Solaris 11

GNU/Linux has been plowing ahead with great speed in the past decade. Other Unix distributions like BSD have coexisted nicely with Linux while commercial brands have taken something of a beating. Sun Microsystems tried to keep their Unix variant – Solaris – alive by making it free to use. Unfortunately the editions on offer were outdated at release date compared to Linux and BSD so only people needing compatibility with existing software and stability(which I expect such a conservative piece of software would have) would use Solaris. Next Sun Microsystems created something called OpenSolaris which was a big hit as it took on many of the traits that made Linux so successful but when Oracle bought Sun they effectively canceled OpenSolaris. It was quite a surprise when they came out with Solaris 11 Express and then proper Solaris 11 as it was a great piece of kit!

Solaris 11 has many great features but the ZFS file system(actually Zettabyte File System) is what I think draws most people in. ZFS is available for FreeBSD and Linux but in decreasing order of functionality and performance. Indeed this is why I have undertaken a project to make myself a Network Attached Storage device based on Solaris 11. Oracle are the current developers of ZFS and I trust them to put out thoroughly tested versions of it. FreeBSD is otherwise seen as the best choice for using ZFS but the developers of FreeBSD have to do some tinkering to make ZFS work in their OS. Linux can't include the ZFS code properly for licensing reasons and so can only run the code in user space. I don't mind the user space requirement all that much but between Oracle's native ZFS code for Solaris 11 and Ubuntu's old version that has been tweaked to function in Linux, I suspect a lot of reliability falls away.

Solaris 11 NAS

It should be noted that my project to make a Solaris 11 NAS isn't quite what you might think it is. I'm not trying to replace my existing four-disk, low-power QNap TS-419 with a similar device using Solaris 11. Few such devices are sold as ZFS is quite a resource-intensive workload compared to simpler file systems and I'm not all that keen on buying several hundred dollars worth of hardware only to notice that Solaris 11 doesn't run on it. This is actually quite a problem. Linux has remarkable hardware support while Solaris 11 “kind of” runs on x86 hardware. UEFI? Not so much. New kind of chipset on your motherboard? Well that *might* work... Solaris 11 has a hardware compatibility list(HCL) that mainly includes commercially available servers from Oracle, Dell and Fujitsu. Since I know that Solaris 11 runs fine on both my workstation and server I first ran with the server as the basis for my NAS but when it turned out to only have four SATA-ports I ditched that and re-purposed my workstation. I will refer to this device as both server and NAS depending on the context.

Solaris 11 Zones

As it's quite clear that the server has the power needed to run lots of virtual machines I hope to do under Solaris 11. This allows me to merge the services of file storage and server into one device! We do however run into difficulty. Oracle boasts how good Solaris 11 is at virtualization but that is classical marketing stuff. Solaris 11 has great **container** virtualization but other kinds of virtualization are only supported as an afterthought. What then is the problem with this container virtualization that Oracle called Solaris Zones? Well, you can only run Solaris 10 and Solaris 11 zones in Solaris 11... Now I don't hold this against Oracle. Most people running Solaris 11 will be have one or more dedicated servers to this end. If they need to host a bunch of Linux, Windows and BSD virtual machines they can afford to set up another couple of servers with VMware vSphere or Windows Server Hyper-V. Technically there is nothing stopping me from doing this either, I'm just very frugal!

What good are Solaris 11 guests under Solaris 11? Well, what about web-hosting? Each customer gets his own Solaris 11 environment that gets updated whenever the host server is updated. Container virtualization is extremely efficient so you can run a huge number of guests compared to full virtualization like VMware vSphere. Solaris 11 Zones can also be used to migrate certain workloads from one physical server to another, much like VMware and Xen can do but **only** on SPARC hardware! All in all Zones are quite insufficient for me and need to be supplemented with a more traditional virtualization solution. At this time the only real candidate is VirtualBox, most likely because it is also an Oracle product.

Install

Installing Solaris 11 is easy enough. I choose the GUI version because that allows for simple modification of the network cards but there is also a purely text-based installer. Either way you will install the OS on a storage unit called rpool(presumably *root pool*). The rpool has to consist of a single Solaris2 partition on a single drive. You can't place rpool on a mirror or raid-z array at this stage. The install takes a long time so let it churn along.

