CECIL FILE SYSTEM

By Ted Gress

In CECIL everything is a file, similar to Linux. This includes devices, memory blocks, and mount points. CECIL uses a dual space manager (DSM) that keeps a redundant copy of everything on a separate partition, using a RAID style effect on one drive. I think this reasonable considering the size of hard disks today. In this case partition one is identical to partition two. Also, we trade security for speed. All files on the drive are in constant encryption unless being accessed. In addition we trade speed versus space. All inactive data files are compressed and only uncompressed when accessed.

As far as administration goes, all file system information is kept in memory except for the actual bytedata (chunk data). This allows for quick lookup.

As far as authentication goes, every file has a group, user, world, and roles. The permissions are read, write, and execute. Roles is stored in the beginning of the file system. Roles can be created on the fly. For example, say I want to create a role, animefan. Suppose there is a directory that animefan only wants access to. We can tag the directory with the role animefan and turn off ugo and animefan will have exclusive access.

The filesystem appearance is modeled after the Internet. A URL behaves just like a member of the filesystem. For example:

<http://www.google.com>

<ftp://download.com>

sfile://ted\_gress/ (secure file protocol – file system. In this case user name ted\_gress)

sfile://root/ (admin level access)

sfile://dev/hd0 (devices, first hard drive)

sfile://dev/mnt (mounted devices folder)

sfile:// (machine root)

sfile://localhost (network root)

sfile://ted\_gress <- user space

sfile://application.html <- application space

sfile://shell/ <- shell space

sfile://kernelspace <- kernel space

sfile://devicespace <- device space

Everything is a file, including directories, devices, drivers, Windowed Systems, user programs, system programs, etc.

File System

File Systems should not be able to append or change a newly created files properties. Also, basing a files type off of an extension is bad practice. In CECIL when bjan executable is created a property in the file header is marked. In the case of an executable the property is ‘.ext’. Secure objects such as executables are considered secure objects as well as files like shared libraries ‘.so’. Secure objects, to be created, need to petition the operating system for permission to be created. The operating system takes the new primed file, throws it in a quick sandbox, asks to run the file, and then after a few seconds asks the user for his approval. If approved, the file is created. The sandbox process is so fast it isn’t even noticed. Same goes for static libraries ‘.a’ , configuration files ‘.config’, drivers ‘.driver’, etc. User level programs such as word documents or pictures don’t require approval but still have types embedded in them, ‘.png.’, ‘.jpg’, ‘.doc’, ‘.docx’, ‘rtf’, ‘txt’, etc.

The filesystem supports partitioning and also supports a swap partition. The filesystem can’t have circular links and supports symbolic links. The file system also supports aliases and shortcuts. CECIL uses a lookup table similar to FAT32 but uses hashing for the table? CECIL uses live defragmentation, defragmenting in the background every so often. It uses a special algorithm for live defragmentation similar to the bin packing problem. CECIL supports “locking” so that nobody can acess that file while its locked. CECIL supports the feature where two identical copies of a file are in two different places at once. Essentially however, there is one file. Directories are files.

CECIL supports, from the user perspective, one directory at the device root and a tree hierarchy. The single dot refers to the current directory and the double dot refers to the previous directory. CECIL has a single root directory disk for mounting drivers. The devices/drives automount to a directory mount/devicename.

The directory sfile://ted\_gress is the home directory for ted\_gress for example.

The devices example for instance: sfile://devicespace/hdd1 driver for hard drive 1

sfile://devicespace/usb1 driver for usb stick 1

sfIle://devicespace/wifi1 driver for wifi card

The mount points are file: sfile//dev/mnt and for the three above:

sfile://dev/mnt/hdd1

sfile://dev/mnt/usb1

sfile://dev/mnt/wifi1

Paths are bad. Look at Linux. You have to dig around everywhere just

to find a file. CECIL uses search based paths. Not linear search, to slow. CECIL uses an implementation of a hash table.

CECIL uses a master control block. This is at the beginning of the disk. The disk is divided into 64KB chunks.

The master control block doesn’t hold the chunks in memory but links to multiple chunk servers. Chunk servers contain blocks of chunks and maps to them.Master control block links to chunk servers.

Metadata referring to things like file and chunk namespace, mapping files to chunks, location of replicates (doubling) are kept in memory in the metadata cache. Essentially very little lookup while the PC is running. When a “pulse” occurs while running (a certain amount of data has been changed) then the filesystem is wrteen from memory back to disk. Periodically checksums are calculated between the twin storage volumes.

