

File Systems

Optimized C++

Ed Keenan

20 February 2019

13.0.6.4.12 Mayan Long Count

Copyright 2019, Ed Keenan, all rights reserved.



Goals

- Basics
- Seeking
- Archives
- OS Calls
- Protocol Buffers

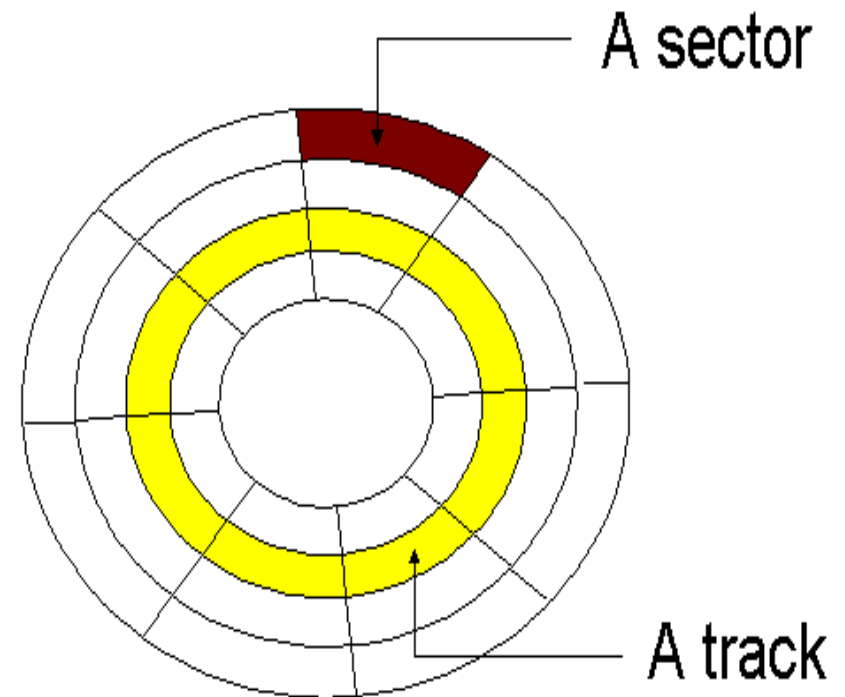


File Systems
Rule!



Basics

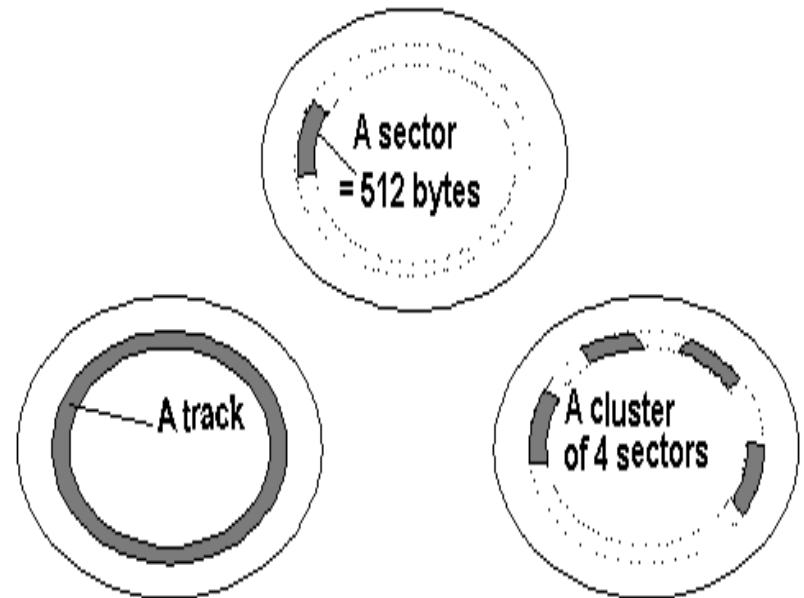
- **Sectors**
 - All disks are divided in 512 byte sectors.
- **Track**
 - Each track is divided into sectors.
- **Clusters**
 - Associated sectors
- **Cylinders**
 - Number of platters or discs





Clusters

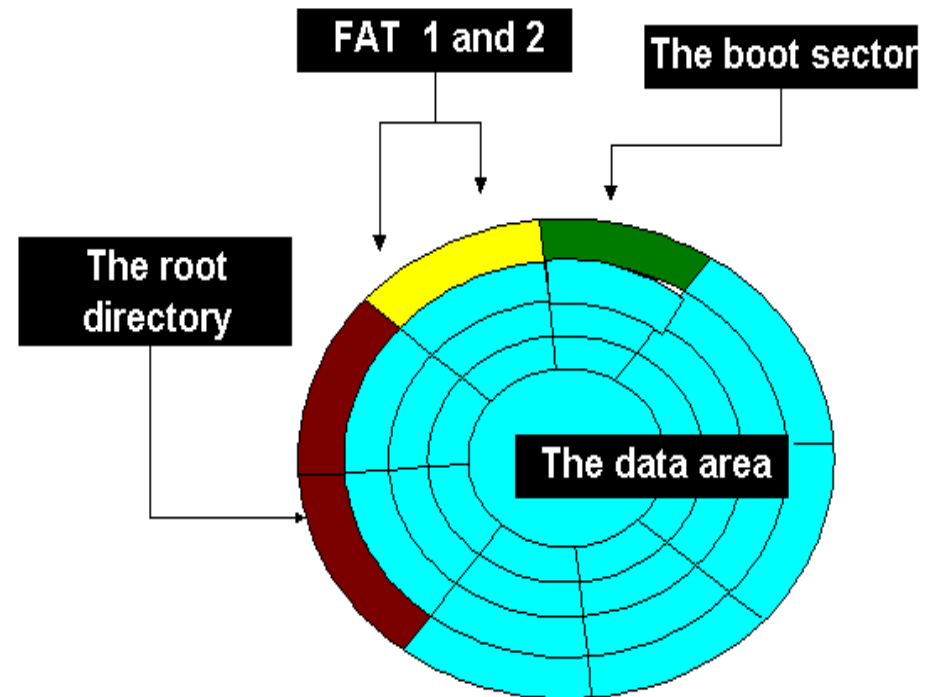
- Clusters
 - FAT formatting the sectors are gathered in *clusters* of 2, 4, 8, 16, 32, or 64 sectors
- FAT 32
 - 32-bits of range
 - The clusters represent a size of one value
- Hard Drive
 - <8 GB - 4K cluster
 - <16 GB - 8K cluster
 - <32 GB - 16K cluster
 - >32 GB - 32K cluster





