UNIX process scheduling

Tamás Mészáros

http://www.mit.bme.hu/~meszaros/

Department of Measurement and Information Systems

Budapest University of Technology and Economics

Previously....

- Definition of a **process (task)**
  - Running program (solving a particular problem)

- Relation between the process and the kernel
  - execution mode (user, kernel) and context (process, kernel)
  - syscall interface
  - process states and transitions
    - two running states, zombie, suspended (stopped) states, etc.

- Administrative data of a process (u-area and proc structure)
  - PID (process identifier) and PPID (parent PID)
  - credentials (UID, GID)
  - actual process state
  - scheduling information
  - etc.

Scheduling in general

- Main task of scheduling
  selecting the next task to be run from the set of runnable processes
- Basic notation
  - preemptive and cooperative scheduling
  - priority
  - static and dynamic scheduling
  - measuring quality (CPU utilization, throughput, avg. wait time, etc.)
- Basic operation
  - maintain a set of runnable tasks (FIFO, red-black binary tree, ...)
  - choose the next task to be run
- Requirements
  - small overhead, small complexity (O(1), O(N), O(log N))
  - optimality (according to the selected measures)
  - deterministic, fair, avoids starvation and system breakdown
  - there might be special application needs (real-time, batch, multimedia, ...)

Overview of this lecture

- Traditional UNIX scheduling
  - characteristics
  - operation in user and kernel mode
  - calculating priorities
  - detailed algorithm (user mode)
- Practice
- Modern UNIX schedulers
  - requirements and characteristics
  - short summary of the Solaris and Linux schedulers

Overview of the classical UNIX scheduling

- UNIX scheduling is **preemptive**, **time-sharing**, and **priority-based**

- Priority-based
  - every process has a dynamically calculated priority
  - the process with the highest priority runs

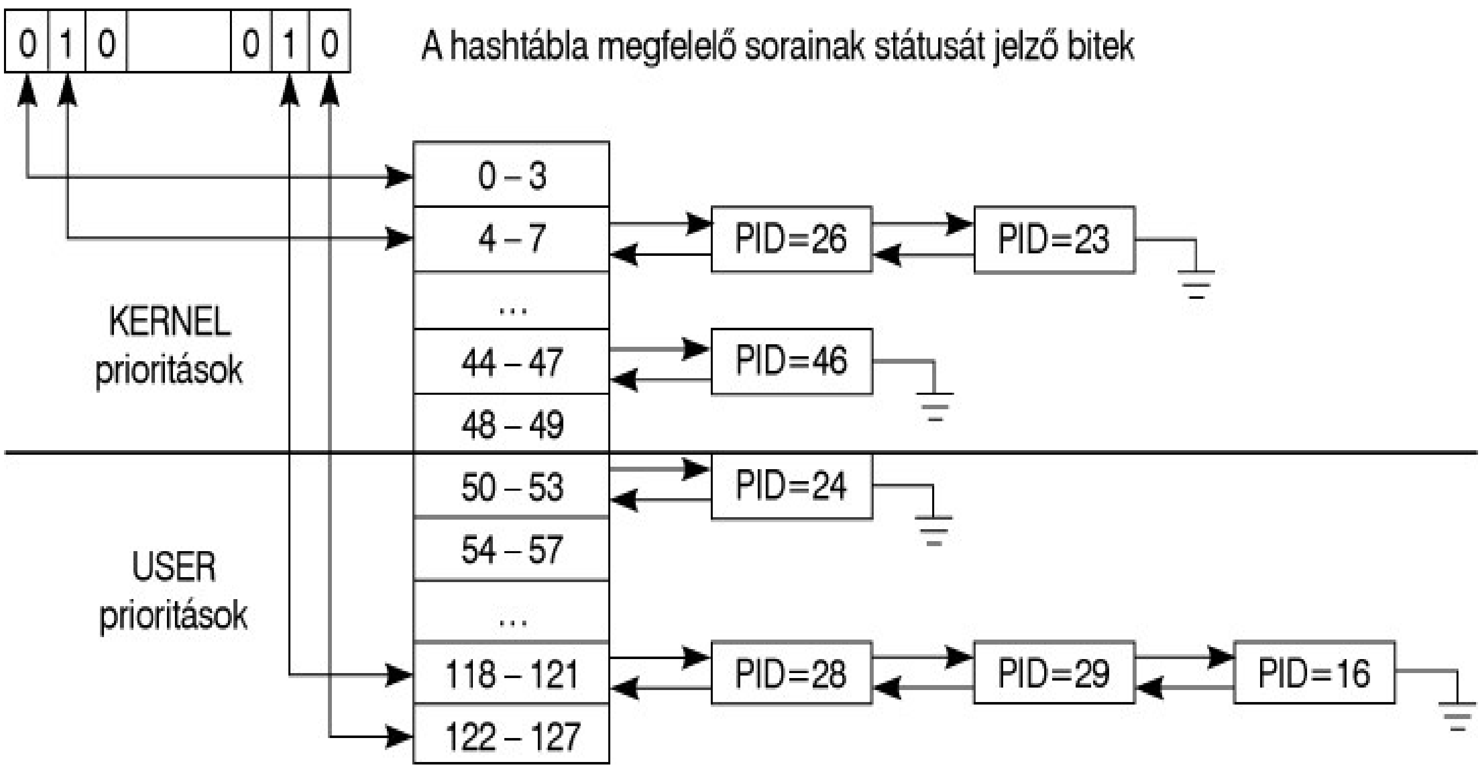
- Time-sharing
  - multiple processes at the same priority level can run in parallel
  - each of them is given a time slice (e.g. 10 msec)
  - after the time is expired the next process with the same prio will run

