Interface Stability

1) fd = open(“file”, O\_RDONLY)

Local open

2) fd = remote\_open(“code.google.com”, “file”, O\_RDONLY)

Remote directory open, but no local!

3) fd = open(“code.com”, “file”, O\_RDONLY)

4) open(“code.com:file”, O\_RDONLY)

Opens files with options!

5) putenv(“FILESERVER” = code.com”)

Fd = open(“file”, O\_RDONLY)

6) fd = open(“file\0code.com”, O\_RDONLY)

Problem is, now wont distinguish between null byte for local open, meaning random garbage data can be accessed

7) fd = open(“http://code.com/file”, O\_RDONLY)

Can have http folders, smth about the / too

8) fd = open(‘/u/class/cs111/file” , O\_RDONLY)

Look in file systems

Sample app: Paranoid Professor

Make new OS, nobody knows how to crack

Want to count number of words in proposal

Want the equivalent of wc (unix program for word count) in our new, unknown OS

But don’t want to use Linux AT ALL since could be bugged

Want to count words on computer in office, unattached to internet

“Ascii words only, I don’t want any foreign words bc I don’t trust foreigners”

Bootstrapping

x86 boot procedure

Sets 0xffff0 as instruction pointer

Divide the physical RAM into two parts

Real DRAM

Wiped at shutdown

Read Only Memory - ROM/EEROM

Cannot be changed, survives power outage

What does it do?

Tests system

Makes sure all the stuff works properly

Looks for devices

Finds a bootable device

Device that contains program in specially marked sectors of that device, one that we want to run

Reads the device’s first sector into RAM

At 0x7c00 - 0x7dff

Jmp 0x7c00

Master Boot Record 512 Bytes (Sector 0)

Last 2 bytes are reserved, so we can do step 3

Whether we can boot it, and whether we should boot it

Then 64 bytes in groups of 4 16 bytes each

Each 16 bytes describes partition

Contains start sector (where partition starts)

Size (number of sectors)

Type (1 byte quantity linux filesystem etc.)

Is Executable?

The rest of the 446 bytes is x86 code

Bootloader Source Code Sneak peak

[c]

for (i=1; i < 20; i++)

read \_ide\_setenv(i, 0x10000 + (i-1)\*SR);

At location 0x7c00 put that code ^

read\_ide \_sector(int s, int a){

//PIO

while((inb(0x1f7)&0xc0) !=0x40)

Continue;

outb(0x1f2, 1); //When command issued, want to do 1 sector

outb (ox1f3, s&0xff);

outb(ox1f4, (s>>8)&0xff);

outb(0x1f5, (s>>16)&0xff);

outb(0x1f6, (s>>24)&0xff); //need to shift bc outb uses bytes not words, so can only read in byte per cant do all 32 bytes at the same time.

outb(0x1f7, 0x20);//Read sectors command/ATA Spec

insl(0x1f0, a, 512/4); //copies into a and executes?

}

[end]

1/18/2017

Talking about booting from sectors, read\_ide\_sector

>>>CODE>>>

[c]

void main(void) {

int nwords=0;

bool inword=false;

int sectornum=10000;

While (true) {

Char buf[512];

read\_ide \_sector(s, buf);

for(int j = 0; j < 512; j++){

If(buf[j] == ‘\0’){

write\_out(nwords);

//Return;

//Really want an infinite loop to display number on screen

while(true) continue;

}

Bool thisalpha = isapha(buff[j]);

Nwords += !inword && thisalpha

inword = isalpha;

}

}

}

[end]

Use memory map I/O

[c]

void writeout(int n){

unsigned char \*screen = (unsigned char\*)

(0xb8000 + 80\*25\*2/2 + 8)

do{

screen[0] = n%10 + '0' //low order byte

screen[1] = 7;

screen -= 2;

n/=10;

}while (n != 0)

}

[end]

Problems in this code

1) integer overflow

2) read\_ide\_sector multiple copies

1) wc

2) mbr

3) in ROM

Everything that loads from the master boot record will need a copy!

Can we arrange for a singular copy thats "bulletproof"

Copy 3 is never going away, it's always going to be there

Call functions only in ROM if we know where it is [read\_ide\_sector = 0x7f000]

3) busy-waiting

theres no need to keep waiting at this point

wc can countwords and then wait for the next sector -> then read sector after that into new buffer

technique is called double buffering

trading memory for better cpu performance

Usually --> CPU --- RAM --- Disk Controller

But thats too slow

Now --> RAM --- Disk Controller

Called DMA (Direct Memory Access)

4) Use DMA to speed it up

5) Run several apps simultaneously

Unified Extensible Firmware Interface (UEFI)

smarter Firmware

standard format for bootloader (EFI)

OS independent

Globally Unique Identifiers (GUID)

for partitions, 128 bit numbers in hex

each partition has a unique label

Be able to know whether it has seen a drive before

firmware will remember the GUID's

Only trust certain

GUID partition table (GPT)

interpretable by the firmware for any comp

EFI Partitions use MS-DOS file format

UEFI Boot manager

in firmware/bios

is configurable via non-volatile RAM (NVRAM)

reads GPT tables

can access files in MS-DOS format

can run EFI programs

Essentially, an OS whose job it is to boot other OS's

Our paranoid program in review

- its too much of a pain to change

- can't reuse code in other programs

- can't run several programs simultaneously

- assumes no failures in hardware

- can't recover from faults/display error messages

Modularity to the rescue

- Break into smaller/easier to handle chunks

- Software engineering advantage, n lines of code, k modules

- blah blah blah but time to find code is O(N^2)

- modularity cuts debugging time by how many modules you have

- is lowkey bullshit but w/e idea is there

Modularity can be good or bad!

- Need metrics, way to measure if good or not idea

- performance

- does adding modularity add performance?

- generally adding modularity hurts performance

- robustness/tolerance of faults in hardware and Software

- how well does system handle violations to the interface you set up?

- flexibility/neutrality/Lack of assumptions

- Simplicity (learn/use/short manual)

Our WC program emphasizes 4 super hard, but everything else is low

How to implement modularity

0) don't do it

1) function call caller/callee modularity

ex [c]

int fact (int n){

if(n==0)

return 1;

else

return n\*fact(n-1);

}

[end]

gcc didn't like it, tried to optimize,

when asked to not-optimize it put out this

pushq %rbp

movq %rsp, %rbp

subq $16, %rsp

movl %edi, -4(%rbp)

cmpl $0, -4(%rbp)

jne .L2

movl $1, %eax

jmp .L3

.L2

movl -4(%rbp), %eax

subl $1, %eax

movl %eax, %edi

call fact

mull -4(%rbp), %eax

.L3

leave

ret

function call modularity is the most common type of modularity in Software

Not what we want in OS though, since it can go wrong often/fail/robustnessis not there

- callee can mess up caller's head pointer

- callee can loop forever

- stack overflow

- callee can change the caller's registers

- can't rely on function call modularity fully

This is soft modularity we want hard modularity

2) Ways to get hard modularity

1) use different computers for each module communicate via a network API

- cloud computing etc taking world by storm

- cost is p large

- worried about robustness of network as well

2) write an interpreter, which runs each untrusted module

- basically checks for bugs in program before it actually executes

- stops it before the line runs

- tells command module failed

- cant use this approach to catch an infinite loop

- want reliability > performance

- subject for the next 5-6 lectures!

