BS

Evolution of Computers

first gen 1945-55

electronic vacuum tubes or relays
patch bays

second gen 55-965

transistors + reliability
punch cards + ~~but~~ batch processing
early languages: FORTRAN & COBOL

third gen 65-80

IC - Integrated Circuit
multiprogramming + multicasting
e.g. fairly between users cpu time

fourth gen 77 - now
personal computers
fifth gen 90 - now
mobile computers

OS

Hardware Abstraction + portability, reuse

VM for applications / execution environment
access to hw. resources Memory, IO, Devices, -

Resource Management & Control
shared access
fair & efficient
optimize
caches
Concurrency & Sync
Isolation
memory protection
Access control
to sensitive ops

System Design Concepts
Abstractions
processes, threads, sockets, files, disks, devices
Mechanism
open, create, fork, schedule, exec, close, send
Policies
how...?

Computers

Assembly Language Intel x86-32, x64-64 types
mnemonic operands / instructions opcode. param

Endianness
intel → Big endian MSB first
intend → Little endian LSB
e.g. 0xAF11
0x0000AF11
0x11AF0000

CPU Registers tiny, fast
(Segment r.: identify segments in memory)
General Purpose r.: immediate values, mem addresses
Program status and control r.
eip - extended Instruction pointer to next to be executed
other eflags

8x 32 bit general purpose on x86
EAX - Accumulator
EBX - Base (mem addr.)
ECX - Counter (loops, strings)
EDX - Data (I/O Pointer)
ESI - Source Index for String OP
EDI - Dest. "
EBP - Base Pointer (stack)
ESP - Stack Pointer

User Interface Program
OS Interface
OS
Hardware Interface

OS Protection Boundary

User Mode / Kernel Mode — by privilege bit in processor

privileged ops in kernel m. else fault

0 Kernel
1 Drivers
2 "
3 Applications
→ Modern OS

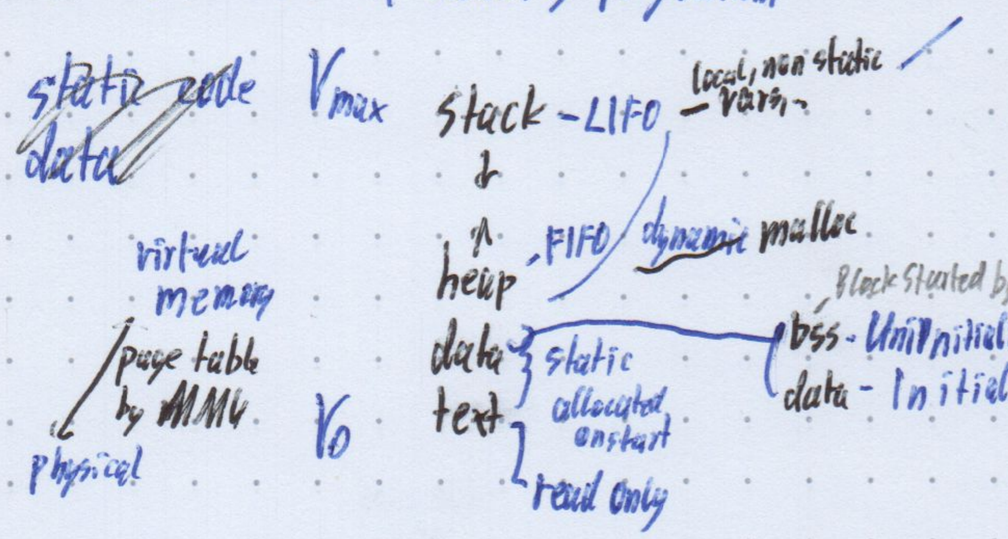
System Calls - request service from kernel

- hardware related
- creation & exec processes
- communicate with kernel services eg. scheduling

1. call libary syscall wrapper
2. accumulator := syscall numbe & param
3. trap function → software interupt
4. Processor switch kernel mode invokes syscall dispatcher
5. look up number → handler
6. processing
7. return control in user mode

0	read	uint fd	char* buff, size_t count
1	write	"	
2	open		
3	close		
...			
9	mmap	map additional memory to process	
10	mprotect	change permission to mem area	
11	munmap	remove mem from virtual addr spce	
...			

Process - abstraction of running programm



Stack Frame per Function Call

args
ret. addr
BP of calling frame
var 1
...

via x86 call instruction
eg. call foo
↳ automatically pushes ret. addr ↳ instruction after call

ret
↳ pops ret addr, loads it into EIP

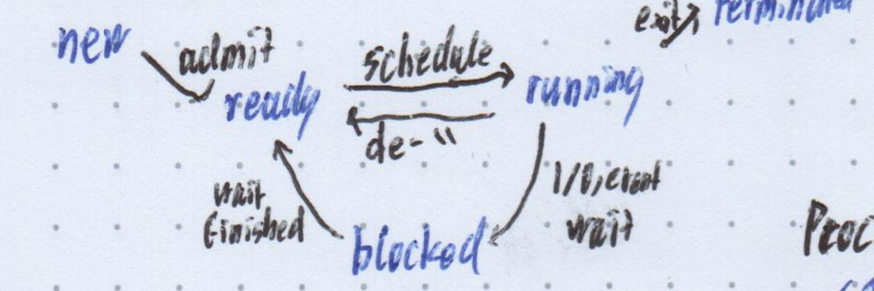
Multiprogramming

multiple processes on system conceptually on own virtual CPU
pseudoparallelism - through rapid context switches
↳ few ms

Process table.