- **Note: scheduling is different in user and kernel mode**
  - user mode: preemptive, time-sharing, dynamic priority
  - kernel mode: non-preemptive, no time-sharing, fixed priority

(Modern UNIX schedulers are quite different.)

Scheduling (priority) levels

- The priority is between 0 and 127 (classical UNIX)
  - 0 is the highest, 127 is the lowest priority level
  - 0-49: kernel mode   50-127: user mode
- Processes are organized into 32 priority levels:



Scheduling in kernel mode

- Characteristics (very simple and almost zero overhead)
  - fixed priority
  - non preemtive
  - no time-sharing (since there is no preemtion)

- The priority does ***not*** depend on
  - the priority in user model
  - processor usage in the past

- Calculating the kernel mode priority
  - It is based on the reason why the process went to sleep.
  - **sleep priority**: attached to reasons to be in sleep state
  - *After waking up* a process the kernel sets its priority to the sleep priority.
  - Examples for sleep priority

20	disk I/O
28	user input from the character terminal

Scheduling in user mode

- The dynamically calculated priority is the basis of scheduling
  - the process with the highest priority runs
  - the scheduler checks the priority levels and chooses the first process from the highest non-empty queue (see the figure before)

- Task of the scheduler
  - in every cycle (typically 100 times a second)

    Is there any process at higher levels than the currently running process?
    If so then switch to the highest priority process.

  - after each time slice (typically 10 cycles, i.e. 10 times a second)

    Is there another process at the same priority level?

    If so the switch to the next process in the queue of that priority level.

    Round-Robin scheduling algorithm

Scheduling data for processes

- For each process:

p_pri	the actual priority of the process (kernel or user mode)
p_usrpri	the user mode priority of the process
p_cpu	CPU usage in the past
p_nice	priority modifier given by the user

- Scheduling decisions are based on p_pri
- Entering kernel mode saves p_pri into p_usrpri
- Returning from kernel mode recalls p_pri from p_usrpri
- p_cpu shows how much CPU was given to the process.
  The scheduler will „forget” past CPU usage slowly to avoid starvation.

Calculating the priority in user mode

- In every cycle p_cpu is increased for the running process

  p_pcu++

- After 100 cycles p_pri is calculated in the following way
  - p_cpu is „aged” by a correction factor (CF)

    p_cpu = p_cpu * CF

  - then the new priority is calculated according to this equation:

    p_pri = P_USER + p_cpu / 4 + 2 * p_nice   P_USER = 50 (constant)

- Calculating the correction factor:
  - SVR3: CF = 1/2 (What is the problem with this?)
  - 4.3 BSD: CF depends on the average number of processes (load_avg) in runnable (ready to run) state:

    CF = 2 * load_avg / (2 * load_avg + 1)

    See the following commands: `w`, `top` and the graphical system monitor

Summary of the user mode scheduling

- In every cycle
  - If there is a process at a higher priority level then switch to that process
  - Increase p_cpu for the running process.

