Where's all the time going?

Microstate accounting in Linux 2.5

Peter Chubb[*]

September, 2003

Abstract

In a UNIX system, `time command` reports real, system and user time used by `command`. For various reasons, the reported times are likely to be at variance with the actual resources used by the program.

We added high resolution timing of the actual time spent in various states in the kernel to Linux 2.5.

1 The Problems

When you time a process using **time**(1), you get back three numbers: real time, user time, and system time. These numbers are generated statistically; at regular intervals, an interrupt occurs and the current process gets an extra tick in either its system time counter or its user time counter.

These numbers should be viewed with extreme scepticism.

- Multi-threaded processes are not accounted for correctly. Only the 'main' thread is counted; time used by threads that it starts are accounted for separately. As LinuxThreads has an extra manager thread which starts all the other threads, created threads are grandchildren not children of the main thread, so their times are never accumulated into the main thread's timers.
- There are also processes that wake at regular intervals, do a small amount of processing (less than a tick) then sleep (e.g., Apache) — their wakeup interval tends to become synchronised with the timer interrupt so their time is never accounted for (they are almost *never* running when the timer tick happens even though they are runnable then).
- Time spent in interrupt handlers is accounted to the process that was interrupted, not to the process that caused the interrupt.
- In addition, there are sources of poor performance that just aren't captured by these times — scheduler latency, IO latency, etc., etc.

Most modern processor architectures have a high precision clock that can be used to time things with low overhead. A simple solution appears to be to add timers using the processor's inbuilt high-resolution clock to track the time spent by each thread in each state.

2 Related Work

The Solaris® operating system has had microstate accounting for its light-weight processes for a long time. The state transition diagramme for its threads is different from that for Linux tasks, so the states tracked are different.

The IRIX® operating system uses hardware timers to track system and user time, rather than relying on statistical sampling.

3 Thread States

Threads move through many states as they run (see figure 1 for a simplified picture).

Some of these states are reflected in the `state` variable in `struct task_struct`; but not all.

[*]This work was funded by HP through the Gelato programme (`http://www.gelato.org`) and the University of NSW

®Solaris is a registered trademark of Sun Microsystems Inc
®IRIX is a registered trademark of SGI