Nomenclature

Zpool	A set of storage devices used to accommodate file systems and volumes. Similar to a Volume Group in Linux LVM.
Volume	A chunk of raw storage area in a zpool. Similar to a Logical Volume in Linux LVM.
File system	A ZFS dataset. Equivalent to an LV in Linux LVM that has been formatted with a file system.
Snapshot	A copy of a volume of file system as it was at a given point in time.

Mirror rpool

Of course we don't much like the idea of having the all-important rpool on a single drive. Unfortunately we have to do some minor hacking to remedy the situation. A man named Constatin Gonzalez has written the authoritative guide on this process[http://constantin.glez.de/blog/2011/03/how-set-zfs-root-pool-mirror-oracle-solaris-11-express] and I recommend using to create a mirrored rpool. Herein follows my own approach which is a boiled down version of what mr. Gonzalez proposes. Basically we create a Solaris-style partition on the spare drive, copy the partition table from our existing rpool drive to the future mirror and finally attach it to the rpool. I recommend starting with the format command as it will list existing hard drives on the system and their names.

It will look something like this:

```
root@sol:~# format
Searching for disks...done


AVAILABLE DISK SELECTIONS:
       0. c3t0d0 <ST2000DM-          S240CFB-0001 cyl 60797 alt 2 hd 255 sec 252>
          /pci@0,0/pci-ide@1f,2/ide@0/cmdk@0,0
       1. c3t3d0 <ST350032-          9QM24X5-0001 cyl 60797 alt 2 hd 255 sec 63>
          /pci@0,0/pci-ide@1f,5/ide@0/cmdk@0,0
       2. c3t4d0 <ST2000DM-          S1E082C-0001 cyl 60797 alt 2 hd 255 sec 252>
          /pci@0,0/pci-ide@1f,2/ide@1/cmdk@0,0
       3. c3t5d0 <ST2000DM-          S1E083G-0001 cyl 60797 alt 2 hd 255 sec 252>
          /pci@0,0/pci-ide@1f,5/ide@1/cmdk@0,0
Specify disk (enter its number):
```

The interesting information is the c3t0d0, c3t4d0 and so on. On my system I have a single SATA-controller – c3 – and six targets on that controller – t0-t5. The d-bit refers to device and is superfluous for my setup. Next we see which drive is used for the rpool.

```
root@sol:~# zpool status

  pool: rpool
 state: ONLINE
  scan: none requested
config:

      NAME        STATE     READ WRITE CKSUM
      rpool       ONLINE       0     0     0
        c3t3d0s0  ONLINE       0     0     0

errors: No known data errors
```

So it's c3t3d0 that currently contains the rpool. The s-bit means the device has a single slice in which Solaris then forms its own kind of partition system. Let's say we want c3t3d0 to be the mirror for our existing rpool drive. Then we would execute the following commands to initialize c3t4d0 with the right partition setup and then copy the actual partition table from c3t3d0.

```
root@sol:~# fdisk -B c3t4d0p0
root@sol:~# prtvtoc /dev/rdsk/c3t3d0s0 | fmthard -s - /dev/rdsk/c3t4d0s0
```

Note the odd use of slices and physical addressing. When we talk about the part of a hard drive that contains actual Solaris data we talk about slices like c3t3d0s0. When we are referring to the entire drive with the kind of partition tables that other operating systems use, it's a physical unit c3t4d0p0.

Finally we add attach the new drive to the rpool and write the boot loader to the it as well.

```
root@sol:~# zpool attach rpool c3t3d0s0 c3t4d0s0
root@sol:~# installgrub /boot/grub/stage1 /boot/grub/stage2 /dev/rdsk/c3t4d0s0
```

Typing `zpool status rpool` should now give you information about the pool now being mirrored and that it is being resilvered, i.e. the new drive is being brought into sync with the original drive.

