Linux, the time, and the noise...

Emiliano Betti

System Programming Research Group
DISP, University of Rome "Tor Vergata"

Linux Kernel Hacking Free Course
Rome, 14 May 2008

Outline

Outline

1. **Time in Linux**
   - Introduction
   - Hardware devices
   - Why does Linux need timekeeping?
   - Timer
   - Tick
2. OS noise
   - Operating system overhead
   - OS noise on computer cluster
3. Bibliography

Does Linux need to keep track of flow of time?

Obviously, yes. . . but why?

- clock (`gettimeofday()` system call)
- CPUs time sharing among processes
- to support other system and user activities:
  - handling timeouts
  - delayed execution of procedures
  - keeping track of resource usage statistics
  - User mode alarm (`alarm()`, `setitimer()` system calls)

Does Linux need to keep track of flow of time?

Obviously, yes. . . but why?

- clock (`gettimeofday()` system call)
- CPUs time sharing among processes
- to support other system and user activities:
  - handling timeouts
  - delayed execution of procedures
  - keeping track of resource usage statistics
  - User mode alarm (`alarm()`, `setitimer()` system calls)

Does Linux need to keep track of flow of time?

Obviously, yes... but why?

- clock (`gettimeofday()` system call)
- CPUs time sharing among processes
- to support other system and user activities:
  - handling timeouts
  - delayed execution of procedures
  - keeping track of resource usage statistics
  - User mode alarm (`alarm()`, `setitimer()` system calls)

Does Linux need to keep track of flow of time?

Obviously, yes. . . but why?

- clock (`gettimeofday()` system call)
- CPUs time sharing among processes
- to support other system and user activities:
  - handling timeouts
  - delayed execution of procedures
  - keeping track of resource usage statistics
  - User mode alarm (`alarm()`, `setitimer()` system calls)

Hardware devices (1/2)

Which hardware devices are available for timekeeping?

- Real Time Clock (RTC): battery-energized clock that keeps the current date and time even when the computer is switched off (e.g., Motorola 146818 chip in PC's)
- Time Stamp Counter (TSC): monotonically increasing counter, generally coupled with the system bus clock signal
- Programmable Interval Timer (PIT): programmable device that generates (periodic) interrupts (e.g., 8254 chip in PC's)

Hardware devices (1/2)

Which hardware devices are available for timekeeping?

- Real Time Clock (RTC): battery-energized clock that keeps the current date and time even when the computer is switched off (e.g., Motorola 146818 chip in PC's)
- Time Stamp Counter (TSC): monotonically increasing counter, generally coupled with the system bus clock signal
- Programmable Interval Timer (PIT): programmable device that generates (periodic) interrupts (e.g., 8254 chip in PC's)

Hardware devices (1/2)

Which hardware devices are available for timekeeping?

- Real Time Clock (RTC): battery-energized clock that keeps the current date and time even when the computer is switched off (e.g., Motorola 146818 chip in PC's)
- Time Stamp Counter (TSC): monotonically increasing counter, generally coupled with the system bus clock signal
- Programmable Interval Timer (PIT): programmable device that generates (periodic) interrupts (e.g., 8254 chip in PC's)

Hardware devices (1/2)

Which hardware devices are available for timekeeping?

- Real Time Clock (RTC): battery-energized clock that keeps the current date and time even when the computer is switched off (e.g., Motorola 146818 chip in PC's)
- Time Stamp Counter (TSC): monotonically increasing counter, generally coupled with the system bus clock signal
- Programmable Interval Timer (PIT): programmable device that generates (periodic) interrupts (e.g., 8254 chip in PC's)

Hardware devices (2/2)

- CPU Local Timer (LOC): circuit integrated in the CPU that raises local periodic interrupts (e.g., APIC Local Timer in IA-32 and Intel-64 CPU's, or Decrementer in POWER CPU's)
- High Precision Event Timer (HPET): device providing a number of high resolution counters and timers
- ACPI Power Management Timer (ACPI_PM): monotone counter included in all ACPI-compliant computers

Hardware devices (2/2)

- CPU Local Timer (LOC): circuit integrated in the CPU that raises local periodic interrupts (e.g., APIC Local Timer in IA-32 and Intel-64 CPU's, or Decrementer in POWER CPU's)
- High Precision Event Timer (HPET): device providing a number of high resolution counters and timers
- ACPI Power Management Timer (ACPI_PM): monotone counter included in all ACPI-compliant computers