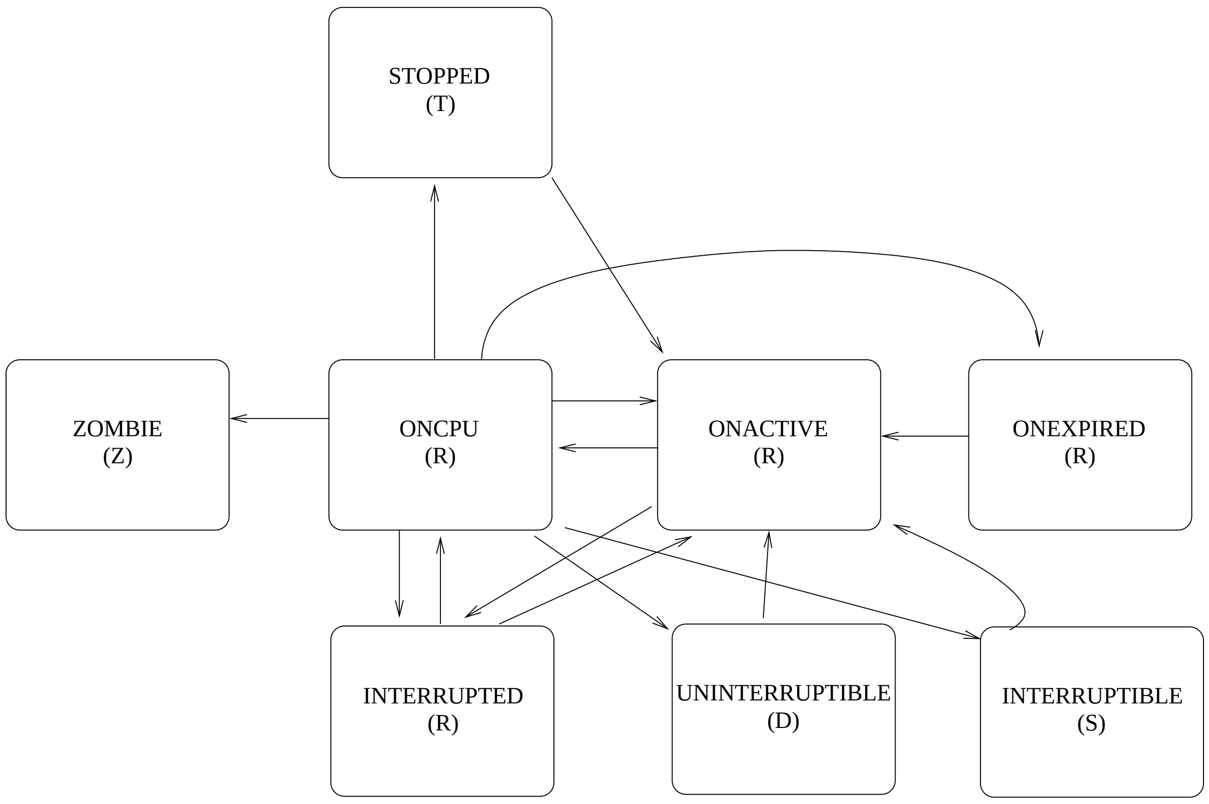

Figure 1: Simplified Thread State Transition Diagramme

In particular, the states tracked in `state` do not distinguish between running and runnable; nor between active and expired.

It'd be nice to capture the time spent in every possible state; however, it's simpler just to capture the most interesting states. From the point of view of being able to tune the Linux scheduler, and being able to measure an application's actual impact on the system, the interesting states are:

1. The time spent running on a processor.
2. The time spent on the active queue,
3. The time spent on the expired queue,
4. The time spent sleeping uninterruptibly (e.g., waiting for disc I/O)
5. The time spent sleeping interruptably (e.g., waiting for TTY I/O)
6. The time spent as a zombie, before the parent lays the process to rest.

4 Implementation

4.1 Data structures

We added to each `struct task_struct` a new data structure, `struct microstates`, defined as in figure 2

Obviously, more states can be added if that turns out to be desirable.

The *timers* array keeps track of times as the thread states change; the *child_timers* array is updated when a thread waits for zombie children.

In addition, a per-cpu array per interrupt keeps track of time spent handling each interrupt.

4.2 In-kernel interfaces

We added new functions:

'`msa_set_timer(tsk, newstate)`'
which accumulates the time since *last_change* into *timers[cur_state]* then sets *cur_state* to *newstate*.

```
typedef u64 clk_t;

enum thread_state {
        UNKNOWN = -1,
        ONCPU_USER,
        ONCPU_SYS,
        ONACTIVEQUEUE,
        ONEXPIREDQUEUE,
        UNINTERRUPTIBLE_SLEEP,
        INTERRUPTIBLE_SLEEP,
        ZOMBIE,
        STOPPED,
        INTERRUPTED,
        NR_MICRO_STATES /* must be last */
};

struct microstates {
        enum thread_state cur_state;
        enum thread_state next_state;
        int lastqueued;
        unsigned flags;
        clk_t last_change;
        clk_t timers[NR_MICRO_STATES];
        clk_t child_timers[NR_MICRO_STATES];
};
```

Figure 2: `struct microstates` definition

‘`msa_init_timer(tsk)`’ , which sets all timers to zero, *last_change* to the current time, and *cur_state* to `UNINTERRUPTIBLE_SLEEP`.

‘`msa_next_state(tsk, next_state)`’ sets the *next_state* field in the task to *next*.

‘`msa_flip_expired()`’ which tracks the time when the active and expired queues were last flipped on the current processor (Linux has a separate pair of queues for each processor).

‘`msa_switch(prev_task, next_task)`’ tells the microstate tracking infrastructure that *prev_task* is coming off the processor, and *next_task* is going onto the processor. The timers are updated appropriately. *cur_state* in *prev* is set to *prev*’s *next_state* variable if it is other than *UNKNOWN*; otherwise, *cur_state* is mapped from the task’s state.

‘`msa_start_irq(cpu, irq)`’ is called from the architecture-dependent generic irq handler to mark that the current state is `INTERRUPTED` (it must previously have been `ONCPU`). The code also starts timing how long this IRQ will take.

‘`msa_continue_irq(cpu, oldirq, newirq)`’ is called when the generic interrupt routine starts handling a new interrupt without returning to normal processing. IA64 does this; IA32 code doesn’t use this function.

