The *Virtual Time Subsystem* for Linux Kernel

A patch is available as VirtualTimeKernel [*]

Jiaqi Yan[†]
Illinois Institute of Technology
West 31st Street, Chicago
Illinois, USA
jyan31@hawk.iit.edu

Dong (Kevin) Jin
Illinois Institute of Technology
West 31st Street, Chicago
Illinois, USA
dong.jin@iit.edu

ABSTRACT
TODO

Categories and Subject Descriptors
I.6.3 [**Simulation and Modeling**]: Application—*Miscellaneous*; D.4.8 [**Operating Systems**]: Performance—*Measurement, Simulation*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms
Operating System

Keywords
Virtual Time, Linux Kernel, Emulation

1. INTRODUCTION
What is virtual time? The short answer is: abstraction of time. When we abstract out the physical hardware, we can build an operating system that manage them intelligently; when we abstract out the behavior of operating system, we can run guest operating system inside a completely different host operating system; when we abstract the time of a process, what we can do is only limited by our imagination.

2. THE FEATURES OF VIRTUAL TIME SUBSYSTEM
Virtual time, in essential, abstract time for processes from the operating system they reside on. With this abstraction, now we can dilate a process's clock or freeze a process's clock.

2.1 Time Dilation
When a process is `dilated`, it's clock advances by a factor faster or slower than wall-clock time. We borrow the definition of **Time Dilation Factor** from [8]: the ratio of the time passing rate in physical world to the time passing rate perceived by process or virtual machine(VM). For example, when TDF equals 10, as 10 seconds elapsed in physical world, VM only perceives that 1 second passed.

An interesting application of time dilation is to emulate a link with 1 Gbps bandwidth while we only have a 100 Mbps link. Ideally, when a process or a VM keeps sending data through this link for 1 seconds, it actually take 10 seconds in real world and hence 1 Gb data is transferred. In other words, the dilated process think the network bandwidth is 1 Gbps.

2.2 Freeze
Processes in any operating systems can be stopped and resumed. Usually it is through sending signal and handling signal. Imagine we not only stop the execution of a process, but also stop *its* clock. Here "its" is not accurate; all processes inside a particular operating system share a global software clock. For example, process in Linux queries current time by `gettimeofday` system call. Now with virtual time, process finally have its own clock and we are able to stop particular processes' clocks.

What a **freeze** does is both stop the execution of a process and the advance of its clock. A **unfreeze** does the opposite things: resume the execution and the clock. A once frozen process will not counting the freezing duration into its clock. A nontrivial application of freeze and unfreeze is to cooperate simulation and emulation of a system. For simulation, there is no concept of real time; time advances according to software's wish. So even if a calculation in simulation may take a great deal of time, for example solving a multiple variable differential equation, this mount of time is not counted inside the simulating model. Emulation, on the other hand, runs real software or on real hardware in real time. As a result it uses physical system's real clock. As an illustration, consider this scenario: in a network emulator, host `sender` wants to transmit a certain amount of measured data to another host `receiver`; these measured data are actually gathered from simulator *s* and it takes a fair amount of time for simulator to do the calculation, during which we need to stop the emulation as well as emulation's time. Just stop both hosts by sending `SIGSTOP` is not enough because time may be a relevant variable in emulation. For example,

[*]https://github.com/littlepretty/VirtualTimeKernel
[†]Also the implementer

if a process who maintaining a timer is just stopped execution, possibly it will miss a timeout during its halt and when it resumes, it is unable to trigger the action associated with the missed timer.

3. THE NAMESPACE STORY

3.1 Linux Namespace

The first namespace `mnt` namespace was introduced in Linux 2.4.19. Since then, more type of namespaces are implemented and currently we have totally 6 namespaces:

- Mount namespaces(`CLONE_NEWNS`), the first born, isolate the set of file system mount points seen by a group of process [9]. It is a more secure and flexible alternation of `chroot jail`. Related system calls are `mount()` and `unmount()`
- UTS namespaces(`CLONE_NEWUTS`) may be the most simple one to implement. It make containers to be able to have its own `nodename` and `domainname` [9]. Related system calls are `uname()`, `setnodename()` and `setdomainname()`
- IPC namespace(`CLONE_NEWIPC`) isolate System V IPC objects and POSIX message queues, e.g. `sem, shm, msg`.
- PID namespace(`CLONE_NEWPID`) isolate the process ID number space. Between different PID namespaces, processes can have the same PID. It is used to enable containers to be migrated to different host system while still keeping the same PID for processes inside that container. Processes inside a particular container, following the tradition of Linux(Unix) holds unique, sequentially assigned ID number. [10]
- Network namespaces(`CLONE_NEWNET`) isolate the entire network stack in Linux kernel. With help of network namespace, container can have its own network device, addresses, ports, routes, firewall rules, etc [7]. It is widely applied in network emulation, Mininet [3] for example, to create isolated network hosts.
- User namespaces(`CLONE_NEWUSER`) are the most complicated namespace, taking five years to complete, spanning from Linux 2.6.23 to Linux 3.8 [11]. The resource user namespace provided to process is its user ID with root privilege. A process can do full privilege operations inside its user namespace, even like creating other types of namespaces. Outside it user namespace, process can only do normal unprivileged operations.

3.2 Clock Namespaces

Virtual time, by its definition, seems to fit into the category of namespace: it isolates time from the system wide wall clock so that a process can advance faster or slower by an offset or by a constant factor. If so, virtual time should actually be called **Clock Namespace**. However, a problem hangs here: virtual time is really a per process feature; there is no need to share time by a group of processes, like sharing network or file system. In other word, what can we do when we can identify that a group of processes belong to the same Clock namespace? We will see an satisfactory answer in section 5.3.

4. THE SOFTWARE INTERFACE

This section present 2 different software interfaces for virtual time. The first one will be deprecated.

4.1 Through Additional System Calls

Probably the most straightforward and destructive way of exposing virtual time to user space is through modifying existing system calls and adding new system calls. Our first working implementation use this method to interfacing clock namespace [12, 13]. To enable the virtual time perception to processes, we added/modified the following new system calls.

- `unshare()` system call with flag `CLONE_NEWTIME`[1] is modified. It is used by container-based emulators, such as Mininet, to create emulated nodes. When `CLONE_NEWTIME` is set, `unshare()` creates a new process with a default TDF 1 in a different namespace from its parent process.
- Newly added system call `set_dilaiton()` offers an interface to change the TDF of a process.
- Newly added system call `freeze()` and `unfreeze()` comes as a combo to control the execution and pause of a process. Here pause is more than just send a signal `SIGSTOP` to process so that is stop executing; the process's clock, on freezing, actually is detached from OS's clock and is stopped. See section 5.3 for detail.

In container based network emulation, usually we want all processes inside the same container having the same time dilation factor. In other words, to keep the persistence, `set_dilation()` should better set time dilation factor for a group of processes. We can take advantage of the identification of the *network host*, who is usually the root of the process tree inside a particular container. We only invoke the `set_dilation()` once for this *root* process; as a leader in the container, it cascade `set_dilation` further down to the process tree so that every process's TDF will be updated.

4.2 Through Proc Virtual File System

It is also possible to implement virtual time without adding system calls. Virtual file system provides a interface between kernel and user space. Since virtual time is a per process thing, it is more appropriate to create `/proc` file entry under the directory that specific to process. Virtual time interface thus consists of two extra file entry under `/proc/$pid`.

- `/proc/$pid/dilation` can be read and written so that process `$pid` can enter virtual time(write non-zero value to zero TDF), exit virtual time(write zero to non-zero TDF) and change to new TDF.
- `/proc/$pid/freeze` expects a boolean value and freeze or unfreeze process `$pid` according to the written value.

[1] `CLONE_NEWTIME` takes value `0x02000000`, which previously unused by `CLONE_STOPPED`

Change time dilation factor from zero to non-zero value will detach the process's clock from system clock, thus entering virtual time. Conversely, setting a process's TDF to zero will make it exit virtual time. The traditional `unshare()` system call together with `CLONE_NEWTIME` flag is kept as an alternative for entering virtual time.

While being more elegant interface and persisting with the philosopher of Proc virtual file system, this interface is doomed with more overhead due to that introduced by necessary `open(), write()` and `close()` operations. However, in practice these operations usually should not be invoked very often.

5. ALGORITHMS AND IMPLEMENTATIONS

In this section, we first present the extra 52 bytes variables added to process data structure. Then the pseudo-implementation/algorithms are elaborated in order.

5.1 Additional Data Structure

Following new fields are added into `task_struct` so that a process can have its own perception of time.

