Scheduling

Scheduling of Threads

Contents

- Scheduling Principles
  - Round-Robin-Scheduling
  - Priority based scheduling
  - Dynamic priority adjustments
  - Hard and soft affinity
- Scheduling in UNIX
  - Traditional Scheduling Algorithm
  - Innovations in SVR4 and Solaris 2.x
  - Scheduling in Linux
- Scheduling in Windows

What is Scheduling?

- Scheduling is the assignment of the CPU(s) to threads.
- Criteria for the scheduler:
  - Justice: every thread shall obtain a „fair“ part of the CPU time (in particular: none shall starve).
  - Efficiency: the CPU(s) shall be as busy as possible.
  - Response Time: the response time for interactive users shall be minimal.
  - Special requirements for realtime systems, for example guaranteed response time to events.
- Two basic scheduling principles
  - Preemptive Scheduling: After a certain time on the CPU, a currently running thread will be preempted (i.e. removed from the CPU, and set to the ready state).
  - Run to completion (non-preemptive): a thread keeps the CPU until it voluntarily relinquishes it (yields the CPU, or blocks, or ends).

Round-Robin-Scheduling

- All ready threads are kept in a queue.
- Each thread is assigned a quantum (time slice) of CPU time.
- When a thread has used up its quantum, and the ready queue is non-empty
  - the thread is preempted, i.e. put in the ready state, and inserted at the tail of the ready queue,
  - the thread is assigned a new quantum,
  - the CPU is assigned to the first thread in the ready queue.
- When a blocked thread becomes ready again, it is inserted at the tail of the ready queue.
- Criteria for choosing the quantum:
  - The size of the quantum must be in a reasonable relation to the duration of a context switch.
  - Big quantum: long delays and waiting times possible.
  - Small quantum: short response time, but bigger overhead because of frequent context switches.

Priority-Based Scheduling

- Each thread is assigned a priority.
- The ready thread with the highest priority obtains the CPU. (Exception: Linux, see below)
- When there are several ready threads with the highest priority, the round robin algorithm is used for choosing among them.
- When a thread becomes ready, which has a higher priority than the currently running thread, the currently running thread is preempted.
- Assigning priorities:
  - The priority may be specified by the program or the system manager.
  - The priority may, for example, be assigned depending on the kind of process: interactive, batch, network, etc.

Dynamic Priority Adjustment

- Most operating systems dynamically alter the priority of threads at run time, distinguishing
  - the base priority of a thread, which is externally assigned,
  - the current priority of a thread, which is
    - dynamically assigned by the operating system,
    - used in the scheduling decisions.
- The most important goal of dynamic priority adjustments is to make
  - CPU-intensive threads run with a lower priority,
  - I/O-intensive threads run with a higher priority.
- Further reasons for dynamically changing thread priorities may be
  - to make interactive programs run with a higher priority,
  - to make the thread in the foreground window run with a higher priority,
  - to avoid that threads with a low base priority starve completely.

When is Scheduling Done?

- In the following situations the scheduler has to check, whether some other thread should obtain the CPU, and possibly initiate a context switch:
  - when the running thread blocks,
  - when the running thread ends,
  - when a blocked thread becomes ready,
  - when a new thread is created,
  - when a thread has used up its quantum,
  - when the priorities of ready threads were adjusted.
- The scheduler is invoked by the system call or interrupt handler handling the respective situation.
  Examples:
  - When the clock interrupt detects that the the currently running thread has used up its quantum, it invokes the scheduler.
  - A system call which blocks the current thread will afterwards invoke the scheduler.

How is the Scheduler Invoked?

- The scheduler is itself implemented as a (software) interrupt handler of low interrupt priority. „Invoking“ the scheduler hence means raising this interrupt.
  - The scheduler will not run immediately, but only after all pending activities with higher interrupt priority have been finished, immediately before returning to thread activity.
  - System routines can prevent context switches by running at a sufficiently high interrupt priority.
- The scheduler needs to perform two steps:
  - Choose the thread which is to run next.
    The scheduler may also decide that the currently running thread is to continue running.
    In this case it will skip the second step:
  - Perform the context switch to the chosen thread:
    - Save the thread context of the running thread
    - Restore the thread context of the chosen thread

Data Structures for Scheduling

Most systems use the following data structures for scheduling:

- One ready queue per priority value for the ready threads of this priority.
- A bitmap with one bit per priority value indicating whether the ready queue for this priority value is empty or not.

