

# kvm-tlc

Copyright © 2010 vasabiLab

## **kvm-tlc: The Thread-Base Live Checkpointing Implementation on the Kernel-based Virtual Machine**

### **A User Manual**

version 0.1

by

Kasidit Chanchio and Vasinee Siripoonya

Thread-base live checkpointing (TLC) is the term coined in Vasinee Siripoonya's Thesis [1] for the state collection operation of the virtual machine software, which spawns a separate thread, namely the checkpointing thread, to collect the state of a virtual machine concurrently with the virtual machine's computation. While the virtual machine performs computation on one thread, the checkpoint thread repeatedly collects state of the memory and other devices of the virtual machine and store them to a file.

TLC allows the virtual machine to utilize the underlying multiprocessor of the host computer. On a checkpointing event, It relieves loads off the current processors the virtual machine is running on by assigning the checkpointing thread to an available idle processor. Please see [1] and [2] for more information.

The TLC has been integrated into the Kernel-base Virtual Machine (KVM) source code [3] by Vasinee Siripoonya as a part of her Master Thesis at Thammasat University. The extended KVM source code is called “kvm-tlc”. In particular, we build kvm-tlc based on the kvm-88.tar.gz distribution.

This document intends to give a short guideline on how to compile and run the kvm-tlc source code released at <http://vasabilab.cs.tu.ac.th/projects.html>. Note that our distribution is an experimental prototype. Please beware to use at your own risks.

## **1. Basic Requirements**

kvm-tlc can run on any hardware platform that support Linux and KVM. However, for desirable checkpointing performance, the underlying computer should have more than one processor core and have enough physical memory to facilitate the guest OS and applications of virtual machines.

In our test, we run kvm-tlc on a desktop computer with Quad Core CPU that has support VT-x hardware assist virtualization. The machine has 4GB RAM, 500GB disk space, and run Ubuntu 9.10.

## **2. Compilation and Installation**

After obtaining the kvm-tlc software from [4], unzip and extract the tar file into a directory, and do the followings:

```
$ gzip -d kvm-88_tlc.tar.gz
$ tar xvf kvm-88_tlc.tar
$ cd kvm-88_tlc
$ ./configure
$ make
```

To install the compiled binary to public installation location, switch to super user status.

```
$ sudo su
# make install
```

### 3. Testing

To run the software, do exactly the same way when you want to run the KVM software. Please consult [3] if you don't know how. We use the following command to run ours:

```
$ /usr/local/bin/qemu-system-x86_64 -m 512M -hda myvm1.ovl -localtime -net nic -net user
```

where “myvm1.ovl” is our “qcow2” formatted disk image file. The figure 1 below shows a fedora 11 guest OS running as a result. We also runs the “bt.B” benchmark from the NAS parallel benchmark on the guest OS as shown in the Figure.