Non-active saved in PCB - Process Control Block
contains state
PC, Registers, Stack Pointer
Memory allocations, open files
accounting & scheduling info
ID, status

Process States

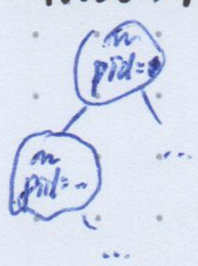


POSIX - Portable Operating System Interface

Process Creation

```
pid = fork();
if(pid == 0){ //child process
  variant of exec() replace mem space with new program
} else {
  wait(NULL); //waits for childs
  exit(0); //terminates, deallocates
}
parent can kill() child processes
```

Process tree



IPC - Inter Process Communication

Message Passing - based
send() → Message queue → read()
- syscall overhead

Shared Memory - based
mem region mapped into both virtual mem
+ faster - complex
error prone

Memory Management / Organisation

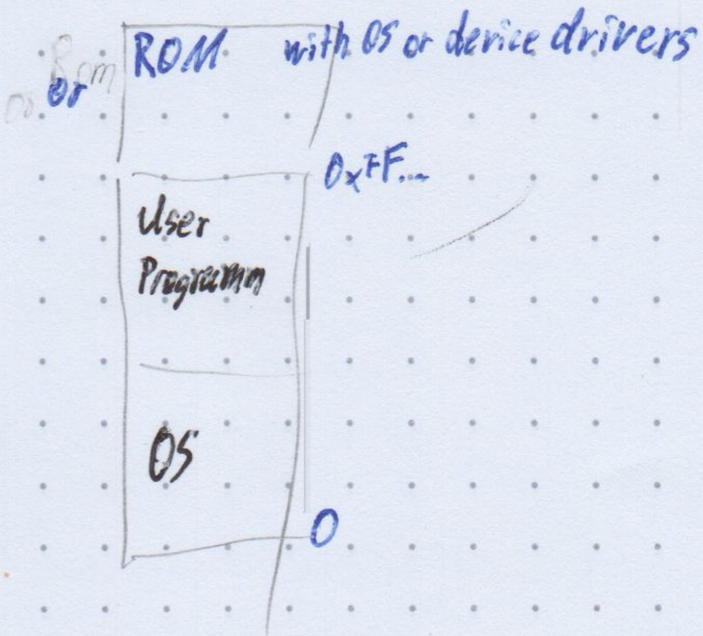
Requirements

Protection between processes
Fast translation
context switch

No Memory Abstraction

processes see physical memory
⇒ programm needs to know where it lies ⇒ parallel only with swapping
or Static Relocation during loading
+slow
-needs additional info

time for prompt-driven systems
only one process at a time



Allocate Memory to Process

+multiple processes in parallel

Fixed Partitions

physical addr = base addr + logical addr.
each process gets partition of same size

+easy
+fast context switch
-internal fragmentation mem inside partition not used by process

zu viel
zu wenig
⇓

Variable Partitions

variable size partitions
base and limit register
physical > base+limit ⇒ addressing error

+memory protection
-external fragmentation loading, unloading creates empty holes between partitions

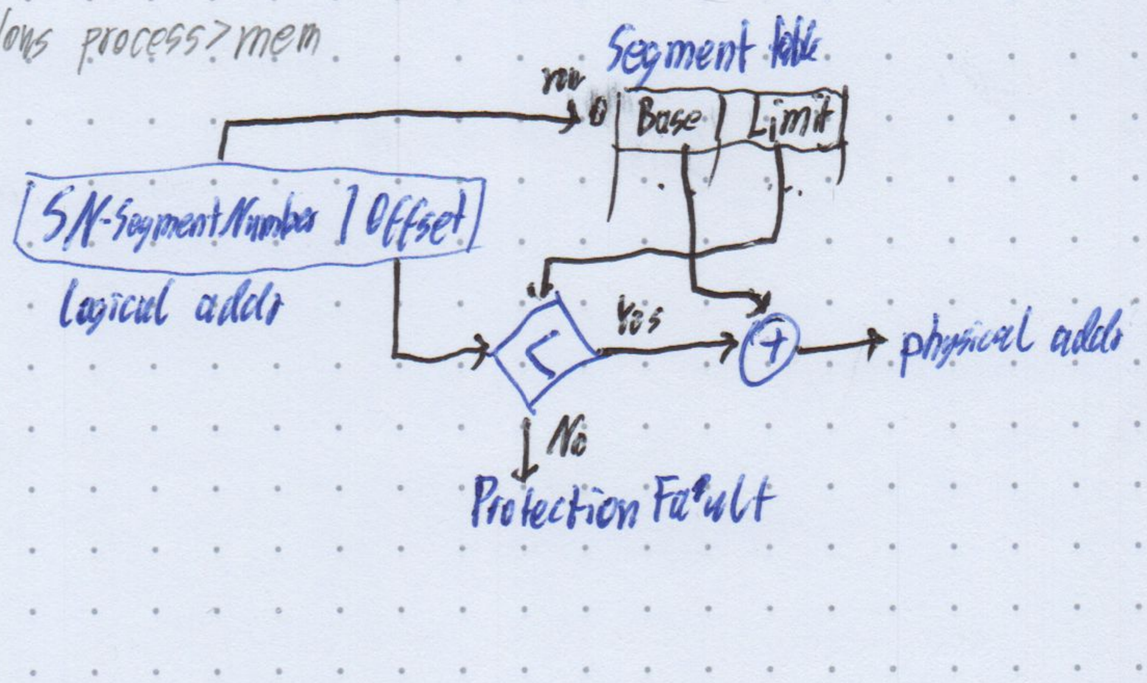
intel x86 segmentation Global Deskriptor table
Code, Data, Stack, Extra, F, G → Segment Deskriptor
-Segments
Access | Limit
base
...
Validation bit
read/write/exec

Virtual Memory

physical location can change
be not continuously
virtual addr must not ~~not in physic~~ be in physical Memory ⇒ allows process > mem

Segmentation extension variable Partition

allows many segments per process
STBR - Segment table base register
STLR - " length



+ only segments need continuous memory
+ no internal frag, + mem protection
- external fragmentation