Reserved Sectors

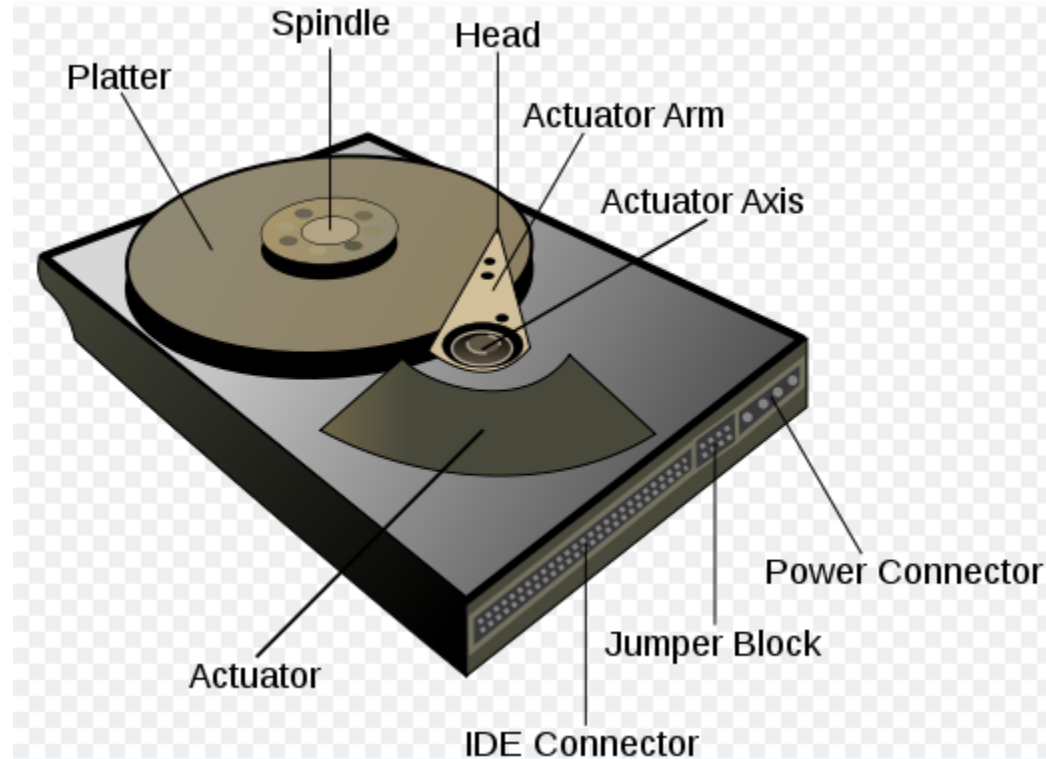
- Outside track
 - Root, Fat and Boot sectors, are on the outside track
 - Why?
- Sectors
 - Formatting and partitioning of disc is broken is through reserved sectors.





Mechanics

- Head moves back and forth
 - In – towards the center
 - Out – towards the outside edge
- Outside Edge == fast
 - Outside tracks have faster data rates since the head can read data faster





Seeking

- Seeking
 - Read or write data in a particular place on the disk
 - Head of the disk needs to be moved to the correct place.
- Time it takes for the head to move to the right place is the **Seek Time**.





Seeking is a PLG

- Seek time is very dominant
 - Extremely slow
 - Roughly 2 ms per seek.
 - Yikes!
 - Seeking is proportional to distance the head moves
- Real-time games:
 - Video games run at 60 Hz
 - That's 16.66ms to do everything
 - 2 seeks is an eternity!



Less is more

- Load in the beginning of game
 - Between levels
 - Mask off loads while playing movies
- Minimal Run-time loads
 - Make sure any file loading is necessary
 - No way around it
- Do not use file system as a swap
 - Try to plan the game
 - Stay in memory



Restrictions

- Game is mostly Read ONLY use
 - Very rarely do we write to media
- DVD / CD / Blu-Ray
 - Same principles for DVD as for hard drives
 - Except generally you do not write to media.



Archive

- Open / Closing files are costly
 - Many small files == **SLOW**
- Combine many files into an **Archive**
 - Parse through the file to get the chunks out.
 - Many Chunking schemes
 - IFF, TIFF, RIFF, PNG, WAV
- Chunks
 - Each chunk contains a header which indicates some parameters (e.g. the type of chunk, comments, size etc.)
 - There is an area containing data which are decoded by the program from the parameters in the header



Disk Layout

- Data on outside edge is faster to read.
 - Put critical data on outside area.
- Repeat Data
 - Multiply pack same data in several archives
 - Depending the head location
 - Critical files located in several locations
 - minimize seek
- DRY principle (don't repeat yourself)
 - Might want to break the DRY concept
 - You might want a small file in several archives instead of separating into one common file



Fragmentation

- Create your own ISO to control physical layout
 - Keep files aligned to Cluster boundaries
 - Small files should still take integer number of clusters
 - Better caching.
- If amount of data is smaller than disk.
 - Keep everything packed towards outside track
 - Faster



Elevator Algorithm

- Very advanced file systems
 - Control the head access
 - Queue several requests
 - Sort based on track positions
 - Read in order of tracks
 - Just like the elevator,
 - Read outside track first, then next inner and so forth.
 - Reverse the process for next set of reads.
- Several different algorithms
 - SCAN, LOOK, C-SCAN, SSTF, C-LOOK



Handles

- Files systems use handles
 - Keeps the critical resources isolated from abuse.
 - Necessary for multiple processes and threads
- **HANDLE is an opaque data type**
 - An object that controls access to another.
 - Controls the acquisition and release of resources



File should be Runtime Ready

- Files systems should deal with binary data
 - Not a parse friendly text format like XML
 - Should process this offline and store results
- Do as much preprocess as possible
 - You can even have files ready for runtime execution.
 - Load In Place (file system)
 - Convert data's endianness offline
 - Big or little, data can be converted for the processor



Streams

- File functions versus streams
 - Your preference
 - There are many pros / cons on both.
 - I don't like them
 - Asynchronous interfaces don't support streams
- You can wrap file functions inside streams
 - So if are a stream person, your good.



Basic functionality

- Learn all the basic functionality
 - Open – open file
 - Close – closes file
 - Seek – moves the head to a specific offset
 - Flush – writes and buffers to media
 - Read – read data
 - Write – writing data
 - Tell – returns the location of the head
 - Rewind – moves back to beginning of file



Use Specialize OS calls

- Every system has specialize functions to use.
 - They are faster than the standard:
 - Fopen, fclose, fwrite, ...
- Win32 has many functions to speed up access:
 - CreateFile(), ReadFile(), WriteFile(), CloseHandle()....
- Other specialized
 - Xbox, PS3, Iphone, Wii have their specialize functions as well.



Buffers are your friend

- File reads happens in buffers
 - Even if the read is a few bytes
 - Whole buffer is read
 - Then divide up based on the function calls
- Only use buffer reads
 - Do all of your file access in terms of the buffer cache size
 - Create functions to deal with subdivision of buffer
 - Buffer management
- Find what's the natural size of buffers
 - Do various read tests on buffers of different sizes
 - Measure your results
 - That's the optimal size for all data reads



Synchronous File access

- Most files systems are synchronous
 - They block, while the file transfer is happening
 - Slows down performance
- There are asynchronous file system calls
 - You get a call back when file is done
 - Can be processed on a different thread.
 - Doesn't slow down the game



What if you need XML?

- Sometimes you need XML – like capabilities
 - Relationship data
 - Unstructured data
 - Stuff other than Binary



Protocol Buffers

- Google Protocol Buffers

- Website

- <http://code.google.com/p/protobuf/>
 - SDK, documentation located on this site

- Definition

- A language-neutral, platform-neutral, extensible way of **serializing structured data**
 - Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data
 - Think XML, but smaller, faster, and simpler



Protocol Buffers

- Define how you want your data
 - Write and read your structured data to and from a variety of data streams
 - C++, C#, Java, Python languages supported
 - Update your data structure without breaking deployed programs that are compiled against the "old" format.



Why not just use XML?