- `dilation` represents the time dilation factor of a time-dilated process.
- `virtual_start_ns` represents the starting time that a process detaches from the system clock and uses the virtual time, in nanoseconds.
- `physical_start_ns` represents the starting time that a process detaches from the system clock and uses the virtual time, in nanoseconds. It differs from `virtual_start_ns` when process's dilation changes from non-zero value to another non-zero value. See section 1 for details.
- `virtual_past_ns` represents how much virtual time has elapsed since the last time the process requested the current time, in nanoseconds.
- `physical_past_ns` represents how much physical time has elapsed since the last time the process requested the current time, in nanoseconds.
- `freeze_start_ns` represents the starting moment that a process or process group is frozen. It is always zero when the process is not frozen.
- `freeze_past_ns` represents the accumulative time, in nanoseconds, that a running process or process group was put into freezing state. A special process *leader* is responsible for maintaining and populating this variable for its members.

5.2 Algorithm/Implementation for Change Time Dilation Factor

Setting a new dilation for a process may falls into several possible cases, as shown in the condition clause of Algorithm 1. The difficulty here is that setting TDF may happen during the lifetime/execution of a process and user should not be bothered with handling the virtual time with old dilation. So, in the 3rd case following, kernel should do the old dilation's virtual time keeping.

- We return immediately when user repeat set an old time dilation factor.
- Then we handle the case that TDF was or is going to be zero; corresponding actions are either enter or exit virtual time space by invoke helper functions `init_virtual_time()` and `clean_up_virtual_time()`, defined in Algorithm 3.
- Finally the primary case goes when both previous and future TDF is non-zero value. We first clear *dilation* and *virtual_start_ns* so that we can get undilated time moment *now*. Then the algorithm can do an audit for the old dilation: we calculate the time offset from *now* to the last moment that *physical_past_ns* is updated; *virtual_past_ns* thus advances by a factor of physical past time according to the old dilation. At last, we can say goodbye to previous dilation factor and update *physical_start_ns* to *now*, minus the time that this process does not perceived at all $freeze\_past\_ns$, and dilation to the new TDF. As we will see in subsection 2, if a process does not play the role of leader, how long it is frozen, the time it should not perceived, is obtained from its leader's $freeze\_past\_ns$.

Finally, we need to cascade the update-TDF operation to all tsk's children.

5.3 Algorithm/Implementation for Freeze and Unfreeze

At first, we intended to let *process group leader* deals with the freeze part of time keeping for all its members. Every process can just access its process group leader's frozen time when doing virtual time keeping. This design would be simple and straightforward to implement if the field `group_leader` in `task_struct` is, by our wish and its naming, the leader/root of a process group/sub-tree. Unfortunately, it is not.[2]

Anyway, we can implement this process group leader from scratch. However, the modification for this leader may span many different source files in kernel; it may even worsen the already complicated process relationships. So far this design doesn't smells very good. Perhaps freeze/unfreeze should be restricted to per process granularity.

On the other hand, freeze and unfreeze in most cases should be done with respect to a entire process group. As an illustration, consider the case that we wants to freeze a container in the network emulation. Recall that container here usually plays the role of a lightweight virtual machine. Therefore when we want to freeze this container, we actually wants to freeze all the processes(usually emulates applications running inside a VM) inside this container. For example, when we freeze host h_1(container), we should also stop the clock of `ping`, `iperf` or other applications running inside $h1$. Obviously, for applications running in $h1$, we can make $h1$ their leader. Doing freeze/unfreeze in group, we can ensure that every member process will access the same frozen duration. So instead of let every process maintain its own frozen time and, maybe, communicate with other members in the same group for consistency/synchronization, we'd better make the

[2]It turns out to be the thread group leader.

