To: Distribution

From: John J. Bongiovanni

Date: 01/20/81

Subject: CPU Time Accounting

1. INTRODUCTION

The intent of this MTB is to describe a method for measuring CPU time consumed under the aegis of a Multics process and for attributing this CPU time to appropriate categories. Examples of the latter include virtual CPU time, system overhead time for handling interrupts, etc. The motivation for developing this method is to improve the accuracy of per-process timers. However, only CPU time measuring and accounting will be discussed in this MTB. Full support of more accurate per-process timers will be the subject of a separate MTB. Following some definitions, the current method of measuring and accounting for CPU time on Multics will be described, and the limitations of this method will be discussed. Then the proposed design will be discussed in some detail.

Send comments by one of the following means:

By Extended Mail Facility, on MIT or System M:
Bongiovanni.Multics

By Telephone:
HVN 261-9314 or (617)-492-9314

2. DEFINITIONS

Total CPU Time - the amount of time during which a process is executing instructions on any processor (including DIS instructions). This time includes the following for a process:

- o time executing in user ring
- o time executing ring-0 routines called explicitly by the user process
- o time executing ring-0 routines called implicitly by the user process (e.g., page faults, segment faults, etc.)
- o time executing system overhead functions under the address space of this process (e.g., handling interrupts and connect faults, traffic control processing, etc.)

Note that during any period of time, the length of that period multiplied by the number of processors configured is equal to the sum of total cpu time accrued by all processes active during the period. System idle time is a subset of the total CPU time accrued by the idle processes. Note also that the total CPU time accrued by a given process accomplishing a fixed task is dependent on the load of the system and its configuration (hardware and software).

Virtual CPU Time - the time a process spends executing instructions satisfying explicit requests of user-ring. This time includes the following:

- o time executing in user ring
- o time executing ring-0 routines called explicitly by the user process

Virtual CPU time is intended to be a repeatable measure of user CPU demand which is independent of system load and configuration (other than the speed of the CPU configured).

Overhead CPU Time - the time any process spends executing instructions satisfying explicit requests of a specific system overhead routine or set of routines (e.g., page fault processing, interrupt processing, etc.). This time is somewhat analogous to virtual CPU time in that it does not include the time spent in lower-level routines called implicitly or asynchronously. For example, the overhead CPU time for handling segment faults does not include any CPU time spent handling page faults encountered while handling segment faults, and it does not include any CPU time spent handling interrupts while the interrupted process was handling a segment fault.

3. CURRENT CPU TIME ACCOUNTING

A process running on a CPU is accumulating total CPU time at a rate equal to the speed of the real-time clock in the bootload SCU. A value is stored in pds$cpu_time which, when subtracted from the current value of the real-time clock, yields the total CPU time of the process. When a process is not running on any CPU, it cannot be accumulating total CPU time (or, for that matter, any kind of CPU time). In this case, the locally constant value of total CPU time for the process is stored in the apte. So far, quite reasonable.

The decomposition of this time in virtual CPU time and various overhead CPU times is not so clean. When a process is running on a CPU, it is accumulating virtual CPU time at a rate equal to the speed of the real-time clock. pds$virtual_delta contains all of the known non-virtual CPU time accumulated so far. So virtual CPU time is computed (for a running process) by first subtracting pds$cpu_time from the current value of the real-time clock (yielding the total CPU time for the process), and then subtracting the value of pds$virtual delta. pds$virtual_delta is updated discretely, at the end of an overhead function (e.g., page fault processing stores the total virtual CPU time for the process when it is invoked, subtracts this value from the total virtual CPU time for the process when it is finished, and then updates pds$virtual_delta to reflect the additional overhead).

This has the curious and undesirable effect that virtual cpu time for a process, when sampled over sufficiently small intervals, appears to run backwards. An undesirable result of this effect is that per-process CPU timers can (and do) go off early, and can be noticed in user-ring as having gone off early. Since no CPU time spent in user-ring can be overhead time of any sort, time (fortunately) does not appear to run backwards in user-ring.