- Protocol buffers have many advantages over XML for serializing structured data
 - Protocol buffers:
 - are simpler
 - are 3 to 10 times smaller
 - are **20 to 100 times faster**
 - are less ambiguous
 - generate data access classes that are easier to use programmatically



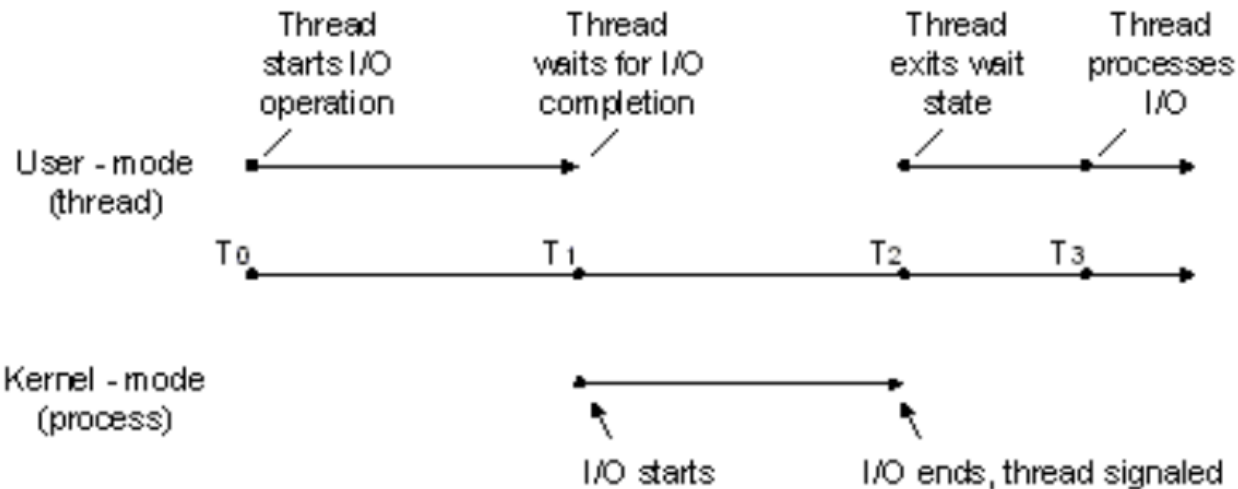
History of Protocol Buffers

- Protocol buffers were initially developed at Google to deal with an index server request/response protocol
 - Prior to protocol buffers
 - format for requests and responses that used hand **marshalling / unmarshalling** of requests and responses
 - Supported a number of versions of the protocol
- Protocol buffers were designed to:
 - New fields could be easily introduced
 - Formats were more self-describing, and could be dealt with from a variety of languages

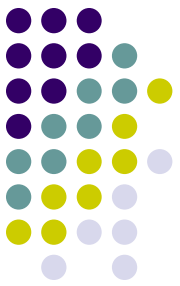


Synchronous File I/O

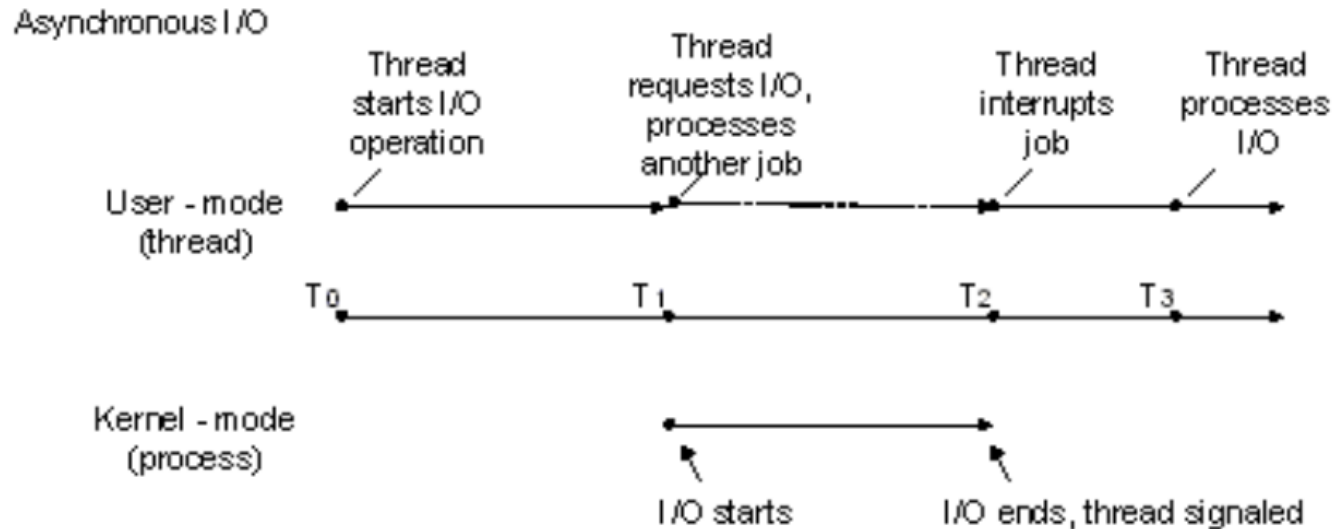
Synchronous I/O



- ***Synchronous file I/O***
 - A thread starts an I/O operation and immediately enters a wait state until the I/O request has completed.



Asynchronous File I/O



- **Asynchronous file I/O**

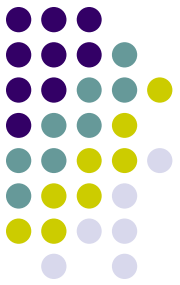
- A thread sends an I/O request to the kernel
- Thread continues processing another job until the kernel signals to the thread that the I/O operation is complete
- It then interrupts its current job and processes the data from the I/O operation as necessary



Async file I/O

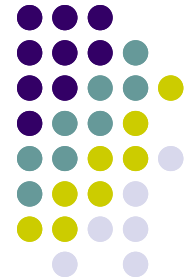
- Beyond the focus of this class
 - Should be in a Multi-Thread class
 - CSC 462/362
- Very tied to the OS kernel and support
 - Need to make platform specific calls
 - Race / Starvation / Synchronization problems
 - Threading issues

Thank You!



- Questions?





Load in Place

Optimized C++

Ed Keenan



Goals

- Load in Place
 - Motivation
 - How to Use it
- In place loading
 - Not used enough in practice
 - Critical for embedded systems
 - Extremely beneficial
- Useful at parties?
 - Yes
 - Just as sure as pizza falls
cheese side down on the ground



Finally a valid
reason



Motivation

- What's the fastest way to draw a polygon?
 - Determine that you don't need to draw it
 - Skip it
- Common knowledge
 - Optimize the most used function
 - Often people optimize the most used function
 - Getting 10-40% FASTER



Motivation

- If you can skip the function call all together
 - 10 x faster
 - 100 x faster
 - 1 million x faster
 - or even a 1 billion x faster
 - 1,000,000,000 😊
- Nothing is faster than skipping
 - Doing no work takes 0 seconds

Problem:

Construction from File



- Constructions of objects from a File
 - Reads data from file
 - Creates object
 - Through operator new
 - Call constructor
 - Copy data from file to new object



Problem:

Construction from File

- What happens if you can skip construction of objects?
 - That would be fast!
 - No unnecessary copies
 - Touch memory once and only once
 - No new operators
 - Remember they are slow



Load in Place

- **Definition:**
 - Instead of loading many discrete objects from media, load one large load that contains many small elements
 - Loading is contiguous
 - Pointers are fixed up, after load
 - File access is done in large reads of buffers at a time.
 - File reads write to dynamic buffer directly