Paging

Page - equally sized mem block (usually 4kB)
managed independently

rwx Permissions of mapping

0..1 → memory frame / physical page

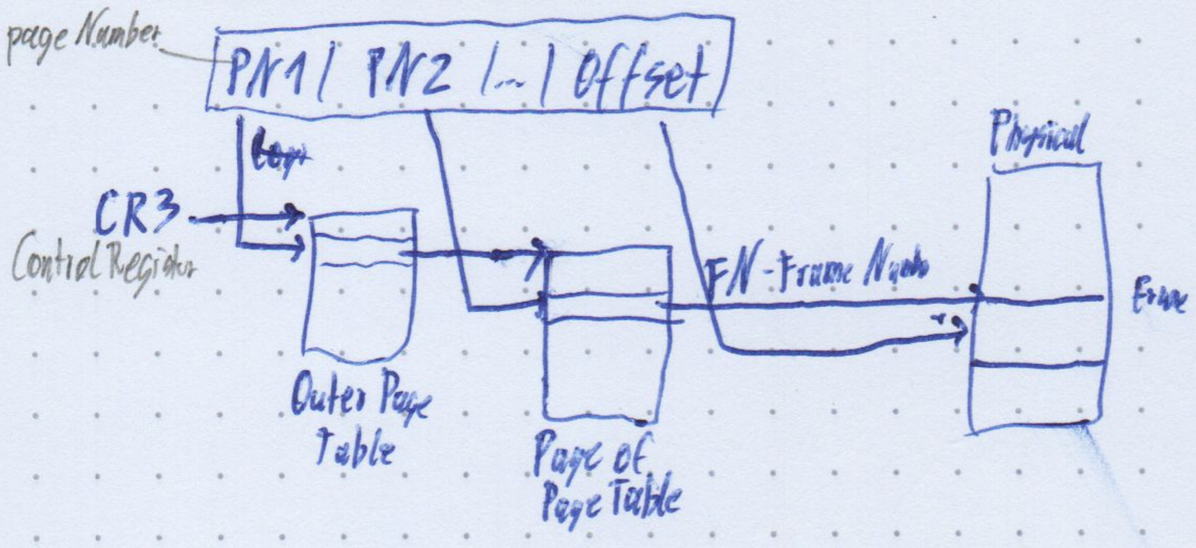
Page Sharing

n virtual pages → 1 physical page

z.B. load common data only once
(read only)

shared mem for IPC

Address Translation



large ⇒ not in Registers

⇒ TLB - translation Look-aside Buffer
cache for translations

x86 4 Level Paging
PML4E
Page-Directory-Pointer-Table
Page Directory
Page Table
) can be skipped for larger pages

Page Protection

access control information in Page Table Entrys

- protection bit
- Valid bits
  - present in Memory
  - page fault

Read / Write
User / Supervisor
Write Through
Cache Disabled
Accessed
Dirt
Global

Scheduling & Threads

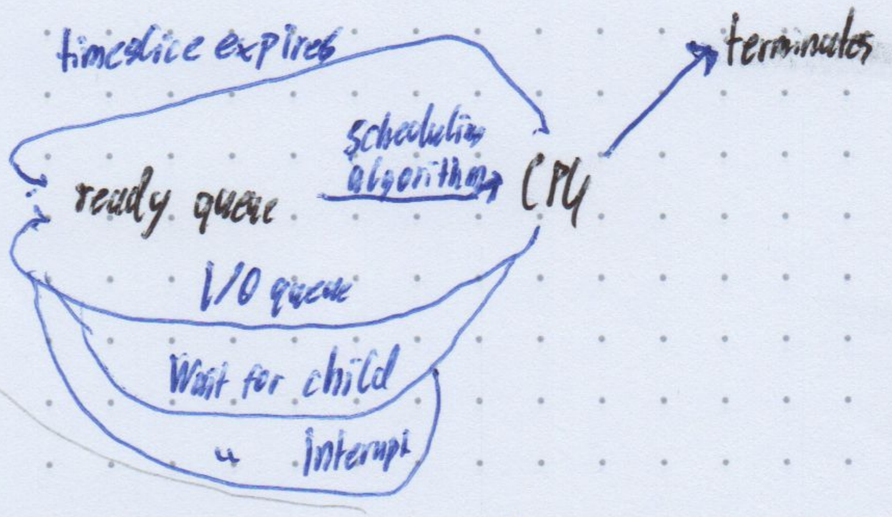
multiprocessing ⇒ ↗ CPU utilization = $1 - p_L^n$ (n = degree of multiprogramming; p_L = fraction of time waiting for I/O)

CPU- & I/O-bound processes

Scheduler - lowest level of OS

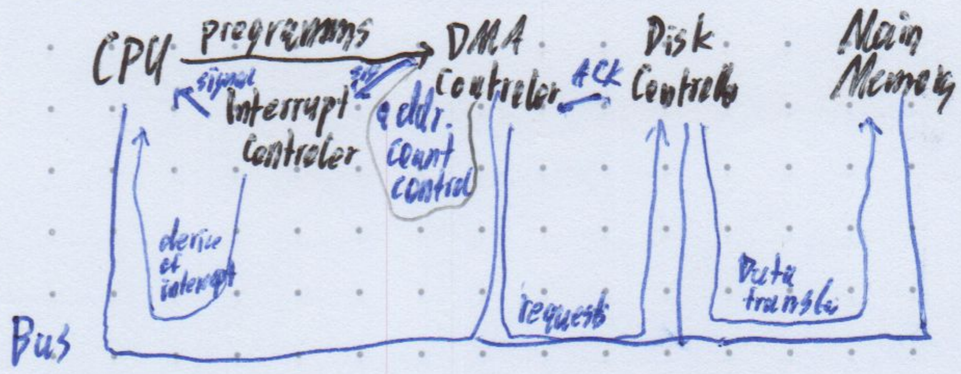
handles interrupts & scheduling
└ which first?

process creation →



Interrupts - signals event to OS — MUST BE EFFICIENT (100s per sec)
e.g. input ready/output complete
exception (I/O, attempt protected mem/privileged instr.)
clock interrupts
CPU checks Interrupt-request line after V instr.
└ dispatches to routine at specific addr.
└ context switch

e.g. DMA - Direct Memory Access



When to schedule?

fork < parent or child next?

process terminates
blocked

interrupt incl. clock interrupts eg. 250 Hz

How?

environment
- batch - big tasks ⇒ throughput, turnaround time, utilisation ⇒ non-preemptive (turnaround time = avg. time to complete job)
- interactive - active users ⇒ response time (for user active), enforce proportionality
  - └ quick jobs fast, big can take time
- Real-time - hard timing requirements (e.g. Multimedia) ⇒ +predictability
  - └ else deterioration in media

general goals
- fairness in same priority
- Policy enforcement
- Balance of computation/I/O usw. for utilization with

Preemptive ↔ Non-preemptive scheduling
- └ suspend, when max timeslice
- └ until blocks or releases ⚠

Strategies (resp. time)
- First Come, First served ⚠ diff. types
- SJF - Shortest Job First - needs estimate + optimal avg. turnaround time
- Shortest Remaining Time Next - preemptive SJF version - long jobs wait long

└ Round-Robin
First come, first serve with quantum - max timeslice at a time
- too long - resp time
- too short - many context switches

└ Priority
queue for each priority ~~- higher scheduled before lower~~
higher before lower

└ Multiple Queues
lower priority queues get ↗ e.g. 2× mehr quanta when scheduled

└ Shortest Process Next
estimates e.g. based on previous observations
+ if alternating pattern of comand-exec
$T_n = \alpha T_{n-1} + (1-\alpha) T_{prev}$ (n: runtime number; aging param)

└ Guaranteed $\frac{1}{n}$ CPU time (n = # users)
~~next process with min~~ min $\frac{\text{received}}{\text{entitled}}$ time ratio
+ fair
e.g. Linux CFS Completly fair Scheduling

└ Lottery
winning ~ # tickets
└ Co-operating may share
z.B.: user A: A, B
user B: E
⇒ AEBEAE...