Algorithm 1 Set Time Dilation Factor

```
1: function SET_DILATION(tsk, new_tdf)        /* Set new_tdf to process tsk */
2:     old_tdf ← tsk.dilation
3:     vsn ← tsk.virtual_start_ns
4:     if new_tdf = old_tdf then
5:         return 0        /* do nothing */
6:     else if old_tdf = 0 then
7:         INIT_VIRTUAL_TIME(tsk, new_tdf)        /* enter virtual time with new_tdf */
8:     else if new_tdf = 0 then
9:         CLEAN_UP_VIRTUAL_TIME(tsk)        /* exit virtual time */
10:    else if new_tdf > 0 then        /* real works here */
11:        tsk.dilation ← 0
12:        tsk.virtual_start_ns ← 0
13:        __getnstimeofday(ts)        /* ts: a timespec temp */
14:        now ← timespec_to_ns(ts)
15:        tsk.virtual_start_ns ← vsn
16:        delta_ppn ← now − tsk.physical_past_ns − tsk.physical_start_ns − tsk.freeze_past_ns
17:        delta_vpn ← delta_ppn / old_tdf
18:        tsk.virtual_past_ns += delta_vpn
19:        tsk.physical_start_ns ← (now − tsk.freeze_past_ns)
20:        tsk.physical_past_ns ← 0
21:        tk.dilation ← new_tdf
22:        return 0
23:    else
24:        return -EINVAL
25:    end if
26:    for all tsk's child do
27:        SET_DILATION(child)
28:    end for
29: end function
```

leader a central controller of the freeze time keeping. Thus everyone can just query the leader when needed, for example, in Algorithm 1 and Algorithm 3. The only drawback is that we waste 16 bytes for every process other than the leader.

We must admit that the above discussions, and thus our virtual time's freeze implementation, is highly tailored for the purpose of container based emulation. After all this is our motivation of implementing freeze/unfreeze inside virtual time. Ideally, freeze/unfreeze should be done in the unit of per process. Sometimes we need this fine granularity and sometimes it makes more sense to do it in a bundle.

In short, how can we implement freeze/unfreeze with flexible scope as well as elegant design and implementation?

Here enters **Clock namespace**.

The implementations for freeze/unfreeze is shown in Algorithm 2. First, after *kill* a group of processes, we make a record of current moment; this record is used to calculate how long this group is frozen when we unfreeze it. It is a little bit subtle that when unfreeze, sending `SIGCONT` to all processes is behind the time keeping part. The reason is that if we resume the process group first, by any chance, an unfreeze process may be schedule to run and possibly query time before we populate the $freeze\_past\_ns$ to entire container.

5.4 Algorithm/Implementation for Virtual Time Keeping

Firstly, we have 2 helper functions to initialize and cleanup virtual time.

- `clean_up_virtual_time()` reset all relevant variables to zero, which is also the default value for these variables when a `task_struct` is created.
- Since physical and virtual time will fork as we enter virtual time, `init_virtual_time()` initialize both to the current moment *now* and then set a non-zero dilation to the process.

Unsurprisingly, `do_virtual_time_keeping()` maintains virtual time including dilation and freeze. It first call `update_physical_time()` to get the physical duration since last time we request time, excluding the frozen part; then `update_virtual_time()` dilates this physical duration. At the ending of both helper methods, we finish the timekeeping by updating *physical_past_ns* and *virtual_past_ns*. As an example, we show in system call `gettimeofday` the way to dilate system's wall clock time.

6. MISCELLANEOUS PARTS

Our primary motivation of virtual time system is to benefit network emulation. This section discuss issues we come across when virtual time system cooperate with other part of the operating system (even sometimes, mingling with applications is necessary). This section also serve as a guideline

Algorithm 2 Freeze and Unfreeze Process

```
1: function FREEZE(tsk)
2:     kill_pgrp(task_pgrp(tsk), SIGSTOP, 1)
3:     __getnstimeofday(ts)        /* timespec ts */
4:     now ← timespec_to_ns(ts)
5:     tsk.freeze_start_ns ← now
6: end function
7:
8: function POPULATE_FROZEN_TIME(tsk)
9:     for all child of tsk do
10:        child.freeze_past_nsec ← tsk.freeze_past_nsec
11:        POPULATE_FROZEN_TIME(child)
12:    end for
13: end function
14:
15: function UNFREEZE(tsk)
16:    __getnstimeofday(&ts)        /* timespec ts */
17:    now ← timespec_to_ns(ts)
18:    tsk.freeze_past_ns += now − tsk.freeze_start_ns
19:    tsk.freeze_start_ns ← 0
20:    POPULATE_FROZEN_TIME(tsk)
21:    kill_pgrp(task_pgrp(tsk), SIGCONT, 1)
22: end function
```

about how virtual time's user should consider tweeting(or **NOT**) the OS kernel in their expertise fields.

