previously blocked thread is awakened.
- A thread uses up its time quantum.
- A thread blocks itself.
- A thread terminates.
- A thread’s priority changes.
- The preferred processor of a thread in a multiprocessor system (called the processor affinity) changes.
Scheduling Algorithm

Scheduler() {
    do { Find highest priority ready_a process p;
        Find a free cpu;
        if (cpu != NIL) Allocate_CPU(p,cpu);
    } while (cpu != NIL);
    do {
        Find highest priority ready_a process p;
        Find lowest priority running process q;
        if (Priority(p) > Priority(q)) Preempt(p,q);
    } while (Priority(p) > Priority(q));

    if (self->Status.Type != 'running') //then
        Preempt(p,self);
}

Scheduling Methods

• Decision mode: - “when”

• Arbitration rule: - “who”
  – Priority function  P = F(a1, a2, …)
  – Preemptive
  – Non-preemptive

• Long-term / process scheduling
  – “job” – batch
  – “task” – interactive program (process / thread)
  – “real-time” – hard / soft deadlines
**Priority Attributes**

- attained service time
- real-time in system
- total service time
- deadline
- periodicity
- external priority
- memory requirements

**Common Scheduling Algorithms**

- First-in / First-out (FIFO)
- Shortest-Job_first
- Shortest-Remaining_time
- Round-Robin (timeslice - “quantum”)
- Multilevel-Priority (static)
- Multilevel-Feedback (dynamic)
- Rate-Monotonic (periodic)
- Earliest-deadline_First (real-time)
- Lottery scheduling
### Priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>maximum time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$T_n$</td>
</tr>
<tr>
<td>$n-1$</td>
<td>$2 \cdot T_n$</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>$2^{n-1} \cdot T_n$</td>
</tr>
<tr>
<td>error</td>
<td>$\infty$ (inf)</td>
</tr>
</tbody>
</table>

### Scheduling Strategies

<table>
<thead>
<tr>
<th>Scheduling Algorithm</th>
<th>Decision mode</th>
<th>Priority function</th>
<th>Arbitration rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>Nonpreemptiv</td>
<td>$r$</td>
<td>random</td>
</tr>
<tr>
<td>SJF</td>
<td>Nonpreemptiv</td>
<td>-$t$</td>
<td>chronological or random</td>
</tr>
<tr>
<td>SRT</td>
<td>Preemptive</td>
<td>$-(t - a)$</td>
<td>chronological or random</td>
</tr>
<tr>
<td>RR</td>
<td>Preemptive</td>
<td>0</td>
<td>cyclic</td>
</tr>
<tr>
<td>ML</td>
<td>Preemptive</td>
<td>$e$ (same)</td>
<td>cyclic chronological</td>
</tr>
<tr>
<td>MLF</td>
<td>Preemptive</td>
<td>$n - \lfloor \log_2(a/T + 1) \rfloor$ (same)</td>
<td>cyclic chronological</td>
</tr>
<tr>
<td>RM</td>
<td>Preemptive</td>
<td>-$d$</td>
<td>chronological or random</td>
</tr>
<tr>
<td>EDF</td>
<td>Preemptive</td>
<td>$-(d - r/d)$</td>
<td>chronological or random</td>
</tr>
</tbody>
</table>

### Organization of MLF Scheduling Queues

Processes exceeding maximum time
SPN - SJN with Aging

- SJN produces the minimum average response time
- $T_0$ - estimate of required processing time
- $T_1$ - measurement of required processing time
- New estimate: $T_{n+1} + aT_{n-1} + (1-a)T_n$

  $a$ - aging factor

Lottery Scheduling

- Processes are allocated “tickets”
- Scheduling decisions are based on random selection of tickets
- Priority can be observed by issuing a corresponding number of tickets to a process
- Cooperating processes may exchange tickets
**Turnaround Time**

Time spent in the system from initial start to completion = active time ($r$) + waiting time.

Average turnaround time for a set of processes:

$$\frac{\sum_{i=1}^{n} r_i}{n}$$

$$(t_1 + (t_1 + t_2))/2 \mid (t_2 + (t_2 + t_1))/2 \quad 1.4 \sim 3.45$$

process completion under different scheduling disciplines
**Turnaround Time**

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>0+4</td>
<td>4+2</td>
<td>3+1</td>
<td>14/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>=4.66</td>
</tr>
<tr>
<td>SJF</td>
<td>2+4</td>
<td>0+2</td>
<td>3+1</td>
<td>12/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>=4.00</td>
</tr>
<tr>
<td>SRT</td>
<td>3+4</td>
<td>0+2</td>
<td>0+1</td>
<td>10/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>=3.33</td>
</tr>
</tbody>
</table>

**Fig 5-7a: VAX/VMS priority levels**

- Real-time priority (static, FIFO)
- Current priority + increment
- Base priority
Schedules

- Real-time processing must meet deterministic timing constraints.
- Deadlines can only be met if processing requirements are known in advance – at least a good estimate
- A schedule is feasible if all deadlines can be met.
- A scheduling method is optimal if all deadlines can be met whenever a feasible schedule exists.
Fig 5-9: Linux priority levels
Scheduling Goals

• All systems:
  – Fairness – each process gets a fair share of the CPU
  – Policy enforcement – seeing that stated policy is carried out
  – Balance – keeping all parts of the systems busy

• Batch systems
  – Throughput – maximize jobs per hour
  – Turnaround time – minimize time between submission and termination
  – CPU utilization – keep the CPU busy all the time

• Interactive systems:
  – Response time – react and respond to requests quickly
  – Proportionality – meet user’s expectations

• Real-time systems
  – Meeting deadlines – avoid losing data or synchronization
  – Predictability – avoid quality degradation in multimedia systems
  – Integrity – guarantee correct reaction to external events within specified time frame
Timing Aspects

- Deterministic timing constraints
- Real-time systems and deadlines
- Schedule: assignment of processors to processes
  - Feasible schedule – keeps all deadlines
  - Optimal method – produces a feasible schedule

Real-Time Scheduling

\[ U = \sum_{i=1}^{n} \frac{t_i}{d_i} \]

A periodic process \( p_i \) will use \( t_i/d_i \) of the CPU time, where \( d_i \) is the period length and \( t_i \) is the total service time for process \( i \) for one period.

For \( n \) processes, the overall CPU utilization is defined as:

A schedule is feasible as long as \( U \leq 1 \) using EDF.

With RM a schedule is feasible only for \( U \leq 0.7 \) (approx. - remember queuing effects ?!)
Queuing Effects

 Priority Inversion

- Preemptive priority-based scheduling systems can suffer from this phenomenon which can lead to deadlock or violation of deadline restrictions.

- Example problem: 3 processes with priorities p1>p2>p3. Assume p1 and p3 interact through a common semaphore mutex, p2 is totally independent.

  - p1: . . . P(mutex); CS 1; V(mutex); . . .

  - p2: . . . program 2; . . . /* independent of p1 and p3 */

  - p2: . . . P(mutex); CS 3; V(mutex); . . .
Priority Inheritance

- A possible solution to priority inversion is dynamic priority inheritance; in this case a low priority process that has acquired a lock and is in a CS “inherits” the priority of a higher priority process attempting to enter the CS.
- This protocol works for any number of higher-priority processes becoming blocked on the same lock. The lock holder keeps inheriting the priority of the highest-priority blocked process.
- It also is applicable to nested CSs.
- Blocking times for the higher priority processes are bounded by the CS execution times of lower-priority processes.
Fig 5-11: priority inheritance

End