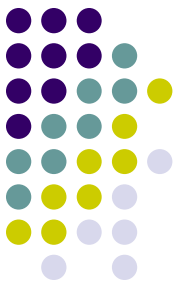
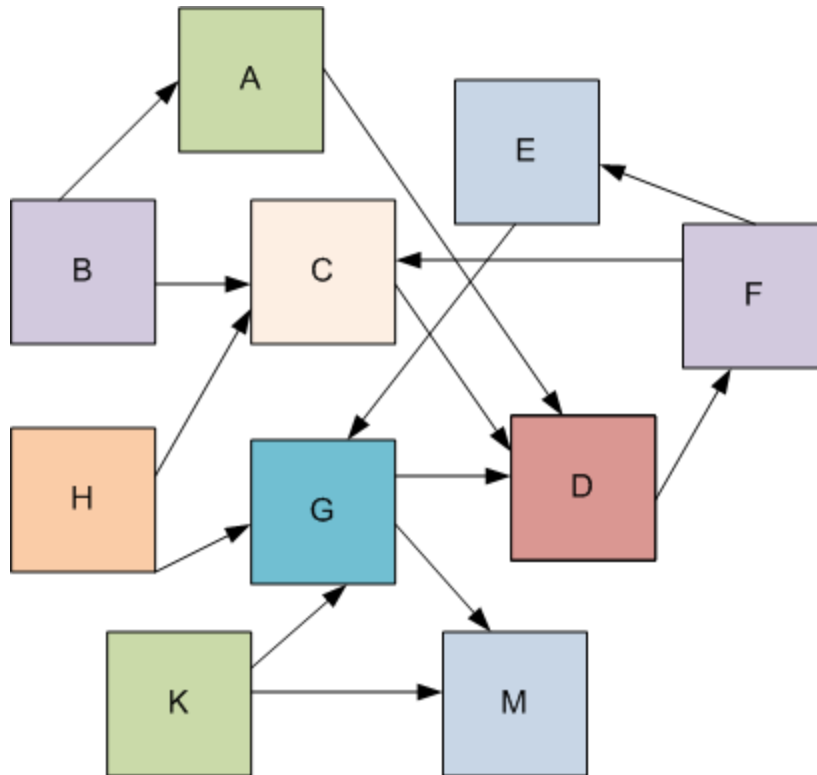
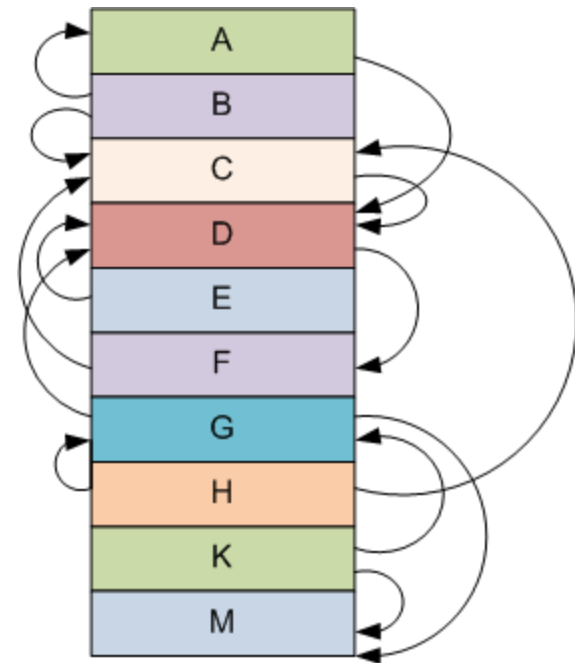


Photo == 1K Words



Disjointed Memory Objects



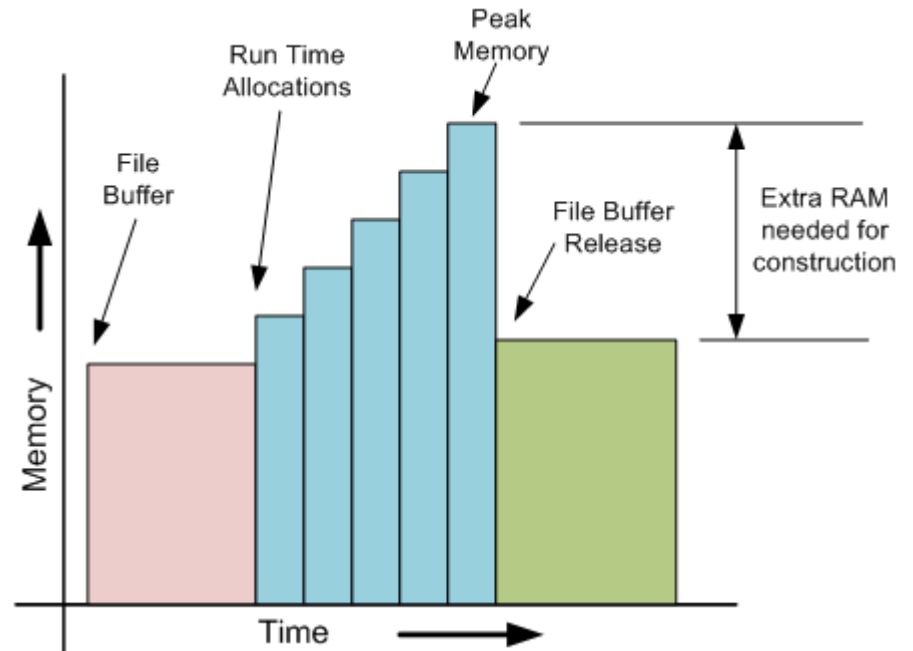
Contiguous Memory Objects



Memory Spike

● Common Use Pattern

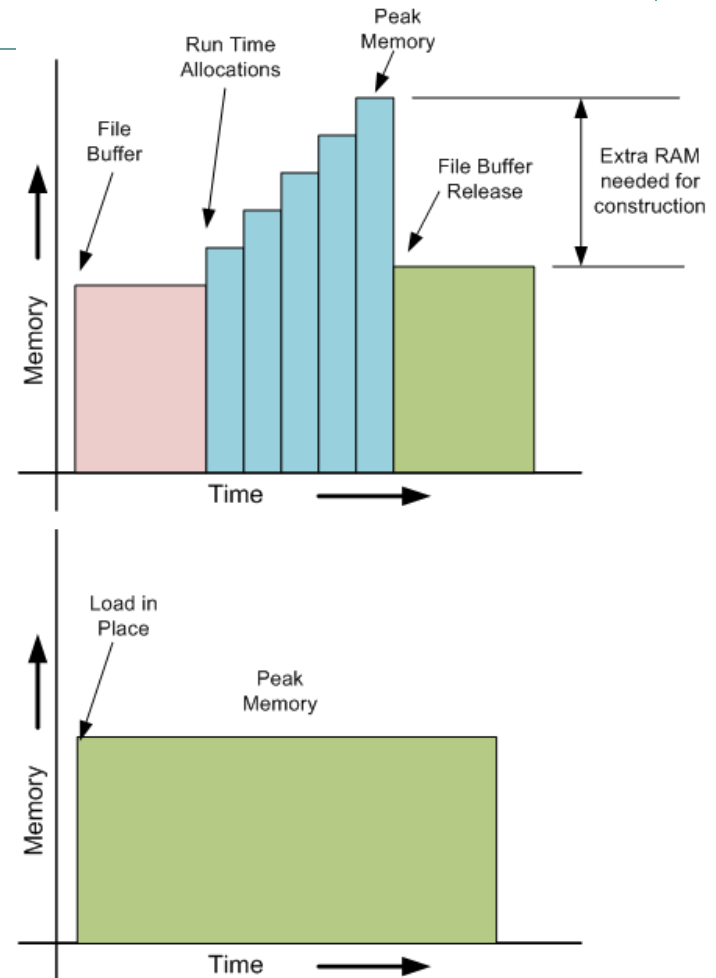
1. Load large buffer from file system into RAM
2. Create objects from the buffer
 - Dynamically allocated blocks
 - Load / initialize data from buffer
3. Free the large file buffer