6.1 Bypass virtual Dynamic Shared Object

One issue we notice is that the 64-bit Linux kernel running on Intel-based architectures provides the Virtual Dynamic Shared Object (vDSO) and the `vsyscall` mechanism to reduce the overhead of context switches caused by the frequently invoked system calls, for example, `gettimeofday()` and `time()` [6]. Therefore, applications may bypass our virtual-time-based system calls unless we explicitly use the `syscall` function. Our solution is to disable vDSO with respect to `__vdso_gettimeofday()` in order to transparently offer virtual time to applications in the containers. However, as we seen in the very next subsection, sometimes it is necessary to mingle with application so that the application's time goes with virtual clock.

6.2 Inconsistent Time

Let's start with a widely used application in network emulation: `ping`. The goal of `ping` is simply estimate the **network delay** without affected by the processing delay on host ends. Basically, it [2] works by sending and receiving *ICMP* packet with `SOCK_RAW` socket [4] in the communication domain of *IPV4* network layer protocol[3].

By using raw socket that providing raw access to network layer, `ping` can bypass transport layer processing so that it can directly and manually parse the receiving ICMP packet. As a result, at the sending side, ping-sender will fill only 8 bytes of the data portion with the Unix `timeval`, which the sender obtained from system call `gettimeofday()`. Along with the process ID and an ascending sequencing number, this very tiny `ICMP ECHO REQUEST` packet will be handed to IP layer.

[3]Format `AF_INET` for IPV4 `ping`

Correspondingly, the receiving side receives a message from the raw socket. This message are sometimes called ancillary data because it is not a part of the socket's payload but a sequence of `cmsghdr` structure with appended data [1]. Ideally, `ping` should parse this raw message to decide if it belongs to itself and calculate latency by querying system time again and subtract it from sending time extracted from the ICMP packet.

So far everything goes perfectly for both sender and receiver applications. If in an virtual time emulation where all the hosts are using virtual time, since applications inherit "virtual time" from hosts, any `gettimeofday()` system call mentioned above are covered by virtual time subsystem. Therefore the measured delay should be either dilated (by TDF) or adjusted (subtract frozen time in the case of freeze/unfreeze). However, in our benchmark experiments, if we freeze ping for 1 second, the network delay will increase 1 second very time freeze/unfreeze pair occurs. In other words, either the sending or the receiving side is not using virtual time.

The reason lies in just one line of code: `ping` set the `SO_TIMESTAMP` option unless the user want to see **full user-to-user latency**. With `SO_TIMESTAMP` option on, socket will report the time stamp via `recvmsg()` in the control message as struct `timeval`[4]. More specifically, when raw socket (`raw_prot`'s virtual function `raw_recvmsg()`) is receiving a packet *skb* from `skb_recv_datagram()`, `__sock_recv_timestamp()` will query system time if any of `SO_TIMESTAMP`, `SO_TIMESTAMPNS` or `SO_TIMESTAMPING` flags is set in socket options. Time returned by `ktime_get_real()` will be put inside the `SCM_TIMESTAMP` filed at `SOL_SOCKET` level in the receiving message's ancillary

[4]Higher resolution time stamp in nanoseconds is possible with `SO_TIMESTAMPNS` option. Moreover, general timestamping(`SO_TIMESTAMPING`) can be done on reception, transmission or both with multiple timestamp sources

data, e.g. `CMG` sequence. `ping` knows this timestamping feature in socket and prefers this time value embedded inside socket message to getting time by itself when calculating round trip time.

6.3 Dilate Bandwidth Rate in Traffic Control

One particular case related to network emulation is the usage of `tc`, a network quality-of-service control module in Linux [5]. For instance, one can use `tc` to rate-limit a link to 100 Mbps using Hierarchic Token Bucket (HTB) `qdisc`. However, as a network module in Linux, `tc` does not reference Linux kernel's time as the way user application does. Therefore, `tc` is transparent to our virtual time system. If the TDF is set to 8, the link bandwidth would be approximately 800 Mbps from the emulated hosts' viewpoints as we observed from `iperf3` application.

One way to solve this problem is to modify the network scheduling code in kernel to provide `tc` with a dilated traffic rate. In the earlier example with TDF set to 8, the experiment will run 8 times slower than the real time. To emulate a 100 Mbps link, `tc` module should configure rate limit as $rate/TDF = 12.5$ Mbps.

