Memory Allocation

The Heap

- A block of memory, starting at x and ending at y.
- Some parts are free, other parts are in use
- The "new" command takes some free memory and makes it as in-use.
  - It is marked free again when no longer needed.

Chosing a Hole

- n bytes are requested: choose a hole of size m $(m \geq n)$; any leftover space is a new hole of size $m-n$.
- First-Fit: choose the first hole of size n $(m \geq n)$
- Best-Fit: choose the smallest hole of size $m \geq n$
- Worst-Fit: choose the largest hole, leaving the biggest possibe new hole.

Storing the list of Holes

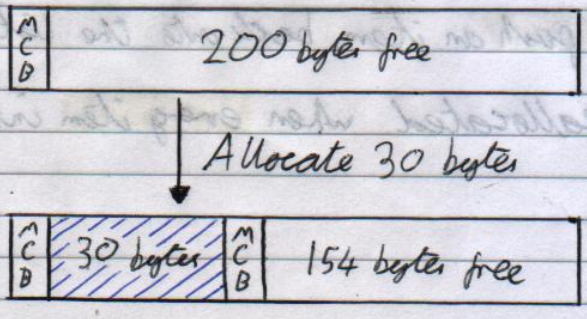
```
public class MemControlBlock {

    boolean available;        1 byte
    int size;                 4 bytes

    MemControlBlock prev;     4 bytes
    MemControlBlock next;     4 bytes

}
```

13 bytes, but an MCB consumes 16 bytes due to alignment.



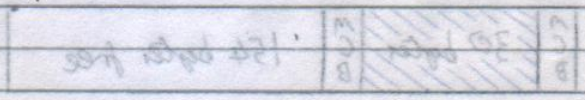
- MCBs form a doubly-linked list so that contiguous free sections can be "collapsed".

Fragmentation

- **Internal Fragmentation** - a block is large enough for the request, but not enough will be left free for the MCB.
  - Eg. If requesting 90 bytes, a 100 byte block does not have enough spare for the 16 byte MCB.
  - The entire large block is allocated - this is wasteful.
- **External Fragmentation** - enough memory may be free in total, but no one block is large enough.
- **Avoiding Fragmentation**
  - External fragmentation between processes can be resolved by "sliding" blocks to close the free segment, aka. Compaction.
    - This is easy to do when processes access memory with a "base and limit" system (CS1).
    - This does not work for internal fragmentation as all memory addresses in the code would have to be updated.
  - Internal fragmentation can be avoided by using memory pools to keep similarly sized objects together.

Memory Pools

- Allocate space for several small objects in one contiguous block.
- Keep a singly-linked list of free addresses in that block.
- Allocation request: pop an item off the list
  - None left? → Make a new block.
- Deallocation request: push an item back onto the list.
- The block is only deallocated when every item inside it is free.

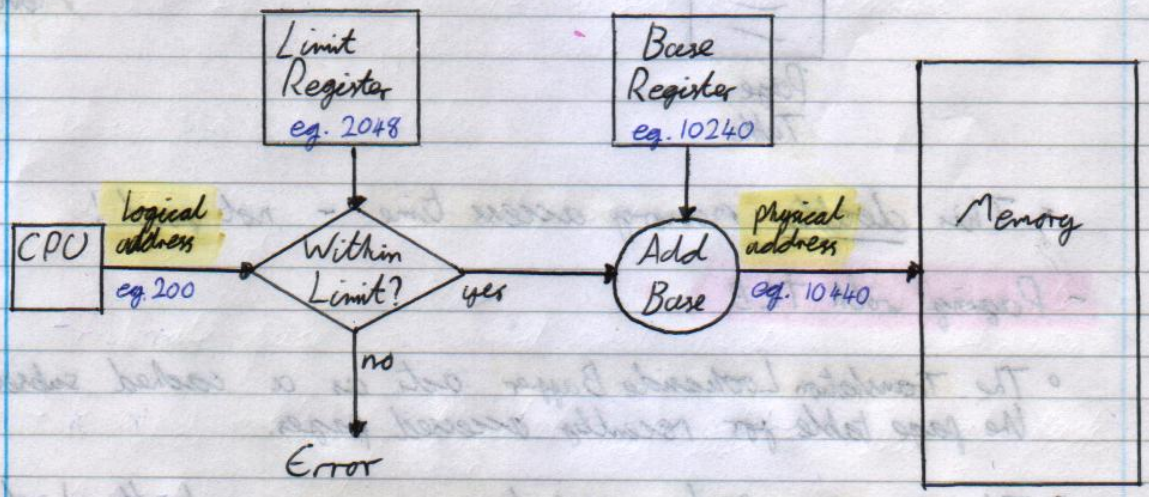


- Example:
  - Dave is an 8 byte object.
  - 1024 Daves with 1024 MCB = 1024 × 8
    + 1024 × 16
    24 kb.
  - 1024 Daves with a pool size of 60 = 18 × 60 × 8 ①
    + 18 × 16 ②
    + 54 × 8 ③
    9.1 kb

    ① 18 pools of 60 8-byte Daves
    ② 18 MCBs
    ③ 54 unused slots in the last pool, each of which has an 8-byte list node.
  - Space is wasted in the pools (54 empty slots), but overall a lot of space is saved.

