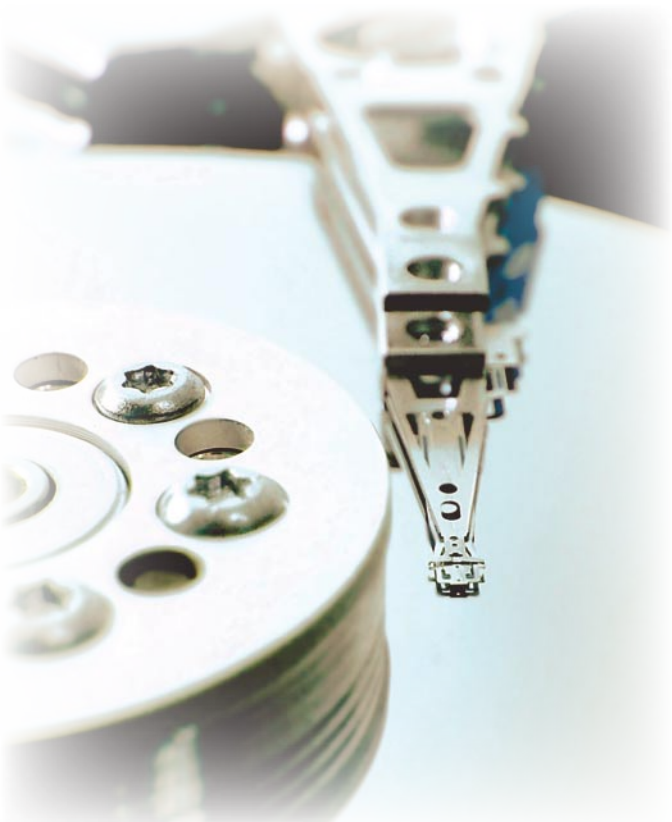
The Linux RAM Disk

Petros Koutoupis

I have always been fascinated about the topic of RAM disks as it covers some extremely fast performing I/O concepts.



The RAM disk has its significant advantages alongside some major disadvantages and despite those disadvantages, it still hasn't prevented it from being used in full production environments to even the local desktop; where it is often used as a temporary cache for various services and applications. The idea is nothing new and has been around for at least a few decades but as we increase our computing power and are capable of utilizing larger amounts of system memory, the concept becomes much more intriguing.

Sometimes referred to as a RAM drive, a RAM disk is a chunk of volatile memory that the operating system is using as a temporary disk drive for temporary data storage. Note that this is a virtual (or software) RAM drive and should not be confused with a hardware RAM drive such as a *Solid State Drive* (hereafter, SSD). I explain these hardware implementations further below. The major advantage to this approach is that memory performs at such high speeds and the computer in turn is capable of handling a significant higher workload with very little latencies. The obvious disadvantage is that when the computer shuts down or when the power is somehow disconnected, all contents within the RAM disk disappear as they only reside in volatile memory. There are approaches to preserving the data with some sort of implemented battery backup and also to synchronize all data contents to a non-volatile form of media. These techniques will be described further in this article.

The Linux Pseudo File System

The way in which most traditional RAM disk modules are presented to the Linux kernel are as pseudo (or virtual) file systems. From the bootup to the post-bootup process the Linux operating system utilizes more than one pseudo file system as a temporary place to mount the kernel image in *rootfs*, or a place to store running processes invoked by applications in *procfs*, to even storing all hardware device information in *sysfs;* all of which are dynamically allocated and populated every time the computer is powered on. They are immediately destroyed when the unit loses power from either a shutdown or any form of unexpected power failure. For

any other type of computing needs, *ramfs* and *tmpfs* have been created; although the Linux kernel does support other pseudo file systems. The */proc/filesystems* file lists all the supported file systems under your running kernel at the time of listing. If the first column is labeled with a *nodev*, it signifies that the associated file system is not associated with a block device. Most if not all of the *nodev* labeled file systems listed will be pseudo file systems (see Listing 1).