1/23/2017

#OS Organization

Goals:

- Protection

- Robustness

- Utilization

- Performance

- Flexibility

- Simplicity

This is great, but can't have all of them!

Need to find best way to orgaize

Function Calls maybe? (soft modularity)

Great for everything

But absolutely terrible for protection and Robustness

only work if every byte of code is trustworthy and works 100%

3 System Abstractions

1) memory

- load and store

- read and write

- Things to think about with memory

- size?

- word size?

- throughput?

- latency?

- volatility?

- linear vs associative?

2) interpreters

- instruction pointer

- environment pointer

- reportive

- normal vs exceptional execution

- exception handlers

- traps/interrups

3) links

- send and receive (I/O bus)

2 ways to hard modularity

1) Client/Server

- Client sends read request to Server

- Server sends response

- Form of (3) link abstraction!

- better than function calls because if server is written well, server can't be messed with

- limit error propagation

- no shared state

- even if loops or crashes

- hassle to set up

- more resources, need more server!

2) Virtualization #will be focus of course

- just have 1 computer that pretends to be 2 or more

- Virtualization via emulation

- e.g. ARM emulator on SEASnet x86-64

- good for debugging but way too slow

- Virtualization via hardware support

- Filter will check if instruction looks safe

- Then hardware executes

- But if instruction looks iffy, then gives up

- unpriveleged vs priveleged instructions

- unpriveleged - run at full speed (addq $1, %eax)

- priveleged - doesn't run, causes the hardware trap -> almost like an invalid instruction (halt)

- Interupt service vector

- Have a trap array, if anything executed in there jump to code to be run when trap occurs

- both array and trap runners have to be in protected memory

- application can't have anything to do about it

- trap saves:

- stack pointer

- instruction pointer

- flags (also holds if priveleged or not)

- stack segment

- code segment

- error code

- where to save all this info?

- Save it on the stack!

- differences between traps and fct calls

- way more expensive, have to save a ton of data vs a return address for function calls

- protected trasfer of control (between application and machine)

- trap handler can execute priveleged instructions

- way it has been done for years, but it is super slow!

- everything is saved in memory, and talking to memory is relatively slow

Application and Hardware

- Application runs unpriveleged commands

- another command, syscall, but run by kernel

- its like application, but can use priveleged instructions!

- almost all of your commands should be unpriveleged, bc priveleged take a ton longer

Layered System

- stdio level

- getchar putchar etc

- syscall level

- read + write

Ring Structured operating System

- several layers each protected from the rest

- files -> virtual memory -> kernel etc.

- every time to cross boundary you need to do a trap

- very expensive and v slow

Linux -> monolithic kernel

Organization via Virtualization

- Have to think of resources of comp for ease of use and efficiency

- ALU (Arithmetic Logic Unit)

- user code has full access

- Registers

- user gets full access to common registers (rbp rsp rax etc) but flags ect no access or limited access

- primary memory

- user gets full access to user memory

- limited access to ISV and kernel access

- boundary between what users can do and system can do

- I/O

- no access!

process: program in execution in an isolated domain

- run on a virtualizable processor

- thinks they have syscall access but rlly they dont

fork vs execvp

- fork

- child = parent, but pID's ARE DIFFERENT

- different file descriptors

- accumulated execution times

- file locks (child has none)

- pending signals (SIGKILL etc)

- execvp is the opposite!

- but you have new

- program

- data

- registers

- Signal handlers and reset

1/25/17

[c]

pid\_t fork(void);

clones current process

returns -1 on failure

returns 0 in child

returns pid of child if parent

void execvp (char const \*file, char const \*argv);

returns -1 //even though void

[end]

Zombie Process cannot do anything, no counter, but its still there

How do we kill?

- not kill, not exit, but wait!

[c]

//process vulture, waits around to watch processes die

pid\_t waitpid(pid\_t pid, int \*status, int options); //use WNOHANG on options so returns -1 if process still running

[end]

Can only wait for your children to die, can't wait for just any process to die

- if you could put any process, you could wait for yourself and it'd be bad

This is called "reaping" the child process

Make sure to always check return value of fork!

[c]

bool printdate(void){

pid\_t p = fork();

if(p < 0)

return false;

if(p == 0){

car \*args[] = {"date", "-u", (char\*)0};

execvp("/usr/bin/date", args);

\_exit(23); //this makes exit without return, error check for execvp bc will only hit this point if execvp fails and returns!

}

int status;

if(waitpid(p, &status, 0) != p){

return false;

}

return WIFEXITED(status) && WEXITSTATUS(status) == 0;

}

[end]

use alarm(10) to make the program only run for 10 seconds

There is an easier way!

[c]

int posix\_spawnvp(pid\_t\* restrict pid, char const \*restrict file, posixspawnfile\_actions\_t const \*file\_acts, posix\_spaw\_aff .........)

[end]

^ Nobody uses this except for microsoft. It is little data structures to put into fork, way too complicated.

Files

- can be anything

- regular files (arrays of bites)

- directories (mappings from file name components to file attributes)

- /dev/null (trash can, write to succeeds read from 0 bytes)

- /dev/full

- /dev/zero

- /dev/tty0

- /dev/dsk/0 -> low level access to disk drive

- pipes

int open(char const\*, int flags, ...);

flags can be all this different stuff

open is to linux => spawnvp is to microsoft

Maybe it does too much?

[c]

creat(char const\* f, mode\_t m);

open(f, O\_RDWR|O\_CREAT|O\_TRUNC, m);

[end]

just use open, it can do everything anyway

1/30/2017

What can go wrong with a file descriptor?

Went over a ton of different error cases

- read past EOF

- step away from keyboard

Dealing with Race Conditions

- behaviour that depends on timing

- "No excitement up ur spine, just chills down ur spine"

- Worst kind of bugs in the world

[bash]

(cat a & cat b) > c

#two cats running at same time, which cat/cpu is faster? Might get a different c each time

(cat>a & cat>b) < c

#both reading from c, writing to a and b splits c up since they'll grab whatever they can

(apache1 & apache2) >> c

[end]

Example of race condition you might have to deal with

gzip: compression program

gzip foo.tar, what does it do?

- reads foo.tar

- writes foo.tar.gz

- removes foo.tar

User hits ^C

[c]

int main(int argc, char\*\* argv){

int fdk = open("/dev/tty", O\_RDONLY | O\_NONBLOCK);

int fdi = open("foo.tar", O\_RDONLY);

int fdo = open("foo.tar.gz", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

do{

if(read(fdk, &c, 1) == 1 && c ==3){

unlink("foo.tar.gz");

return 1;

}

n = read(fdi, ... );

compare();

write(fdo, ... );

}while( 0 < n );

close(fdo);

unlink("foo.tar");

}

[end]

We can use Polling to solve our issues

- keep asking about ^C at same point in loop

- When done right will work but is hassle

- Downside, have to modify at every loop to check ^C

- would rather want to separate out job so we dont have to keep modifying

Using Signals!