Variant on multiprocessor systems:

- Separate ready queues and bitmaps for each processor in the system.
  - Advantage: No shared access to the ready queues and bitmap, hence better performance because no synchronization is needed.
  - Disadvantage: more complicated to implement. Additional problems need to be solved, e.g. access to threads in the ready queues of other processors when the own ready queues are empty.

Hard Affinity and Soft Affinity

On multi-processor systems, the scheduler takes the following concepts into account:

- Hard Affinity
  - It can be specified that certain threads may only run on some subset of the CPUs.
- Soft Affinity
  - The system remembers on which CPU a thread was last running („soft affinity“ CPU).
  - It is likely that the cache of a CPU still contains some data of a thread which was running on this CPU shortly before.
  - When several CPUs can be chosen for a thread, the scheduler should select the soft affinity CPU.
  - In favor of soft affinity, the scheduler can deviate from the order of ready threads in the queue.
  - The priority of a thread has precedence over the consideration of soft affinity!

Scheduling in UNIX

Traditional Scheduling Algorithm
Scheduling in System V R4 and Solaris 2.x
The Linux Scheduler

Traditional Scheduling in UNIX (SVR3, 4.3BSD)

- Priorities:
  - Separate priority values in kernel mode (logically higher than user mode priorities).
    - When a process blocks, it is assigned a fixed (kernel mode) priority, which depends on the event the process is waiting for.
    - When switching from kernel mode to user mode, the process again gets its previous user mode priority.
  - The first ready process of highest priority obtains the CPU (exception: Linux).
- Round-robin Scheduling for processes with the same priority.
- Dynamic Priority Adjustment: Once a second, the priority for every process in the system is recalculated such that a process which recently got a lot of CPU time gets a logically lower priority.
- Non-preemptive Kernel: a process will not be preempted while running in kernel mode. Preemption can only occur when the process switches back to user mode again.

Scheduling in System V R4: Principles

- Priority values from 0 (lowest) to 160 (highest).
- SVR4 allows the use of different scheduling policies by introducing scheduling classes.
  By default there are the classes:
  - Timesharing (Default: priorities 0-59)
  - Realtime (Default: priorities 100-159)

  (The priorities 60-99 are the kernel priorities.)

  Other scheduling classes (and hence scheduling policies) can be defined and added.
- The scheduling class determines
  - the range of priority values for this class,
  - if, and under what circumstances, the priority of a process will change,
  - the size of the quantum.
- The SVR4 kernel defines several preemption points, at which a process running in kernel mode can be preempted by processes in the realtime class.

SVR4: The Timesharing Class

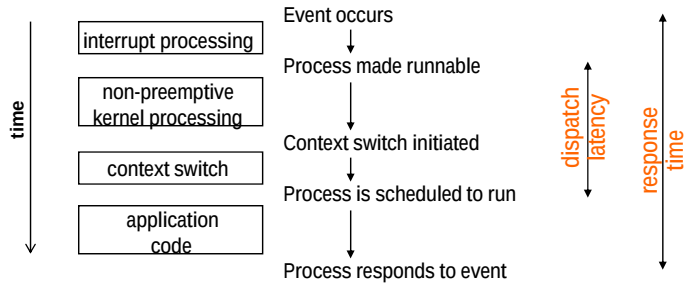
- Priority adjustment is done in response to specific events:
  - The priority is increased, when a blocked process becomes ready.
  - At quantum end, the priority is
    - decreased, when the quantum was used within a certain amount of time (maxwait),
    - increased, when using the quantum took longer than maxwait.
- The scheduling is driven by a „dispatcher parameter table“:

index	globpri	quantum	tqexp	slpret	maxwait	lwait
0	0	100	0	10	5	10
1	1	100	0	11	5	11
...	...	...	...	...	...	...
15	15	80	7	25	5	25
...	...	...	...	...	...	...
40	40	20	30	50	5	50
...	...	...	...	...	...	...
59	59	10	49	59	5	59

tqexp: priority at quantum end before maxwait
lwait: priority at quantum end after maxwait
slpret: priority when returning from a sleep state

SVR4: The Realtime Class

- Uses the priorities 100-159, which are higher than both timesharing and kernel priorities.
- Can only be used by superuser processes.
- Uses fixed priorities and quantum values, which can only be changed explicitly by the process using the system call *priocntl*.
- Together with the kernel preemption points this results in a reduced dispatch latency and hence response time.