Embedded systems

- You have a **fixed amount** of memory
 - Memory spikes are sometimes not possible.
- **Load-in-Place**
 - Removes memory spikes
 - As well as speed up loading





Off-line > Runtime

- Off-line: Good
 - Pre-process as much as possible.
 - Doesn't matter how much time or effort to convert data.
 - Reason why **God** invented computers
- Runtime – be minimal
 - If there is a way any conversion could have been known in advanced
 - Do it off-line
 - Only must have operations in Runtime
 - Pointer fix-up
 - Dynamic allocations



Example Class

```
// Example Class
class Dog
{
public:
    // pointer to next Dog object
    Dog *next;

    // member data A & B
    int a;
    int b;
};
```

- Sample class Dog
 - Has pointer to next
 - Data members
- Example
 - Show case issues with pointers and data



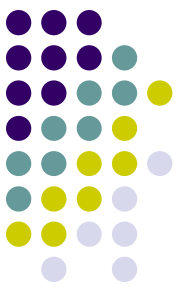
Use pattern

- Pseudo code:
 - Load many discrete objects from file
 - Create dynamic space
 - Copy the data
 - Insert into some active list.
- When done
 - Delete all files

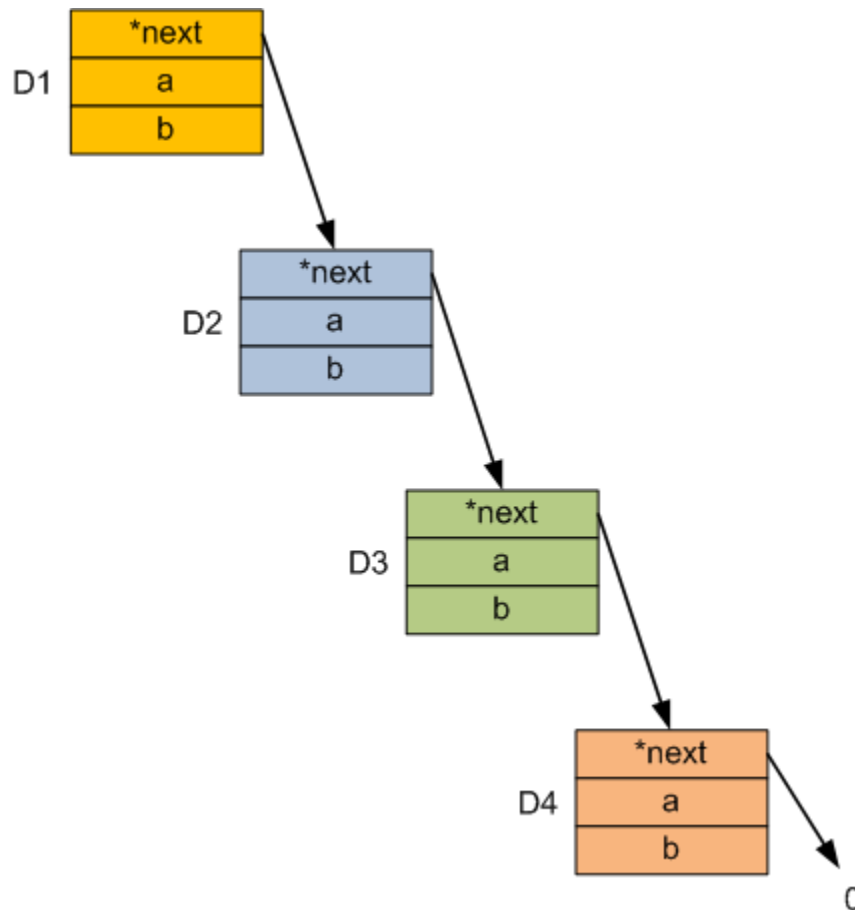
```
// load many objects from file into RAM
for( i = 0; i<ObjectCount; i++)
{
    // Create object
    Dog *p = new Dog;

    // Get Dog object from file
    loadObjectFromFile(...);

    // insert object to list or system
    insertObjectToList(...);
}
```



Load several objects



```
Dog *d1 = new Dog;  
Dog *d2 = new Dog;  
Dog *d3 = new Dog;  
Dog *d4 = new Dog;
```

```
d1->a = 0x11111111;  
d1->b = 0x11111112;  
d1->next = d2;
```

```
d2->a = 0x22222221;  
d2->b = 0x22222222;  
d2->next = d3;
```

```
d3->a = 0x33333331;  
d3->b = 0x33333332;  
d3->next = d4;
```

```
d4->a = 0x44444441;  
d4->b = 0x44444442;  
d4->next = 0;
```

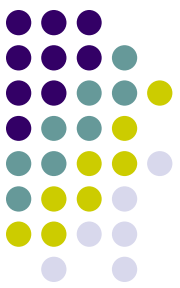


Objects in Memory

Address	Value
0x0034A3E8	0034a430
0x0034A3EC	11111111
0x0034A3F0	11111112
0x0034A3F4	fdfdfffd
0x0034A3F8	abababab
0x0034A3FC	abababab
0x0034A400	00000000
0x0034A40C	000
0x0034A40C	next 009
0x0034A40C	0018073a
0x0034A410	0034a3c8
0x0034A414	0034a458
0x0034A418	00000000
0x0034A41C	00000000
0x0034A420	0000000c
0x0034A424	00000001
0x0034A428	000000a6
0x0034A42C	fdfdfffd
0x0034A430	0034a478
0x0034A434	22222221
0x0034A438	22222222
0x0034A43C	fdfdfffd
0x0034A440	abababab
0x0034A444	abababab
0x0034A448	00000000
0x0034A44C	00000000
0x0034A450	00090009
0x0034A454	00180731

- Every allocation is disjointed
 - Not continuous in memory
 - Pointers are correctly pointing to the next block

Converting to: Contiguous Memory



- Need to have all allocations into contiguous memory
 - Placement new
 - Starting to be our friend in this class
- Data Structure Layout
 - Data needs to be correctly padded for alignment

```
// Example Class
class Dog
{
public:
    Dog *next;
    int a;
    int b;
};
```

- Structure alignment
 - 4-byte alignment
- 12 – byte structure
 - No padding
 - 4 – (Dog *next)
 - 4 – (int a)
 - 4 – (int b)



Load data into Single Buffer

```
// size of Block
int sizeInPlaceBlock = 4 * sizeof(Dog);

// create buffer
char *p = new char [sizeInPlaceBlock];
char *pOrig = p;

// instantiate d1
Dog *d1 = new(p) Dog;

// instantiate d2
p = p+sizeof(Dog);
Dog *d2 = new(p) Dog;

// instantiate d3
p = p+sizeof(Dog);
Dog *d3 = new(p) Dog;

// instantiate d4
p = p+sizeof(Dog);
Dog *d4 = new(p) Dog;
```