- Every 10 cycles
  - Round-Robin scheduling among the processes at the same priority level

- Every 100 cycles
  - calculating the correction factor based on the last 100 cycles
  - „aging" p_cpu
  - calculating priorities for all processes
    please note that the priority does **not** depend on the past priority
  - switching to the highest priority process

Practice: scheduling 3 processes

(see scheduling_examples.xls)

Evaluating the classical UNIX scheduling

- Pros
  + simple and efficient
  + suitable for general-purpose time-sharing systems
  + avoids starvation
  + supports processes with I/O operations

- Cons
  – does not scale well
  – no guarantee for processes
  – users can not configure scheduling (except for nice)
  – no support for multi-processor, multi-core systems
  – kernel mode is non-preemptive:
    A process running in kernel mode for a long time can hold up the entire system (**priority inversion**)

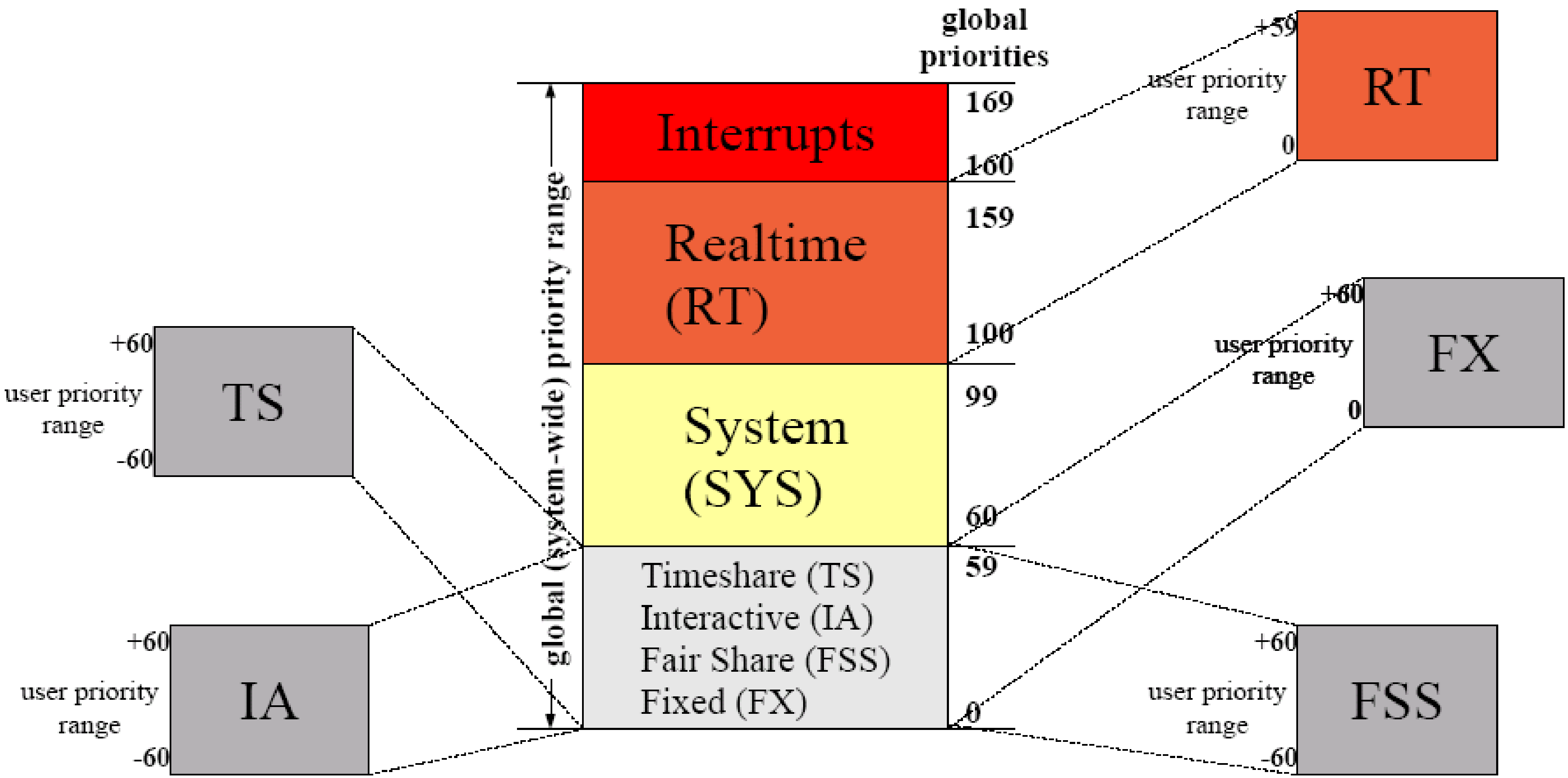
Modern UNIX schedulers (requirements)