RAID-Z

RAID 5 is an old technique where you have N drives containing data and one drive containing parity bits. That way all data can be regenerated as long as no more than one drive fails. The probability of two simultaneous hard drive failures is quite low but RAID 6 uses two parity disks for just that eventuality. RAID-Z is a ZFS-version of RAID 5. By obtaining information about which drives are available on the computer in question like we did when mirroring rpool we can choose a set of 3+ drives to form a RAID-Z array.

```
root@sol:~# zpool create raided raidz c4t2d0 c4t3d0 c4t4d0  c4t5d0
root@sol:~# zpool list
NAME      SIZE  ALLOC   FREE  CAP  DEDUP  HEALTH  ALTROOT
raided   19.9G   130K  19.9G   0%  1.00x  ONLINE  -
rpool    11.9G  4.05G  7.89G  33%  1.00x  ONLINE  -
```

```
root@sol:~# zpool status -v raided
  pool: raided
 state: ONLINE
  scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        raided      ONLINE       0     0     0
          raidz1-0  ONLINE       0     0     0
            c4t2d0  ONLINE       0     0     0
            c4t3d0  ONLINE       0     0     0
            c4t4d0  ONLINE       0     0     0
            c4t5d0  ONLINE       0     0     0

errors: No known data errors
```

This particular printout came from an experiment I ran in a VMware vSphere guest with lots of failures popping up. I'm quite confident that there is some incompatibility between Solaris 11 and the vSphere hypervisor leading to corruption of hard drives used by Solaris 11.

Disaster Recovery

One thing that made me reluctant to use RAID-Z was the issue of retrieving data in case of failure. My previous NAS used Linux RAID 1 and I had confirmed that I could just yank out one of the drives in a mirror, plug it into another computer running Linux and extract all my data. With RAID-Z this isn't possible but it turns out that a serious failure where both the NAS itself and one of the drives in a RAID-Z array also fails is quite recoverable. Just plug in the remaining drives from the array and run the command `zpool import` and you will be presented with information about the array that the remaining drives used to be part of. As there is one missing drive the array will be reported as degraded.

Let's say that the array used to be called raidstore. Running `zpool import raidstore` is not going to work in the scenario given above as the existing Solaris 11 instance will recognize that raidstore was last used by some other Solaris install. Running `zpool import -f raidstore` will however work and you will now have a degraded but readable zpool called raidstore on the new system. The performance of this degraded array will be awful but I don't mind as long as I get all my data back. After testing this in various ways I grew confident that disaster recovery of a RAID-Z array is reliable.

Boot Environment

Boot Environment stem from ZFS and allows you to choose which snapshot of the root file systems you wish to boot. I used the following commands to first create a BE for the fresh install on a test system:

```
root@nastest:~# beadm create pristine20120722
root@nastest:~# beadm list
BE                  Active Mountpoint     Space  Policy Created
--                  ------ ----------     -----  ------ -------
pristine20120722    -      -              128.0K static 2012-07-22 19:54
solaris             NR     /              4.12G  static 2012-07-22 16:44
```

But this was mainly meant to be used to restore the system to that point in time, not something I would boot into. For that I created another BE where I intended to proceed by installing VirtualBox:

```
root@nastest:~# beadm create vboxtest
root@nastest:~# beadm activate vboxtest
root@nastest:~# beadm list
BE               Active Mountpoint Space  Policy Created
--               ------ ---------- -----  ------ -------
pristine20120722 -      -          128.0K static 2012-07-22 19:54
solaris          N      /          43.0K  static 2012-07-22 16:44
vboxtest         R      -          4.12G  static 2012-07-22 19:55

REBOOT

root@nastest:~# beadm list
BE               Active Mountpoint Space  Policy Created
--               ------ ---------- -----  ------ -------
pristine20120722 -      -          128.0K static 2012-07-22 19:54
solaris          -      -          5.12M  static 2012-07-22 16:44
vboxtest         NR     /          4.14G  static 2012-07-22 19:55
```

Performance

There are many configuration options both in terms of zpools, file systems and network protocols. I tried various combinations and used the Bonnie++ benchmarking utility on Linux to compare these combinations with the old NAS. First off, make sure you enable AHCI in the BIOS if you have that available. I forgot that and got pitiful performance for any setup. With AHCI enabled things got better but creating lots of small files turned out to be a problem for ZFS. Adding an SSD as a kind of write-cache(ZIL, or ZFS Intention Log) for the underlying storage device improved matters further.