Hardware devices (2/2)

- CPU Local Timer (LOC): circuit integrated in the CPU that raises local periodic interrupts (e.g., APIC Local Timer in IA-32 and Intel-64 CPU's, or Decrementer in POWER CPU's)
- High Precision Event Timer (HPET): device providing a number of high resolution counters and timers
- ACPI Power Management Timer (ACPI_PM): monotone counter included in all ACPI-compliant computers

Wall clock time (1/2)

In Linux the *wall clock time* is the current time and date

Linux must keep track of it for:

- the `gettimeofday()` system call
- User Mode POSIX clocks and timers based on CLOCK_REALTIME (`clock_gettime()` and `timer_create()` system calls)
- for timestamps (filesystems, TCP protocol...)

Wall clock time (2/2)

Real Time Clock (RTC)

- When system is switched off the *Real Time Clock* keeps the current date and time
- typically reading RTC is slow
- RTC has low resolution and low accurancy

Wall clock time

- During the system boot Linux reads it and stores in the `xtime` variable the number of seconds elapsed since the Epoch (00:00:00 UTC, January 1, 1970)
- then continues to keep track of time without using RTC (we'll see how)

Wall clock time (2/2)

Real Time Clock (RTC)

- When system is switched off the *Real Time Clock* keeps the current date and time
- typically reading RTC is slow
- RTC has low resolution and low accurancy

Wall clock time

- During the system boot Linux reads it and stores in the `xtime` variable the number of seconds elapsed since the Epoch (00:00:00 UTC, January 1, 1970)
- then continues to keep track of time without using RTC (we'll see how)

Wall clock time (2/2)

Real Time Clock (RTC)

- When system is switched off the *Real Time Clock* keeps the current date and time
- typically reading RTC is slow
- RTC has low resolution and low accurancy

Wall clock time

- During the system boot Linux reads it and stores in the `xtime` variable the number of seconds elapsed since the Epoch (00:00:00 UTC, January 1, 1970)
- then continues to keep track of time without using RTC (we'll see how)

CPUs time sharing

Linux distributes CPUs time among processes according to a policy based on:

- processes' priority
- how long each process has been running

Processes with highest priority run first (with preemption!)

Once scheduled on a CPU, a process can run until:

- another process with highest priority claims a CPU
- a specified amount of time elapse (then another process with same priority obtains the CPU)

. . . So Linux needs to know long time each process has been running!

CPUs time sharing

Linux distributes CPUs time among processes according to a policy based on:

- processes' priority
- how long each process has been running

Processes with highest priority run first (with preemption!)

Once scheduled on a CPU, a process can run until:

- another process with highest priority claims a CPU
- a specified amount of time elapse (then another process with same priority obtains the CPU)

. . . So Linux needs to know long time each process has been running!

CPUs time sharing

Linux distributes CPUs time among processes according to a policy based on:

- processes' priority
- how long each process has been running

Processes with highest priority run first (with preemption!)

Once scheduled on a CPU, a process can run until:

- another process with highest priority claims a CPU
- a specified amount of time elapse (then another process with same priority obtains the CPU)

...So Linux needs to know long time each process has been running!

CPUs time sharing

Linux distributes CPUs time among processes according to a policy based on:

- processes' priority
- how long each process has been running

Processes with highest priority run first (with preemption!)

Once scheduled on a CPU, a process can run until:

- another process with highest priority claims a CPU
- a specified amount of time elapse (then another process with same priority obtains the CPU)

. . . So Linux needs to know long time each process has been running!

Support for User and System activities

Linux (and in particular drivers) needs a mechanism to schedule an activity in the future and to handle timeouts:

Example (timeout)

- timeout for networking protocols
- waiting for data from a device

Example (User alarm)

- `alarm()` and `setitimer()` system calls

Support for User and System activities

Linux (and in particular drivers) needs a mechanism to schedule an activity in the future and to handle timeouts:

Example (timeout)

- timeout for networking protocols
- waiting for data from a device

Example (periodic activities)

- polling for data on a device
- flush disk cache

Example (User alarm)

- `alarm()` and `setitimer()` system calls

Support for User and System activities

Linux (and in particular drivers) needs a mechanism to schedule an activity in the future and to handle timeouts:

Example (timeout)

- timeout for networking protocols
- waiting for data from a device

Example (periodic activities)

- polling for data on a device
- flush disk cache

Example (User alarm)

- `alarm()` and `setitimer()` system calls

Support for User and System activities

Linux (and in particular drivers) needs a mechanism to schedule an activity in the future and to handle timeouts:

Example (timeout)

- timeout for networking protocols
- waiting for data from a device

Example (periodic activities)

- polling for data on a device
- flush disk cache

Example (User alarm)

- `alarm()` and `setitimer()` system calls

Software Timer (1/2)

To handle delayed functions, periodic activities, and timeouts Linux implements a very useful mechanism, the *software timer*

A *software timer* allows to schedule the execution of a function in the future

In the function it’s possible to reactivate the timer, so it is easy to realize periodic activities

Software Timer (1/2)

To handle delayed functions, periodic activities, and timeouts Linux implements a very useful mechanism, the *software timer*

A *software timer* allows to schedule the execution of a function in the future

In the function it's possible to reactivate the timer, so it is easy to realize periodic activities

Software Timer (2/2)

There are two types of timer:

High Resolution Timer

An hardware device is programmed to raise an interrupt when the HR timer expires and the related function is executed as soon as possible

Traditional Timer

Periodically the Linux kernel checks if there are expired timers and in such case executes the related functions

Software Timer (2/2)

There are two types of timer:

High Resolution Timer

An hardware device is programmed to raise an interrupt when the HR timer expires and the related function is executed as soon as possible

Traditional Timer

Periodically the Linux kernel checks if there are expired timers and in such case executes the related functions

Tick: why?

To summarize, Linux needs to:

- keep track of current date and time
- keep track of how long each process has been running and execute periodically the scheduler (for CPU time sharing)
- check for traditional timer expiration
- . . . and execute other periodic activities, such as:
  - keep track of other resource usage statistics
  - perform profiling
  - etc. . .

How can Linux do everything? . . . Using ticks

Tick: why?

To summarize, Linux needs to:

- keep track of current date and time
- keep track of how long each process has been running and execute periodically the scheduler (for CPU time sharing)
- check for traditional timer expiration
- . . . and execute other periodic activities, such as:
  - keep track of other resource usage statistics
  - perform profiling
  - etc. . .

How can Linux do everything? . . . Using ticks

Tick: why?

To summarize, Linux needs to:

- keep track of current date and time
- keep track of how long each process has been running and execute periodically the scheduler (for CPU time sharing)
- check for traditional timer expiration
- . . . and execute other periodic activities, such as:
  - keep track of other resource usage statistics
  - perform profiling
  - etc. . .

How can Linux do everything? . . . Using ticks

Tick: why?

To summarize, Linux needs to:

- keep track of current date and time
- keep track of how long each process has been running and execute periodically the scheduler (for CPU time sharing)
- check for traditional timer expiration
- ...and execute other periodic activities, such as:
  - keep track of other resource usage statistics
  - perform profiling
  - etc...

How can Linux do everything? ...Using ticks

Tick: why?

To summarize, Linux needs to:

- keep track of current date and time
- keep track of how long each process has been running and execute periodically the scheduler (for CPU time sharing)
- check for traditional timer expiration
- . . . and execute other periodic activities, such as:
  - keep track of other resource usage statistics
  - perform profiling
  - etc. . .

How can Linux do everything? . . . Using ticks

Tick: why?

To summarize, Linux needs to:

- keep track of current date and time
- keep track of how long each process has been running and execute periodically the scheduler (for CPU time sharing)
- check for traditional timer expiration
- . . . and execute other periodic activities, such as:
  - keep track of other resource usage statistics
  - perform profiling
  - etc. . .

How can Linux do everything? . . . Using ticks

Tick: what is it? (1/2)

A tick is a periodic event with a frequency defined during kernel configuration (macro `HZ`) raised typically by an hardware device that works as a metronome

Linux counts the number of ticks occurred since the system boot increasing monotonically a 64 bits variable called `jiffies_64`

Tick: what is it? (1/2)

A tick is a periodic event with a frequency defined during kernel configuration (macro `HZ`) raised typically by an hardware device that works as a metronome

Linux counts the number of ticks occurred since the system boot increasing monotonically a 64 bits variable called `jiffies_64`

Tick: what is it? (2/2)

`jiffies_64`'s value is used to keep track of flow of time and thus to handle traditional timers, timeouts, etc. . .