- New scheduling classes
  - special application needs (multimedia, real-time, etc.)
  - „fair share”: it is possible to plan the resource allocation
  - multitasking at kernel level
  - modular scheduling with and extendable framework
- Kernel preemption
  - it is necessary for multiprocessor scheduling
- Performance, overhead
  - scheduling became more and more complex (requirements, hardware)
  - scheduling algorithms should scale well
- Threads or processes?
  - modern applications use threads a lot (e.g. Java)
  - schedulers should focus on threads not on processes

Scheduling in the Solaris operating system

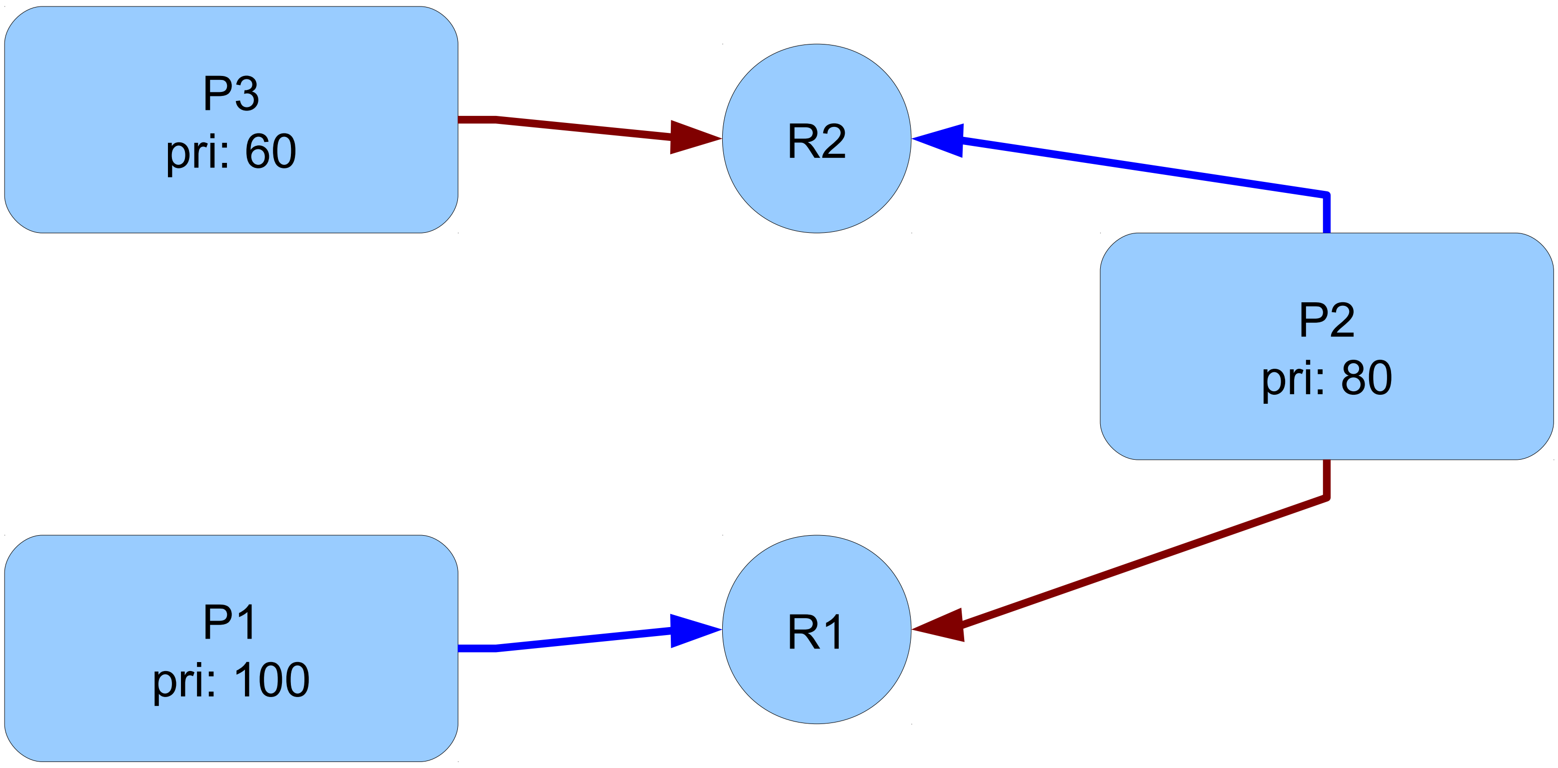
- Characteristics
  - scheduling is thread-based
  - the kernel is fully preemptible
  - supports multi-processor systems and virtualization
- New scheduling classes
  - Time Sharing (TS): similar to the classical scheduling
  - Interactive (IA): same as above but puts more emphasis on the active window on the graphical user interface
  - Fixed priority (FX)
  - Fair share (FSS): allocating CPU resources to process groups
  - Real-time (RT): provides the shortest resonse time
  - Kernel threads (SYS)