The best performance for writing lots of small files was obtained by removing the requirement to honor the implicit guarantee that any data written to a file system is *permanently* stored. So the command `zfs set sync=disabled mirstore/testarea1` tells Solaris that when someone asks to write a file to mirstore/testarea1 they should be told that the write is complete as soon as the file has been placed in the server's working memory. This is much faster than waiting for the write to actually be committed to the underlying hard drives. The problem of course is that the server could crash at which point all writes in memory would be lost.

The old NAS turned out the be quite slow in sustained reads and writes but really fast when it came to writing lots of small files. I suspect it uses an aggressive version of sync=off to accomplish this but I'll admit Linux ext3 is probably inherently faster when it comes to this kind of operation as it isn't transactional, CoW or most other things that make ZFS so attractive. As far as using large block sizes for NFS communication actually degraded performance so that was abandoned right quick.

The table below contains the most interesting information from the various Bonnie++ runs carried out over a 1GB network. The first three numerical columns are in the unit KB per second while the last two are files read/created per second. Clearly the Solaris 11 NAS is two to three times faster in throughput and reading many files. As mentioned earlier, the old NAS had blazing performance in creating lots of small files.

	Block(Write)	Rewrite	Block(Read)	Reads/s	Create/s
RAIDZ+SSD	61082	21254	46778	432	388
NASX	18603	13005	40171	210	1734
MIRROR+SSD	69420	21306	54172	636	554
MIRROR+SSD+NFSOPTS	60985	20190	47764	473	364
MIRROR+SSD+NFSOPTS-SYNC	72652	21292	48383	502	595

In the end I chose the top entry, RAID-Z with three 2 TB drives and a 120 GB SSD ZIL. I was going to test using the SSD as a more generic cache device but this all took a lot of time so I skipped it.

Raidpool

In line with the previous section I blanked the three 2 TB drives and formatted them using the prescribed method:

```
root@sol:~# dd if=/dev/zero of=/dev/c3t4d0p0 bs=512 count=1000
root@sol:~# format -e
* Choose a disk *
format> partition
partition> label
[0] SMI Label
[1] EFI Label
Specify Label type[0]: 1
Warning: This disk has an SMI label. Changing to EFI label will erase all
current partitions.
Continue? y
partition> quit
format> verify
* printout *
format> quit
```

Repeat for the other drives. Then create the actual pool:

```
root@sol:~# zpool create raidpool raidz c3t1d0 c3t2d0 c3t3d0 log c3t4d0
root@sol:~# zpool status
  pool: mirstore
 state: ONLINE
  scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        mirstore    ONLINE       0     0     0
          raidz1-0  ONLINE       0     0     0
            c3t1d0  ONLINE       0     0     0
            c3t2d0  ONLINE       0     0     0
            c3t3d0  ONLINE       0     0     0
        logs
          c3t4d0    ONLINE       0     0     0
```

Network share

I used CIFS/SMB for Windows clients and NFS for Linux clients with my old NAS. One thing bothered me though, CIFS would label the files I created with the UID/GID of the user I had on the NAS. NFS on the other hand would just use the UID/GID of my client's user. This is a known issue with NFS and can be solved in two ways.

1. Synchronize all user accounts on Linux clients throughout the network.
2. Use the Kerberos centralized authentication system.

The first option is cumbersome when you use several different Linux distributions and the second is cumbersome all the time. Right now I'm trying to use CIFS even with Linux clients but I have setup a flaky proof-of-concept Kerberos system that I could refine in case CIFS plays up. Right now the biggest issue with CIFS is that it gets really concerned about minor time-differences between the NAS and the client.