In actuality CECIL is a flat file system, not a tree. Each file has a virtual parth that is stored. Operations on the file directory (find,open,read,write) is performed by searching for the file in that flat directory and matching its metadata with what is being searched for. Periodically mergesort is run, sorting the flat file system alphabetically, making retrieval of entries faster. Also hash tables are incorporated instead of an array for speed. I claim this is faster than a typical file tree.

There are several large buffers kept in memory that maintained with recent files. If accessed, they are rtouched (reverse touched) and have to be swapped from the buffer and written back to the disk. Then the rtouch is removed.

Objects larger than the chunk size are split up. Garbage collection is used instead of deletion. If a deleted chunk is overwritten it is fine. After a while the garbage collector would have gotten it. This is not top priority so the GC is a good thing.

Most important thing. The filesystem resides partially in memory. When n CECIL starts it finds the largest block of free memory. It examines the filesystem that will fit in that free memory and then loads it from the filesystem into memory. We have more hard drive space than memory. So we essentially we transfer what relevant filesystem data we have into memory. This is sort of a reverse virtual memory.

How to implement a cache?

Struct master\_boot\_sector

{

disk\_size //size of the entire disk in bytes

location\_of\_file\_table //?

list of logical partiion servers //?

address of logical partition servers //addresses of the logical partitions

version\_number //version number of CECIL

boot\_code //?

jmp instruction //?