While still implemented in the Linux kernel, there was a time when the generic Linux RAM disk (listed as a `/dev/ram` device) acted and was treated as a physical storage volume. It was able to be formatted with a traditional block device file system and accessed as such. For example, it could be formatted with an `ext2` file system, as anything with a journal would have been a wasted effort. The journal only provided redundancy in an event of failure. With the contents gone, there was nothing to recover from. This method brought with it numerous handicaps. For instance, once memory was allocated for use in this virtual volume, it could never be deallocated until a system reboot. Also, once the memory was allocated, it could never be resized. I do not know if it has changed much recently but during the Linux 2.4.x kernel days, it was only useful when dealing with small files as the default limit of the disk was 4 MB and it could only be adjusted at boot time as a kernel option. As a result of this being used like a normal block device, it required unnecessarily copying memory from the virtual block device into/from the page cache, as well as creating and destroying dentries. It also needed a file system driver to format and interpret this data. This wasted memory, created unnecessary work for the CPU, wasted memory bus bandwidth, and polluted the CPU caches. Something had to change. That is when developers started looking in other areas such as an implementation in the form of a file system. This would provide some flexibility and easier manageability.

Enter ramfs and tmpfs

Both *ramfs* and *tmpfs* have been around since the days of the 2.4.x Linux kernel. The main differences between the two types of RAM disks is that *tmpfs* allocates its memory dynamically thus reducing its contents from being moved into swap space. *Tmpfs* also imposes a size limit (as specified by the user) which does not allow it to grow dynamically. If more space is needed, this is not a problem as *tmpfs* can be resized while online by utilizing the mount command along with the remount and newly defined size options. *Ramfs* does not take advantage of this method of virtual memory allocation and therefore also never uses swap space. *Ramfs* does grow in size dynamically (consuming more system memory) which leaves the administrator to monitor and control all processes writing to it. If the processes exceed the defined size limit it may eventually prove to be fatal for the computing host in the sense that there will be no memory left to function and process on anything else. For those of you coming from a Microsoft background, you may be used to this concept but a simple reboot would clear all that up.

Tmpfs was actually built on top of the *ramfs* framework and picking a suitable RAM disk usually depends on the types of features that the administrator is looking for. For the simplicity of this topic, as they are typically similar in setup, I will be using *tmpfs* during the examples of this article.

Configuring a *tmpfs* mount is extremely simple. First you must create a directory to mount the memory-based virtual volume to:

```
petros@debian5:~$ sudo mkdir /mnt/tmp
```

You can then mount the virtual volume with any supported options. In the example below, I have chosen a size limit of 50 MB. You can view all supported options for *tmpfs* under the manual page for the *mount* command (`man 8 mount`).

Listing 1. List of supported file systems under running kernel

```
petros@debian5:~$ cat /proc/filesystems
nodev    sysfs
nodev    rootfs
nodev    bdev
nodev    proc
nodev    cgroup
nodev    cpuset
nodev    debugfs
nodev    securityfs
nodev    sockfs
nodev    pipefs
nodev    anon_inodefs
nodev    tmpfs
nodev    inotifyfs
nodev    devpts
nodev    ramfs
nodev    hugetlbfs
         iso9660
nodev    mqueue
nodev    usbfs
         ext3
         ext2
```

Listing 2. Listing of mounted devices using 'df'

```
petros@debian5:~$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/debianLinux-root
                       7611872   3278496    3946716  46% /
tmpfs                   518044         0     518044   0% /lib/init/rw
udev                     10240        80      10160   1% /dev
tmpfs                   518044         0     518044   0% /dev/shm
/dev/hda1               233335     16410     204477   8% /boot
tmpfs                    51200         0      51200   0% /mnt/tmp
```

```
petros@debian5:~$ sudo mount -t tmpfs
-o size=50m tmpfs /mnt/tmp/
```

When listing all mounted volumes, the newly created RAM-based volume has been appended to the bottom of the list (see Listing 2).

Utilizing a basic method for I/O generation I wrote to a file on the *tmpfs* mount and filled the entire 50 MB with an I/O profile of 50x 1 MB sequential transfers. It only took 0.15 seconds (see Listing 3).

When I wrote to another location, local to my SATA drive, with a similar I/O profile, it took as much as 0.55 seconds to accomplish the same task. That is over 3 times the RAM disk performance for a small file write (see Listing 4).

I even expanded the virtual volume to a size of 512 MB (536870912 bytes) and reran the same test. You can resize the virtual volume with the following command:

```
petros@debian5:~$ sudo mount -o
remount,size=512m /mnt/tmp
```

The RAM disk took 1.5 seconds to write approximately 512 MB while the SATA drive accomplished the same task in 19.7 seconds. Note that the SATA volume is formatted with an `ext3` file system and the device is mounted with the default *ordered* journaling method.