Linux kernel can be configured in *dynamic tick mode*, so that when the CPU is idle ticks are stoped. In this case ticks do not respect `HZ` frequency, so `jiffies_64` needs to be adjusted using a clock source, such as TSC or ACPI_PM

Tick: what is it? (2/2)

`jiffies_64`'s value is used to keep track of flow of time and thus to handle traditional timers, timeouts, etc...

Linux kernel can be configured in *dynamic tick mode*, so that when the CPU is idle ticks are stoped. In this case ticks do not respect `HZ` frequency, so `jiffies_64` needs to be adjusted using a clock source, such as TSC or ACPI_PM

Tick: how?

During the boot Linux selects an hardware device, typically *APIC Local Timer* for Intel architecture or *Decrementer* for PowerPC, and, depending on kernel configuration:

- periodically programs a device to raise one-shot interrupts at the nearest event (with High Resolution Timer support or in dynamic tick mode) , or
- programs a device to raise periodic interrupts (without High Resolution Timer support and dynamic tick mode)

On Symmetric MultiProcessor (SMP) systems timer interrupt must be local to the CPUs, so if a global interrupt is raised (i.e. PIT) the CPU that handles it has to send an IPI (Inter Processor Interrupt) to each other CPU

Tick: how?

During the boot Linux selects an hardware device, typically *APIC Local Timer* for Intel architecture or *Decrementer* for PowerPC, and, depending on kernel configuration:

- periodically programs a device to raise one-shot interrupts at the nearest event (with High Resolution Timer support or in dynamic tick mode) , or
- programs a device to raise periodic interrupts (without High Resolution Timer support and dynamic tick mode)

On Symmetric MultiProcessor (SMP) systems timer interrupt must be local to the CPUs, so if a global interrupt is raised (i.e. PIT) the CPU that handles it has to send an IPI (Inter Processor Interrupt) to each other CPU

Tick: how?

During the boot Linux selects an hardware device, typically *APIC Local Timer* for Intel architecture or *Decrementer* for PowerPC, and, depending on kernel configuration:

- periodically programs a device to raise one-shot interrupts at the nearest event (with High Resolution Timer support or in dynamic tick mode) , or
- programs a device to raise periodic interrupts (without High Resolution Timer support and dynamic tick mode)

On Symmetric MultiProcessor (SMP) systems timer interrupt must be local to the CPUs, so if a global interrupt is raised (i.e. PIT) the CPU that handles it has to send an IPI (Inter Processor Interrupt) to each other CPU

What happens at each tick?

each CPU has to

- perform profiling activities
- check for expired traditional timer (and possibly execute related functions)
- account CPU time to the current process
- execute the scheduler (only if needed)

only one (elected) CPU has to

- update `jiffies_64`
- update wall clock time (`xtime`)

What happens at each tick?

each CPU has to

- perform profiling activities
- check for expired traditional timer (and possibly execute related functions)
- account CPU time to the current process
- execute the scheduler (only if needed)

only one (elected) CPU has to

- update `jiffies_64`
- update wall clock time (`xtime`)

Outline

Tick and operating system overhead

The *operating system (OS) overhead* is the amount of time a CPU spends while executing system's code

Ticks periodically interrupt the running process and can launch other activities (such as timer functions or scheduler)

After updating `jiffies_64` some timeouts can expire, so related functions are executed

This means that part of *OS overhead* is concentrated after a tick

Tick and operating system overhead

The *operating system (OS) overhead* is the amount of time a CPU spends while executing system's code

Ticks periodically interrupt the running process and can launch other activities (such as timer functions or scheduler)

After updating `jiffies_64` some timeouts can expire, so related functions are executed

This means that part of *OS overhead* is concentrated after a tick

Tick and operating system overhead

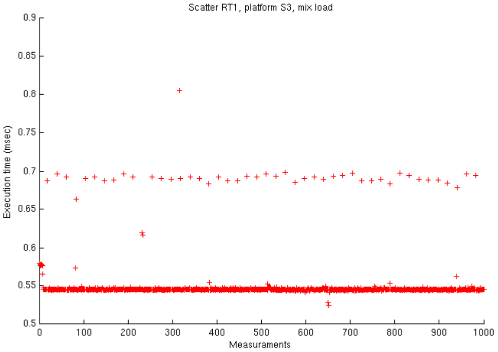
The *operating system (OS) overhead* is the amount of time a CPU spends while executing system's code

Ticks periodically interrupt the running process and can launch other activities (such as timer functions or scheduler)

After updating `jiffies_64` some timeouts can expire, so related functions are executed

This means that part of *OS overhead* is concentrated after a tick

What about a running process?