Addressing Memory

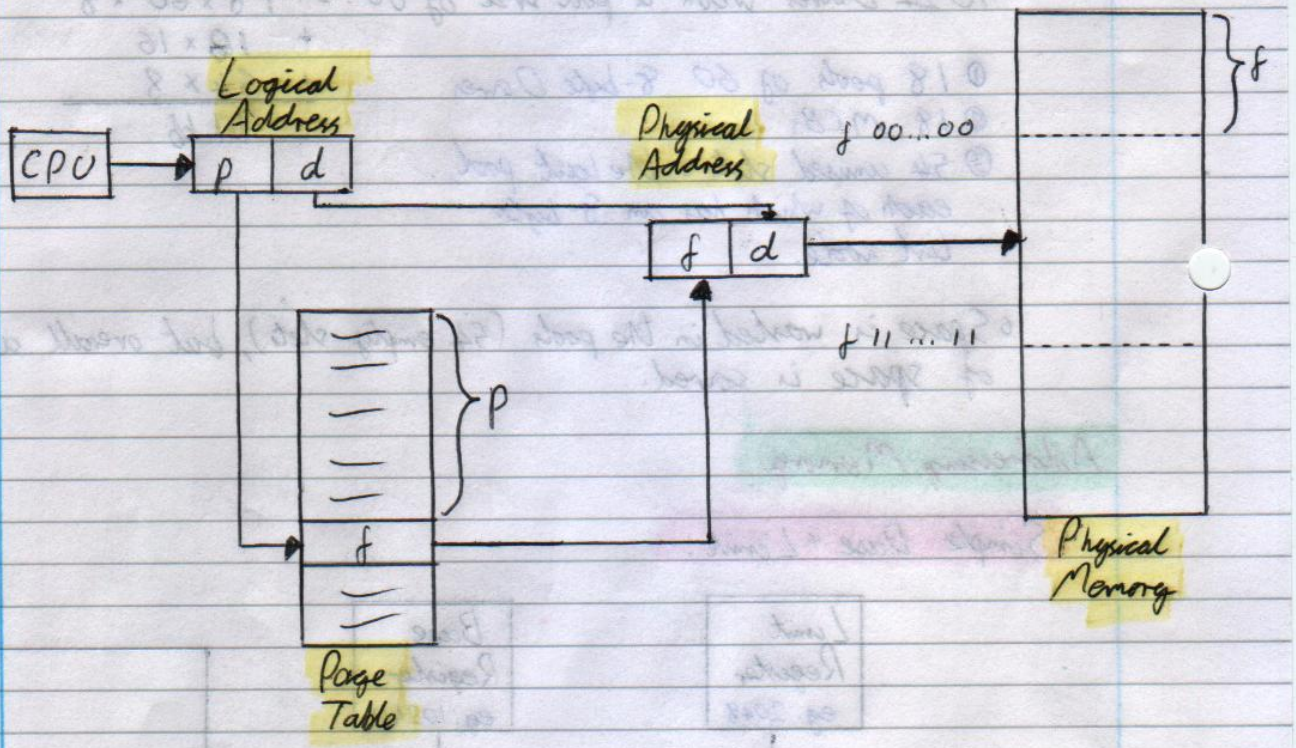
- Simple Base + Limit:



- Paging

- Split physical memory into frames of some size, eg. 4kb.
- Split logical memory into pages of the same size.
- Any page can be stored in any frame.
- A process gets one or more pages of memory.