└ Fair-Share
user gets fraction CPU time regardless # processes

BRUNNEN

Threads - lightweight processes

per Thread: PC, Reg, Stack, State

per process: code, data, files, accounting info, ...

+thread scheduler
+ no IPC between
+faster creation, destruction

POSIX Threads

Pthread_ create
exit
join wait for spezific thread
yield release CPU
attr_init
attr_destroy

User ↔ Kernel-Level implementation (Runtime System, Thread Table)

+ faster switching if no blocking syscall anyway
+ no OS support needed

+ better blocking syscall handling
- replicate V on fork?

Hybrid Kernel-Level Threads having User-level threads

Swapping

no mem ⇒ swap process to disk

Manage free mem

Bitmaps
0 free
1 occupied
allocation unit size ▽ may waste space
larger bitmap slower finding

Linked Lists
ordered (Process/H, start pos, length)
↳ Hole

Allocation of segments

Allocation Strategies

First Fit

Next Fit from last place of search

Best Fit - searches all
smallest large enough - leaves small rest holes

Worst Fit
largest available

Quick Fit + fast find
list for each hole size - slow combining on swap

Paging only pages 'swapped out'/'evicted' to ~~mem~~ disk

Page Fault

minor in mem but not mapped in process addr. space ⇒ upd. page table
major not in mem ⇒ load
Segmentation fault invalid addr. ⇒ kill p

Page Replacement Algorithm ~~frame limit per proc~~

Optimal
least likely referenced in near future
impossible, but after run + baseline to eval other

Not recently used NRU

Class	R	M
0	0	0
1	0	1
2	1	0
3	1	1

random of lowest class

First-In, First-Out + easy - problematic

Second-Chance
FIFO, but when R=1 requeued as R=0 like new page

Clock
Second Chance but circular list (clock hand moves not pages)

Least Recently Used LRU - needs update @ every reference
impl. with list last used in front, least in back

Not Frequently used NFU
counter ~ page
↳ @ clock interrupt += R
evict lowest counter
- doesn't forget long past references

Aging
counter ~ page
↳ @ clock interrupt (R, ctr) >> 1
bit width limits remembering

Working Set
↳ set of pages process currently using utilizes locality of reference
↳ p tends to only refer subset at a time
last time ~ page
@ page fault: R=1 update time R←0
R=0 age > τ ⇒ evicted
if none random with oldest time

WSClock last time/age ~ page, clock hand
R=1 set age, advance
R=0 age > τ ∧ M=1 schedule write, advance
M=0 evict
if no and no write scheduled, evict any clean, else current

Demand Paging / Lazy Loading
no preloading + no unnecessary load

Page table entry
Caching Disabled bit - e.g. for memory mapped I/O
Referenced bit ←0 @ clock interrupt ←1 @ referenced
Modified / Dirty bit
Protection bits rwe
Present/absent

Thrashing if process ~~larger~~ > available frames
Seitenflattern
evicted soon needed ⇒ many page faults

Allocation Policies

Local ↔ Global page replacement
↳ selected among frames of p
↳ among all
- risk thrashing or wasting frames

#frames dynamic
+ if working set size varies

Allocation size

periodic fixed ∀ active processes fixed fraction
- doesn't processes of diff. sizes

proportional to process size

adjusting based on Page Fault Frequency PFF
PFF > A ⇒ increase alloc
PFF < B ⇒ decrease alloc

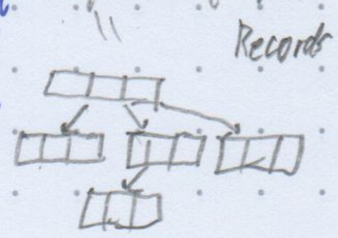
File Systems - controls how data stored/received

File - abstraction data blocks on disk

Types
- regular files
- directories - system files for structure
- Character special files - model I/O devices
- Block special files - model disk

Naming
- allows other processes to reference it
- conventions
  - Unix case sensitive, extension only convention
  - Windows not " , extension link to application

File Structure
- byte oriented
- record oriented
- tree structure

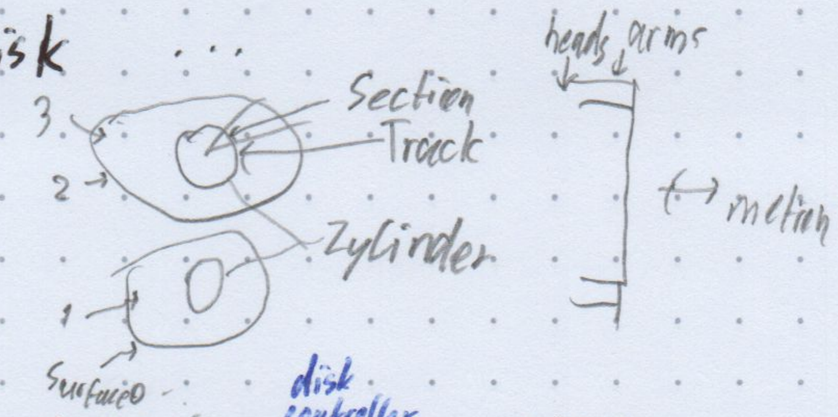


Types of Access
- sequential (e.g. magnetic tapes)
- random out of order possible
- Positioning File Reads read() specifies start or seek() moves position

Attributes
- Protection
- Password
- Creator
- Owner
- Read-only flag
- Hidden "
- System " system file?
- Archive " 0 has been backed up
- ASCII/binary " 0 ASCII, 1 binary
- Random access " 0 sequ access, 1 random
- Temporary " 1 delete on process exit
- Lock " allows only one p access
- Record length # bytes in record
- Key position Offset of key within each record
- length # bytes in key field
- Creation time
- Time of last access
- " change
- Current size #bytes
- Maximum "

Operations
- Create
- delete (name, O_RDONLY)
- Open ~~(name)~~ gets attr., adds. in mem returns file deskriptor
- close (deskriptor) remove from mem
- read (in. fd, char buf[size], size)
- write " at current pos may overwrite or extend
- append
- seek
- get/set attr.
- rename

Disk



disk blocks ⟷ (disk controller) physical blocks

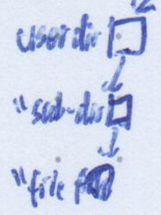
Delays
- Seek time to move arms to cylinder
- rotational delay for sector to
- read/write time

e.g. executeable
- Header (Magic Number, Text size, Entry point, "", Flags)
- Text
- Data
- Reloc bits
- Symbol table
- Data "
- Bss "
- Symbol table "

binary archive
- Header (Module name, Date, owner, Protection, size)
- Object module
- Header(...)
- Object module
- ...