master\_drive\_server\_offset\_lists //addresses of master\_driver\_servers

master\_drive\_server\_size\_lists //sizes of each master\_driver\_server

block size //size of struct

bytes free //amount of space free

int offset; //offset of block

}

Struct master\_drive\_server

{

struct logical\_partition\_server l\_list //controls a partition

struct logical\_partition\_server l\_list\_bak; //redundancy

struct swap\_partition s\_list; //swap partition

struct filesystem\_type f\_type; //file system type

struct master\_drive\_ops; // ops for master\_drive\_server

struct addresses; //table of addresses of logical partitiosn

struct operations\_buffer; //a buffer of operations to disk

//empties buffer when done

struct operations\_buffer2; ///another buffer of operations to disk

//empties buffer when done

struct operations\_buffer3; //another buffer of operations to disk

//empties buffer when done

stuct writecache; //a data cache of file writes

struct readcache; //a data cache of file reads

unsigned long md5 writecache //store md5 hash in memory

//if data being written hashes differently

//the cache is flushed

` //otherwise the cache is sent to write

unsigned long md5 readcache //store md5 hash in memory

//if md5 is different from cache

//flushes cache otherwise

//reads from cache

unsigned long md5readcache; //actual readcache

unsigned long md5writecache; //actualwritecache

unsigned long md5quickflag //flag in memory for quick\_cache

struct quick\_cache //see below. Used for both read and write operation

struct bytes\_in\_use //number of bytes in use

unsigned long number\_logical\_partitions; //number of logical partitions

logical\_partitions list //addresses of partitions

unsigned long partitions\_size; //size of partitions

number\_of\_types\_mounted //number of mounted types

status\_of\_filesystem //status of filesystem – OK, DAMAGED,FAIL

amount of freespace to partition //available space for partitioning

freespace name //given name for free space on drive

filesystem //file system

mirror //concurrency lock

roles\_Table //lookup roles here

file\_identifier\_ctr

logical\_partition\_offset\_lists //addresses of logical partitions

logical\_partition\_size\_lists //sizes of logical partitions

block size //size of block

bytes free //freespace - padding

int offset //location of block

}

struct master\_drive\_ops

{

void (\*force\_replicate) //force dual partitions to sync

void (\*swap\_partition) //force swap

void (\*beat\_defrag) //perform a defrag operation on heartbeat

void (\*flush) //flush buffering of data and commands

void(\*flushcache) //flush read write cache

void (\*create\_file) //creates a flie

void (\*read\_file) //check cache flag. Read from cache or

//filesystem depending on flag

//pass command to partition\_server

//to read. Read from quick cache if available

void (\*update\_file) //check cache flag. Write to cache or

//filesystem depending on flag pass command to

//partition //server to write. Write to quick //cache

//send update to logical\_partition\_server

void(\*delete\_file) //delete a file from the table call part\_server

void (\*rename) //rename a file

void(\*new) //sends a command to partition server

void(\*consistency) //auto consistency check on heartbeat

void(\*delete\_chunk) //delete a chunk

void (\*delete\_chunk\_srvr) //removes a chunk server

void(\*read\_chunk) //read chunk

void(\*read\_chunk\_srvr) //read chunk server

void(\*delete\_logical\_part) //delete a logical partition

void (\*read\_logical\_part) //read a logical partition

void(\*add\_logical\_part) //add a logical partition

void(\*remount) //remount master\_drives

void(\*add\_chunk) //add a chunk to the chunk server

void(\*mkdir) //make a directory

void(\*rmdir) //remove a directory

void(\*followalias) //follow alias

void(\*followshortcut) //follow shortcut

void(\*setattr) //set attributes (dugorx)

void(\*getattr) //get attributes (dugorx)

void(\*find\_file) //find a file in the flat list

void(\*open\_file) //open a file

void(\*flat\_list\_merge\_stort) /perform merge sort on the flat list

void(\*garbage\_collect) //perform garbage collection on chunks

void(\*compress) //compress logical partitioning

void(\*encrypt) //encrypt logical partitioning

struct quick\_cache

{

struct chunkservcer quickcache //on read operation store in quickcache

paritybit; //calculate parity bit of incoming/outgoing

//if parity bit is the same perform read/write

//from cache. Else read/write cache and set

//parity bit in memory along with w or r

//for read or write

numberOfPartitionServers;

sizeOfPartitionServers;

numberOfChunkServers;

sizeOfChunkServers;

}

Struct logical\_partition\_server

{

struct chunk\_servers c\_list ; //list of chunk serveres

struct directory\_flat d\_list ; //list flat directory and their chunks

unsigned long part\_size; //logical partition size in bytes

unsigned long avail\_size; //available space

chunk server addreses; //table of chunk server addresses

number\_chunk\_servers //number of chunk servers

volume\_name; //name of partition

chunk\_server\_offset\_lists; //addresses of chunk\_servers

chunk\_server\_size\_lists; //size of chunk serves

block size //size of block

bytes free //freespace - padding

int offset //location of block

}

struct logical\_partition\_ops

{

void (\*create\_file) //create a file in this partition

void (\*read\_file) //search directory\_flat list for file

//use directory list to call correct chunk\_server

void(\*update\_file) //search flat list and get chunk\_server to update

void(\*delete\_file) //check flat list and send a command to delete

void(\*rename\_file) //pass new name to rename op for file

void(\*new\_file) //pass new command to a chunk\_server

void(\*delete\_chunk) //delete a chunk

void(\*delete\_chunk\_server) //delete chunk

void(\*read\_chunk) //read a chunk

void(\*read\_chunk\_srvr) //read chunk server

void(\*add\_chunk) //add a chunk manually

void(\*add\_chunk\_server) //add a chunk server manually

void(\*mkdir) //make a directory

void(\*rmdir) //remove a directory

void(\*followalias) //follow alias

void(\*followshortcut) //follow shortcut

void(\*setattr) //set attributes (dugorx)

void(\*getattr) //get attributes (dugorx)

void(\*find\_file) //find a file in the flat liset

void(\*open\_file) ///open a file in the flat list

void(\*flat\_list\_merge\_stort) ///perform merge sort in the flat list

void(\*garbage\_collect) //perform garbage collection

void(\*compress) //compress the logical partition

void(\*encrypt) //encrypt the logical partittion

}

struct chunk\_servers //kept in memory

{  
 struct chunks ck\_list; //list of 64 KB chunks.Each kept in memory?

struct directory\_flat d\_list; //list flat directory and its chunks

hashtable file\_lookup\_table f\_tbl //look up files spread over chunks

unsigned long chunk\_server\_size //sizes of a chunk server

unsigned long avail\_servier\_size //available space

struct chunks addresses; //table of addresses of chunks

buffer\_read\_data //read data from buffer

buffer\_write\_data //write data to buffer

number\_of\_chunks //number of chunks

number\_of\_files //number of files

number\_of\_unallocated\_chunks //number of unallocated chunks

block size //size of this block

free bytes //free bytes for this block

int offset; //address of this struct

sandbox //whether this chunkserver and its files rae in the sandbox

timestamp mergesort //every so often the chunks are merge sorted on disk

chunk\_offsets\_lists; //addresses of chunk offsets

chunk\_size\_lists; //sizes of chunk offsets

}

struct chunk\_servers\_op

{

pulse(); //write memory back to disk

perform\_checksum(); //run a checksum between the two partitions

perform\_mergesort(); //run on flat files to speed things up

perform\_garbage(); //garbage collector

create\_file(); //create a file in this chunk server

//and in file structures

read-file(); //passed from above. Search directory flat

//list for correct chunks making up file.