Goals

1) signal handler

2) Uncooperative programs in loops

3) invalid program illegal instructions

- SIGILL or floating point exception

- invalid address FIGSEGV SIGBUS

- I/O Errors

- SIGIO

- SIGPIPE write to pipe but no readers

- Process

- SIGCHILD - a child died

- used to call waitpid immediately

- SIGKILL SIGSTOP SIGTSTP SIGCONT -> cannot be caught by handler, !not catchable! (signal nuum 9)

Signals

- change to processes abstract machines

- between any two instructions, a signal can arrive

- a signal handler function is executed

- this function is specified by the application

[c]

void hadle\_control\_c(int sig) {

unlink("foo.tar.gz");

\_exit(126);

}

[end]

Poss race condition here!

- race condition, could call signal and unlink foo.tar.gz without anything being there

- Signal arrives after we remove uncompressed file

- can use special args, SIG\_IGN

- Ignore the signal at that point

[c]

int main(int argc, char\*\* argv){

int fdk = open("/dev/tty", O\_RDONLY | O\_NONBLOCK);

int fdi = open("foo.tar", O\_RDONLY);

int fdo = open("foo.tar.gz", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

do{

if(read(fdk, &c, 1) == 1 && c ==3){

unlink("foo.tar.gz");

return 1;

}

n = read(fdi, ... );

compare();

write(fdo, ... );

}while( 0 < n );

close(fdo);

signal(SIGINT, SIG\_IGN);

unlink("foo.tar");

}

[end]

critical parts of program

- unlink and say

can handle with sig ignore to signal handler

[c]

signal(SIGINT, SIG\_IGN);

//critical part

signal(SIGINT, hadle\_control\_c);

[end]

user can get mad, seems like ^C is getting ignored

theres another flag, more lightweight that blocks signal but stores it

[c]

what you were blocking = pthread\_sigmask(block SIGINT)

//critical part

pthread\_sigmask(old set);//^C arrives here

[end]

signal handlers can arrive between machine instructions

Signal handler acts like function call, but will splice into instruction array

[c]

void handle\_signal(int sig){

FILE \*f = fopen("file", "W");

fprintf(f, "caught signal %d\n", sig);

fclose(f);

}

[end]

this code has an issue,

- we want to keep signal handlers as simple as possible

- they arent allowed to break anything!

fopen calls malloc, and this is super dangerous

- it modifies the heap data structure

malloc calls take a long time on the stack, is possible to get interrupted in the middle

if malloc gets called again during signal, it will crash

#only call async\_signal\_safe functions

Signals

+ can manage processes better

+ fixed some robustness & performance issues

- processes are less isolated

- can be signaled at any time => races

- signal handling is notoriously buggy

- emacs Example

Threads come in

- get rid of memory isolation

- all memory is shared, there is no memory isolation

- threads can communicate

- this makes race conditions crazy

- no need for signals or pipes, threads can pick up info on their own

Threads share as much as possible

- memory (address sace) as well as code (instruction pointers are different)

- file descriptors

- owner, pid, ppid ...

They don't share

- instruction pointers

- each one must be separate to run properly

- registers ae separate as well

- they dont share the stack

- thread id

- errno

Try to keep differences to a minimum, we want them to be super lightweight

The Resource Allocation Problem

8-Core CPU, and 1000 threads what do we do?

- use cpu to run program to schedule cpu

2/1/2017

[c]

#include<pthread.h> //usr/included/pthread.h

int pthread\_create(pthread\_t \*thread, pthread\_attr\_t const \*attr, void (\*start) (void \*), void \*arg);

void pthread\_exit(void \*value);

void pthread\_join(pthread\_t thr, void \*value); //like waitpid for processes

void pthread\_kill(pthrea\_t, int); //like kill for process, zombie thread

[end]

how do they work without stepping on each other?

If simple, no need to listen to the rest of the lecture

Cooperative Threads

- yield the CPU to other threads (for fairness)

- at every system call, kernel chooses which thread to return to

^ poss infinite loop issue

This is where sched.h comes in, can schedule threading

Poss Solution, every 100ms or less, use a system call so can exit if needed (very hard to maintain)

- maybe get OS to fo it for us

Hardware Timer Interrupt

- clock connected to cpu

- variable time (10ms on linux) CPU traps via interupt service vector to clock interrupt code

- equivalent of sched\_yield, essentially is a system call

- jumps to other portion of kernel code, but have to save registers of thread

- so kernel decides to return to some other thread or stay in trap

"Sunny Side of Threads"

Solvng infinite loop problem

- timeout for the process

- user initiated interrupts

- don't use loops!

- make sure there is an upper bound on number of times to execute

Cooperating across shared memory and I/O devices

3 basic techniques

1) busy waiting

- [c] while(!ready()) continue; [end]

2) Polling

- keep asking cpu if ready

- [c] while(!ready()) sched\_yield(); [end]

3) blocking

- [c] while(!ready(device)) wait\_for(device);[end]

- don't wake me up until device is ready.

Include a thread state, where its running, runnable, or blocked

- scheduler uses these states to decide what to run next

Which thread to run next?

Based all on scheduling policy (not a mechanism)

scheduling: policies and dispatch mechanisms

Example is airline scheduling, have to schedule airplane and gate, flight crew etc.

Scheduling scale

- long term

- which processes are admitted to the system (admission control)

- medium

- which proccesses are in RAM?

- short

- which threads have CPU?

- preemptive - clock interrupt

- Cooperative - yield()

Issues still arise,

- priority queues and multilevel systems

- threads trying to run but are in different categories

- system thread queue (high priority)

- interactive threads (next priority)

- batch threads (lower priority)

- student threads

Realtime Schedulers

1) Hard Realtime Schedulers

- hard real time (nuclear power)

- has to arrange for some deadline to be met, cannot miss deadline at all

- equivalent to dumping core essentially

- bug just as serious as miscalculating temp of reactor

- in this approach, predictability trumps performance

- e.g. caches disable, keep caches off

2) Soft Realtime Schedulers

- Some deadlines can be missed, if there is too much work

- Ex, video playback get data, render it

- deadline for each frame, if can't render just skip it

- e.g. earliest deadline first

Scheduling Metrics

- average/expected wait time

- response time

- turnaround time

- can also calculate the variance

throughput: how may jobs per second can we do

- useful work per unit of time

utilization: 90% utilized if 90% of the time its doing actual threadwork

- minimize wait/response time

fairness: max fair, min variance, competes with utilization

Simple Scheduling policy

- first come first serve (FCFS)

- priority = arrival time (no yielding)

jobs arrival time run time

A 0 5

B 1 2

C 2

Lost rest of lecture, ran out of battery. 10 minutes left

2/6/17

Scheduling:

FCFS = fairer

SJF = better average wait time ...

Fair?? Minimize standard deviation of wait times? Equal priorities??