Directories

hierarchical:



Pathnames
- absolut starts with root / , \ unix win
- relativ to working dir of process
  - syscall can change it
- Special names . current
- .. parent

operations
- create ., .. only entries
- delete
- opendir for reading
- closedir free mem
- readdir next entry
- rename
- link new link file→dir
- unlink removes "

file n—n dir
- removed if not referenced

BRUNNEN

Disk Organisation

MBR - Master Boot Record in sector 0
- used to boot
- partition table at end
  - always one marked active (contains OS)

one or more Disk Partitions
- all start with boot block
- typically: but file system dependent
  - superblock (magic num to identify type, #blocks, ...)
  - Free space mgmt
  - I-nodes
  - Root dir
  - Files and dirs

File Allocation

Continuous Allocation
+ easy — - external fragmentation
+ efficient — - know max size beforehand

Linked-List Allocation
first word of block pointer to next
- 0 für Ende
+ no external frag.
+ dir entry only needs start block
- slow random access
- block size not pow 2

File Allocation Table FAT
data structure in main mem
- physical block → {next block, -1 end of file, -2}
+ faster
+ Block pow of 2
- Large fully in mem

Index I-Nodes
- File attr.
- Addr. of disk block 0
- "
- ↑
- ↓
- 7
- Addr of block of pointers
  - containing additional disk addresses

+ only in mem when file open ⇒ less mem than FAT

Directory Impl

file with entry per file init
- or
  - fixed size with file attr (z.B start block)
  - Link to I-Node

Shared Files
Hard ↔ Symbolic Links
- to I-node
- Special file specifying target filename
  - additional overhead

Counter for #references

Design Considerations
Large ↔ small block size
- Long seeking - performance
- Internal frag - mem

Management of Free Blocks
Bitmap
Linked List of free blocks

LFS - Log-Structured File System
treats disk as circular log
buffer for small writes + less seek & rotational delay
then continuous at end of log
- scatters i-nodes
  - ⇒ i-node map to locate them

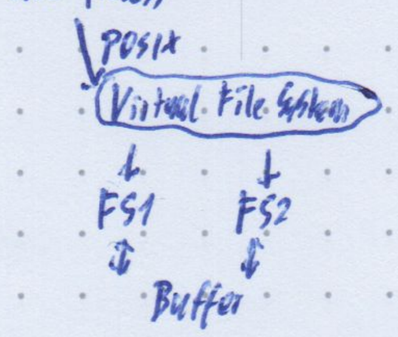
Cleaner thread through log (circular)
- removes obsolete data
- copies valid data to mem; then written to new segment

Journaling File System
assures consistency on sys. crash
1. record planned ops in log
2. write log to disk
3. exec. ops
4. upon sucess remove log entries

recover by reexecute pending ops
- need to be idempotent

normally problematic example
1. remove file from dir
2. release I-node
3. return disk blocks

Virtual File System
standardized interface for file ops



examples

MS-DOS
fixed sized dir entries of 32 Bytes
FAT-12, -16, -32 variants
- only 28 used

Disk blocks multiple of 512 bytes

⟩ results in max partition size

UNIX V7
filenames up to 14 chars
dir entries (filename, i-node num of file)

for large files
- indirect blocks

- file size, creation, last access, mod time
- Owner, group, protection (r/w/x),
- count of dir references, pointers to block addr. of data blocks

retrieve name
root dir → i-node —(if folder)→ mem block → ...

Disk Quotas
file/i-nodes and disk block limits
utilizes disk quota file on disk
link from open file to quota record
checked if file extended

Hard ↔ Soft limits
- error
- checked on login
  - decrementing warning counter
  - then ch>0 login denied

Remote Procedure Call RPC

between processes on networked system

! may fail or duplicate
⇒ timestamps, request history, Acks, resend

stubs - proxy for actual procedure
- client-side: locates server, marshalls call param
- server-side: receives mesg, unpacks param, performce procedure

Race condition

result depends on order each processes ops on shared data

ctr++ could be impl.
r=ctr
r=r+1
ctr=r

Critical region/section

part of programm modifying shared mem

Avoiding Race Conditions

1. No 2p. simultaneously in crit. region
2. No assumption on CPU speed, number
3. No blocking other p. outside crit. region
4. No waiting forever to enter crit. region

Disabling Interrupts after entering

enables just before leaving

- guarantee enabling?
- multiprocessors

Lock Variables

- descheduling right after lock test

shared var
init 0
@enterin:
lock==0 ⇒ =1
lock==1 ⇒ wait

Process Alternation / Spin lock

shared var turn

- busy waiting only reasonable for short
- ! especially for high-prio waiting
- p's of diff. speed

p0:
while(turn!=0) /*loop*/;
crit. region();
turn = 1;
non crit region;
p1: analog

Peterson's Solution

shared vars: turn, flag array interested[N]

interessted[process] = TRUE
turn = process;
while(turn==process && interested[other])
  //busy waiting
crit region
interested[process] = FALSE;

Atomic Instructions by locking mem bus

TSL RX, LOCK - loads LOCK into RX
Test&Set Lock — stores nonzero val in Lock
XCHG Rx, Lock - exchanges content
vie TSL nach MOVE Rx, #1

then check RX==0
Move Lock, #0 after crit region

Monitors - programming language - level

collection procedures, vars, data structures
only 1p at a time in monitor
impl bei compile.

monitor Name
  type name;
  procedure name(); begin ... end
end monitor

BRUNNEN

no busy waiting:

Sleep and Wakeup + processes blocked while waiting

e.g.

producer:
```
item = produce_item();
if(count==N) sleep();
insert_item(item);
count = count+1;
if(count==1) wakeup(consumer);
```

consumer:
```
if(count==0) sleep();
item = remove_item();
count = count-1;
if(count==N-1) wakeup(producer);
consume_item(item);
```

! deschedule right before sleep
missing wake up
go to sleep for good

Semaphores Dijkstra 1965

down(semaphore)
value > 0 ⇒ decrement
value == 0 ⇒ sleep, down op remains pending

up(semaphore)
increment
if 1 <= # processes sleeping on semaphor
⇒ wake one up

e.g.