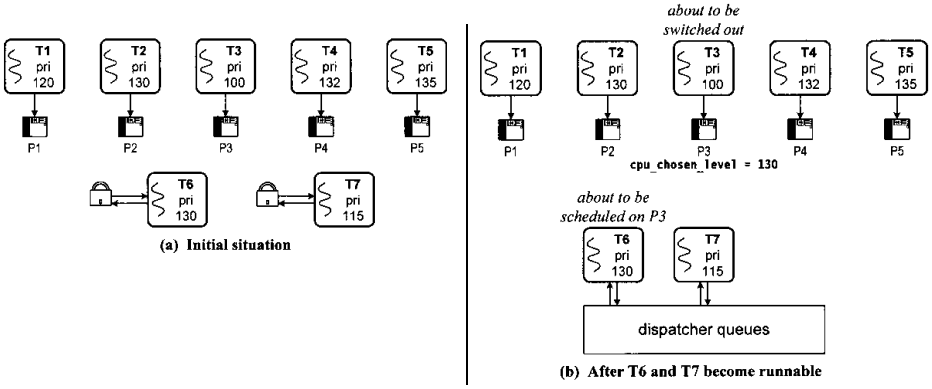


Scheduling in Solaris 2.x

- Solaris 2.x (1992) enhances the scheduling architecture of SVR4 by
  - making further optimizations to lower the dispatch latency of realtime processes,
  - supporting multithreading and SMP-systems,
  - taking soft affinity into account (only as of Solaris V9 [2002]).
- The kernel of Solaris 2.x is fully preemptive.
  - Kernel-Code can be preempted after every machine instruction.
  - In order to make this possible, all global kernel data structures must be protected by appropriate synchronization mechanisms such as mutexes (see chapter „Synchronization“).

Solaris 2.x: Preemption on Multiprocessor Systems

- When a thread becomes ready, the scheduler
  - determines the CPU which runs the lowest-priority thread,
  - preempts the thread on this CPU, if the thread that just became ready has higher priority.



Scheduling in LINUX

- The Linux Scheduler treats priorities in a totally different way than usual:
  - Priority values are used to determine the percentage of CPU time that a process should get relative to all all other ready processes.
    Example: If thread 1 has twice the priority of thread 2, it will get twice as much CPU time.
  - A process which has used up its quantum will not get a new one until all ready processes have used up their quantum.
- Up to version 2.4 of the Linux kernel (2001) there only is a single ready queue. The scheduler
  - calculates a „goodness“ value for each ready process (whatever this is ...),
  - assigns the CPU to the process with the highest goodness value.
- As of version 2.6 of the Linux kernel (2003)
  - there is a separate ready queue for each „goodness“ value,
  - the scheduler simply chooses the first process in the non-empty queue with the highest „goodness“ value (O(1) scheduling),
  - if an *interactive* process uses up its quantum, it immediately gets a new one.

The Scheduler in Linux

```
/*
 * 'schedule()' is the scheduler function. It's a very simple and nice
 * scheduler: it's not perfect, but certainly works for most things.
 * The goto is "interesting".
 * NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
asmlinkage void schedule(void)
{
      int c;
      struct task_struct * p;
      struct task_struct * prev, * next;
      unsigned long timeout = 0;
      int this_cpu=smp_processor_id();
/* check alarm, wake up any interruptible tasks that have got a signal*/
      if (intr_count)
            goto scheduling_in_interrupt;
      if (bh_active & bh_mask) {
            intr_count = 1;
            do_bottom_half();
            intr_count = 0;
      }
      run_task_queue(&tq_scheduler);
      need_resched = 0;
      prev = current;
      cli();
      /* move an exhausted RR process to be last.. */
      if (!prev->counter && prev->policy == SCHED_RR) {
            prev->counter = prev->priority;
            move_last_runqueue(prev);
      }
      switch (prev->state) {
            case TASK_INTERRUPTIBLE:
                  if (prev->signal & ~prev->blocked)
                        goto makerunnable;
                  timeout = prev->timeout;
                  if (timeout && (timeout <= jiffies)) {
                        prev->timeout = 0;
                        timeout = 0;
            makerunnable:
                        prev->state = TASK_RUNNING;
                        break;
                  }
            default:
                  del_from_runqueue(prev);
            case TASK_RUNNING:
      }
      p = init_task.next_run;
      sti();
#ifdef __SMP__
      /*
       *    This is safe as we do not permit re-entry of schedule()
       */
      prev->processor = NO_PROC_ID;
#define idle_task (task[cpu_number_map[this_cpu]])
#else
#define idle_task (&init_task)
#endif
```