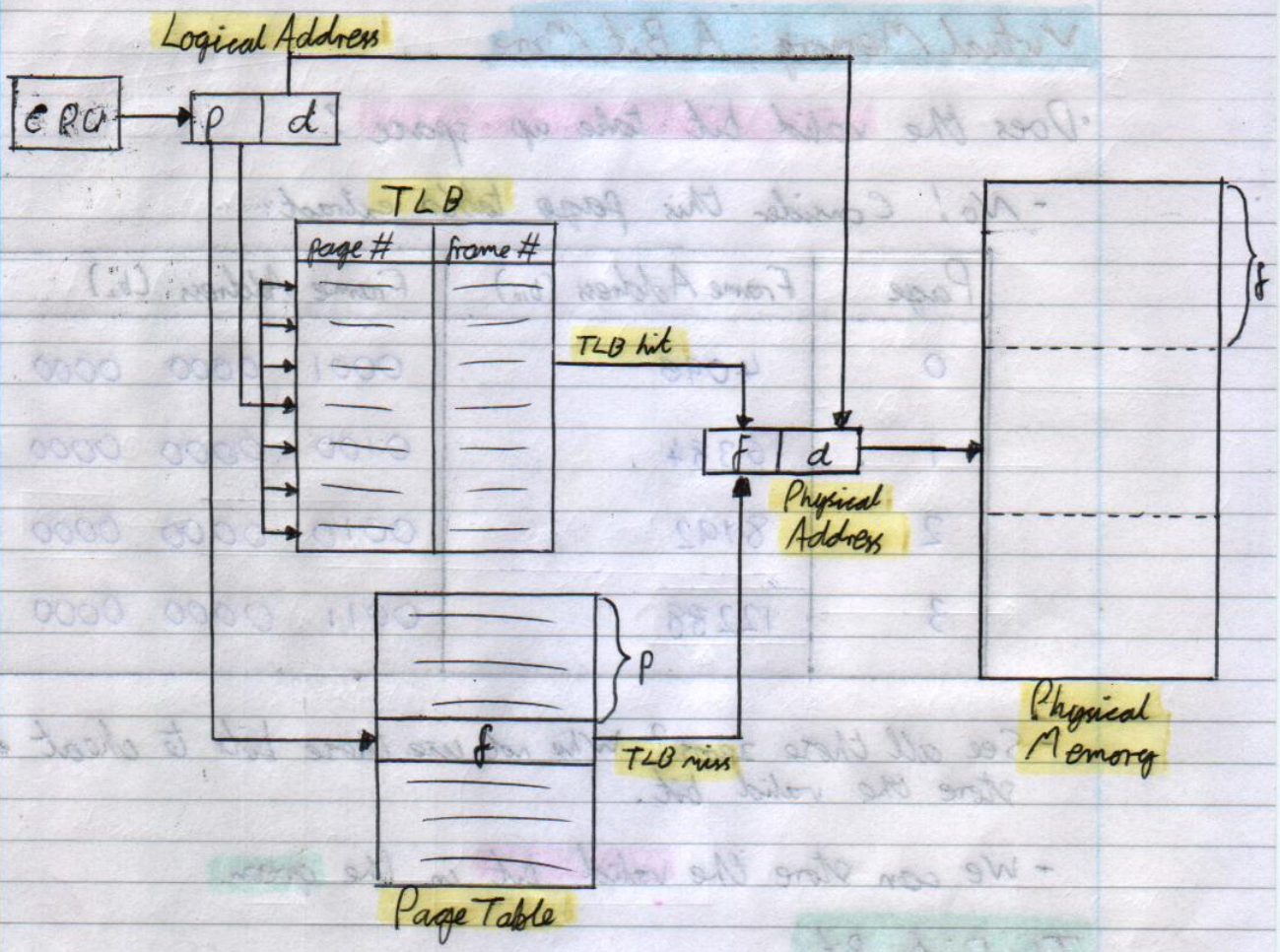
- A page table stores the memory address for each page.
- A logical memory address has two components:
  - a page number (p)
  - a page offset (d)
- A physical address is calculated as page-table[p] + d



- This doubles memory access time - not good!

- Paging with TLB

- The Translation Lookaside Buffer acts as a cached subset of the page table for recently accessed pages.
- It uses very fast associative memory - actually built in to the hardware.
- TLB hit: the correct frame for the requested page was found in the buffer and returned
- TLB miss: the frame was not found for the requested page and lookup continues as before.



- This is faster (if the TLB hit rate is good) but we will eventually run out of frames.

- Virtual Memory

- Each entry in the page table has a valid bit:
  - 1 = yes, the page is loaded in memory
  - 0 = no, it is not → page fault.
- Accessing a page p with valid bit = 0 causes the kernel to handle the page fault:
  - Choose a victim frame f in memory to copy to disk and set the valid bit to 0 in any corresponding page table entry.
  - Load p from disk into f
  - page-table[p] = f, and mark as valid
  - Get the data from (f, d) as before.
- This is called swapping
- The process doesn't know/care about it.

Pun intended.

Virtual Memory: A Bit More

- Does the valid bit take up space?
  - No! Consider this page table extract:

Page	Frame Address (b_{10})	Frame Address (b_2)
0	4096	0001 0000 0000 0000
1	16384	0100 0000 0000 0000
2	8192	0010 0000 0000 0000
3	12288	0011 0000 0000 0000

  - See all those zeros? Why not use these bits to cheat and store the valid bit.
  - We can store the valid bit in the green.

The Dirty Bit

- Swapping uses the HDD and takes time.
- What if a page was swapped out, then in, then back out?
- If the page didn't change while it was in memory then we already have a copy on disk, so we don't need to swap it out.
- We use the dirty bit to keep track:
  - 0 when a page is swapped in
  - 1 when edited
  - Don't swap out a page with DB=0
  - Orange above.

More Bits

- We can use all these zeros for things like read-only bits and execute-only bits.
- More on them later.

Choosing a Victim Frame

- We have to pick a frame to swap out.
- The page fault handler deals with this
- Aim: minimise the number of page faults
- Comparing Algorithms
  - Fix the number of free frames
  - Take some sequence of page accesses
  - Count the page faults.
  - We will use 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 for examples.
- Optimal Solution:
  - Replace the page that will not be used for the longest time.

Page Req.	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	1	1	1	4	4
Frame 2		2	2	2	2	2	2	2	2	2	2	2
Frame 3			3	3	3	3	3	3	3	3	3	3
Frame 4				4	4	4	5	5	5	5	5	5

6 Page Faults

  - Caveat: This requires looking into the future.
  - But it does give a theoretical lower bound to use as a benchmark.

- First-In, First-Out

- Replace frames in a round-robin order.
- With 3 frames:

Page Req.	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4

9 Page Faults

- With 4 frames:

Page Req.	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	3	3	3

10 Page Faults

- **Belady's Anomaly**: more frames does not necessarily mean less page faults.