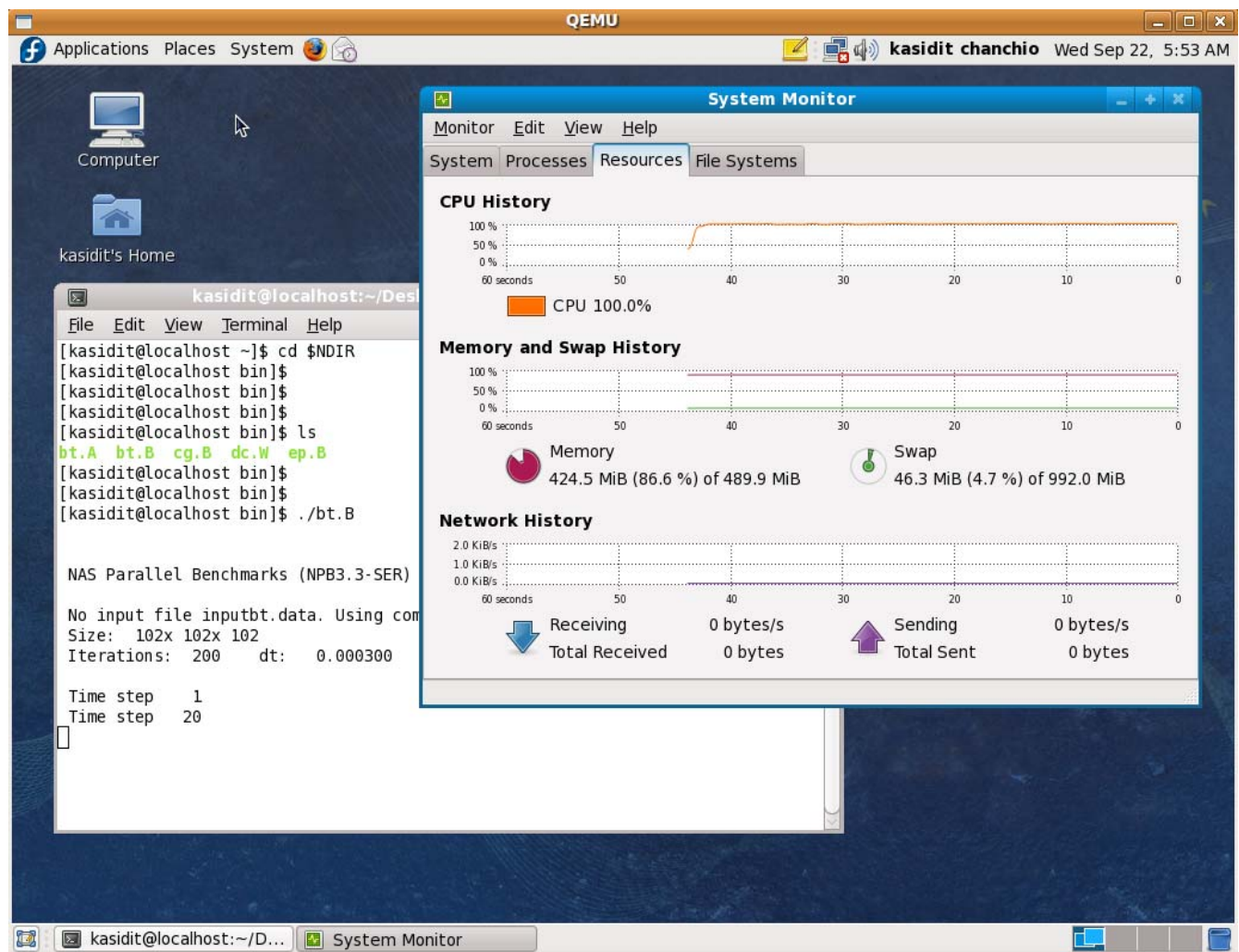


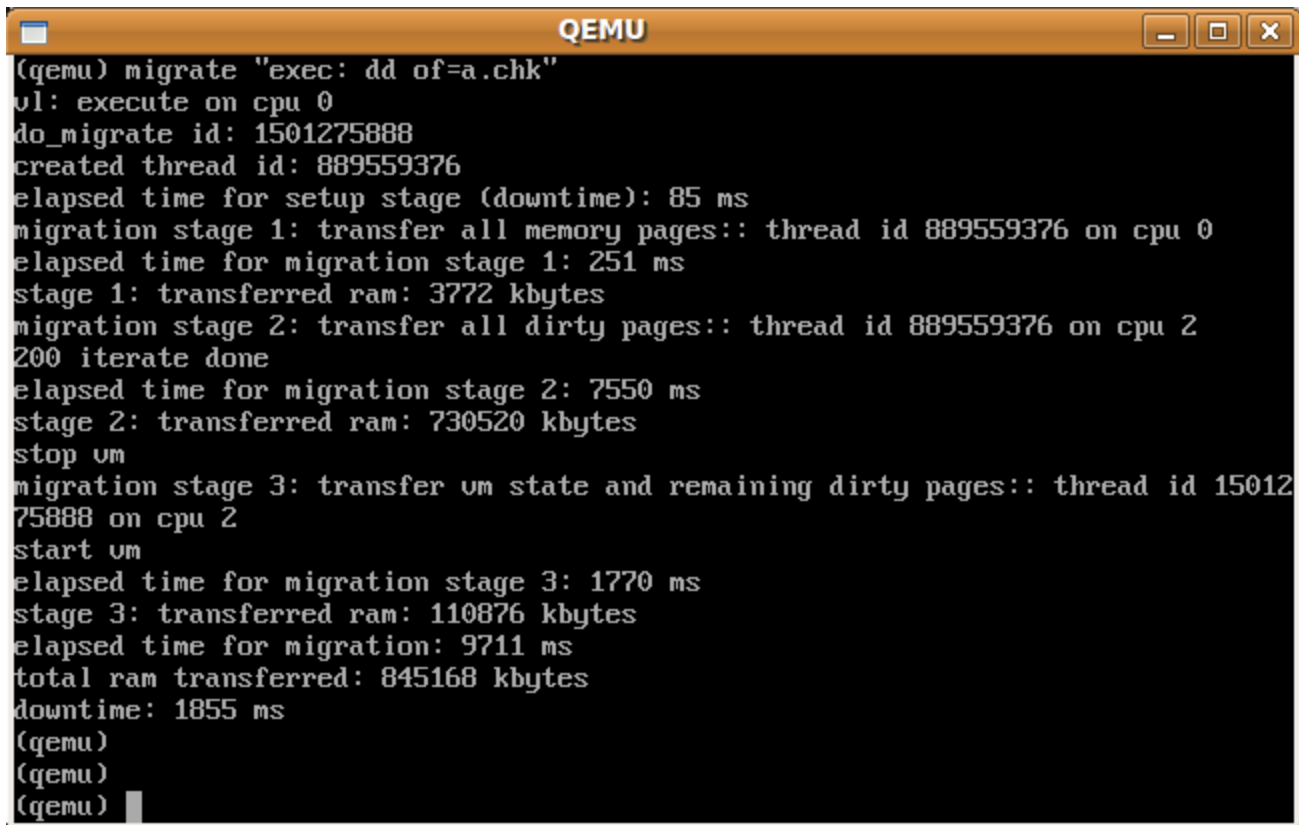
Figure 1: fedora 11 guest OS running on kvm-tlc