SAN

My previous NAS also had SAN functionality, namely iSCSI. Thankfully Solaris 11 also has support for iSCSI and it can be combined with the features of ZFS! Let's first install a basic system:

```
root@sol:~# pkg install group/feature/storage-server
root@sol:~# zfs create -V -s 200g raidpool/disk1
```

Note: I missed the -s switch at first which lead to problems when it came time to make snapshots. Without the -s switch the volume isn't "sparse" and that means it reserves all necessary space up front, including snapshots. So rather than a 120 GB placed in the volume taking up 120 GB, it takes upp 200 GB. And a snapshot will take up 200 GB as well, which makes the whole ZFS-thing kind of pointless.

```
root@sol:~# stmfadm create-lu /dev/zvol/rdsk/sanpool/disk1
Logical unit created: 600144F05F200F0000004FC805280001
root@sol:~# stmfadm list-lu
LU Name: 600144F05F200F0000004FC805280001


root@sol:~# stmfadm add-view 600144F05F200F0000004FC805280001
root@sol:~# stmfadm list-view -l 600144F05F200F0000004FC805280001
View Entry: 0
    Host group   : All
    Target group : All
    LUN          : 0

root@sol:~# svcadm enable -r svc:/network/iscsi/target:default
root@sol:~# itadm create-target
Target iqn.1986-03.com.sun:02:8a029619-609d-e56a-9e5c-beeee7b2cf93 successfully
created

root@storage:~# itadm list-target -v
TARGET NAME                                                  STATE    SESSIONS
iqn.1986-03.com.sun:02:8a029619-609d-e56a-9e5c-beeee7b2cf93  online   0
        alias:                  -
        auth:                   none (defaults)
        targetchapuser:         -
        targetchapsecret:       unset
        tpg-tags:               default
```

Now we can try mounting it on another computer, in this case a virtual machine running Ubuntu.

```
cjp@privatevm:~$ sudo iscsiadm -m discovery -t sendtargets -p 192.168.0.126
192.168.0.126:3260,1 iqn.1986-03.com.sun:02:8a029619-609d-e56a-9e5c-beeee7b2cf93

cjp@privatevm:~$ sudo iscsiadm -m node -T iqn.1986-03.com.sun:02:8a029619-609d-
e56a-9e5c-beeee7b2cf93 -p 192.168.0.126 --login
Logging in to [iface: default, target: iqn.1986-03.com.sun:02:8a029619-609d-e56a-
9e5c-beeee7b2cf93, portal: 192.168.0.126,3260]
Login to [iface: default, target: iqn.1986-03.com.sun:02:8a029619-609d-e56a-9e5c-
beeee7b2cf93, portal: 192.168.0.126,3260]: successful
```

It's as simple as that! It's not much more difficult to take a snapshot of the underlying volume and make a writable copy(called a clone) available over iSCSI:

```
root@sol:~# zfs snapshot raidpool/disk1@initial
root@sol:~# zfs clone raidpool/disk1\@initial raidpool/disk1initial
root@sol:~# stmfadm create-lu /dev/zvol/rdsk/raidpool/disk1initial
Logical unit created: 600144F05F200F0000004FC92BA00004
```

```
root@sol:~# stmfadm add-view 600144F05F200F0000004FC92BA00004
```

You can then reload the iSCSI session on the client and you will now have the writable clone available next to the current version.

```
cjp@privatevm:~$ sudo iscsiadm -m session -R
```

Solaris 11 Zones(again)

While Zones aren't sufficient for my needs for virtualization they are still useful. I have had stability problems when running VirtualBox in what is known as the global zone(basic install) but I haven't had any trouble yet running it in a dedicated zone. It's also practical for testing configurations intended for the global zone. You could work with Zones for months and still have lots to learn but I use a simple setup. First we create a file system for Zones:

```
root@nastest:~# zfs create -o mountpoint=/zones rpool/zones
```

Next comes a starting a starting zone:

```
root@nastest:~# zonecfg -z client1 -f client1_zone.cfg
root@nastest:~# zoneadm -z client1 install
* long non-interactive install process *
```

The file client1_zone.cfg:

```
create -b
set zonepath=/zones/client1
set brand=solaris
set autoboot=true
set bootargs="-m verbose"
set ip-type=exclusive
add anet
set linkname=net0
set lower-link=auto
set configure-allowed-address=false
set link-protection=mac-nospoof
set mac-address=random
end
```

Next we boot the zone and login into it by way of ***console***:

```
root@nastest:~# zoneadm -z client1 boot; zlogin -e @ -C client1
```

The -C is crucial because if you don't include that switch you will be logged in to the command line. That's fine later on but first we need to configure the zone and we can only do that through the console. It is a simple curses-type program and you enter information about IP addresses, hostname and so on. I recommend installing some basic utilities that you might want to use, like nano emacs23-nox and so on.