- Least-Recently Used

- Use the frame used longest ago as a victim.
- Each page has a **counter**:
  - When a page is accessed, copy the **clock** to the counter
  - The **LRU** has the **lowest clock** value.

Page Req.	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	1	1	1	1	1	5
Frame 2		2	2	2	2	2	2	2	2	2	2	2
Frame 3			3	3	3	3	5	5	5	5	4	4
Frame 4				4	4	4	4	4	4	3	3	3

8 Page Faults

- The **clock implementation is inefficient**: each time a victim is selected it requires a search.
- Alternative: use a DLL to form a **priority queue** of pages
  - When a page is accessed, move it to the head (6 pointer changes)
  - The victim is at the tail - **no search**.
  - The 6 pointer changes and extra memory for the list makes this **also inefficient**.
- Approximating LRU with a **reference bit**:
  - Each page gets a single reference bit; **rb(i)** for page i
    - Set to **1** when the page is accessed.
  - Use a "**second chance**" algo:

i is some page table entry; **n** is the size of the table

```
while(rb(i)==1):
    rb(i) = 0;
    ++i;
    if(i>=n) i=0;
```

- After this, the **victim is at i**
- This avoids victimising pages used between page faults.

Swapping and Performance

- Suppose...

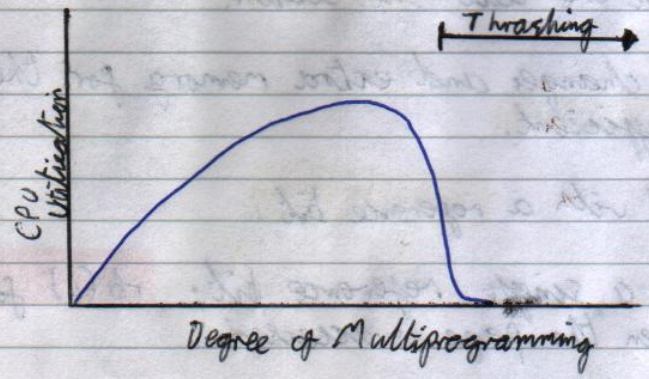
RAM access = 5 ns
HDD access = 50,000 ns
Page fault prob. = p

$$EAT = (1-p) \times (5 \times 2) + (p) \times (5 + 50000)$$

- Why does it even work if disks are so slow?
  - A: we minimise p

Thrashing

- If the sum of processes' working sets (localities) exceeds the available memory, page faults are frequent.
  - Low CPU use as time is spent swapping.
  - OS switches between waiting processes, causing even more faults
  - This situation is called thrashing.



-Locality Matters

- Say we have this square array:

```
int[][] data = new int[128][128];
```

- And say each row takes up one page, and we have one free frame.
- Compare these:

```
for (col from 0-127) {
    for (row from 0-127) {
        data[row][col] = 0;
    }
}
```

16,384 page faults

```
for (row from 0-127) {
    for (col from 0-127) {
        data[row][col] = 0;
    }
}
```

128 page faults

-Pre-fetching

- Some languages allow the program to hint at what data may be accessed next:

```
__builtin_prefetch(data[0]);
for (row from 0-127) {
    if (row < 127) {
        __builtin_prefetch(data[row+1]);
    }
    for (col from 0-127) {
        data[row][col] = 0;
    }
}
```

The Free Frames List is (Almost) a Lie

- We want a compact representation of where free memory is, and to try to keep contiguous free memory together.
- Use a buddy system
  - Free space list is split into blocks covering a 2^n amount of memory.
    - Each has a start frame and frame count.

- When a 2^n amount of memory is requested:
  - While the best-fit block is too big, split it into two buddies and put the two buddies back on the list.
  - This leaves a block of just the right size.

-Example:

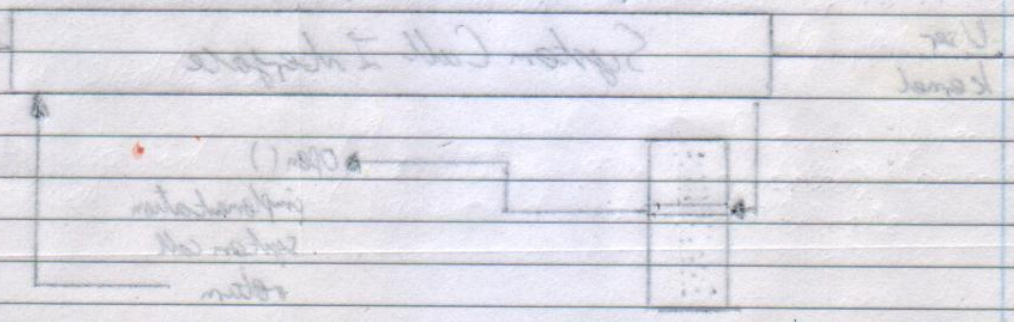
- One free space block: start 512, size 512.
- 64 frames requested:
  - Split [512, 512] into [512, 256] and [768, 256]
  - Split [768, 256] into [768, 128] and [896, 128]
  - Split [896, 128] into [896, 64] and [960, 64]
  - Mark [960, 64] as used and return 960 as the start of the block of memory.
  - Left with 4 blocks for 448 frames.