Beneficial Implementations of a RAM Disk

From the rudimentary benchmarks posted above, one can see the significant performance gains when utilizing a RAM disk. After seeing such great results, the first question to come up is: Where can I use this? The best answer I can give is that for your normal end-user it can significantly speed up caching performance from various applications such as the Mozilla Firefox web browser. It usually does not matter that the contents disappear on a system shutdown. Without these contents cached locally to disk, you also run less of a risk from potential security threats. On the business or enterprise computing scene, it can serve well in an area where constant database queries or other web services are being cached and routinely accessed. In some situations, it may not necessarily be important to preserve such information in an event of failure and if it were, I would hope that the storage administrator would have employed some form of redundancy under the idea of high availability (i.e. clustering, load-balancing, multipathing/ failover, etc.)

Let us say that you wanted to improve the caching performance of your Firefox web browser, you can do so in the following steps (specify the mount point and volume size that you are most comfortable with):

```
petros@debian5:~$ sudo mkdir /mnt/tmp
petros@debian5:~$ sudo mount -t tmpfs
-o size=96m,nr_inodes=10k,mode=0777
tmpfs /mnt/tmp/
```

To have the virtual volume automatically mount at bootup, modify your `/etc/fstab` file by appending the following line:

```
tmpfs    /mnt/tmp tmpfs size=96m,nr_
inodes=10k,mode=777 0 0
```

The final step is to open up the Firefox browser and in the URL address entry bar, type `about:config`. You will be prompted with a message about how you must be careful and tampering with the configuration parameters could result in your warranty on this product being void. If it does not already exist add the new key `browser.cache.disk.parent_directory` with the value of `/mnt/tmp`. When you restart you browser (or reboot your host and open your web browser), you should see a directory entry of `Cache` listed in `/mnt/tmp`.

Synchronizing to Non-Volatile Media

In fact, there are a couple ways that an individual can go about this. If more complex and intelligent methods and approaches are needed, then the creation of a custom application/daemon would be beneficial. If only a basic synchronization is needed over a defined period of elapsed time, then it can always be configured into a simple shell script utilizing the rsync command. Please reference its manual page for details on supported options (`man 1 rsync`). If my RAM disk was mounted to `/mnt/tmp` and my archival destination was mounted to `/mnt/backup`, the command for a remote synchronization can look something like this:

```
petros@debian5:~$ sudo rsync -av /mnt/
tmp /mnt/backup
```

Using the same command string with the source and destination directories swapped can restore the data back into the virtual volume.

Vendor Implementations of Non-Volatile RAM Disk Devices

In more recent years, there has been a trend in the use of SSD technologies for data storage. These devices have been manufactured in a few forms; that is, Flash-based, DRAM-based and also a hybrid of the two technologies. Vendors such as Texas Memory Systems, Fusion-io to even Violin Scalable Memory have been standing in the forefront of this set of technologies. They have been manufacturing products utilizing both Flash-based and also DRAM-based Solid State Media.

The Flash-based SSD provides great read and sequential/random performance as there is very little latency in seeking, but with regards to write performance it suffers. Other handicaps come from the life expectancy of cell erase/rewrite methods. Both NAND and NOR based technologies have their own maximum erase/rewrite ratings. More recently this has been controlled by the manufacturer

Listing 3. Writing to tmpfs mounted device

```
petros@debian5:~$ dd if=/dev/zero of=/mnt/tmp/test.txt bs=1M
dd: writing '/mnt/tmp/test.txt': No space left on device
50+0 records in
49+0 records out
52371456 bytes (52 MB) copied, 0.157543 s, 332 MB/s
```

Listing 4. Writing to local SATA disk drive

```
petros@debian5:/mnt$ dd if=/dev/zero of=/tmp/test.txt bs=1M count=50
50+0 records in
50+0 records out
52428800 bytes (52 MB) copied, 0.550141 s, 95.3 MB/s
```

implementing their own mechanism for wear-leveling so that all write operations are spread across the entire medium; completely transparent to the operating system writing to it. And despite the cell sizes a write operation for a standard Flash SSD operates in pages of 128K. So if I were to write 127K or 1 byte of data, an entire page of 128K is pulled into memory, the bytes are then modified, the location on the SSD is erased and finally rewritten (not exactly in this order). This takes a lot of time. Especially when performing with larger files and I/O transfers. The write performance (after a few hours of use out-of-box) takes a tremendous hit and decreases. This is where implementing a tmpfs or ramfs mount for applications that do a lot of caching can become advantageous for a node using a Flash SSD as its local hard drive.