//piece together chunks to make file and

//return as a buffer

update\_file(); //search flat list for file and piece together

//update pieced together file and

//write back as chunks in a buffer

delete\_file(); //delete file f rom flat list

rename\_file() //rename a file from flat list

new\_file() //assign chunks to new file

//write to flat\_list

delete\_chunk() ///delete a chunk from the chunk server

read\_chunk() //read CHUNK

add\_chunk() //add a chunk

add\_chunk\_srvr() //add a chunk server

void(\*mkdir) //make a directory

void(\*rmdir) //remove a directory

void(\*followalias) //follow alias

void(\*followshortcut) //follow shortcut

void(\*setattr) //set attributes (dugorx)

void(\*getattr) //get attributes (dugorx)

void(\*find\_file) //find a file

void(\*open\_file) //opena file

void(\*flat\_list\_merge\_stort) ///merge sort flat list

void(\*garbage\_collect) //garbage collection

void(\*compress) //compress

void(\*encrypt) //encrypt

}

struct chunks

{

compression\_flag //is this chunk compressed?

struct reverseVirtualMemory ; //take disk and swap to memory

//take this chunk and put in a data structure in memory

//whenever the one on disk is polled it is directed to

//memory

Boolean inMemory; //is this chunk in memory or stuck on disk

unsigned long chunk\_size; //chunk size in bytes

unsigned long avail\_size; //available size in bytes

address data\_block; //the address of the data block in bytes on disk

file\_with\_association; //file associated with this chunk

sectors\_per\_cluster; ///number of sectors per clusters

chunkname //name of chunk

block size //block size

bytes free //bytes free

int offset; //location ondisk

keptinmemory //amount kept in memory

hashtable //enter in chunk name, returns file struct

databytes; //the actual data of the chunk!!!!!

}

struct chunks\_op

{

create\_file\_portion(); ///create a portion of a file as this chunk

}

struct file //virtual table to represent a file from a bunch of chunkcs

{

bool encrypted\_flag; //currently encrypted

bool compressed\_flag; //currently compressed

struct user perm\_list\_user;

struct group perm\_list\_group;

struct world perm\_list\_world;

struct roles perm\_list\_roles

string filetype; //exec, so, dll, etc.’

bool directory\_flag; //is this a directory

struct dentry mount; //directory mount point

struct mntpoints m\_list; //what things are mounted here

string virtual\_path; //fake a hierarchy

unsigned long file\_size; //file size in bytes

unsigned long address; //address of start of file

string name; //human readable name

chunks filechunks; //the chunks that make up this file

file\_forks; //invisible redundancy to the file

creation\_time

last\_modification\_time

deletion\_time

dirty\_flag //was modified

encrypted\_flag

locked //

struct roles\_authorized

alias

shortcut

symbolic link

hidden

uniqueidentifier

location\_idx //number in the flat list

}

struct sandbox {

chunkservers

}

struct file\_ops

{

create\_file(); //create a file in file structure

create\_file\_fork() //create an invisible redundancy of the file

void(\*mkdir) //make a directory

void(\*rmdir) //remove a directory

void(\*followalias) //follow alias

void(\*followshortcut) //follow shortcut

void(\*setattr) //set attributes (dugorx)

void(\*getattr) //get attributes (dugorx)

}

READ PROCESS

Master\_drive\_ops.readfile() -> check quick cache ->check read cache -> logical partition server -> chunk servers -> request buffer data into DO (chunks making up file) -> getflatlist -> piece together chunks and return

Master)drive\_ops.writefile() -> check quick cache -> check write cache -> logical partition\_server -> chunk servers -> -> break file into chunks -> write buffer data into a DO as chunks -> get flatlist -> piece together chunks and write

Master\_drive\_ops.readfile() -> check quick cache ->check read cache -> logical partition server -> chunk servers -> request file from chunks->compressionflag(off)->encrypted\_flag(off)-> request file into buffer

MSTR LOGICAL LOGICAL CHNKSRV CHUNKS CHNKSRV CHUNKS LOGIC…..