- Real defn: No job waits forever ... can be implemented as a boolean

Preemption & scheduling is one possible solution to be more fairer

- quantum = time between when we can reschedule (if a process takes longer than the quantum)

- in GNU/linux, we have 10 ms roughly

One example is Round Robin = first come first serve with preemption

A arrive 0, run 5

B arrive 1, run 2

C arrive 2, run 9

D arrive 3, run 4

Order: A, B, C, D, A, B (b done), C, D, A, C, D, A, C, D (d done), A (a done), C, C, C, C, C (C done)

- 0 wait time for A, B, C, and D (great average wait time)

- Average turnaround time = (15 + 5 + 18 + 11)/4 = 12.25 -> + 19 delta (bigger overhead than before)

- technically deltas of the CCCCC part will be smaller, but still some overhead

Round robin can also have starvation ... solution is to put newly arrived job at the end of the queue

- else lots of new jobs will come in the front and starve out things in the end

Priority Scheduling:

User-Assigned (static)

System-assigned (dynamic)

"Niceness" is used to see which process is most or least greedy.

- nice make linux

runs from (-19 to 19) ... 19 is best, -19 is greediest and needs more processing

- nice -n -3 make linux (negative numbers are only for super users/root users)

- nicer processes are put later in the queue and get less CPU time ... so be greedier if possible

- bigger number is lower priority job

Synchronization:

- Threads' biggest problem!!!

- Default is unsynchronized

- BUT things usually work anyways! ! because only a small time frame for things to crash

Coordinating Actions in a shared address space is one way to avoid synch ... hard for large systems

- Maintain data consistency,

- do so efficiently (utilization and latency)

- make it clear and simple

Bugaboo: race conditions (races) ... CPU is faster than another

Observability:

- System and outside world ... syscalls can load and store

Old State loads to new state does getpid yet another state, and then stores for final state

- but in reality maybe not sequentially but rather 2 threads in parallel doing both

- serialization says the OS should conform to the spec ... can come up with a single step by step

explanation for what happened but might not actually be serial

Isolation:

- actions X and Y are unrelated (St 1000, ld 2000)

Atomicity:

- actions X and Y are related, but system arranges for either one to finish first before doing the other

Example:

unsigned long balance = 10000;

2 functions: deposit (unsigned long amount){ balance += amt; }

- bool withdraw (unsigned long amt){if amt <= balance, then balance -= amt, return true. else return false }

Thread 1 will deposit 10 dollars.

Thread 2 will try to withdraw 20 dollars

movq balance, %rax

addq $1000, %rax

movq %rax, balance

Movq balance, %rax

cmpq $2000, %rax

jl failure

subq $2000, %rax

movq %rax, balance

Cannot write code like this ... if (balance + amount < balance = means synch error)

Java has a keyword synchronized void deposit ... and synhronized boolean withdraw

- But how does JAVA do it?? ... everything in the method must be done all at once (make them atomic)

Critical section is a section of your code that is all atomic

- executed indivisably ... at most one thread's CP is in a critical section

Don't want critical section too large (basically makes ur 8 core useless) (lost performance due to bottlenecking)

- but too small and you might not have capture enough of the possibilities of thread Safety

1) To minimize critical secion you need to look for writing to shared states

2) Lok for dependent reads that then used to grow critical sections

- C has one flag called \_\_button\_addoverflow\_p (balance amt)

3) Reads to large objects will also cause errors if you put them not critical since it will take a long time to read or write data

to these objects. you want them to be fully read or written to otherwise you will get race conditions

footnote:

watch out for small objects toos

struct{

unsigned int a:1;

unsinged int b:1;

t1

sa = true;

sb = true;

}

Even small objects should be made atmoic because load/store is atomic for 1, 2, 4, 8 bytes, but nothing else

- Must also be aligned as well ... everything else is not atomic so can be errors

atomic operations are allowed to have benevolent side effects as long as update chancing is invidible to the user

- don't want two differnet insruction pointers in the same part for the same account, but ok for different accounts

Critical section for two accounts at once (transwer function)

- or audit\_all\_accounts is a super critical section ... should lock out all other threads and users

Enforcement of Critical Section:

- Simple version, 1 CPU and syscalls for all actions (no preemption)

- every syscall is a critical section, and no hardware traps (kinda like how a realtime system())

IN Hardware, we can termporarily mask out interrupts (use SIGBLOCK to ignore signals)

- block and unblock interrupts is a primitive way to do critical sections!!

More generally though, multiple CPUs have more of a problem

1) mutual exlcusion: if one processor is pointing in critical section, other thread from using that

2) Bounded wait: any thread in a critical section will exit it quickly. Prevents starvation

Struct pipe {

char buf [1024 ... power of 2];

size\_t r, w;

};

bool write (struct pipe\*p, charc){

if (p->w & p->r == 1024) return false;

p-> buff[p>w++%1024] = C;

return true;

}

int read (struct pipe\*p){

if (p->r == p->w) return -1;

return p-> buff[p->rH%1024];

}

we need more checks in this. If no data, return -1

also make sure u dont write to a pipe that has already been writen to.

2/13/2017

[c]

#define N 1024

mutex\_t m;

struct pipe{

unsigned char buf[N];

size\_t r,w;

}

bool writec(struct pipe \*p, char c);\{

//disable\_int();

lock(&m);

if(p->w - p->r == N) { /\*enable\_int();\*/ return 0; }//returned, but interrupts are still disabled!

p-> buf[p->w++%N] = c;

unlock(&m);

return 1;

//enable\_int();

}

int readc (struct pipe \*p){

//disable\_int();

lock(&m);

if(p->w - p->r == N) { /\*enable\_int();\*/ return EOF; }

int x = p-> buf[p->r++%N];

//enable\_int();

unlock(&m);

return x;

}

[end]

Implementing Crit Sections on a uniprocessor

Only problem is the interrupts,

- disable in critical Sections

- Doesn't rlly work well in multi core CPUS

api for answer!

[c]

typedef int mutex\_t;

void lock(mutex\_t \*m);//precondition, don't own lock

void unlock(mutex\_t \*m);//precondition, own lock

[end]

Coarse Grained Locking

- force mutex at every critical Section

- often creates a huge bottle neck bc this affects all pipes

- solve this by making mutex a member variable of each pipe

[c]

struct pipe{

unsigned char buf[N];

size\_t r,w;

mutex\_t m; //finer grain locking

}

bool writec(struct pipe \*p, char c);\{

//disable\_int();

lock(&p->m);

if(p->w - p->r == N) { unlock(&p->m); /\*enable\_int();\*/ return 0; }//returned, but interrupts are still disabled!

p-> buf[p->w++%N] = c;

unlock(&p->m);

return 1;

//enable\_int();

}

int readc (struct pipe \*p){

//disable\_int();

lock(&p->m);

if(p->w - p->r == N) { unlock(&p->m); /\*enable\_int();\*/ return EOF; }

int x = p-> buf[p->r++%N];

//enable\_int();

unlock(&p->m);

return x;

}

[end]

implementation of lock/unlock

[c]

/\*

void lock(int \*m){

\*m = 1;

}

void unlock(int \*m){

\*m = 0;

}

\*/

void lock(int \*m){

while(\*m)

continue;

\*m = 1;

}

//assumes precondition that we own lock, so it's fine

void unlock(int \*m){

\*m = 0;

}

[end]

Intel says,

- loads & stores are atomic for aligned address-multiple-of-size (32) words only

- lock incl x <- memory address

- adds 1 to x atomically

[c]

void lock\_incl(int \*p){

asm(incl lock%......)