‘`msa_finish_irq(cpu, irq)`’ marks interrupt handling as over for the time being.

‘Time’ here is some architecture-specific monotonically increasing clock. For uniprocessor systems, we use the in-built high-resolution clock (TSC for IA32 systems, ITC for IA64). For multiprocessor systems, there’s no guarantee that these clocks are synchronised between processors, so for IA32 systems, the code calls ‘`monotonic_time()`’, which is implemented in various ways according to what hardware is available. On IA64 SMP systems, the ITC *is* synchronised between processors; on IA64 NUMA systems, more work is needed to determine an acceptable clock (see section 6.1).

4.3 User-land interfaces

4.3.1 System Call

We provide a new system call, ‘`msa()`’ that returns a snapshot of the current timers, or the children’s timers. It takes as argument the number of timers to retrieve, a pointer to them, and a flag to say whether the child or self timers should be retrieved,

```
int msa(unsigned ntimers, unsigned which, clk_t *timers);
```

The ‘basic’ timers are first in the array; as new timers are added for specific purposes, the existing code can remain backwardly compatible.

4.3.2 Sample output

In addition, a new file ‘`/proc/`*pid*`/msa`’ gives an ASCII representation of a snapshot of the current timer set.

The contents of /proc/*pid*/msa for an instance of xemacs:

```
State:          Interruptible
ONCPU_USER         512707809528
ONCPU_SYS                     0
INTERRUPTIBLE     6723140348728
UNINTERRUPTIBLE     81179247984
INTERRUPTED          8850680192
ACTIVEQUEUE        121728272508
EXPIREDQUEUE           65523052
STOPPED                       0
ZOMBIE                        0
UNKNOWN                       0
```

As you can see, *xemacs* spends most of its time waiting for user interaction, and most of the rest either running on a processor, or waiting for disc I/O.

These times are normalised to nanoseconds before being output. This instance had run for just over two hours, and according to **ps**(1) had used 4 minutes 20 seconds of processor time. Compare this with the measured 8 minutes 41 seconds (not counting the time in INTERRUPTED state).

The reason for the discrepancy is that user time is accrued only when the process is running at the time a scheduler tick occurs. For an interactive process, most of the time it is sleeping. It wakes, runs for less than a tick, then goes back to sleep again without accruing any time to the standard UNIX accounting mechanism.

4.4 Scheduler interaction

4.4.1 Scheduler Queues

The O(1) scheduler has two run queues: the *active* queue and the *expired* queue. Whenever a thread uses its time slice entirely, it is requeued with a new time slice to the expired queue; threads waking from sleep are queued to the active queue. The scheduler runs threads from the active queue until it is empty.

Whenever the active queue is empty, the active and expired queues are swapped, so what was the (now empty) active queue becomes the expired queue; and what was the expired queue becomes the active queue.

In this way, threads that run for a little and then sleep (typical interactive behaviour) are favoured over processor-bound jobs. The microstate accounting infrastructure keeps track of the time at which the queues are swapped. When a thread whose last state was `ONEXPIRED` finally gets some CPU, times are accumulated to both `ONACTIVEQUEUE` (*now - lastflip*) and `ONEXPIREDQUEUE` (*lastflip - last_change*).

It’s guaranteed that all threads on the active queue will get to run before any threads on the expired queue, so this is good enough for keeping track.

4.5 Task migration

When a processor is idle, it looks for work on other processor’s expired queues before looking on active queues. Thus a possible state transition is

`EXPIRED`(0) → `ONCPU`(1) This case is detected by looking to see if

1. the last flip time for the processor the task was last queued on is before the time the task entered the `ONQUEUED` state, and
2. the processor it is about to be run on is different from the processor it was last queued on.

In this case, the `EXPIRED` timer update uses the current time, not the last queueflip time.

4.5.1 Scheduler Ticks

At regular intervals, a timer tick occurs, and decrements the remaining time slice of the current task. When the time slice reaches zero, the task is given a new time slice, and requeued, possibly to the *expired* queue, and the `NEED_RESCHED` flag set to cause the scheduler to be called when the interrupt returns to the interrupted context.