- Create a buffer
 - Calculate the total size
 - Instantiate the object
- Objects are place
 - Sequentially in memory
 - No padded or unused memory

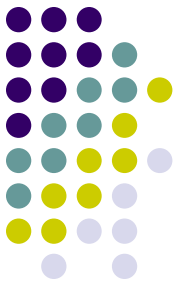


Data layout memory

- Data is contiguous
 - Pointers are correct
 - Data is valid
- What happens if the data is loaded into a different buffer?
 - All the pointers would change
 - Need to get this into reproducible format

0x0034A368	0034a374	d1
0x0034A36C	11111111	
0x0034A370	11111112	
0x0034A374	0034a380	d2
0x0034A378	22222221	
0x0034A37C	22222222	
0x0034A380	0034a38c	
0x0034A384	33333331	d3
0x0034A388	33333332	
0x0034A38C	00000000	
0x0034A390	44444441	
0x0034A394	44444442	d4

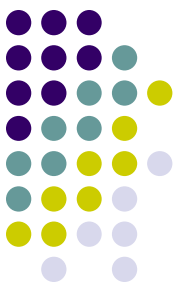
2nd Copy Shows pattern



Memory 1	
A - Copy	
0x0034A368	0034a374
0x0034A36C	11111111
0x0034A370	11111112
0x0034A374	0034a380
0x0034A378	22222221
0x0034A37C	22222222
0x0034A380	0034a38c
0x0034A384	33333331
0x0034A388	33333332
0x0034A38C	00000000
0x0034A390	44444441
0x0034A394	44444442

Memory 2	
B - Copy	
0x0034A3D8	0034a3e4
0x0034A3DC	11111111
0x0034A3E0	11111112
0x0034A3E4	0034a3f0
0x0034A3E8	22222221
0x0034A3EC	22222222
0x0034A3F0	0034a3fc
0x0034A3F4	33333331
0x0034A3F8	33333332
0x0034A3FC	00000000
0x0034A400	44444441
0x0034A404	44444442

- Instantiate objects
 - Into a second buffer.
- What changed?
 - Only the pointers
 - NULL pointers didn't changed



Difference Buffer

Memory 1	Memory 2	Memory 3
A - Copy	B - Copy	C = B-A
0x0034A368 0034a374	0x0034A3D8 0034a3e4	0x0034A458 00000070
0x0034A36C 11111111	0x0034A3DC 11111111	0x0034A45C 00000000
0x0034A370 11111112	0x0034A3E0 11111112	0x0034A460 00000000
0x0034A374 0034a380	0x0034A3E4 0034a3f0	0x0034A464 00000070
0x0034A378 22222221	0x0034A3E8 22222221	0x0034A468 00000000
0x0034A37C 22222222	0x0034A3EC 22222222	0x0034A46C 00000000
0x0034A380 0034a38c	0x0034A3F0 0034a3fc	0x0034A470 00000070
0x0034A384 33333331	0x0034A3F4 33333331	0x0034A474 00000000
0x0034A388 33333332	0x0034A3F8 33333332	0x0034A478 00000000
0x0034A38C 00000000	0x0034A3FC 00000000	0x0034A47C 00000000
0x0034A390 44444441	0x0034A400 44444441	0x0034A480 00000000
0x0034A394 44444442	0x0034A404 44444442	0x0034A484 00000000

- Subtracting data element by data element
 - Determines pointers versus data
 - Data is Static yields 0
 - Pointers yields offsets between buffers
 - NULL pointers now behave like static data



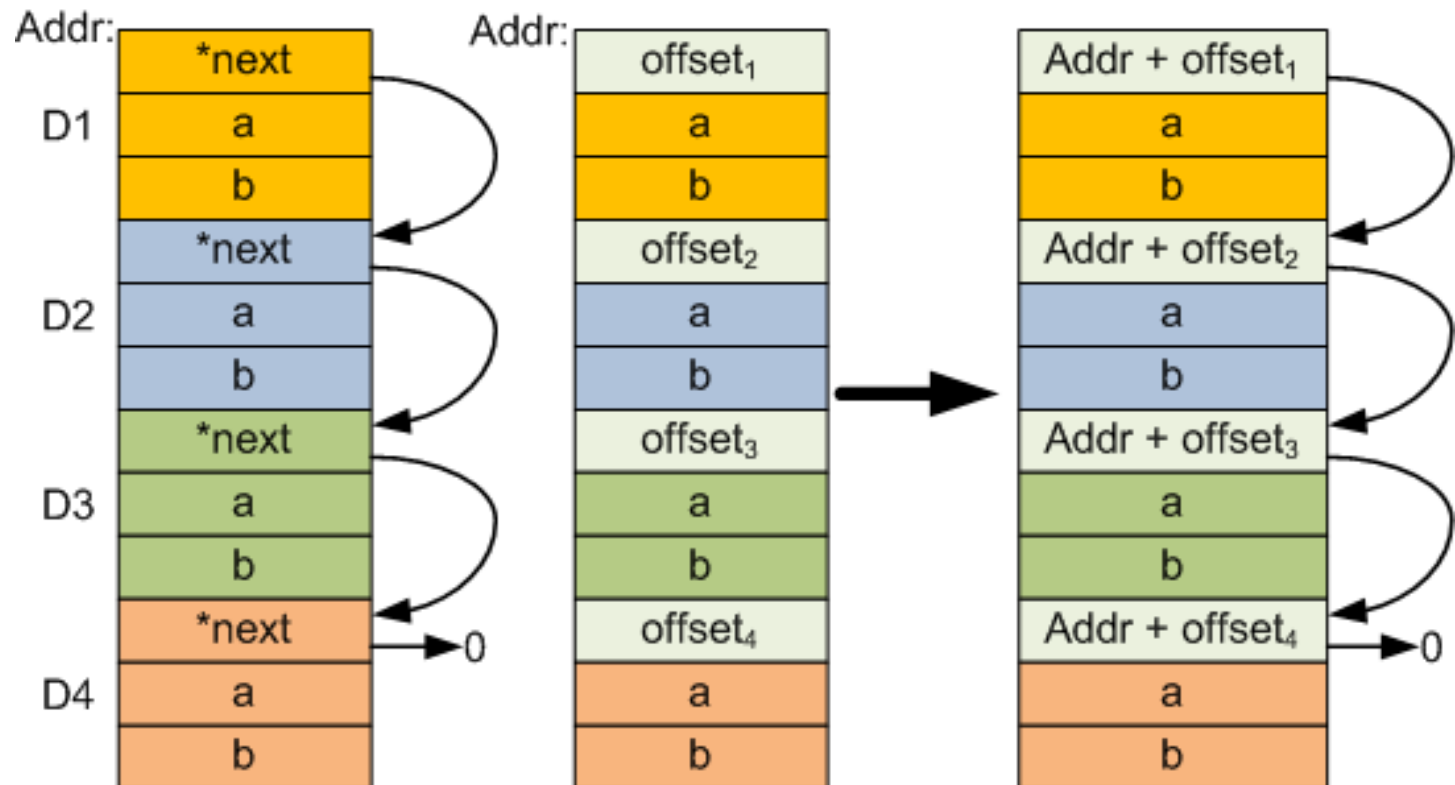
Deeper look at buffer

C = B-A	
0x0034A458	00000070
0x0034A45C	00000000
0x0034A460	00000000
0x0034A464	00000070
0x0034A468	00000000
0x0034A46C	00000000
0x0034A470	00000070
0x0034A474	00000000
0x0034A478	00000000
0x0034A47C	00000000
0x0034A480	00000000
0x0034A484	00000000