//some assembly language => \*p++;

}

[end]

xchgl %eax, %edx

- update and return old value

[c]

int xchgl(int\* p, int v){

int ov = \*p;

\*p = v;

return ov;

}

[end]

Now we have these hardware instructions, we can implement lock better

[c]

void lock(int\* m){

//if returns 1, someone else has lock. Once returns 0, we toggled the value so we have lock now

while(xchgl(m, 1))

continue;

}

//assumes precondition that we own lock, so it's fine

void unlock(int \*m){

\*m = 0;

}

[end]

How to make functions atomic?

[c]

compare\_and\_swap(int\* v, int o, in n){

if(\*v == 0){

return 1; //<^these two lines are a single instruction atomically

}

return 0;

}

for(;;){

int y = f(x);

if(cas(&x, x, y)){

break;

}

}

[end]

Hardware Lock Elision (Intel Haswell Architecture)

lock: movl $1, %eax

try: xacquire lock xchgl %eax, %rbx

cmpl $0, %eax

jnz try

ret

unlock: xrelease movl $0,%rbx

ret

Avoid Polling with Blocking mutex

- try to get lock, if you can't go to sleep

- wait for lock to become available

- can be implemented atop spin locks

[c]

typedef struct bmutex{

mutex\_t m;

bool locked;

struct process \*waiting, \*\*waiting\_tail;//pointer to linked list of processes that are waiting on this mutex to stop, can be null terminated

}bmutex\_t;

void acquire(bmutex\_t \*b){

lock(&b->m);

if(!b->locked){

b->locked = 1;

unlock(&b->m);

return;

}

self->next = NULL;

\*b->waiting\_tail = self;

b->waiting\_tail = &self->next;

unlock(&b->m);

yield(); //let someone else run

}

void release(bmutex\_t \*b){

lock(&b->m);

b->locked = 0;

proc \*p = b->waiting;

if(p){

p->blocked = false;

b->waiting = p->next;

if(!b->waiting){

b->waiting\_tail = &b->waiting;

}

}

unlock(&b->m);

}

[end]

The problem with this implementation for pipes is the OS doesnt know why im going to sleep after writing

- can't distinguish operations, when to read?

This is what we rlly want, wake up when no one uses pipe && pipe is full!

otherwise pipe will be constantly polling.

[c]

acquire(&p->m);

if(p->w - p->r == N){

release(&p->m);

yield("until p->w - p->r != N");

}

[end]

This is where condition variables come in

2/15/2017

Condition Vars API:

void wait (condvar\_t \*c, bmutex\_t \*b); represents boolean expression that "the pipe is full"

- precondition is b is already acquired

- releases b, blocks until some other process notifies

- reacquires b, then returns

notify cond(condvar\_t \*c);

broadcastCond(condvar\_t \*c);

[c]

struct pipe {

char buf[8192];

size\_t r, w;

bmutex\_t b;

condvar\_t pipe\_is\_full;

condvar\_t pipe\_is\_empty; // used for READING

}

void write(struct pipe \*p, charc){

acquire (&p->b);

while(p->w - p->r == 8192)

wait(&p->pipe\_is\_full, p->b);

p->buf[p->w++%8192] = c;

release(&p->b);

notify\_cond(!&p->pipe\_is\_empty);

}

[end]

Semaphores are blocking mutexes but with an int (not a boolean) that represents the amount or number of resources available

- eg = 10 simultaneous users

- P down acquire .... P(&s) prolaag (dutch) for try and decrease

- V up release verhoog (dutch) "increment"

DeadLock:

cat read(0, &buf, 512); write (1, &buf, 512);

copy(0, 1, 512);

Deadlocks are a pain

- only happen a few times but still an issue

- happens when so many locks that nothing happens

Deadlock is a race condition

- 4 conditions to get a Deadlock

1) circular wait

- constant cycle of waiting for each process to release something

- process could start waiting for self

2) mutual exclusion

- if one thing has an acquired resource, no one else can access

3) no preemption of locks

- If preemption offered, no Deadlock

- basically opportunity to break the lock and force threads off

4) Hold and Wait

- holding one resource while waiting for another to be available

- if only 1 lock at a time its fine, but multiple locks for one thread simultaneously is killer

If and of these conditions is prevented, deadlock is broken

#Detecting Deadlocks Dynamically#

Here, the OS tracks every process and the resources they hold or are trying to hold. It builds a directed graph and makes sure it’s acyclic. If a cycle were to be established based on the new arc, then the lock request will be rejected, giving an error status like EDEADLOCK for lock().

The kernel is helping applications, but apps must now be modified to watch out for deadlock

Deadlock is over-synchronization

- everything gets jammed up

- can get more complicated though, example

Parent process that forks a child

sends data there by pipe, gets results back through another pipe

They both have buffers trying to write to each other

P- write(small command); write(small command);

c- read(small command); write(large result); hangs

=> still have to worry about poss of deadlock

=> but develop as if it doesnt exist

- only way to stop deadlocks is by understanding whole system

New Bug: Priority Inversion (Mars Pathfinder 1997)

3 threads/Priority

T-low T-med T-high

- T-high is super priority, bug fixes etc

- T-med is science stuff, is important but eh

- T-low is like battery, is also import but meh dont really care

If we start with T-low being the only runnable thread

All of a sudden, there's a context switch to T-high

T-high wants the resource from T-low, but its blocked and yields

T-medium comes in and starts running, and T-low is hung

!Because T-low is hung, T-high is hung as well!

Some Solutions to help

1) Lock preemption, but can corrupt the data

2) Keep track of who had the lock

- if someone with higher priority wants lock, run low priority with priority of calling thread so it will finish faster

- this way no starvation and runs faster

This is kind of crazy, talking so much about threading in such simple sections

Needs to be a better way

Our Saviour: Event Driven Programming

Very popular in embedded systems

[c]

for(;;){

e = get\_next\_event();

handle\_event(e); //<-this must finish quickly!

}

[end]

Event handlers have to be very fast

Essentially, EDP is the inverse of threads

Completely single-threaded system

- use asynch events instead

The thing is, this does not scale to multi-core

- if want scaling, many single cores communicate via network

While no Deadlock, EDP does not kill all race conditions

Biggest Example is Livelock

Livelock: CPU is always doing something, but nothing useful is happening

Specifically, receive Livelock

requests are coming in from external place

|------ capactity

| (actions/sec)

|\_\_\_\_\_\_ work done y axis

load ->

requests/sec

Want it such that we work until we hit capacity and work at capacity optimally

----------

/

/

/

but often, in EDP this process goes to 0

.-----.

/ \

/ \

/ \

/ \

Scenario, the overhead of all the subevents after the handle event call ends up killing overhead of getting event

This is called #receive livelock#

Solution: be disciplined about which requests to handle

Discard events more enthusiastically

- disable interrupts when the system is loaded

Talkin' bout file systems

120 PB = 120000 TB

200000 drives each 600 GB

gpfs: General parallel file system

Distributed metadata

The data about the data locations is distributed

metadata: the result of ls -l

--rwrrrx anshul hello-------

Needs to be distributed since if everyone wants to ask about it, then it wont be returned

Multiple copies of the metadata

- this way we can ask about metadata then find the file, makes it more efficient and can ask from cache

Efficiend Directory Indexing

lets say we have a directory containing 20,000,000 subfiles

- pls don't run ls

dir representation is essentially an array of entries

entries are name: pointer-to-file:

if call ls on this, read name of all of them, then go back through and linearly search through the directories

GPFS solves this with smarter indexing pf directories

We want to retain ability to create and delete files, but still find files fast

So instead of an array or hash table (which sucks at sorting) or a balanced binary tree (too bushy), we use a B+ tree, an n-ary tree with a large number of children per node.