The Scheduler in Linux (continued)

```
/*
 * Note! there may appear new tasks on the run-queue during this, as
 * interrupts are enabled. However, they will be put on front of the
 * list, so our list starting at "p" is essentially fixed.
 */


/* this is the scheduler proper: */
      c = -1000;
      next = idle_task;
      while (p != &init_task) {
            int weight = goodness(p, prev, this_cpu);
            if (weight > c)
                  c = weight, next = p;
            p = p->next_run;
      }

      /* if all runnable processes have "counter == 0", re-calculate counters */
      if (!c) {
            for_each_task(p)
                  p->counter = (p->counter >> 1) + p->priority;
      }
#ifdef __SMP__
      /*
       *    Allocate process to CPU
       */

       next->processor = this_cpu;
       next->last_processor = this_cpu;
#endif
#ifdef __SMP_PROF__
      /* mark processor running an idle thread */
      if (0==next->pid)
            set_bit(this_cpu,&smp_idle_map);
      else
            clear_bit(this_cpu,&smp_idle_map);
#endif
      if (prev != next) {
            struct timer_list timer;

            kstat.context_swtch++;
            if (timeout) {
                  init_timer(&timer);
                  timer.expires = timeout;
                  timer.data = (unsigned long) prev;
                  timer.function = process_timeout;
                  add_timer(&timer);
            }
            get_mmu_context(next);
            switch_to(prev,next);
            if (timeout)
                  del_timer(&timer);
      }
      return;

scheduling_in_interrupt:
      printk("Aiee: scheduling in interrupt %p\n",
            __builtin_return_address(0));
}
```

The Scheduler in Linux: The subroutine *weight*

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 *     -1000: never select this
 *         0: out of time, recalculate counters (but it might still be
 *          selected)
 *       +ve: "goodness" value (the larger, the better)
 *     +1000: realtime process, select this.
 */
static inline int goodness(struct task_struct * p, struct task_struct * prev,
int
this_cpu)
{
      int weight;
#ifdef __SMP__
      /* We are not permitted to run a task someone else is running */
      if (p->processor != NO_PROC_ID)
            return -1000;
#ifdef PAST_2_0
      /* This process is locked to a processor group */
      if (p->processor_mask && !(p->processor_mask & (1<<this_cpu))
            return -1000;
#endif
#endif
      /*
       * Realtime process, select the first one on the
       * runqueue (taking priorities within processes
       * into account).
       */
      if (p->policy != SCHED_OTHER)
            return 1000 + p->rt_priority;

      /*
       * Give the process a first-approximation goodness value
       * according to the number of clock-ticks it has left.
       *
       * Don't do any other calculations if the time slice is
       * over..
       */
      weight = p->counter;
      if (weight) {

#ifdef __SMP__
            /* Give a largish advantage to the same processor...   */
            /* (this is equivalent to penalizing other processors) */
            if (p->last_processor == this_cpu)
                  weight += PROC_CHANGE_PENALTY;
#endif