Solaris scheduling levels



Inherited priorities (Solaris)

- The problem of priority inversion (blue: waiting, red: holding)
- Solution: increasing the priorities according to the waiting scheme



Linux schedulers

- Before kernel V2: based on the classical UNIX scheduler
- Before V 2.4
  - scheduling classes: real-time, non-preemptive, normal
  - scheduling algorithm with O(N) complexity
  - single runnable queue (no SMP support)
  - non-preemptive kernel
- Kernel v2.6 (Ingo Molnár)
  - O(1) scheduler (scales very well)
  - multiple runnable queues (better SMP support)
  - a heuristic algorithm to differentiate between I/O and CPU-bound tasks
    - comparing running and waiting (sleeping) times (takes considerable time)
    - prefers I/O-bound processes
- 2.6.23 kernel: CFS (Completely Fair Scheduler)
  - designed and implemented by Ingo Molnár, some ideas from Con Kolivas
  - a new data structure for runnable processes: self-balancing red-black tree
  - tries to be fair by calculating a „virtual" runtime for all processes

Linux scheduling information (practice)

- Acquiring information using the /proc filesystem
  - /proc/cpuinfo – available CPUs
  - /proc/stat – CPU and scheduler properties
  - /proc/loadavg – average system load (past 1, 5, 15 minutes)
  - /proc/sys/kernel/sched* – scheduler information
  - /proc/<PID>/status – process state, Cpus_allowed, ...
  - /proc/<PID>/sched – process scheduling data

- What is happening on my computer?
  - Interesting story: Peeking into Linux kernel-land using /proc filesystem
    - Uses `ps, strace, /proc/PID/...` to debug a database problem
  - Other interesting things to know:
    - What Your Computer Does While You Wait

Linux CFS

- It replaces the previous *O(1)* scheduler with an *O(log n)* algorithm

- It uses a self-balancing *red-black tree* instead of simple linked lists
  - this is a binary tree with *O(log n)* complexity search
  - lower values to the left, higher to the right
  - insert and delete is more simple

- Calculating the priority is based on
  - number of virtually running processes (`nr_running`)
  - virtual run time (`vruntime`) in the rbtree index

- Basic operations
  - enqueue_task: New task arrived (nr_running++)
  - dequeue_task: Task no longer ready to run (nr_running--)
  - pick_next_task: who is the next to run

UNIX CRON and AT: long term scheduling

- Executing tasks at given time(s)
  - e.g. simple backup, maintenance tasks, etc.

- Usage
  - AT: execute a task at a given time (`at now + 1 day`)
  - CRON: periodically execute a task (see `man crontab`)

    minute, hour, day of month, month, day of week

    ```
    0 6 * 1-6,9-12 2 /local/bin/lets_play_soccer
    ```
    Send an invitation every Tuesday morning at 6am (except during summer)
    ```
    */20 * * * * /local/bin/clear_old_temp_cache
    ```
    Clear temporary and cache files in every 20 minutes

- This scheduling is <u>not</u> performed by the kernel
  - It is part of the userspace program set.
  - It starts certain tasks but does not govern them while they are running.
  - After started these tasks belong to short term scheduling.

Summary

- Classical UNIX scheduling
  - user mode: priority-based, time-sharing, preemptive
    - the process with the highest priority runs first
    - round-robin time-sharing scheduling between processes at the same prio. level
    - priority is calculated based on previous CPU usage and the nice value
  - kernel mode: fixed priority, non-preemptive
    - sleep priority assigned to resources will be given to awaking processes
  - simple, avoids starvation, handles I/O jobs very well
  - no SMP support, does not scale well, no support for spec. app. needs

- Modern UNIX schedulers
  - modular
  - several scheduling classes according to applications' needs
  - supports multi-cpu, multi-core systems (including CPU affinity)
  - better resource allocation (guaranteed CPU resources)
  - schedule threads

Try this at home: Linux scheduler simulator

- Install and get familiar with LinSched!

  http://www.cs.unc.edu/~jmc/linsched/

- Experiment with several task setups like
  - a mixture of I/O and CPU bound processes
  - processes with different nice values
  - a typical web server scenario (web + db + programs)

- This guide will help you on the way

  http://www.ibm.com/developerworks/library/l-linux-scheduler-simulator/