There are many reasons for creating a memory based file system in Linux, not least of which is to provide a near zero latency and extremely fast area to story files. A prime use of a RAM disk is for application caching directories or work areas.

There are two main types of RAM disk which can be used in Linux and each have their own benefits and weaknesses:

* ramfs
* tmpfs

The **free** command shows the amount of RAM availale on your system in addition to the amount of memory used, free and used for caching. SWAP space is also displayed and shows if your system is writing memory to disk.

Create a folder to use as a mount point for your RAM disk.

|  |  |
| --- | --- |
| 1 | mkdir /mnt/ramdisk |

Then use the **mount**command to create a RAM disk.

|  |  |
| --- | --- |
| 1 | mount -t [TYPE] -o size=[SIZE] [FSTYPE] [MOUNTPOINT] |

Substitute the following attirbutes for your own values:

* [TYPE] is the type of RAM disk to use; either **tmpfs** or **ramfs**.
* [SIZE] is the size to use for the file system. Remember that ramfs does not have a physical limit and is specified as a starting size.
* [FSTYPE] is the type of RAM disk to use; either **tmpfs**, **ramfs**, **ext4**, etc.

Example:

|  |  |
| --- | --- |
| 1 | mount -t tmpfs -o size=512m tmpfs /mnt/ramdisk |

You can add the mount entry into **/etc/fstab** to make the RAM disk persist over reboots. Remember however, that the data will disappear each time the machine is restarted.

|  |  |
| --- | --- |
| 1 | vi /etc/fstab |
| 1 | tmpfs       /mnt/ramdisk tmpfs   nodev,nosuid,noexec,nodiratime,size=1024M   0 0 |

**tmpfs vs. ramfs**

The two main RAM based file system types in Linux are tmpfs and ramfs. ramfs is the older file system type and is largely replaced in most scenarios by tmpfs.

**ramfs**

ramfs creates an in memory file system which uses the same mechanism and storage space as Linux file system cache. Running the command **free** in Linux will show you the amount of RAM you have on your system, including the amount of file system cache in use. The below is an example of a 31GB of ram in a production server.

|  |  |
| --- | --- |
| 1  2  3  4  5 | free -g         total used free shared buffers cached  Mem:   31    29   2    0      0       8  -/+ buffers/cache: 20 11  Swap:  13    6    7 |

Currently 8GB of file system cache is in use on the system. This memory is generally used by Linux to cache recently accessed files so that the next time they are requested then can be fetched from RAM very quickly. ramfs uses this same memory and exactly the same mechanism which causes Linux to cache files with the exception that it is not removed when the memory used exceeds threshold set by the system.

ramfs file systems cannot be limited in size like a disk base file system which is limited by it’s capacity. ramfs will continue using memory storage until the system runs out of RAM and likely crashes or becomes unresponsive. This is a problem if the application writing to the file system cannot be limited in total size. Another issue is you cannot see the size of the file system in **df** and it can only be estimated by looking at the **cached** entry in **free**.

**tmpfs**

[tmpfs](http://www.jamescoyle.net/cheat-sheets/1659-what-is-tmpfs)is a more recent RAM file system which overcomes many of the drawbacks with ramfs. You can specify a size limit in tmpfs which will give a ‘disk full’ error when the limit is reached. This behaviour is exactly the same as a partition of a physical disk.

The size and used amount of space on  a tmpfs partition is also displayed in **df**. The below example shows an empty 512MB RAM disk.

|  |  |
| --- | --- |
| 1  2  3 | df -h /mnt/ramdisk  Filesystem Size Used Avail Use% Mounted on  tmpfs      512M 0    512M  0%   /mnt/ramdisk |

These two differences between ramfs and tmpfs make tmpfs much more manageable  however this is one major drawback; tmpfs may use SWAP space. If your system runs out of physical RAM, files in your tmpfs partitions may be written to disk based SWAP partitions and will have to be read from disk when the file is next accessed. In some environments this can be seen as a benefit as you are less likely to get out of memory exceptions as you could with ramfs because more ‘memory’ is available to use.

## initrd vs initramfs

initrd (initial ramdisk) is a scheme for loading a temporary file system into memory in the boot process of the Linux kernel. Initrd and initramfs refer to slightly different methods of achieving this. Both are commonly used to make preparations before the real root file system can be mounted, but there is a difference.

Initrd is a fixed-size block device, which requires to be 'formatted' by a filesystem such as ext2. It sits on /dev/ram0 by default, and cannot be enlarged or shortened.

On the other hand, initramfs is a cpio archive which is simply unpacked during boot to ramfs memory. This memory is of dynamic size and thus can be shortened or enlarged as needed.

/linuxrc or /init is the first executable started once the initrd or initramfs is loaded by the kernel /linuxrc is used in the older initrd /init is used in the newer initramfs This executable is usually a shell script; Debian uses /bin/sh but Red Hat uses /bin/nash; it does not matter which shell/interpreter is used

**When using initrd, the system typically boots as follows:**

1.     The boot loader loads the kernel and the initial RAM disk

2.     The kernel converts initrd into a "normal" RAM disk and  frees the memory used by initrd

 3.   initrd is mounted read-write as root

 4.   /linuxrc is executed (this can be any valid executable, including shell scripts; it is run with uid 0 and can do basically everything init can do)

 5.  linuxrc mounts the "real" root file system

 6.  linuxrc places the root file system at the root directory using the pivot\_root system call

 7.  The usual boot sequence (e.g. invocation of /sbin/init) is performed on the root file system

 8) The initrd file system is removed

[System.map](http://en.wikipedia.org/wiki/System.map) contains a [symbol table](http://en.wikipedia.org/wiki/Symbol_table), i.e. a list of function names in the Linux kernel, with for each function the address at which its code is loaded in memory (the addresses are not physical addresses, they're in the kernel's address space, like any executable symbol table is in the loaded process address space). This isn't limited to system calls (the interfaces exposed to user processes): the file also lists functions that might be called by a loaded module, and even internal functions. The system calls are the symbols whose name begins with sys\_.

The addresses are associated to a particular kernel binary (vmlinux, bzImage or other format; the image format doesn't change the addresses, it's just an encoding); they are reproducible for a given kernel source, configuration and compiler. The file is generated by [scripts/mksysmap](http://lxr.linux.no/#linux+v2.6.37/scripts/mksysmap) near the end of the kernel build process; it is the output of the [nm](http://sourceware.org/binutils/docs-2.21/binutils/nm.html) command.