            /* .. and a slight advantage to the current process */
            if (p == prev)
                  weight += 1;
      }
      return weight;
}
```

Scheduling in Windows

Principles and Data Structures

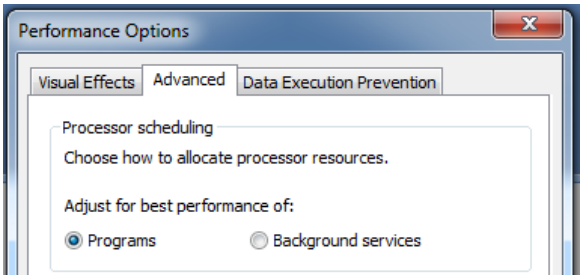
- Windows uses priority based, preemptive scheduling.
- The component performing the scheduling is called the dispatcher.
- The dispatcher runs at interrupt request level (IRQL) 2.
- The scheduling policy is modified by
  - several kinds of priority adjustment,
  - adjustment of the size of the quantum.
- Data structures for scheduling:
  - One queue per priority for ready threads with this priority („ready queues“).
  - A 32-bit bit mask („ready summary“), each single bit indicating, whether the corresponding ready queue contains threads or not.
  - A 32-bit bit mask („idle summary“), each single bit indicating, whether the corresponding CPU is idle or running a thread.

  (As of Windows Server 2003, these data structures exist per CPU.)

Quantum Size

- For threads with the same priority round-robin scheduling is performed. The length of the quantum by default is
  - 180 ms for Windows Server Systems (more precisely: 12 clock intervals),
  - 30 ms for Windows Workstation Systems (and on servers running terminal services),
  - 90 ms for the foreground thread on a Windows Workstation.

- The setting can be changed:



Priorities

- Internal priority values:
  - priorities from 0 (lowest) to 31 (highest)
  - two classes of priorities:
    - realtime priorities: 16 - 31
      no priority adjustment
    - dynamic priorities: 0 - 15
      dynamic priority adjustment
  - Priority 0 is only used by the system (e.g. for the idle thread).

- The priority of a thread is calculated from
  - the priority class of the process, and
  - the priority level of the thread (priority relative to the priority class)

Specifying Priorities in the Win32-API

- Specification of a priority class for a process:
  - IDLE (priority = 4)
  - BELOW_NORMAL (priority = 6)
  - NORMAL (priority = 8)
  - ABOVE_NORMAL (priority = 10)
  - HIGH (priority = 13)
  - REALTIME (priority = 24)
- Specification of a priority level (relative priority) for a thread:
  - IDLE (priority = 1 resp. 16)
  - LOWEST (priority = priority class - 2)
  - BELOW_NORMAL (priority = priority class - 1)
  - NORMAL (priority = priority class)
  - ABOVE_NORMAL (priority = priority class + 1)
  - HIGHEST (priority = priority class + 2)
  - TIME_CRITICAL (priority = 15 resp. 31)

Dynamic Priority Adjustment (1)

- Every thread has two priorities:
  - base priority: assigned during thread creation
  - current priority: dynamically determined
- For threads with a dynamic base priority (0-15) priority adjustments are performed, always making sure that
  - base priority $\leq$ current priority $\leq$ 15.
- Priority boosts for GUI-threads

  A thread belonging to a GUI application gets its priority boosted by 2
  - when it returns from a wait for user input or for a window message,
  - when it's the foreground thread and returns from a wait for some kernel object.

  After using one quantum, the priority of the thread returns to its base priority.

Dynamic Priority Adjustment (2)

- Priority boosts when returning from a wait state

  When a thread returns from an I/O or other wait operation, its priority is boosted by some value depending on the I/O device:

Device waited for	Boost
Disk, CD-ROM, Parallel Port, Video	1
Network, Mailslot, Named Pipe, Serial Port	2
Keyboard, Mouse	6
Sound	8

  Whenever a thread has used up a quantum, its priority is decremented (down to the thread's base priority).

Dynamic Priority Adjustment (3): Anti Starvation Boost

- A thread of low priority
  - can „starve“,
  - can block a thread of higher priority, which is waiting for it.
- Once a second, every *ready* thread not having run for more than 6 seconds (400 clock ticks)
  - gets its priority boosted to 15,
  - gets its quantum tripled.
- As soon as a thread that got an anti starvation boost yields the CPU
  - the priority returns to its previous value,
  - the quantum is reset to its normal value.

Priorities for Multimedia Applications and Games

- Client Versions of Windows incorporate the MultiMedia Class Scheduler Service (MMCSS) whose purpose is to ensure glitch-free multimedia playback, e.g. for audio or games (first implemented in Windows Vista).
- Applications have to register with MMCSS to use its functionality.
- Applications registered with MMCSS belong to one of four scheduling categories:

Category	Priority	Usage
High	23-26	Professional Audio threads.
Medium	16-22	For threads which are part of a foreground application.
Low	8-15	All other threads.
Exhausted	1-7	Threads that have exhausted their share of the CPU.

- By default, multimedia threads get 80% of the CPU time available, while other threads receive 20%.

Scheduling on SMP-Systems

- Criteria for assigning a CPU:
  - Hard Affinity: A thread can only run on specific processors.
  - Soft Affinity: A thread should preferrably run on the processor, on which it ran last time (because of the caches!).
  - Ideal Processor: The preferred CPU for a thread (a specialty of Windows).
- Choosing a thread for an idle CPU:
  - Find the non-empty ready queue with the highest priority ready threads. The first thread in this queue is the „primary candidate“.
  - The primary candidate is given the CPU, if
    - it last ran on this CPU (soft affinity), or
    - this CPU is its ideal processor, or
    - it already was in the ready state for more than three clock ticks, or
    - it has a priority greater than or equal to 24.
  - Else, the queue is searched further for the first thread satisfying one of these four conditions, and the CPU is given to this thread. If there is no such thread, the primary candidate obtains the CPU.

Scheduling on SMP Systems as of Windows Server 2003

- Windows Server 2003/2008 maintains
  - separate ready queues for each processor in the system,
  - a separate ready summary bitmask for each processor in the system.
- When searching for a thread to run on a processor, only the ready queues for this processor are searched for a ready thread.
- When all ready queues of a processor are empty, the idle thread is run on the CPU.
- The idle thread scans other processor‘s ready queues for threads in order to run them.