## 4. Migration

While the guest virtual machine is running, you can tell it to checkpoint the virtual machine's state by pressing Ctrl-Alt-2 to switch the screen of the virtual machine to KVM monitor and enter the following command

```
(qemu) migrate "exec: dd of=a.chk"
```

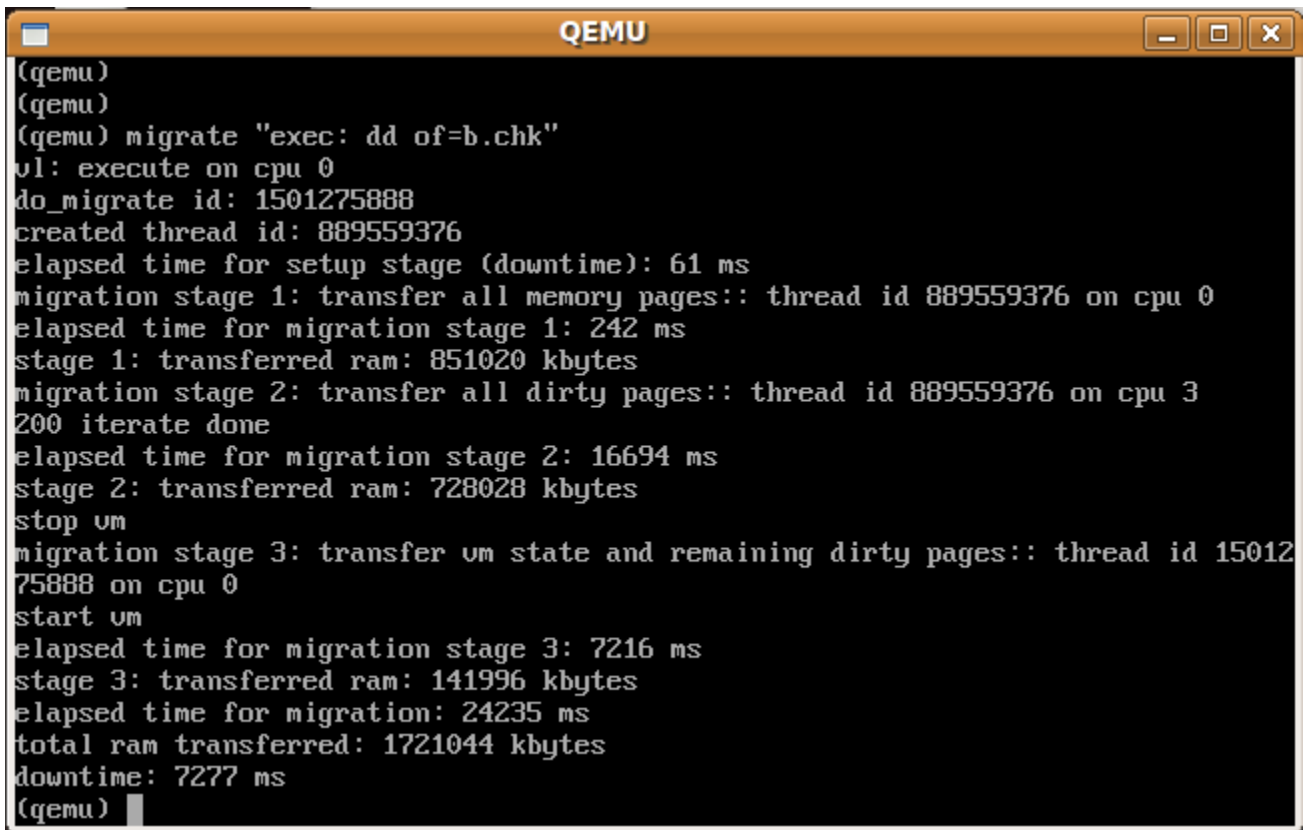
where a.chk is the name of your checkpoint file as shown in Figure 2.