--Distributed Locking--

We want to be able to make changes atomically

- essentially have a distributed system of nodes that handles different parts

- super complicated, beyond scope of current lecture

--Partition Awareness--

Even when network is split up into a ton of nodes, want to know when any part is cut

- Solution, count how many nodes on each side of partition. Larger one keeps read/write ops while smaller one can only read

--File system stays up during maintainence--

- linux has file system check (fsck)

- checks for errors in FS and tries to repair them

- if it is under maintainence DO NOT TOUCH IT

- reading is ok but writing is nono

- theoretically should do it during boot

- takes too long to read all the data at once

- made it robust so it can be fixed while live

Today's lecture was supposed to be about File System Performance

- get to it later

2/22/2017

File Systems:

GPFS: General Parallel File System

- Distributed metadata: metadata stored on multiple disks

- Efficient directory indexing: use an n-ary tree

- Distributed locking: no single node "owns" an object

- Partition awareness: FS aware of partition. clients on each side can essentially operate independently

- File system stays up during maintenance

File System:

Organization

Robustness

Performance

--- Performance ---

- Throughput, bytes/sec, I/O

- Utilization (0-1)

- Latency (delay)

. How long you have to wait before you get access to a file

. Typically, improving throughput hurts latency, et vice versa

. Balance between the two

File system devices

Hard disks: 300 Mb/s external data rate (peak rate, data rate of controller)

95 Mb/s sustained data rate (rate for sustained readign)

32 MB cache

5.5 ms avg latency (random access)

5900 RPM spindle speed (±98.33 Hz)

512 bytes sector

50,000 contact starts/stops (how many times the drive can turn off)

10^14 non recoverable read erros/bit (1/10^14 chance a bit will be corrupted)

0.32% AFR (annualized failure rate) (if drive was running for a year, chance the entire disk fails)

6.8 W avg operation, 5.5 W idle

2 A startup current

70 g operating shock

300 g non operating shock

2.5 bels acoustic output

2 TB capacity

Flash:

60 GB capacity

3 Gb/s

1000g shock

7 W

2 Gb/s I/O r/w

50k IOPS (4k writes) (200 MB/s)

Flash wears out

Techniques for performance

- Batching

. r/w continguous sector - low level

. getchar reads into buf - high level

- Dallying

. wait for a bit before doing requests so you can batch them

- Speculation

. Prefetching

- grab data from device before it's needed

. guess what IO will happen next

. Assume locality of reference

- Spatial locality

. Accessing address i means accessing i+1 is likely

- Temporal locality

. Accessing at time t means accessing at time t+1 is likely

--- Organization ---

RT-II FS - made by eggert as a kid

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|/// file /// file|

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In first part, was directory

array of director entries

each entry looks like:

name of file|start location(sector num)|num bytes

files are continguous

+ simple

+ throughput

- pain to grow

- fragmentation

internal fragmentation:

storage assigned to file but not used

worse case: 1 bit. 511/512 will be unused

external fragmentation:

free storage capacity is enough, but it's scattered, so no big files anymore

FAT (File Allocation Table)

(boot sector) | (superblock/version/size/used) | (FAT) | (data)

FAT array

entries are block numbers

stores block number of next block after i in current file

0 block is EOF

-1 is free

N: next block in current file

Accessing Nth block in file is O(N)

directories in FAT: are files (specially marked)

. contain data

. array of directory entries

(name)|(size)|(1st block number)

block number of root directory stored in superblock, which will then contain the rest of the files

very hard to move files...instead copy then delete

Unix File System design:

(system block)|(inode table-1 entry per file)

each inode entry:

- timestamps (m - inode,c,a - access)

- uid, guid, permissions

- link count (number of directory entries that point to this file)

block numbers

inode table

if file is too big, indirect block

if file is still too big, use double indirect block

the indirect block points to an indirect block table

2/27/2017

Inodes(indirect node) are:

- Fixed size meta data about the file.

- 1 per file which is indeced via the inode number

- $ls -il file 1 file 2 ... which will give 319264 (inode number) -rw-r-r--2 eggert "lastModified" file1, etc etc

- ls -il calls stat which creates a data structure with all the metadata of various things in the file system

Inode does NOT tell you:

- the name of the file

- the directory of the file

This indirection can be a performance problem

- can also be a correctness problem ... so we have a linked count (the 2 is the number of pointers to the same file)

- increment link count when there are more hard links

- there is no primitive to delete the inode (delinode(219376)) or like openinode(219376, O\_RDONLY) (2nd is more efficient)

- downside for openinode is security (ex /a/ is rw---, while /a/b/c is -rw-r-r, the second should not be readable by anyone since the parent is not readable for others, but with openinode, you can just read it with openinode)

- no extra hard links to directories (apart from the fact that every directory is linked to its parent)

- ln /usr /usr/eggert/dir will make inode 37 create a directory entry but that will increment link count which cannot be easily deleted

Linux ext3 v2 directory entry

4 bytes inode number, 2 bytes dir entry length, 1 byte name length, 1 byte for file type, and then some 256 bytes for the name (1 to 255 bytes)

- if the name is not the full 255 bytes, then it will be unused -> ex if entry length is 50, and name length is 5 then you have a total of 4 + 2 + 1 + 1 + 1 + 5 = 13. so 37 bytes that are unused

- the unused entries are used for padding, and alignment

- you can also change an entry to junk by making the name length very long, and then we can never access/get the entry (thats an easy way to unlink)

- we also can get varying length file names by doing this

So how to implement open("file"m,...) and unlink("file",...)

- use namei: filenames -> inode numbers

- "" is invalid :)

- "a" = look for entry "a" in directory 2917 which returns 7634

- "a/bb" = first 2917, then 7634, and then changed to 291 which is the inode for bb

- "." = just be the current direcotry inode

- "a/bb/." = 2917 goes to 7634 goes to 291, stays at 291

- "a/bb/./.." = somewhere in 291, we would have the .. which would point back to 2917

- this is not a violation of the link counts since we cannot unlink directories ... rmdir("foo") will work if directory is empty, so link count is just self pointer

- treats two or more / as just being one single slash

- if you put a slash at the end, then the system treats it ike a dot after the slash so its a directory. CANNOT do it if its a file (so no file/)

- if you put slash at the beginning, like /a/bb/./.., then you go to the root directory section of the process table and get that inode

In our process table, we had the working directory column, and in the process table entry, that means an inode number

cd /usr ... how does this actually work

- ls a, now we need to be able to change the working directory of the process table

- system call to do that is chdir("/usr"), and takes file name, runs through namei and gets the directory number to put into the process table.

- chroot will change the root directory of the current process. fork will give the child the parent working and root directory

- .. is the same as root when we are at the root.