Choosing Page Size

Factor	Small Pages	Large Pages
Page Table Size	Large	Small
Internal Fragmentation	Not much	Lots
I/O Implications	Small pages quicker to swap individually	Quicker to swap one large than many small
Locality	Fine-grained: several small pages; load only the ones needed	Larger: pages must be kept in memory even if only partially used
TLB	Requires several TLB entries	Requires fewer TLB entries; lower miss rate.

TLB and Page Size

- Smaller pages → higher TLB miss rate
- key term: **TLB reach**
  - The amount of memory accessible from the TLB
  - Page size × TLB entry count
- Target: **TLB reach ≥ sum of process localities**
  - Could make pages larger (but see previous page)
  - Could use **multiple page sizes**: larger pages for processes that need a lot of memory.
    - Intel x86: 4kb pages, 4mb large pages



Security

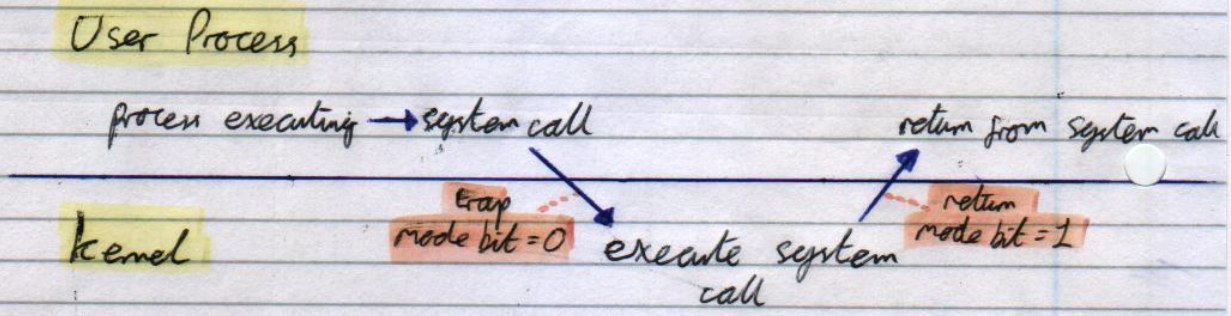
- In a computer, devices are connected by one or more buses. In theory, an unrestrained process could write to any bit of memory it wants or take control of devices - this is bad.

The Mode Bit

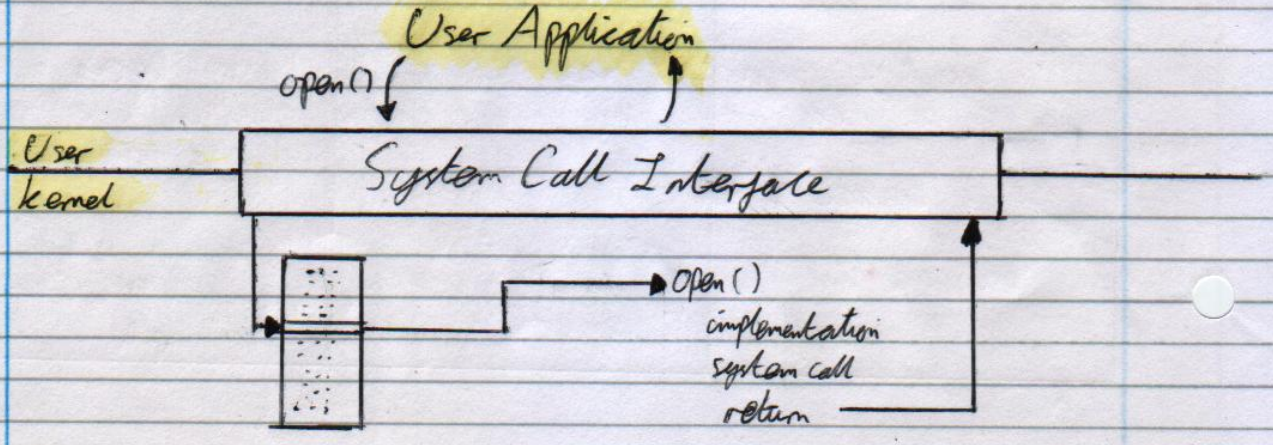
- The mode bit is part of a solution to this.
- Depending on the bit's value, a process is in one of two modes:
  - Mode 0: kernel Mode
    - Full H/W access
    - Can modify page tables, handle page faults, etc.
  - Mode 1: User Mode
    - Can only access memory via the process' page table.

Interrupts

- Handling a system call that requires the kernel mode is usually done with interrupts:



System Call Interface



Parameters

- What about parameters for the system calls?
  - Could use registers, but these are finite
  - Could use memory, but this is complex and slow
  - Could use the stack
  - Could mix and match any of the above.

Direct Memory Access

- DMA allows certain devices to write directly into a buffer in memory, bypassing the CPU.
- This allows the CPU to do useful work during a "slow" transfer and receive an interrupt when it is complete.
- It requires IO Interlock to prevent the buffer from being swapped out while the device is using it.