Producer
```
item = produce();
down(&empty);
down(&mutex);
insert(item);
up(&mutex);
up(&full);
```

Consumer
```
down(&full);
down(&mutex);
item = remove();
up(&mutex);
up(&empty);
consume(item);
```

Mutex simplified Semaphore only locked, unlocked

impl. with atomic instr. in user space

mutex_lock:
```
TSL RX, MUTEX
CMP Rx, #0
JZE ok
CALL thread_yield
JMP mutex_lock //try again
ok: RET
```

mutex_unlock:
```
MOVE MUTEX, #0
RET
```

Condition Variables allows ps to block until condition

wait(condition var) blocks p
signal(condition var) wakes up random p sleeping on it

Deadlocks of set of p's

⇔ $\forall p$ wait for event only another in set can cause

Conditions for Resource Deadlocks

1. Mutual Exklusion of Resources (assigned to 1p or available)
2. Hold & wait p holding Res. can request additional
3. No pre-emption
4. Circular wait condition circular list p: each waiting for R held by next

Modelling

process (A)

assigned ↑ requesting

ressource [R]

Resource graph of errors

Handling

Ignoring
- likeliness?
- more than other crash causes?

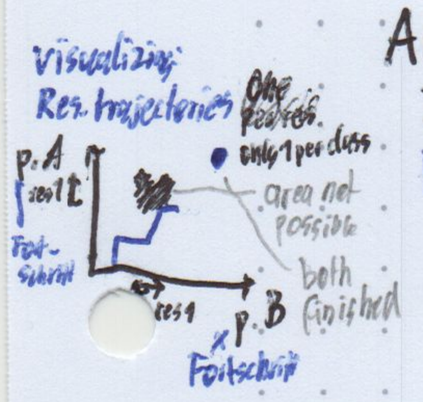
Detection in current situation
- construct resource graph
  - single res:
    - DFS - list of cur. path, mark visited arcs
    - node twice in path ⇒ cycle
  - multiple res of res classes:
    - $E=(E_1,\dots E_m)$ ressource vector
      - #res Existing in class i
    - $A=(A_1,\dots A_m)$
      - #res Available in class i
    - C - Current Allocation Matrix
      - C_{ij} P_i currently holds C_{ij} of class j
    - R - Request Matrix
      - P_i requests R_{ij} of class j

$\sum_{i=1}^{n} C_{ij} + A_j = E_j$

1. unmarked P_i with $R_i \leq A$
   - No → all unmarked deadlocked
   - found → mark P_i, $A+=C_i$

Recovery
- Pre-emption
  - not always gracefully possible
- Killing processes
  - until cycle broken
- Rollback
  - to periodically recorded checkpoints
    - Log (mem, res. state)
  - work lost
  - side effects? idempotent?

Visualizing Res. trajectories



Avoidance only grant res. if safe
- ∃ scheduling order ∀ complete even if ∀p request max #res now

Banker's Algorithm
- like detection with mult res marking as terminated
  - needs rarely known
  - # p's not fixed
  - res may fail

Resources

e.g. Blu-ray disk

Preemptable ↔ Non-preemptable e.g. Mem

└ can be forcibly taken away from p without negative effects

access pattern
- request `down(&semaphore_r);` direct request syscall to open...
- use ...
- release `up(...);`

Prevention of deadlock conditions

Mutual Exclusion Condition - no p exclusive access to res

e.g.
- Data resources in read-only access to data
  - not feasible
- Daemon processes ∀ access through them, they never request other res.

Hold & Wait Cond.

∀p must request ∀res before exec
- only scheduled, if available
- know in advance
- Use of res not optimal (e.g. long analysis in and out)

No-Pre-Emption Cond.

- difficult for some res.
- some can be virtualized
  - e.g. Print job written to mem exec by daemon
    - still deadlocks with disk space
    - not for database records

Circular Wait Cond.

∀p may hold only 1 res at a time - no simultaneous access to 2

global numbering of res
- ∀ request in num. order

Two-Phase-Locking

- 1. phase: req. ∀res for next op
- 2. phase: exec, release
- if 1. couldn't lock all ⇒ release, try again
  - real-time systems
  - process control system

Communication Deadlocks

A, B waiting for msg from other
- e.g. if msg lost ⇒ timeout detection, resend recovery

Livelock

2p acquire and release res simultaneously (held by other p because can't acquire res)
- not blocked but can't proceed

Starvation

e.g. shortest-job-first res allocation policy
- if short jobs with need for r continue to come in
  - long job never gets r
- avoided by first come, first served

Caches ~~& Memory Allocation~~

Motivation

main storage
- shared between processes
- cheap but slow
- low frequency

Hierarchy

CPU
Registers
~~expensive~~
Small Latency

Volatile SRAM
- L1 split 3rd ~10 cycles latency
- L2 unified ~100
- L3 shared ~1000

Volatile DRAM: Main Mem
- cheap
- high capacity

Non Volatile: Disk/Drive
- Hard Disk Drive
- Solid State Drive

Split ↔ Unified Caches

Split: L1 dedicated cache for instructions & data

Non-exclusive / Partially inclusive - can, but doesn't have to

Inclusive ↔ Exclusive Caches

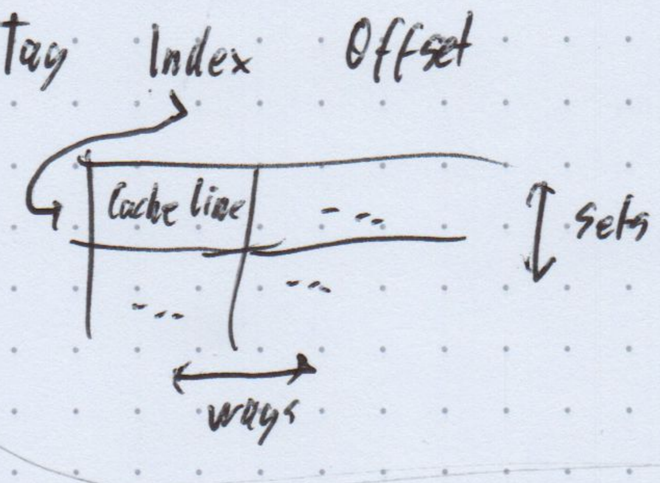
data stored in all cache levels

Exclusive: either core exclusive or shared

X-way set-associative

Fully associative ↔ Directly-mapped

Cache Architecture



Line:
V Tag DATA

Cache Addressing

PIPT - Physically Indexed Physically Tagged
- index, tag extracted from physical mem addr.
- needs virt→phy. translation before cache
  ⇒ latency

VIVT - Virtually Indexed Virtually Tagged
- index, tag extracted from virt. addr.
- only unique per process
- homonym problem: 1 virt. addr. → diff phy.
- synonym: ~~1~~ diff virt. → same phy.