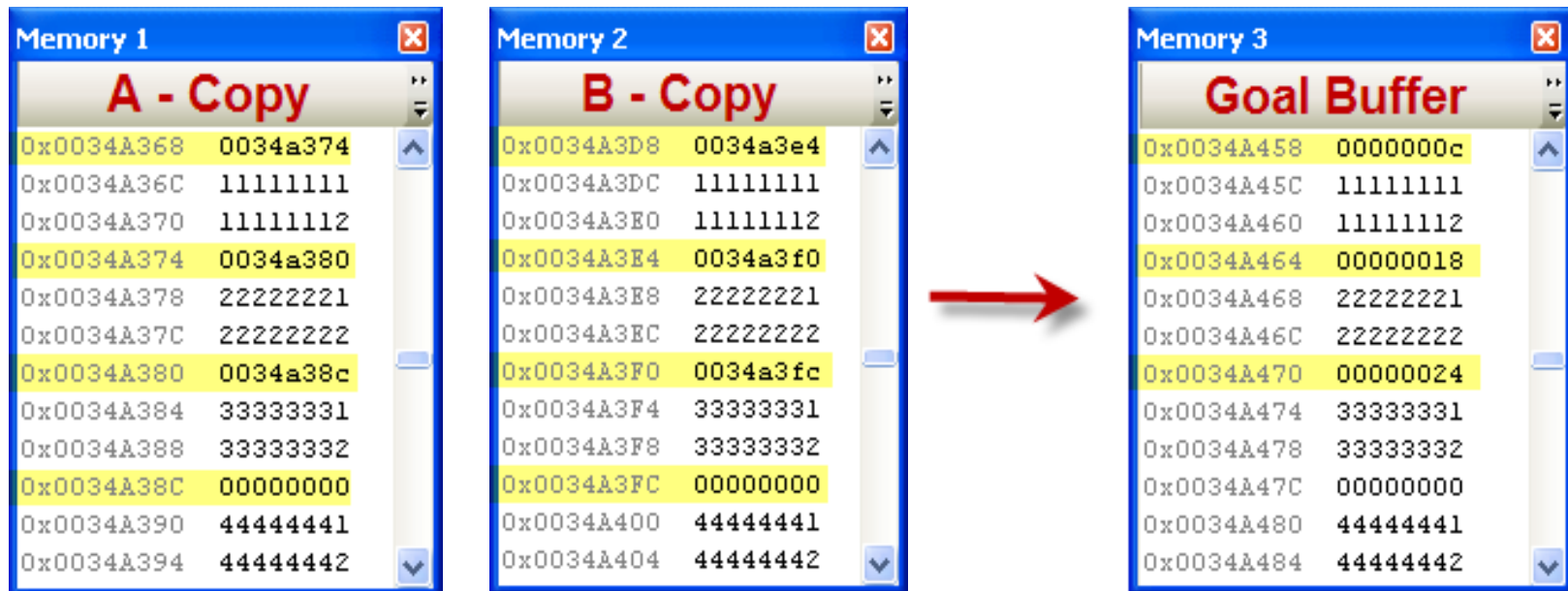
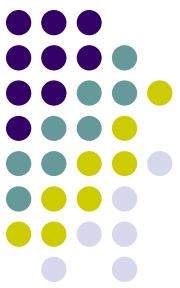
- Highlighted shows the difference in buffers
 - Only pointers that changed
- Why are the same values?
 - Difference between the 2 address buffers
- How do we exploit this?



Self-Relative Buffer

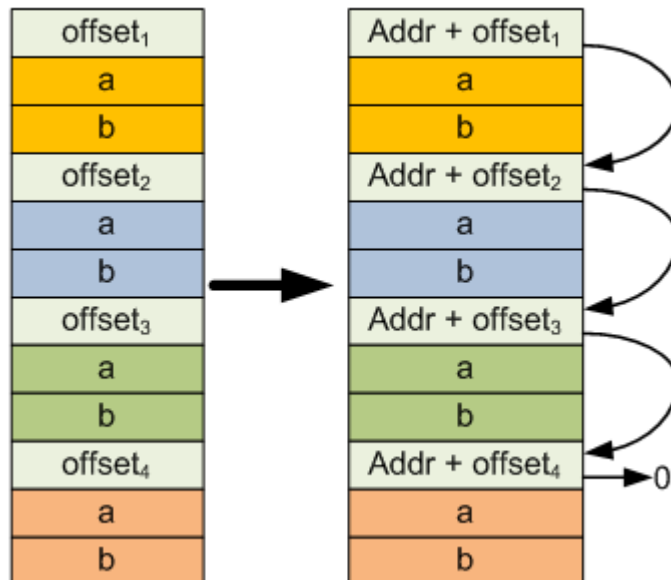
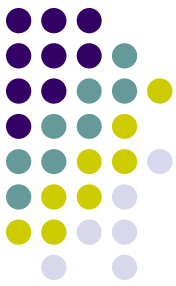


The Goal



- Using Buffer A and B
- Create the desired buffer

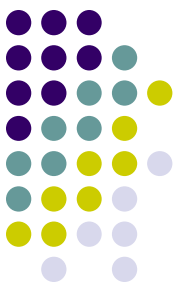
Verify it



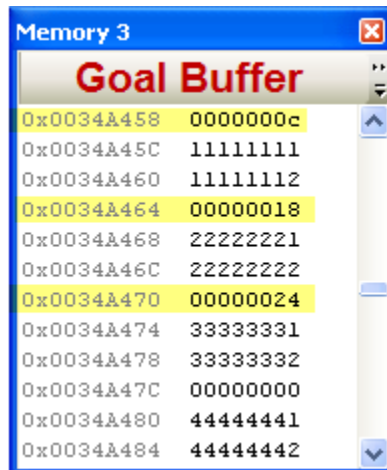
Memory 1	
Buffer	
0x0034A368	0000000c
0x0034A36C	11111111
0x0034A370	11111112
0x0034A374	00000018
0x0034A378	22222221
0x0034A37C	22222222
0x0034A380	00000024
0x0034A384	33333331
0x0034A388	33333332
0x0034A38C	00000000
0x0034A390	44444441
0x0034A394	44444442

Memory 1	
A	
0x0034A368	0034a374
0x0034A36C	11111111
0x0034A370	11111112
0x0034A374	0034a380
0x0034A378	22222221
0x0034A37C	22222222
0x0034A380	0034a38c
0x0034A384	33333331
0x0034A388	33333332
0x0034A38C	00000000
0x0034A390	44444441
0x0034A394	44444442

- Verify
 - Start with the Buffer
 - Add the Buffer Start Address to each of the offsets
 - You get buffer A



Create the destination buffer



Goal Buffer	
0x0034A458	0000000c
0x0034A45C	11111111
0x0034A460	11111112
0x0034A464	00000018
0x0034A468	22222221
0x0034A46C	22222222
0x0034A470	00000024
0x0034A474	33333331
0x0034A478	33333332
0x0034A47C	00000000
0x0034A480	44444441
0x0034A484	44444442

```
// store the pointers' address that you modified
std::vector< int > fixup;
```

```
// num int * in block
int numIntPtrInBlock = (4 * sizeof(Dog) ) / sizeof(int *);
```

```
// setup the pointers
b = (int *)sOrig;
a = (int *)pOrig;
c = (int *)rOrig;
```

```
for( int i=0; i < numIntPtrInBlock; i++)
{
    // This address
    if( (*b - *a) != 0x0 )
    {
        // copy offset data
        *c = (unsigned int)*a - (unsigned int)pOrig;

        // squirrel away which pointers were affected
        fixup.push_back( (unsigned int)a - (unsigned int)pOrig );
    }
    else
    {
        // copy static data
        *c = *a;
    }

    // increment pointers
    a++;
    b++;
    c++;
}
```



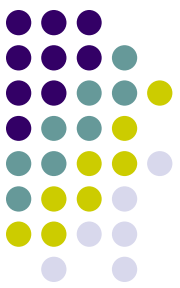
Are we done yet?