Kernel Types

- There are two main types of kernel space:
  - Monolithic kernels - these are large, single-unit entities that include everything about the kernel, drivers, managers, system services, etc.
  - Micro kernels - these include the bare minimum of kernel functions, allowing drivers, system service managers, etc. to reside in the user space. Their small size often makes them faster.

System Calls via an API

- The API/library maps language-level functions to their relevant counterparts, depending on the host OS.
  - Eg. the C command `printf()` maps to `write()` on POSIX systems, but `WriteConsole()` on Windows.
- The model is the same as the System Call Interface from the previous page.

Stack Hacks

- Malicious users can target poorly written code to cause their inputs to overflow their allocated memory and start editing code*, such as return addresses.
  - This is known as a buffer overflow attack.
- This can be defended against using a canary word.
  - The canary word is placed at the end of the buffer that is being written to.
  - After the write, if that canary has been corrupted at all then the program knows that a buffer overflow has occured.
  - Canary words often start with (or include) a zero-byte character, because this causes many string functions to stop.

* or whatever else is nearby in the stack

Other Protections

- Library load-order randomisation
  - Libraries loaded in a random order, but still adjacent in memory.
  - Limited effect if there are a small number of libraries.
- Address space layout randomisation
  - Each time a process is loaded, randomise the memory addresses (space) of shared libraries, data, etc.
- Both of these work to make the stack space less predictable.

Protection Bits

- When allocating space in memory for a file or input, certain bits can be set to create certain restrictions.
- For example mmap() (which maps a file to memory) allows the following:
  - PROT_EXEC - Pages may be executed
  - PROT_READ - " " " read
  - PROT_WRITE - " " " written
  - PROT_NONE - " " " not be accessed.

File-Level Controls

- It is possible to set access controls on a per-user-per-file level.
- In POSIX this is achieved with file attributes:
  - Each file has an owner and a group.
  - Only the owner (or root) can set permissions.
  - 9 control bits are used: read, write and exec. for owner, group and others.
  - rw-r----- : Owner can read/write
    Group can read
    Other has no access
  - rwxr-xr-x : Owner can read/write/execute
    Group can read/execute
    Other " " /"
- Windows has a similar "access control list"

Privilege Escalation

- Allows users to run code as another user.
- A process has:
  - A real user ID (RUID) - the person who started it
  - An effective user ID (EUID) - the user it appears to run as for permission purposes
  - A saved user ID - a back up of RUID to restore it.

ID Bits on Executables

- Set the UID bit:
  - If 1, when the process is started, EUID = owner of executable.
- Set the GID bit:
  - If 1, when the process is started, EGID = group of executable

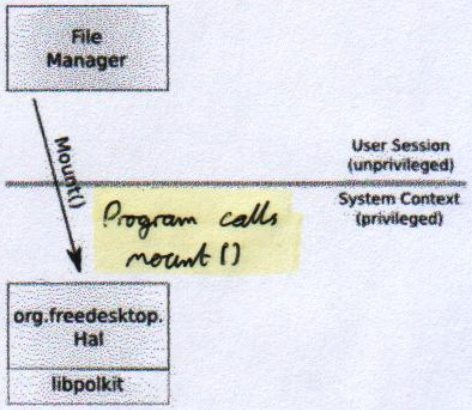
Fine-Grained Privilege Separation

- Example: mounting a flash drive
  - Mounting drives need to run as root
  - Everything else does not
- On modern Linux, this can be done with a Policy kit:
  - Privileged mechanism
  - Unprivileged client.
- Example on next page. →
- On Windows this is achieved with User Account Control.
  - When an admin-level user logs in, they get two tokens:
    - One for a normal user
    - One with admin rights
  - When a process starts, it is given the user-level token.
  - To exercise an admin right, the process must request privilege escalation:
    - Normal user: enter password
    - Admin: swap tokens.

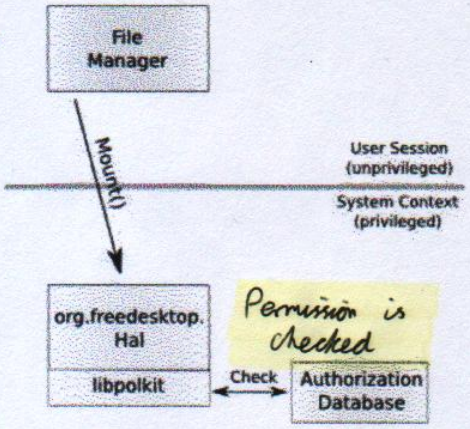
Hash Function

- Turns plain text into an encrypted format.
- Non-reversible.
- Hash Attacks:
  - Obvious passwords
  - brute-force (rainbow tables)
  - social engineering
  - errors in the crypto function
  - bribe standards committees.

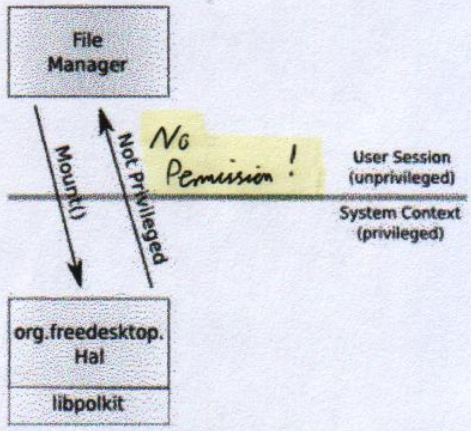
1.