A screenshot of a QEMU monitor window. The window has a title bar with the text 'QEMU' and standard window control buttons (minimize, maximize, close). The main area is a black terminal with white text showing the execution of the 'migrate' command. The output details the migration process, including stage 1 (transfer all memory pages), stage 2 (transfer all dirty pages), and stage 3 (transfer vm state and remaining dirty pages). It also shows the total ram transferred and the downtime.

```
(qemu) migrate "exec: dd of=a.chk"
vl: execute on cpu 0
do_migrate id: 1501275888
created thread id: 889559376
elapsed time for setup stage (downtime): 85 ms
migration stage 1: transfer all memory pages:: thread id 889559376 on cpu 0
elapsed time for migration stage 1: 251 ms
stage 1: transferred ram: 3772 kbytes
migration stage 2: transfer all dirty pages:: thread id 889559376 on cpu 2
200 iterate done
elapsed time for migration stage 2: 7550 ms
stage 2: transferred ram: 730520 kbytes
stop vm
migration stage 3: transfer vm state and remaining dirty pages:: thread id 15012
75888 on cpu 2
start vm
elapsed time for migration stage 3: 1770 ms
stage 3: transferred ram: 110876 kbytes
elapsed time for migration: 9711 ms
total ram transferred: 845168 kbytes
downtime: 1855 ms
(qemu)
(qemu)
(qemu)
```

Figure 2: launching thread-base checkpointing operations on qemu monitor.

After creating the first checkpoint file, we launch another “migrate” command to save the virtual machine's state to a file “b.chk” (in Figure 3). So, there are two checkpoint files created so far. You should notice that during the checkpoint operations, the applications on the virtual machine run normally. You can also interact with the GUI of the guest during that time.



```
(qemu)
(qemu)
(qemu) migrate "exec: dd of=b.chk"
vl: execute on cpu 0
do_migrate id: 1501275888
created thread id: 889559376
elapsed time for setup stage (downtime): 61 ms
migration stage 1: transfer all memory pages:: thread id 889559376 on cpu 0
elapsed time for migration stage 1: 242 ms
stage 1: transferred ram: 851020 kbytes
migration stage 2: transfer all dirty pages:: thread id 889559376 on cpu 3
200 iterate done
elapsed time for migration stage 2: 16694 ms
stage 2: transferred ram: 728028 kbytes
stop vm
migration stage 3: transfer vm state and remaining dirty pages:: thread id 15012
75888 on cpu 0
start vm
elapsed time for migration stage 3: 7216 ms
stage 3: transferred ram: 141996 kbytes
elapsed time for migration: 24235 ms
total ram transferred: 1721044 kbytes
downtime: 7277 ms
(qemu)
```

Figure 3: create another checkpoint file in “b.chk”

Finally, you may quit or close the virtual machine to test the virtual machine restoration next.

## 5. Restoration

To restore the virtual machine state from the checkpoint, start the virtual machine with the following command

```
$ <normal kvm command> -incoming "exec: dd if=abcd.chk"
```

where <normal kvm command> is the original shell command you used to invoke the guest virtual machine. You should get the running virtual machine with applications and guest OS performing whatever they do when the checkpoint “a.chk” was created as shown in Figure 4.