VIPT - Virtually Indexed Physically Tagged
- addr. translation parallel to indexing step
- physical tag prevents homonym
- synonyms possible

Cache Control Flow

Write-hit Policies

Write-back ↔ Write-through
- changed in cache ⇒ needs Dirty bit forwarded to mem on invalidation
- synchron change in cache & mem

Write-miss Policies

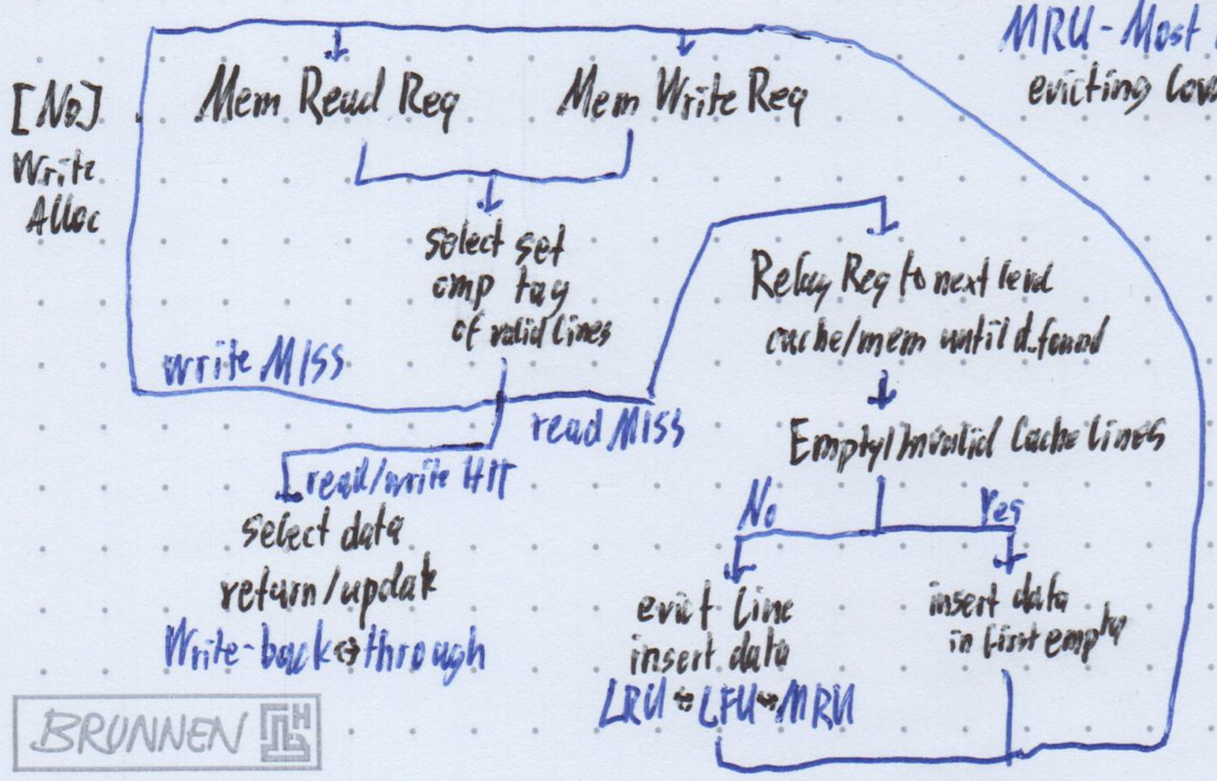
Write-allocate ↔ No-write allocate
- write-miss leads to exception triggering mem read req.
- usually combined with Write-back

Replacement Policies

LRU - Least Recently Used
- Line with oldest age - last time hit performed
- hardware overhead ⇒ pseudo-LRU estimate using less bits

LFU - Least Frequently Used
- ctr. how often line used evicting lowest

MRU - Most Recently Used
- evicting lowest age



BRUNNEN

Memory Mappings

Memory-Mapped Files
disk file mapped to phy. mem.

Copy-on-Write
@ change: apply on copied mem. page
+ same view on shared pages

I/O using I/O Ports ~ 8-/16-bit number
access to device control registers
special I/O instr.
IN Reg, Port
OUT ~~Reg, Port~~
Port, Reg

Memory-Mapped I/O
control reg ~ unique mem addr
+ op's directly on mem addr (instead of loading in regs)
+ no extra instr.
+ protection via mem mapping
- needs selective caching
- complicated with multiple buses (intended device must see)

Hybrid

Seperate Mem. Spaces
seperatly paged, dedicated page tables
z.B. I-space and D-space

DMA - Direct Memory Access
CPU programs DMA Controller
DMA repeatively instructs device to transfer byte-by-byte, device acks to addr in mem
DMA interrupts CPU when done

- if CPU much faster
- complexity (embedded systems)

Operating Modes

Word-at-a-time / cycle stealing
occasional short transfers 'stealing' bus

Block / Burst
series of transfers at once
-=> blocks bus for long time

Fly-by
DMA instructs device
directly read/write to/from mem

DMA stores/reads words itself
+ enables device-to-device
mem-to-mem

internal device buffer
+ checking checksums
+ data at steady rate
but bus volatile

Runtime Attacks

launched @exec
exploit vulnerabilities
affect benign programs
(binaries clean)
Detection @runtime expensive

Vulnerability
flaw enabling attacker using system in a manner not intended

Exploit
taking advantage of vulnerability
0-day not publicly disclosed

Denial of Service Attack
[Remote] Arbitrary Code Execution (e.g. system("/bin/sh"), creating user, opening port)
Privilege escalation

e.g. Buffer-Overflow vulnerability
risk: e.g. override ret.addr. to injected code ⇒ arbitrary code execution
fix: bounds check of untrusted inputs
before COPY(buffer[8], *usr_input

Malware = malicious software
e.g. install backdors
turn into **zombie**
botnet
controlled with command & control server

Ransomware encrypt disk
extort victim for money

Spyware steal sensitive info
e.g. identity theft

Trojan Horses appearing good, voluntarily downloaded

Viruses reproduces by attatching itself to other programms
exec payload potentially time-delayed
to give time to propagate