Now comes the clever bit. We shut down client1 and never use it again(probably). Instead we use it as a template for future zones that we create by cloning client1. Let's say we want a zone called client2:

```
root@nastest:~# zonecfg -z client2 -f client2_zone.cfg
root@nastest:~# zoneadm -z client2 clone -c /root/client-template.xml client1
```

client2_zone.cfg:

```
create -b
set zonepath=/zones/client2
set brand=solaris
set autoboot=true
set bootargs="-m verbose"
set ip-type=exclusive
add anet
set linkname=net0
set lower-link=auto
set configure-allowed-address=false
set link-protection=mac-nospoof
set mac-address=random
end
```

client-template.xml with sensitive information obscured:

```
<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">
<service_bundle type="profile" name="sysconfig">
  <service version="1" type="service" name="system/config-user">
    <instance enabled="true" name="default">
      <property_group type="application" name="root_account">
        <propval type="astring" name="login" value="root"/>
        <propval type="astring" name="password" value="passwordhash"/>
        <propval type="astring" name="type" value="role"/>
      </property_group>
      <property_group type="application" name="user_account">
        <propval type="astring" name="login" value="username"/>
        <propval type="astring" name="password" value="passwordhash"/>
        <propval type="astring" name="type" value="normal"/>
        <propval type="astring" name="description" value="User's real name"/>
        <propval type="count" name="gid" value="10"/>
        <propval type="astring" name="shell" value="/usr/bin/bash"/>
        <propval type="astring" name="roles" value="root"/>
        <propval type="astring" name="profiles" value="System Administrator"/>
        <propval type="astring" name="sudoers" value="ALL=(ALL) ALL"/>
      </property_group>
    </instance>
  </service>
  <service version="1" type="service" name="system/timezone">
    <instance enabled="true" name="default">
      <property_group type="application" name="timezone">
        <propval type="astring" name="localtime" value="Europe/Stockholm"/>
      </property_group>
    </instance>
  </service>
  <service version="1" type="service" name="system/environment">
    <instance enabled="true" name="init">
      <property_group type="application" name="environment">
        <propval type="astring" name="LANG" value="C"/>
      </property_group>
    </instance>
  </service>
  <service version="1" type="service" name="system/identity">
    <instance enabled="true" name="node">
      <property_group type="application" name="config">
        <propval type="astring" name="nodename" value="client2"/>
```

```
      </property_group>
    </instance>
  </service>
  <service version="1" type="service" name="system/keymap">
    <instance enabled="true" name="default">
      <property_group type="system" name="keymap">
        <propval type="astring" name="layout" value="sv"/>
      </property_group>
    </instance>
  </service>
  <service version="1" type="service" name="system/console-login">
    <instance enabled="true" name="default">
      <property_group type="application" name="ttymon">
        <propval type="astring" name="terminal_type" value="vt100"/>
      </property_group>
    </instance>
  </service>
<service name='network/install' version='1' type='service'>
        <instance name='default' enabled='true'>
            <property_group name='install_ipv4_interface' type='application'>
                <propval name='name' type='astring' value='net0/v4'/>
                <propval name='address_type' type='astring' value='static'/>
                <propval name='static_address' type='net_address_v4' value='IP-
address/24'/>
                <propval name='default_route' type='net_address_v4' value='IP-
address'/>
            </property_group>
        </instance>
    </service>
<service name='network/dns/client' version='1'>
        <property_group name='config'>
            <property name='nameserver'>
                <net_address_list>
                    <value_node value='IP-address'/>
                </net_address_list>
            </property>
            <property name='search'>
                <astring_list>
                    <value_node value='domainname'/>
                </astring_list>
            </property>
        </property_group>
        <instance name='default' enabled='true'/>
    </service>
<service version="1" name="system/name-service/switch">
        <property_group name="config">
            <propval name="host" value="files dns"/>
        </property_group>
        <instance enabled="true" name="default"/>
    </service>

<service version="1" name="system/name-service/cache">
        <instance enabled="true" name="default"/>
    </service>
</service_bundle>
```

Now you have a clone of client1 with it's own hostname, IP-address and users.