Some of these same vendors have also invested both time and money in manufacturing rack-mountable storage arrays in which the entire architecture runs entirely on DRAM. In an event of power failure, there is a battery backup device integrated into the system, which when fully charged will synchronize all data to a local backup media. This could be a standard magnetic SAS, SATA or Fibre Channel drive to a Flash SSD. When the power to the unit is re-established, the synchronized data will get loaded back into memory. What makes this concept the most interesting is that vendors such as Violin have been open about using the Linux kernel while also attempting to integrate their device driver (in the name of Ramback) into the kernel tree. I have followed some of the e-mails from the *Linux Kernel Mailing List* (LKML) but to date and as of the 2.6.30 kernel, these drivers have not been integrated. Individuals on the mailing list expressed numerous concerns with the driver. Ideal conditions needed to be met before it could work as expected. These DRAM SSD products operate with limited volume sizes and are usually intended for database operations. For example, between both Texas Memory Systems and Violin Scalable Memory, a 2U/3U/4U product can work with DRAM volumes ranging from 16 GB to 512 GB; some performing as high as 1 million IOPS (Inputs/Outputs Per Second). Most if not all of these units are expandable and an entire cabinet will be able to hold multiple units connected in series and operating as a single array. Obviously this does not compare to the capacities of the traditional magnetic disk drives and therefore is somewhat limited in the applications it can fill. Another drawback to this up and coming technology is that it costs significantly more than an array of magnetic hard drives. Although prices are slowly declining as the concept and uses become more popular.

With regards to the hybrids of the two SSD technologies, Acard Technology has been manufacturing hard drive modules that closely resemble your DVD/CD drive module and can be mounted into your desktop PC as such. They contain a SATA back-end and process a total capacity of 64 GB. The drive contains a rechargeable battery along with a Compact Flash module for data synchronization/ restoration.

Conclusion

As can be seen, RAM disks offer great advantages and can be used to make our computing lives easier and faster. I will however give a word of caution as to when a RAM disk has been made via the *tmpfs* or *ramfs* file system modules, and is being utilized, it will take memory from your system and not return until it has been freed, either through the purging of data file contents or even through the use of the `umount` command to unmount the virtual volume. No more memory should be allocated for the virtual volume than the system can handle. The other concern is that the data is stored in a volatile volume, so if the data is important, you must be sure to employ some method of data synchronization to a non-volatile storage device.

For the developers who are interested in learning more about these device drivers, writing a RAM disk file system module is fairly simple and is a great learning experience when it comes to understanding the mechanics of file systems. There are many tutorials on the same topic posted on the Internet. Although, it must be warned that the majority of tutorials are going to be a bit outdated and some research will need to be conducted during the development process.

As a result of writing this article, I have taken some time to write a generic Linux block device (*rxd*) that resides entirely in memory from module insertion to removal. This should only be used as a learning tool. The source code to version 0.1.1 of the device driver can be found at *www.hydrasysllc.com/downloads/rxd011.tar.gz*. The archived file not only includes the source to the Linux module but also a binary application that performs supported ioctl, seek, read and write operations to the RAM disk device module. Once inserted the disk device operates as any other block device in which a file system can be written to it and the device can then be mounted to any local mount point. Note that once you remove the module or if the system were to reboot, all data written to the RAM disk device will disappear from memory. The device driver defaults to a total volume size of 64 MB. This can be adjusted during the insertion of the module with an optional input parameter (sudo insmod *rxd.ko* sizemb=96). The *README.txt* file packaged with the source code contains instructions for compilation and initialization of all code. All updates to the device driver will be posted on my blog at blog.hydrasysllc.com. This version of the device driver has been tested on 2.6.26 kernels (Fedora 8 and Debian 5.0.1) and will not compile on the 2.6.28 kernel. This is a result of modifications to the `block_device_operations` structure within the kernel code. Support for later kernel revisions will be addressed in the very near future.

Switching back onto the topic of hardware RAM disks, I see a strong future in this direction. As maximum capacities increase and prices drop, will it make our current magnetic disk drives go the way of the dodo bird?

Resources

- Bar, Moshe. *Linux File Systems*
- Pate, Steven. *UNIX Filesystems*
- *http://en.wikipedia.org/wiki/TMPFS*
- *http://wiki.debian.org/ramfs*
- *http://en.wikipedia.org/wiki/Ram_disk*
- *http://www.ramsan.com/*
- *http://www.violin-memory.com/*

About the Author

Petros Koutoupis has been using Linux since 2001 and has been involved with software development and administration even longer. He has been involved with enterprise storage computing from 2005 to the present and currently offers consultation services in the same field. His Web Site is www.hydrasystemsllc.com. He can always be contacted at pkoutoupis@hydrasystemsllc.com.