- sudo at /etc/password or /etc/shadow these are really locked down since only have permission for everything for root user

- only the root user can call chroot, so cannot use a chrooted jail (ln /etc/pass /usr/eggert/jail/etc/pass, then chroot(/usr/egger/jail), but cant be done)

So far we have seen regular file, and directory, so now let's look at symbolic link ->

- ln -s foo bar, or ls -li which might output 239416 lrw-r--r-- \_\_\_\_\_\_\_\_\_\_\_\_\_\_ bar -> foo

- cat bar will do open("bar", ...) which will look at 2917, but then it will try to find the actual foo file instead

- if we link bar to bar, then the procedure will go into infinite look. To avoid this, namei will see if there are too many symbolic links (20+), it will just give up

- for short names, some OS will optimze the inode entry by putting all the symbolic links in the name section

- use readlink("b", buff, sizeof(buf)) -> which will figure out what to print out after the ELOOP error

How to find out if a file is a symbolic link?

- we cannot just always run stat("b", &st) ... because namei will take the b and follow the link

- so we use lstat("b", &st) which, if it is a symbolic link, then it does not follow that link ... ls by default just calls lstat on everything

- unlink("b") will never ever follow a symbolic link

We can do ln a b, and then ln -s b c

- we can also do ln c d, since this will try link("c", "d")

- symlinks have inode numbers in linux ... so we can have multiple hard links to c and d so C not followed

- in other systems, symlinks are directory entries, so C is followed

mkdir a b

touch a/f b/f

ln -s f a/g // just means that symbolic link is treated sorta differently depending on the current directory

ln a/g b/g

ln -s A B

diff A B // you can actually make this have differing outputs

fifos, we can make using mkfifo foo (named pipe).

- echo baz > foo ls -l will make prw-rw....

There are also contigous files -> extra speed, but can lead to fragmentation

Named sockets as well, message queues, sempahores, shared memory objects, typed memory obkects,

mknod (some arguments) like 1 c /dev/null which creates /dev/null ... c which is short for character special file.

Multiple file systems- -> a and b, LINUX has something called mount table which maps inodes to file systems

- root file system, user file system ... whenever we see a certain number, we switch file systems and tries to work with that

- run mount command to see the mount table.

- Identify each of the file systems by using a device number. In reality, every file is actually described by device number AND inode number

3/1/2017

File System Performance, Organization, and robustness:

- Multiple file systems

- Flash drive A, Hard disk B

- How to copy from A to B?

- Mount table: map inodes to file systems

- It will look like nested file systems

- Files are identified by device number inode number pair

Levels -> symbolic links, file names, directories file/name components, inodes, file system/mount, pointers, blocks, sectors (in this order)

At the block level: at any point in time the block level will have many requests

- primary memory will have the current instructions/data and then that will later go on the disk (read + write apps)

- the rest of the pending requests will just be in ram ... eg write 200000 "7 Kib" or read from 1000000 etc ....

- can have worst cases thousands of pending requests ... so order matters (block scheduler)

Block scheduler looks at all pending requests and looks at what to do next

- GOALS: high throughput of IO's per second (IOPS). No starvation either

- Simplest algorithm: do the cheapest request first

Simple model Example: |input - location| for the distance needed from the current head location to the position on the disk

- called SSTF which stands for shortest seek time first ... will leed to starvation if lots of requests near one area, and one in a far far place that will never get called

- Average seek time if random locations .... avg[i-h] randomly chosen [0...1] -> h is the location of the read head

h = 0, avg = 0.5 ... h = 0.5, avg = 0.25 ... h = 1/3, then avg = 1/3(1/6) + 2/3(1/3) = 5/18

take integral from 0 to 1 of h(h/2) + (1-h)(1-h/2) dh ... = 1/3

- another one is called elevator which goes up and down and reaches all the sectors in between in order

- another version is ciruclar elevator which always goes one way and when it reaches the top, and then just reappears at the bottom of the sector again

- example 1 and 2 are in queue ... do 1, done. Then wait a little bit and move the head to the right to finish 2. Then move readhead back to do 3

then 4 comes in and it goes the other way away again ... this is very inefficient since long back and forth moves are not good

- Anticipatory scheduling is used to dally a bit after finishing one operation and see if some requests come in that are close to what you want. This can cause starvation ... so maybe combine with elevator algorithm or have a timer to stop the dallying

Robustness Terminology:

- error is a mistake in the designer's head

- fault is a latent problem in the deployed system

- failure is when the problem actually happens and the system behaves wrongly

File system goals in robustness:

- Durability is the file system survives with a loss of power

- Atomicity is either the write fully happened or not happened ... like money in bank (either transaction happens or no)

- Performance as well!!!!!

Example ... a simple app that opens emacs, reads a file, edits buffer, and then writes to file ... if write is atomic, then you are fine. else its gone

- Golden rule of atomicity ... never overwrite the only copy of your data if you cant assume atomicity

- two regions of data, old and new file ... if system crashes in new file, we at least have an old file ... how to figure out which is the new file though

- if each file has a time stamp at the start and that will tell u which one is newer but that means if we crash, still might be bad since we will chose the new file

- write the time stamp at end ... so if we crash and we dont have a time stamp, discard the "new" unfinished file and keep the old one ... called a commit record

old "A"

in between ?? write ("B")

new "B"

2 copies, non atomic the questions marks are garbage if we crash in the middle and no time stamp ... cant arbritarily take one or the other

A A old

? A

B A intermediate

B ?

B B new

1 2 // solutuon is to have a 3rd copy ....

AAA -> ?AA -> BAA -> B?A -> BBA -> BB? -> BBB done ... at each point you get A or B. if all 3 agree, then we are good. If 2 agree, take that. If all 3 disagree (b ? a) take the first one (get the latest version of data)

-

Lampson-Sturgis created some asssumptions to read/write better

1) Storage writes may fail and may corrupt other pieces of storage. A later read will report this as an error

2) Storage blocks can decay spontaneously (and a later read can report the error)

3) THe rate of failure is relatively low ... if blocks fail at high rate or all fail then its gone but low rate means u should be good

for example, system call rename("d/a", "e/b");

- from user's point of view, either have it happen or not ... but remember that these are just nodes (lets say 77 ad 100)

- somewhere directory entry will have some name, and that name will map to an inode ... delete the old inode for the name (like a)

- we just create a new entry for e and put b in there ... BUT if it crashes, then we might either have 2 links which is bad when recovering space from the disk, or have lost the data

- solution is to make another inode entry in a different inode block (like say inode 12) which has link count 2, then write data block 2, then db 1, then inode block again with link count 1 ... there are leaked storage though if crashes after db 1 and we see link count 2 but really should be 1

- fsck can look for leak by reading the file system into RAM and see which storage blocks arent used ... its a pretty slow program and it will be standalone (cant use/do anything else).