Appendix

I salvaged some old code used for testing the resilience of ZFS and True Crypt given certain kinds of hard drive corruption to make *check_fs*, a script to record the checksum of files in a tree and verifying them at a later date. I used it on very important datasets like documents and pictures before moving them to the NAS device and then afterwards. No corruption at all was detected.

```
#!/usr/bin/python
import os, zlib, sys

oneKB = 1024;
oneMB = 1024*oneKB;
oneGB = 1024*oneMB;

def sum(file):
      try:
            file_handler=open(file, 'rb')
      #except (IOError, OSError) as e:
      except EnvironmentError:
            print("--* File " + file + " not found *--")
            return "NOT FOUND"
      file_reference = file_handler.read()
      generated_digest = str(zlib.adler32(file_reference))
      file_handler.close()
      return generated_digest

def remove_possible_slash(string):
    if(string.endswith(os.sep)):
        return string[0:-1]
    else:
        return string

def formatBytes(size):
      if(size > 10*oneGB):
            newSize = size/oneGB;
            return str(round(newSize, 1)) + " GB";
      elif(size > 10*oneMB):
            newSize = size/oneMB;
            return str(round(newSize, 1)) + " MB";
      elif(size > 10*oneKB):
            newSize = size/oneKB;
            return str(round(newSize, 1)) + " KB";
      else:
            return str(size) + " B";

def check_sums(sums_filename, dir):
      inf = open(sums_filename, 'r')
      recorded = { }
      for line in inf.readlines():
            divided = line.split(' => ')
            thisSum = divided[1].strip()
            thisSize = divided[2].strip()
            recorded[divided[0]] = (thisSum,thisSize)
      inf.close()

      allFiles = 0
      badFiles = 0
      badBytes = 0
      allBytes = 0
```

```
	for a,(b,c) in recorded.items():
		currSum = sum(a)
		if(currSum.__eq__("NOT FOUND")):
			badFiles += 1
			badBytes += int(c)
		elif(currSum.__ne__(b)):
			print(a + " has been corrupted.")
			badFiles += 1
			badBytes += int(c)
		allFiles += 1
		allBytes += int(c)
	allSize = formatBytes(allBytes)
	badSize = formatBytes(badBytes)
	byteCorruptionPercent = str(round(100*(badBytes/allBytes), 2)) + "%"
	fileCorruptionPercent = str(round(100*(badFiles/allFiles), 2)) + "%"
	print("Corruption:")
	print(str(badFiles) + " out of " + str(allFiles) + " files which is " +
	fileCorruptionPercent + ".")
	print(badSize + " out of " + allSize + " which is " + byteCorruptionPercent
	+ ".")


def make_sums(sums_filename, dir):
	out = open(sums_filename, 'w')
	for dir_path, sub_dirs, files in os.walk(dir):
		for file in files:
			absPath = dir_path + os.sep + file
			try:
				generated_digest = sum(absPath)
				print(file + " => " + generated_digest)
				size = os.stat(absPath).st_size
				out.write(absPath  + " => " + generated_digest + " => " +
				str(size) + "\n")
			except EnvironmentError:
				print("--* File " + file + " not accessible *--")
	out.close()

if __name__ == "__main__":
	if(len(sys.argv)) != 4:
		print("check makesums file dir | check checksums file dir")
	else:
		sums_filename = sys.argv[2]
		dirname = sys.argv[3]
		if(sys.argv[1] == "makesums"):
			#print("Makesums! %s %s" % (sums_filename, dirname))
			make_sums(sums_filename, dirname)
		elif(sys.argv[1] == "checksums"):
			#print("checksums! %s %s" % (sums_filename, dirname))
			check_sums(sums_filename, dirname)
		else:
			print("Error!")
else:
	print("Stand alone program")
```

To generate a set of checksums for a directory tree, execute

```
check_fs makesums media_sums.txt /home/user/Media
```

To verify them after they have been moved:

```
check_fs checksums media_sums.txt /home/user/NewMediaLocation
```