Difference between a run with interrupts disabled and another with SCHED_FIFO priority on a CPU that does not handle interrupt, except the local timer interrupt (tick)

OS overhead for an application that calculates an FFT

Computer Cluster

A *computer cluster* is a group of homogeneous computers (nodes) that work together. The nodes of a cluster are interconnected to each other through a fast network

Clusters are widely used in High Performance Computing (HPC) and commonly the applications are based on Message Passing Interface (MPI) or on Parallel Virtual Machine (PVM) libraries

Typically on each node there is a number of processes equal to the number of avalaible CPUs or cores

Common parallel programs alternate computing and communication (synchronization) phases

Computer Cluster

A *computer cluster* is a group of homogeneous computers (nodes) that work together. The nodes of a cluster are interconnected to each other through a fast network

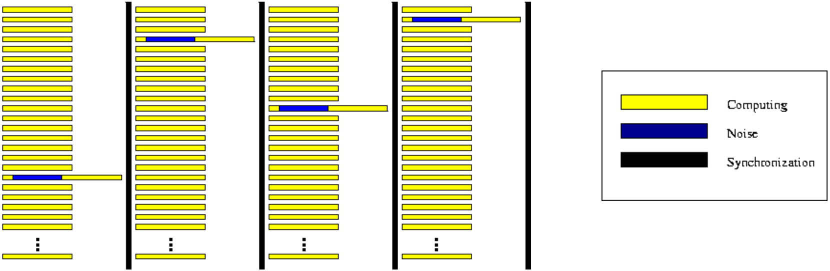
Clusters are widely used in High Performance Computing (HPC) and commonly the applications are based on Message Passing Interface (MPI) or on Parallel Virtual Machine (PVM) libraries

Typically on each node there is a number of processes equal to the number of avalaible CPUs or cores

Common parallel programs alternate computing and communication (synchronization) phases

What about OS overhead in a cluster?

Even if the OS overhead in a node can be 1-2% (on average) the overhall effect on a large cluster increases considerably

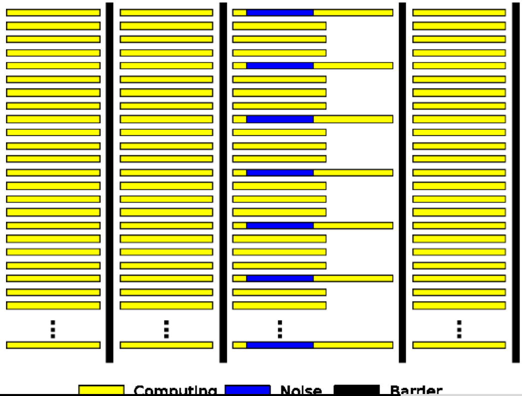


If, on each computational phase, all the processes have to wait for one slow process the whole parallel job is delayed

This can be defined **OS noise**

OS noise and clusters

On large clusters the probability that during each computational phase one process is delayed due to the OS approximates 1



A possible solution is forcing all cluster's nodes to perform system activities at the same time

Nettick

Now, I can leave you in the hands of Francesco. . .

Outline

Bibliography

- **ULK**: “Understanding the Linux kernel, 3rd edition”
  D. P. Bovet and M. Cesati
- **LDD**: “Linux Device Driver, 2nd edition”
  J. Corbet, A. Rubini and G. Kroah-Hartman
- **LWN**: “Linux Weekly News”
  www.lwn.net
- **ASMP**: “Hard Real-Time Performances in Multiprocessor-Embedded Systems Using ASMP-Linux”, EURASIP Journal on Embedded Systems 2008
  E. Betti, D.P. Bovet, M. Cesati, and R. Gioiosa
- **SOPC**: “Analysis of System Overhead on Parallel Computers”, 4th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2004)
  R. Gioiosa, F. Petrini, K. Devis, F. Lebaillif-Delamare