An obvious adjustment to reduce the effect of virtual CPU time running backwards is to update pds$virtual_delta more frequently (at a minimum, immediately before the traffic controller decides whether to set off a CPU timer). This will not work because of an interesting aspect of fault processing. This is illustrated by segment fault processing, which is usually considered as load-dependent overhead and not charged as virtual CPU time to the faulting process. However, if there was an error encountered in processing a segment fault (say, access to the segment has been revoked to the faulting process), the CPU time spent handling the segment fault is not considered as overhead, and it is charged as virtual CPU time to the process. This is an exceptionally reasonable practice, and it applies to some other types of faults as well (e.g., boundfaults). However, this practice has the effect that it is not known how to charge segment fault processing time until that processing is completed.

A minor problem with the present method of CPU time accounting is related to the exceptionally limited implementation of overhead time accounting in the fault processor, in which recursive overhead is restricted severely. As a result, the time spent processing a timer runout or connect fault (for example) will be accounted for differently, depending on whether the fault occured during the processing of a page fault. This anomaly probably has no measurable effect in practice.

4. PROPOSED CPU TIME ACCOUNTING

The model embodied in the design is as follows. A process in ring-0 is running (at any time) a nested set of overhead routines (e.g., from top to bottom, connect fault, page fault, segment fault, initiate). Only the overhead routine currently running (the lowest-level routine) is accumulating CPU time; the CPU time accumulated by higher-level routines is frozen at the value when the lower-level routine was invoked. At the completion of an overhead routine, the time accumulated by it may or may not be propagated to the next higher level. The highest level corresponds to virtual CPU time.

The implementation is as follows. pds$cpu_time_stack is an array (0:N, where N is sufficiently large to handle recursion--7 should be enough) which implements a stack of CPU times. pds$cpu_time_frame contains the index of the current frame, with 0 the index of the frame corresponding to virtual CPU time. When a process is running on a CPU, the CPU time accumulated by the routines owning the current stack frame is defined as the current value of the real-time clock minus the value of pds$running_clock_base. The CPU time accumulated by routines owning other stack frames is defined as the value of pds$cpu_time_stack (level), where level is the index of the stack frame owned by the subject routines. In particular, virtual CPU time is defined as

clock - pds$running_clock_base if the process is running and if pds$cpu_time_frame = 0;

pds$cpu_time_stack (0) if the process is running and if pds$cpu_time_frame ^= 0.

All cells in the pds defined above must reside in the wired portion of the pds.

The following ring-0 wired subroutines will be provided to manipulate the stack. By convention, only these routines will manipulate the stack.

cpu_time_manager$push - pushes another frame onto the stack, clears the cell represented by this frame to zero, and adjusts pds$running_clock_base.

cpu_time_manager$pop - pops a frame from the stack, adjusts pds$running_clock_base, and returns to the caller the CPU time accumulated while running on the popped frame.

cpu_time_manager$pop_and_propagate - pops a frame from the stack, adjusts pds$running_clock_base, adds the CPU time accumulated while running on the popped frame to that accumulated while running on the (now) current frame, and returns to the caller the CPU time accumulated while running on the popped frame.

cpu_time_manager$usage - returns to the caller the CPU time accumulated while running on the current frame.

The cpu_time_manager will perform all operations using inhibited code, to ensure the integrity of those operations. It is the responsibility of the caller to update any cells reflecting overhead time, as appropriate (e.g., the segment fault processing routines will update sst$cpu_sf_time, as they do presently).

A number or error conditions can occur, all of which indicate programming errors in ring-0. Correspondingly, all of these errors, when encountered, will result in a system crash. These errors include stack overflow and underflow. Additionally, the stack level should always be zero when running in other than ring-0. The ring alarm register will be used to trap violations of this principle.

The following is a summary of the benefits of the proposed method of CPU time accounting, compared to the present method:

- o Metering of overhead time is more precise.
- o All measured times are monotonic with real time (i.e., they run forwards). Unfortunately, times are still subject to quantum jumps.
- o CPU time measurement and attribution is centralized in a single routine. This will greatly assist in improving the accuracy of per-process timers. For example, when quantum jumps in virtual CPU time occur, the traffic controller can be called to check for expired timers.
- o Accounting for CPU time by fault processing is much cleaner and more understandable.

5. OBSOLETE CELLS

The following cells in the pds will become obsolete with implementation of this design:

- o cpu_time
- o virtual_delta
- o virtual_time_at_eligibility
- o temp_1
- o temp_2
- o time_1
- o time_v_temp
- o fim_v_temp
- o fim_v_delta