7. EXPERIMENTS

7.1 Functional Tests

7.2 Overhead Measurements

8. CONCLUSIONS

TODO

9. ACKNOWLEDGMENTS

TODO

10. REFERENCES

[1] CMSG(3). `http://man7.org/linux/man-pages/man3/cmsg.3.html`. [Online; accessed 10-Nov-2015].

[2] iputils. `http://www.skbuff.net/iputils/`. [Online; accessed 10-Nov-2015].

[3] Mininet: An instant virtual network on your laptop (or other PC). `http://mininet.org/`. [Online; accessed 6-Oct-2015].

[4] SOCKET(2). `http://man7.org/linux/man-pages/man2/socket.2.html`. [Online; accessed 10-Nov-2015].

[5] W. Almesberger. Linux traffic control: implementation overview. In *Proceedings of 5th Annual Linux Expo*, pages 153–164, Raleigh, NC, 1999.

[6] J. Corbet. On vsyscalls and the vDSO. `http://lwn.net/Articles/446528/`, 2011. [Online; accessed 6-Oct-2015].

[7] J. Edge. Namespaces in operation, part 7: Network namespaces. `https://lwn.net/Articles/580893/`, 2013. [Online; accessed 6-Oct-2015].

[8] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 1–2, New York, NY, USA, 2005.

[9] M. Kerrisk. Namespaces in operation, part 1: namespaces overview. `https://lwn.net/Articles/531114/`, 2013. [Online; accessed 6-Oct-2015].

[10] M. Kerrisk. Namespaces in operation, part 3: PID namespaces. `https://lwn.net/Articles/531419/`, 2013. [Online; accessed 6-Oct-2015].

[11] M. Kerrisk. Namespaces in operation, part 5: User namespaces. `https://lwn.net/Articles/532593/`, 2013. [Online; accessed 6-Oct-2015].

[12] J. Yan and D. Jin. A virtual time system for linux-container-based emulation of software-defined networks. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 235–246, New York, NY, USA, 2015. ACM.

[13] J. Yan and D. Jin. VT-Mininet: Virtual-time-enabled Mininet for Scalable and Accurate Software-Define Network Emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 27:1–27:7, New York, NY, USA, 2015. ACM.

Algorithm 3 Time Dilation Algorithm

```
1: function INIT_VIRTUAL_TIME(tsk, tdf)
2:     if tdf > 0 then          /* tsk.dilaton = 0 before initialization */
3:         __getnstimeofday(ts)          /* ts: a timespec temp */
4:         now ←timespec_to_ns(ts)
5:         tsk.virtual_start_ns ← now
6:         tsk.physical_start_ns ← now
7:         tsk.dilation ← tdf
8:     end if
9: end function
10:
11: function CLEAN_UP_VIRTUAL_TIME(tsk)
12:     tsk.dilation ← 0
13:     tsk.physical_start_ns ← 0
14:     tsk.physical_past_ns ← 0
15:     tsk.virtual_start_ns ← 0
16:     tsk.virtual_past_ns ← 0
17:     tsk.freeze_start_ns ← 0
18:     tsk.freeze_past_ns ← 0
19: end function
20:
21: function UPDATE_PHYSICAL_TIME(tsk, ts)
22:     now ← timespec_to_ns(ts)
23:     delta_ppn ← now − tsk.physical_past_ns − tsk.physical_start_ns
24:     delta_ppn − = tsk.freeze_past_ns
25:     tsk.physical_past_ns + = delta_ppn
26:     return delta_ppn
27: end function
28:
29: function UPDATE_VIRTUAL_TIME(delta_ppn, tdf)
30:     if tdf ≠ 0 then
31:         delta_vpn ← delta_ppn / tdf
32:         tsk.virtual_past_ns + = delta_vpn
33:     end if
34: end function
35:
36: function DO_VIRTUAL_TIME_KEEPING(tsk, ts)          /* timespec ts will be modified */
37:     tdf ← tsk.dilation
38:     if tdf > 0 then
39:         delta_ppn ← UPDATE_PHYSICAL_TIME(ts)
40:         UPDATE_VIRTUAL_PAST_TIME(delta_ppn, tdf)
41:         virtual_now ← tsk.virtual_start_ns + tsk.virtual_past_ns
42:         virtual_ts ← ns_to_timespec(virtual_now)
43:         ts.tv_sec ← virtual_ts.tv_sec
44:         ts.tv_nsec ← virtual_ts.tv_nsec
45:     end if
46: end function
```