- We now have a self-relative buffer.
 - The data is contiguous
 - Address related pointers are removed
 - Replaced with offsets to the buffer address
- How do we load this from a file?
 - Which pointers or fields need to be corrected?
 - How many pointers do we correct?



Need critical info

- Need a header
 - Size of the block to load
 - Number of pointers to correct
 - Offset to pointers to correct
- Many ways to do this
 - Storage of an array of pointers
 - Storage of a vector
 - Zero-sized array
 - How many know what this is?



Zero Size Arrays

- Zero Size Array

- The unsized array at the end of the structure allows you to append a variable-sized string or other array

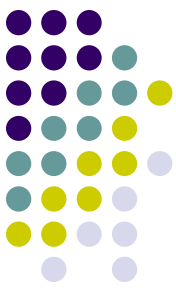
- Automagic pointer

- Avoiding the run-time execution cost of a pointer dereference.

```
struct data
{
    int a;
    int b[];
};
```

- What's the size of data?

- 4 bytes
 - 4 – bytes for int
 - 0 – bytes for b



inPlaceHdr

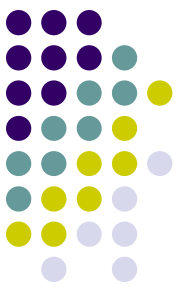
```
struct inPlaceHdr
{
    int sizeBlock;
    int numPtrs;
    int offsetToPtrs[];
};
```

Offsets	
0x0034A458	00000000
0x0034A45C	11111111
0x0034A460	11111112
0x0034A464	0000000c
0x0034A468	22222221
0x0034A46C	22222222
0x0034A470	00000018
0x0034A474	33333331
0x0034A478	33333332
0x0034A47C	00000000
0x0034A480	44444441
0x0034A484	44444442

inPlaceHdr

sizeBlock	(4*12)
numPtrs	3
offsetToPtr[0]	0
offsetToPtr[1]	0xC
offsetToPtr[2]	0x18

- Offset relative to the head pointer
 - Highlighted entries
- SizeBlock
 - Easy, that's the run-time block
- Num of offsets
 - Get this from the vector



Create inPlaceHdr

```
// Get the size of the inPlaceHdr structure in total
int sizeInPlaceHdr = sizeof(inPlaceHdr);
int vectorCount = fixup.size();
int totalSize = sizeInPlaceHdr + vectorCount * sizeof(int );

// create a buffer
char *p_inPlaceHdr = new char[ totalSize ];

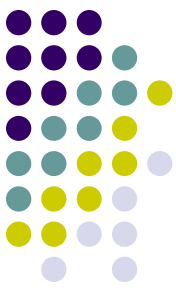
// Cast this to inPlaceHdr * pointer
inPlaceHdr *ip = (inPlaceHdr *) p_inPlaceHdr;

// Fill in the block size
ip->sizeBlock = 4 * sizeof(Dog);
ip->numPtrs = vectorCount;

// Fill in the array of offsets
for( int i=0; i < vectorCount; i++)
{
    ip->offsetToPtrs[i] = fixup[i];
}
```

inPlaceHdr	
0x0034A628	00000030
0x0034A62C	00000003
0x0034A630	00000000
0x0034A634	0000000c
0x0034A638	00000018

- Write this structure to the File



Fixing up pointers

Memory 3	
inPlaceHdr	
0x0034A628	00000030
0x0034A62C	00000003
0x0034A630	00000000
0x0034A634	0000000c
0x0034A638	00000018



Memory 2	
LIP - Buffer	
0x0034A5A8	0000000c
0x0034A5AC	11111111
0x0034A5B0	11111112
0x0034A5B4	00000018
0x0034A5B8	22222221
0x0034A5BC	22222222
0x0034A5C0	00000024
0x0034A5C4	33333331
0x0034A5C8	33333332
0x0034A5CC	00000000
0x0034A5D0	44444441
0x0034A5D4	44444442



Memory 2	
Pointer fixup	
0x0034A5A8	0034a5b4
0x0034A5AC	11111111
0x0034A5B0	11111112
0x0034A5B4	0034a5c0
0x0034A5B8	22222221
0x0034A5BC	22222222
0x0034A5C0	0034a5cc
0x0034A5C4	33333331
0x0034A5C8	33333332
0x0034A5CC	00000000
0x0034A5D0	44444441
0x0034A5D4	44444442

```
// Load in place buffer
int startAddr = (int)rOrig;
int *pCorr;

// correct the pointers relative to the start address
for(int i = 0; i < ip->numPtrs; i++)
{
    // find the pointer to fix
    pCorr = (int *) ((unsigned int)c + ip->offsetToPtrs[i]);

    // add the offset
    *pCorr = *pCorr + startAddr;
}
```

- Updated final buffer



Summary of Load-in-Place

- Runtime code
 - Convert code to use contiguous memory
- Data conversion
 - Write 2 buffers into contiguous memory
 - Buffer A & Buffer B
 - Using Buffer A & B
 - Create relocatable buffer
 - Create inPlaceHdr
 - Write the file
 - relocatable buffer
 - inPlaceHdr



Load In Place: Preconditions

- **Preconditions**
 - File I/O many objects
 - Loading many object from a file at one time.
 - Minimum coupling to external systems
 - Object pointers are point to each other
 - Minimum external coupling
 - No removing of nodes
 - Not deleting or removing any object after it's loaded
 - Can use disable flag in it's place
 - Loading / Initialization Slow
 - The loading from files and dynamic allocations are costly in time.
 - Short on Runtime Memory
 - Can't afford a the memory spike
- **If you have many of the preconditions**
 - Load-in-place is a good option to use



Alternatives

- **inPlaceHdr**
 - Alternatives to fixing up the data
 - If the data is very well behaved
 - Might be able to create a fixup function
 - Exploiting a pattern
 - i.e. every 16 byte is an offset
 - Overload placement new operator in class
- **Skewing between data and runtime structures**
 - Need version controls
 - To protect data



Additions

- Big / Little Endian
 - Pre-swizzle the data to be little or big ending offline.
 - When loading it just magically works without unnecessary conversion
- Useful pattern
 - Can be repeated on networking
 - Generalized for streaming



Additions

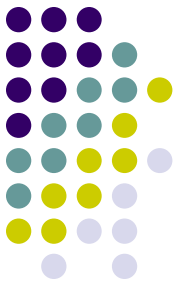
- Hierarchy models
 - Can group systems that work in an individual nodes into these super node structures
 - How do you add / delete to these load in place structures?
 - Deletion or an alternative?
 - Must preserve the individual not character of the system



Additions

- **Memory Mapped File**
 - Padding the file to align file cache
 - Faster access
 - Less page faults
 - Could waste space
 - If caching is 4K space,
 - you have a 5K file,
 - you have to buy 8K of space with 3K wasted.

Thank You!



- Questions?