Make file system invariants (logical statements that should always be true)

- boolean expressions that must be true

- example, for a BSD style file system, bit map of free blocls, inode, direct pointers, and then indirect pointers

1) each reachable block in filesystem contains initialized valid data for its format

2) Each block is used for exactly one purpose

3) All referenced data blocks are marked as used in the bitmap

4) ALl unreachable data blocks are also marked free in the bitmap

All invariants will conflict so decide which ones are more important (3 or 4) ... violate 3 = disaster, violate 4 is u need fsck to work

3/13/2017

--Distributed Systems--

Goal is to increase performance via parallelism

Reliability via redundancy

simple!

via abstraction

r = f(a,b)

s = g(r,a)

^becomes client code, no longer cheap its kind of expensive

Remote Procedure Call (RPC)

- args are copied via network to remote location

- result is copied back

- System Calls on Steroids

- instead of just trapping into kernel, have to shift calls into ethernet

--Diff RPC vs ordinary calls--

- args always copied

- no call-by-reference it doesnt work!

- pass pointer, server side doesn't care about a pointer!

- this is a common approach that is blocked

- this is how C buffers work, the read command

- every java object is internally handled by references

- Caller and callee must agree on format of copies sent over the wire

- pass an integer, client -> server

- if client says int = 16 and server says int = 32, issue arises

- in this case, client should pad the integer with 16 extra bits so server will work

- another issue is representing signed integers

- usually 2's complement, but you dont know

- little endian vs big endian issues

- can reverse the bytes, gets misinterpreted

- big endian most popular

- client "marshals" data structures to server, has to stay intact

- protocol defines how to send over data structures

--Reliability Issues that Arise--

- messages get lost

- messages get corrupted

- messages get duplicated

- Network may be down

- server may be down

--How to solve--

- corruption

- checksum our messages + resend corrupted message

- lost messages

- acknowledgements

- resends if no acknowledgements after timeout

- duplicated

- sequence numbers

- unique id in message lets you know if the same one was sent earlier

- server/network down

- can't tell the differenece between the two tbh

- retry if no answer -> at least once RPC

- this is can be bad

- transfer 100$, dont get a reply, keep transfering

- use on actions that you don't care if it happens multiple times

- clear bank account, just set bank account to 0

- can be called multiple times and nobody cares, really just want it to happen

- report failure to the caller/client -> at most once RPC

- what we really want is exactly once RPC

- super hard to make, very difficult algorithim and its super expensive

|\|

| |

|/|

|\|

| |

|/|

c s

--Performance Issues that Arise--

- distance issues

- move client and server closer

- fatter wires/Smaller Object Representation

- really faster

- helps with throughput, doesnt help with latency

- Multiple Threads

- Cached Version

- big win for latency, can speed it up

- but cache can be wrong/obsolete

- Batching

- send one big question than multiple small questions

- pipelining

- Don't send questions Serially

- send all at once, then wait

- |\|

|\|

|/|

|\|

|/|

|/|

c s

- have to deal with out of order answers and some answers may be lost

- do pipelining if all the questions are independent

Before jumping into these problems and their solutions, we're going to talk about

--Redundancy--

Build on ideas from non-distibuted systems

RAID (Redundant Array of (inexpensive/independent) Disks)

- Assumption with all RAID is that failed drives will be replaced almost immediately

- all finish before next fail

- RAID 0 (inexpensive)

- OG motivation is find the cheapest drives in cost per byte

- 2TB drive is cheapest, but want a 10 TB drive

- Easy, just link 5 2TB drives together, will be cheaper than buying 1 10TB drive

- Problem though, this will fail at a rate 5 times greater than a regular drive, less reliable

- RAID 1 (mirroring)

- Each drive has a copy of itself, so 10 drives

- if any single drive fails, then can be replaced and re-coppied

- while strong on Reliability, very expensive

- Eggert uses this for his computer though

- RAID 2 and 3 were kinda bad so we don't care

- RAID 4 (Parity)

- Have all your data drives, then one extra drive called your parity drive

- |A|B|C|D|E|A^B^C^D^E| (^ = XOR)

- If want to change D -> D', A^B^C^D^E^D^D'

- Theoretically can rebuild another drive by XOR all of the parity bits with everything else

- can rebuild any ine failed disk

- slows down writes pretty hard, have to read from corresponding block in parity bit then rewrite it

- the Parity drive becomes the hotspot, double the IO of the other drives

- SEASnet uses RAID 4

- Adding a new drive is cheap and trivial

- just make the drive all 0's and its already included in the parity drive

- RAID 5

- Split disk into a bunch of "stripes"

|A1|B1|C1|D1|E1|A1^B1^C1^D1^E1|

|A2|B2|C2|D2|A2^B2^C2^D2^E2|E2|

...

- distributes it, much more reliable and distribues the I/O so theres no hotspot

- generally don't grow these types of systems

Mean time to failure (MTTF)

Mean time to recover (MTTR)

Mean time between failures (MTBF) = MTTF + MTTR

availability = MTTF/MTBF

downtime = 1 - availability

Network File System (NFS)

Many clients, few processors, goes to many storage devices

Linux App will use same system calls as before, no idea its an NFS system

- open read write etc

Goes to Virtual File System

- just marshals over wire to NFS server

--NFS Protocol--

looks suspiciously like UNIX syscalls but theyr aren't

LOOKUP(dir fh, name) -> File handle & atrributes

- file handle (fh) short name for file (like an inode #)

- Unix version -> open("a/b/c", ...)

- LOOKUP(cd, "a") -> fh1

- LOOKUP(fh1, "b") -> fh2

- LOOKUP(fh2, "c") -> fh3

CREATE(dir fh, name, attr) -> fh + attr

MKDIR(dir fh, name, attr) -> ""

REMOVE(dir fh, name) -> status

RMDIR(dir fh, name) -> status

READ(fh, ...)

WRITE(fh, ...)

etc

Design goal for NFS is that it uses \*Stateless System\*

Want fileserver to not care or know about client state

- if fileserver crashes and reboots, everything is ok, no important state is lost

- makes simple reliable server but has performance Issues

- can't do batch requests, bc have to store previous questions in cache

- if it crashes then lose that state for the return

3/15/2017

Security (OS < Computer < System)

Computer Attacks

- against privacy

- against integrity

- against service

This is the biggest software constraint

- otherwise, assume universe is benign, let anything run and few bugs no harm no foul

- now have to assume world is out to get you

Step 1: Threat Modeling

- Insiders

- where are your backups stored?

- who is managing your system?

- who are the people involved?

- Social Engineering

- use people and trick them into giving you their password

- these two are the largest security threats

- Network Attacks

- computers running bad software or evil software

- viruses, drive-by downloads

- Denial of Service attacks

- buffer overruns

- device attacks

- USB sticks with malware

Should have probabilities, costs if someone breaks in

Step 2: Design

General functions for defense

- Authentication

- e.g. password

- integrity

- unauthorized changes to data

- checksum

- Authorization

- access control list

- Auditing

- log file check

- Correctness

- static checking of code

- Performance

- make sure overhead isnt too large where users will give up

--How to do it--

Authentication

- based on what they know

- bad if attacker might know it too

- based on what they have

- token, car keys etc

- based on who they are

- biometric Authentication

None are perfect

Use two factor Authentication, ideally use two different formats

^ most of these are susceptible to Social Engineering

- designed to slow down attack, make it more expensive to do

Authentication in practice, external vs internal

external

- check who you are look at you

- done by sentry, expensive

- SEASnet Password

internal

- assumes external is done, just does periodic checks done more often

- process table entry, checks user number Authentication