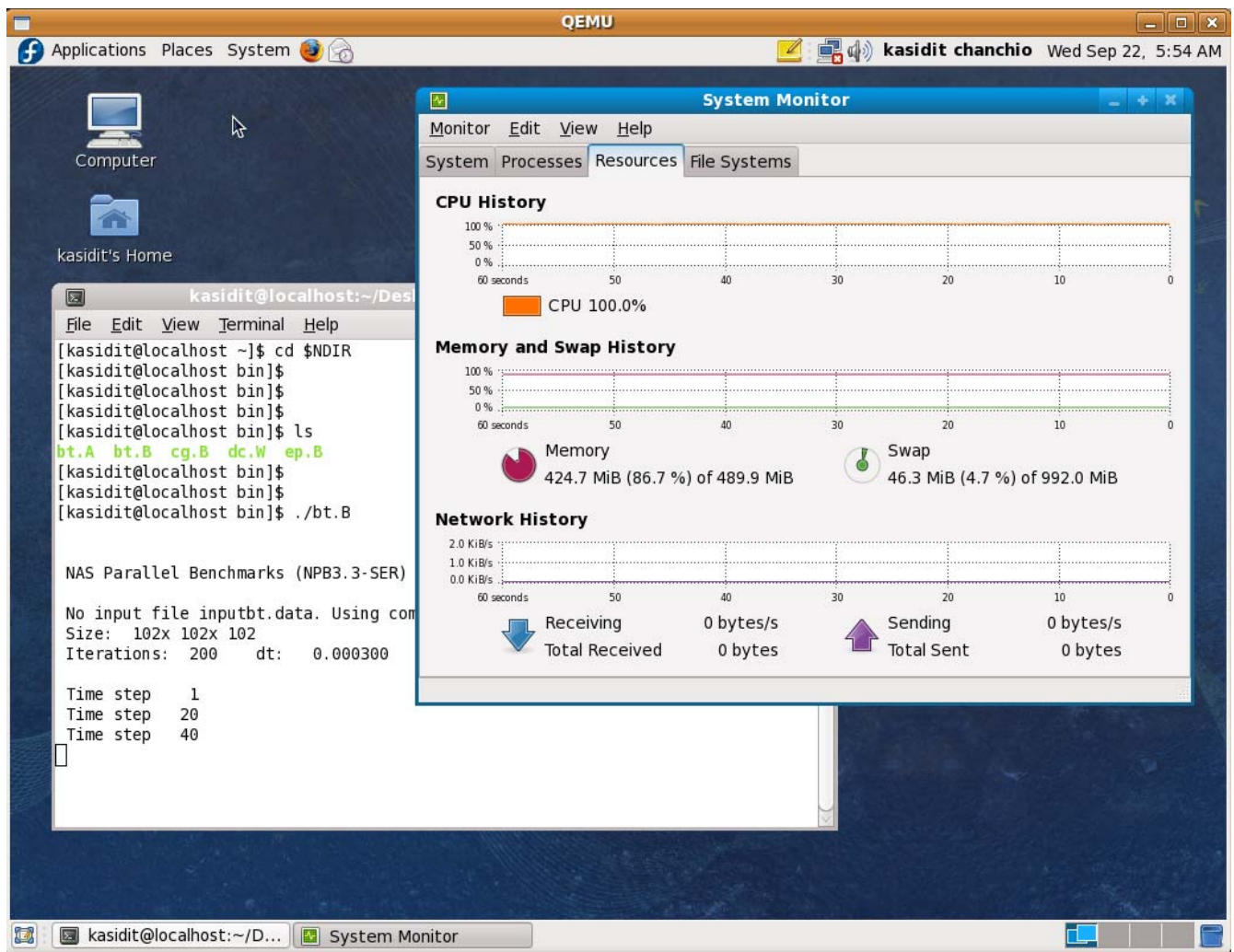


Figure 4: The restored virtual machine from the checkpoint “a.chk”

## References

- [1] Vasinee Siripoonya, Thread-Base Live Checkpointing of Virtual Machines, M.S. Thesis, Thammasat University,
- [2] วสินี ศิริปอญ และ กยติศ ชาญเชื้อว, "ขั้นตอนวิธีการทำเช็คพอยต์สำหรับเวอร์ชวลแมชีนด้วยเทคนิคไลฟ์ไทม์เกรซันแบบเทอร์ด", วารสารวิชาการพระจอมเกล้าพระนครเหนือ, ปีที่ 20 ฉบับที่ 3 พ.ศ. 2553.
- [3] <http://www.linux-kvm.org/>
- [4] <http://vasabilab.cs.tu.ac.th/projects.html>