File Manager
Mount()
User Session (unprivileged)
System Context (privileged)
Program calls mount()
org.freedesktop. Hal
libpolkit

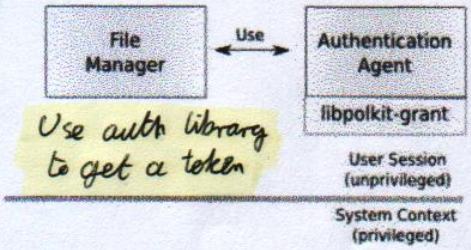
2.


File Manager
Mount()
User Session (unprivileged)
System Context (privileged)
org.freedesktop. Hal
libpolkit
Permission is checked
Check
Authorization Database

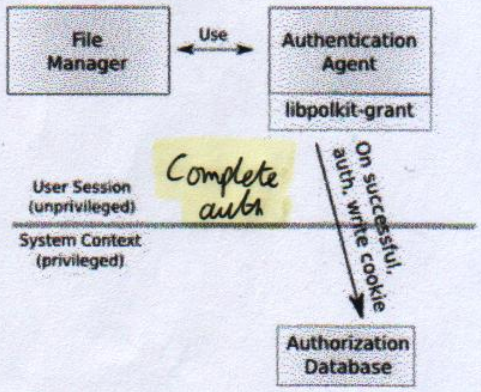
3.


File Manager
Mount()
Not Privileged
No Permission!
User Session (unprivileged)
System Context (privileged)
org.freedesktop. Hal
libpolkit

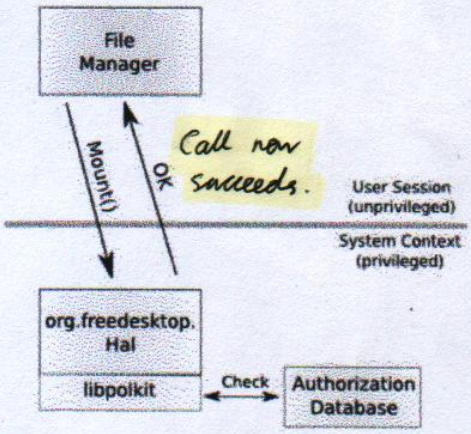
4.


File Manager
Use
Authentication Agent
libpolkit-grant
Use auth library to get a token
User Session (unprivileged)
System Context (privileged)

5.


File Manager
Use
Authentication Agent
libpolkit-grant
Complete auth
On successful auth. write cookie
User Session (unprivileged)
System Context (privileged)
Authorization Database

6.


File Manager
Mount()
OK
Call now succeeds.
User Session (unprivileged)
System Context (privileged)
org.freedesktop. Hal
libpolkit
Check
Authorization Database

SQL Injection

- I know what this is. I'm not writing notes for this.
- Just use prepared statements!

Data Access API Calls

- Think: REST + JSON
- For security, sessions are used:
  - The client authenticates as some user
  - They receive a session key to use in future requests.
- Must be careful: think about the Moon Pig fiasco.

Don't Trust User Input

- XSS: Cross-Site Scripting
  - Forums
  - Search inputs
  - hEATS
- Arbitrary Code Execution
  - Spell-checking user input
  - Generating web pages (eg. CGI Scripts)
  - Request headers may be passed as environment variables
    - Vulnerable to the Shellshock bug:

```
HTTP_USER_AGENT = () { :; }; ping -s 1000000 nsa.gov.
```

Physical Network Access

- All bets are off now.
- eg. MAC address spoofing - one example out of many.
- Packet sniffing
- etc...

Linux

Processes

- Each has, inter alia:
  - An owner (think RUID, EUID, etc.)
  - A parent process
  - A scheduler priority ("niceness")

UNIX Philosophy:
- small is beautiful
- make each program do one thing really well

Shell (eg. Bash)

- aka. the "terminal" or "command line"

- Most commands are executables or disk
  - eg. /bin/ls, /bin/grep, etc.
  - A command starts a process with some arguments (String[] args)
  - Processes return an exit code
    - 0 = okay
    - non-zero = error

- Useful commands:
  - pwd - Present Working Directory
  - cd x - Change Directory to x
  - cat - Loop over a list of files and print the contents
  - grep - Take a regular expression and some input, then print only the lines matching the regex. Can be modified:
    - grep -v print lines that don't match
    - grep -l print file names with a match
    - grep -c count matching lines
  - wc - Word count
    - wc -l line count
    - wc -m character count
  - head/tail - print the first/last 10 lines
    - head -lines=20 print the first 20