Moreover, the timer tick can occur when the processor is handling another interrupt (in fact this seems to happen fairly often — two or three times a second on an active machine). When the interrupt returns, '`schedule()`' picks a new task to run. '`msa_switch()`' then notes that the current state of the previous task is `INTERRUPTED` and sets the current state of the new task to `INTERRUPTED` as well.

5 Experiences

5.1 Overhead

We measured overhead using LMBench2 ([McV]) on 800MHz single and dual processor Itanium 2 machines, and on a single-processor 2.5GHz Pentium-4 machine. The benchmark results are summarised in Table 2.

The net result is that on IA64, there appears to be a small (less than 5%) increase in context switch overhead. 'Real World' benchmarks like KernBench (which is the time to build a Linux kernel from scratch, using twice as many processes as one has processors), show no significant time differences; in fact on UP Pentium-4, the real time taken to compile a kernel is decreased

5.2 Comparison with '`getrusage()`'

On an otherwise unloaded system, '`getrusage()`' results are identical to the results from the '`msa()`' system call for processor-bound jobs.

The standard getrusage() system call under-reports the actual times spent for a large number of common programs: (here say which processes measured, and the degree of under-reporting, also say what optimisations are suggested by the measurements, if any).

5.3 Other measurements

Other things that the '`msa()`' call allows are direct measurement of scheduler latency, correlation with results from lockmeter, etc., to see what in-kernel features affect real user processes, etc (this paragraph to be reworded when we've got some real results)

6 Future Work

6.1 Better Multiprocessor support

Our current code assumes that the hardware timer (ITC or TSC) is a monotonic clock, regardless of which processor a task is running on, and that clocks are synchronised across all processors in a machine. For small SMP systems, this assumption is close to true with recent Linux.

However, for many architectures, clocks are *not* synchronised with nanosecond precision across the machine. In a NUMA multiprocessor, local clocks need not be synchronised, and it can be expensive to get an agreed-upon time.

An approach to solving this could be to add to the *struct microstate* the processor ID of the processor on which the last state change occurred, and to adjust the timers at migration time. Sleep times could be miscalculated (a task that goes to sleep after running on one processor, is then woken and put onto the local percpu runqueue by a different processor could not allow easily for the clock difference between itself and the original processor)

Another approach would be to use an optimised version of '`gettimeofday()`'.

System	With MSA			Without MSA		
	real	user	sys	real	user	sys
IA64 2P						
IA64 UP						
P4 UP	473.6	381.47	41.35	474.57	381.74	40.3

Table 1: Times for make in a kernel tree, average of 3 runs.

Context Switching, no state, microseconds. Smaller is better. stddev in parentheses

System	2proc	4proc	8proc	16proc	32proc
IA64 2P	1.582(0.154)	1.650(0.037)	1.830(0.043)	1.976(0.024)	2.326(0.196)
IA64 2P +MSA	1.664(0.089)	1.688(0.026)	1.850(0.053)	1.992(0.040)	2.480(0.416)
IA64 UP	1.368(0.018)	1.362(0.050)	1.424(0.071)	1.560(0.078)	1.820(0.209)
IA64 UP +MSA	1.538(0.038)	1.524(0.021)	1.608(0.030)	1.704(0.033)	2.038(0.244)

Context Switching, 4k state, microseconds. Smaller is better.

System	2proc	4proc	8proc	16proc	32proc
IA64 2P					
IA64 2P +MSA	1.664(0.089)	1.688(0.026)	1.850(0.053)	1.992(0.040)	2.480(0.416)

Table 2: LMBench2 selected results

6.2 Tracking User and System time

The code we've written tracks all `ONCPU` time in a single timer. An obviously desirable enhancement would be to account separately for the time spent in running user code and the time in system calls.

6.3 New states

With the infrastructure as described in place, adding new (sleep) states is very easy and cheap. The technique is to define the state in '`include/linux/msa.h`' then add a call to '`msa_next_state()`' just before going to sleep (i.e., about the same place where at present the task state is set to `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`.

Interesting states to track might include:

- Sleeping on Futex.
- Waiting for a page after a page fault
- Waiting in **poll**(2) or **select**(2).

7 Code availability

The code is available as a patch against the Linux kernel from the Gelato down-loads page, `http://www.gelato.unsw.edu.au/patches`

References

[McV] Larry McVoy. LMBench2. http://lmbench.bkbits.net/LMbench2.