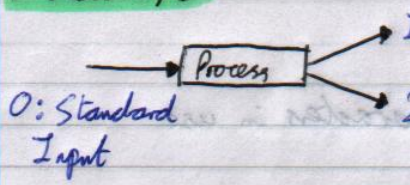
- Commands can be ran in sequence:

  thing a; thing b c; thing d

  - ↑ this does not check the exit code of each command
  - ↓ this makes sure a command exits w/ zero before continuing

  thing a && thing b c && thing d

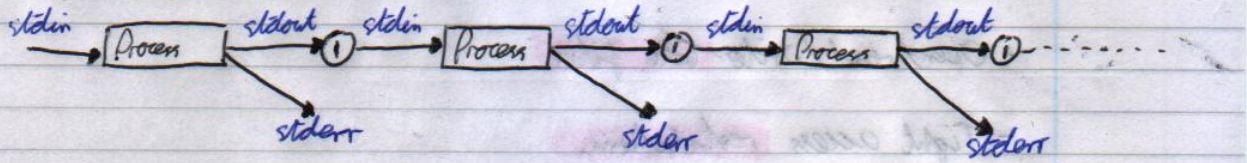
Process I/O



- These are just like writing to files:
  - write (1, "Hello World", 11);
  - write (2, "Uh-oh!", 6);
  - char buff[256];
    read (0, buff, 256);
- There are Java equivalents:

```
InputStreamReader input = new ISR(System.in);
char[] buff = new char[256];
int charsRead = input.read(buff, 0, 256);
```

- We can also loop over input characters by wrapping the ISR in a BufferedReader.
- Exactly the same applies for output:
  - System.out... is a PrintStream: a BufferedOutputStream that also does char → byte conversion.
- Pipes can be used to "chain" commands:



  - How many lines in a file contain "Alan!"?

```
cat file.txt | grep 'Alan!' | wc -l
```

  - Pipes represent a producer/consumer buffer.

Unicode

- In Java, a char is 2 bytes
  - But there are more than 256 characters in use

- In Unicode UTF-8:
  - Read a byte
  - If the top-most bit is 0, convert to a char following ASCII.
  - If the top-most n bits are 1, read n-1 more bytes for a total of n; combine into a char.
  - n may be up to 4.

/proc/...

- /proc is a virtual file system

- /proc/12345 contains information about the process 12345

- /proc/12345/environ - environment variables

- /proc/12345/fd - lists all the file descriptors

/dev/... - Everything is a File

- /dev contains hardware devices
  - /dev/sda, /dev/sdb, /dev/sdc... disks
  - /dev/snd/... audio
  - /dev/video0 web cam

- Open/read/write like files

- Tight access restrictions.

Redirection

- Use > to redirect output to a file

  eg. ls | grep ".txt" > output.txt

- Can be used to "throw away" output:

  eg. command > /dev/null

- By default, stdout is redirected, not stderr.
- We can use 2> to redirect stderr
  - Eg. command > result.log 2> error.log
- We can also merge them:
  - Eg. command 2>&1
- Redirection is buffered
  - In command > file.log, file.log is updated every time the buffer fills or flush() is called.
  - We can control the buffer:

    stdbuf -o 0 command — runs command with a 0-byte buffer.

    stdbuf -oL command — flushes buffer at new lines

Command: sed

- Stream Editor
- sed -e 's,cat,dog,'
  - Replace "cat" with "dog" once
- sed -e 's,cat,dog,g'
  - As above, but globally.
- sed -e 's,\([a-z]+\)[0-9]*,\1,'
  - Replace any block of at least one letter and any number of numbers with just the letter.
  - Eg "abc123" → "abc"

Command: find

- Finds files with certain properties
- find -name "foo" - Search by name
- find -iname "*.jpg" - Search by name (case-insensitive)
- find -type d - Directories
- find -type f - Files
- -exec allows a command to be run on every file found

  eg. find -iname "*.txt" -exec cat {} \;

  {} : Filename Placeholder
  \; : End of Command

ps : Process Status

- ps - shows processes running in the current terminal
- ps ux - " " for the current user in detail
- ps aux - " " " all users in detail
- ps auxf - " " " " " in an ASCII-art tree

Process Signals: kill

- kill 123 terminate process 123
  - sends "SIGTERM" which can be caught to shut down nicely.
- kill -SIGKILL 123 kill proc. 123 immediately.
- kill -SIGSTOP 123 pause 123 (will never be chosen by scheduler)
- kill -SIGCONT 123 resume proc. 123
- The hangup signal is sent when the terminal is closed, or by kill -SIGHUP 123. Usually the same as SIGTERM
- Can be avoided w/ nohup some-command

Returning to Shell

- Normally a command exits and then returns to the shell.
- Add & at the end to return immediately.
  - nohup web-server >& output.log &

Priority

- Processes have a niceness, default = 0
  - 20 = only gets "idle" CPU cycles
  - -20 = highest priority
- /usr/bin/nice command runs command with niceness at 10
- /usr/bin/nice -n 20 command you can guess
- renice -n 20 123 sets proc. 123 to nice 20.
- Processes start at nice 0.
- Root can reassign any value.
- You can change your processes to be more nice
- You cannot make your processes less nice

Environment Vars

- Map of strings → values
- $PWD - present working directory
- $PATH - where to look for commands
- $HOME - path to home directory

Command: cd

- cd x - changes directory to x
- pushd x - pushes $PWD to a stack and does cd x
- popd - pops $PWD off a stack

Scripting

- Better than writing sequences of commands
- Allows loops, variables, etc.
- Could also use a language like Perl
- Best to be written as filters: read from stdin, write to stdout