Worms spread independently over network
using exploits
bootstrap/loader to download actual worm
on victim system

IoT Malware targeting IoT devices
often bad impls/
weak passwds
e.g. Mirai

Device I/O

Layers

I/O req ↓
- User-level I/O software — spooling; I/O call, format
- Device-independent OS — naming, protection; blocking, buffering, allocation
- Device Drivers — set up regs, check state
- Interrupt handlers — wake up driver
- Hardware — perform op

reply ↑

Device types

Block devices
- 1 block at a time
- blocks independent of other

Character devices
- stream of characters
- not addressable, no seek

Device Controller

electronic component of device
- controlled via control regs
- controls device on low level
  - e.g. serial bit stream → blocks
  - error-correction

Interrupts

signals on bus
- disabled until enabled again

Interrupt Controller

- issues prioritized interrupt to CPU
- places number on addr. bus
- interrupt table contains routine addr. for it
- CPU acks to allow next — through I/O port

Interrupt Handling

Where to store interrupted
- Internal Regs
  - can only store fixed amount
- Process Stack
  - Stack pointer no legal value? page fault? ⇒ too slow
  - where to store for page fault routine?
- Kernel Stack
  - switching changes MMU context
  - invalidates cache and TLB

Processors
- pipelined
- superscalar
  - up on 8 instr. in parallel

Precise ⇔ Imprecise Interrupts
- well defined state
  - PC saved
  - instr. before completed
  - no instr. beyond
  - state of instr of PC known
- more state

may differ per type

Intel x86 provides precise for backward compatibility

Programmed I/O — busy waiting

system call:

```
copy_from_user(buffer, p, count);
while for(i=0; i<count; i++){
    while(*printer_status_reg != READY);
    *printer_data_reg = p[i];
}
return_to_user();
```

Interrupt-Driven I/O

Interrupt service procedure:

```
if (count==0) unblock_user();
else {*printer_data_reg = p[i];
  count--; i++;}
ack_interrupt();
return_from_interrupt();
```

I/O using DMA

Aspects

- Device independence
- Uniform naming using identifiers
- Error handling as low as possible — transparent recovery
- Synchronous ⇔ Asynchronous
- Buffering data rate ≠ processing

Device Drivers — device specific

- check params
- translate abstract terms
- queue if device in use
- initialize device status
- interact with device control regs
- block until interrupt (e.g. semaphore)
- check for errors, pass output

Anforderungen:
- reentrant — may be called again before finished
- Error handling
  - graceful termination
  - e.g. on disconnect

Handling (including Program Status Word)

1. save regs not already by hardware
2. setup context for interrupt service procedure
3. " stack "
4. ack interrupt controller / reenable interrupts
5. copy regs from where saved to process table
6. run interrupt service procedure extract info from device control regs
7. schedule new process
8. Set up MMU context, load regs
9. run

Device-Independent I/O-Software

typical tasks:
- Buffering
- Error Reporting
- Allocating, Releasing dedicated devices
- Providing device independent block size

Uniform Interfacing for Device Drivers
- + allows new devices without changing OS
- set of functions per class of devices
  - driver provides table of pointers

Uniform Naming
- UNIX: device's special file i-node contains
  - major number to locate driver
  - minor number passed to driver to identify unit (device)

I/O buffering
- Unbuffered ⇒ interrupt for every character
- User-level
  - + reduces interrupts
  - ~ page fault of buffer?
- Kernel-level then copied to user space
  - + no page fault
  - − buffer full before copied?
- Double buffering in kernel for use if:
  - interrupt while copying filled up buffer to userspace

User Space I/O Software

Libraries provide routines to issue requests

Spooling System for shared access
- daemon has (unlimited) exclusive access
- ps generate files in spooling directory

Security

Goals

Confidentiality - no exposure of data to unauthorised parties

Integrity - no unauthorized modification

Availability - nobody can make system unuseable
denial of service attack

authenticity - source of msg, sender's identity

accountability - e.g. by logs

non-repudiability - can't deny transaction, validity msg

privacy

Protection Domains - names ~~set of~~ objects mapped into it

e.g. user, Processes

e.g. File[RWx]
Plotter[W]

can be mapped into n Domains

Protection Matrix

objects — (level of access (RWx)) — domain

typically sparse ⇒ stored as

ACL - Access Control Lists

File → (domain, level of access)* (* = set of)

or

Capabilities

domain → set of capabilities
↳ (object, level of access)

Authentication

user knows e.g. passwords
has smartphone, smartcard
is biometric

UNIX Passwords

passwd file
username → encrypted password
↳ da von allen usern einsehbar

Encrypted
da z.B. von Admin einsehbar

Salted
user, salt, e(passwd, salt)
to hinder precomputation Attacks
↳ List of common passwords and encrypted

Adversary Model - capabilities of an adversary

in general Turing Machine - able to perform arbitrary ops

e.g. for networks: Dolev-Yao adversary can
intercept any msg
authorized participant (send, receive)
spoof identity

DAC - Discretionary Access Control

users decide access rights

MAC - Mandatory Access Control + high security (military, hospitals)

system enforces rules

Bell-La Padula Security Model

Users ~ security levels
objects

read down
write up
simple security / integrity property
[integrity] * property
⇒ Confidentiality

Biba Model

no write up
no read down
⇒ Integrity

! Covert Channels

P passes info to other
by influencing observable system property
e.g. Modulating CPU usage
paging rate
Locking files
Aquiring devices
...

Side Channel - info extracted from impl. of computer system

e.g. Cache leakage in shared physical system
Timing info for computation
Power consumption
electromagnetic Radiation
...

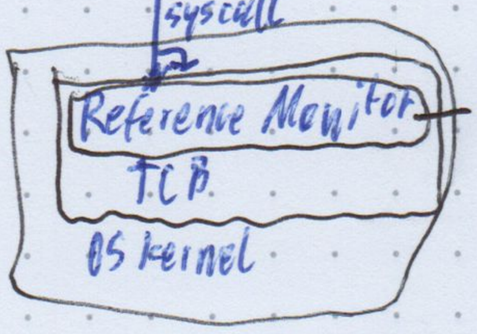
TCB - Trusted Computing Base

hardware & software components nesecary to enforce security rules.

typically
security-relevant hardware
Parts of OS-kernel
User programs with superuser privileges

necessary functionality
P. creation, switching
Mem. Management
Part of File, I/O Management

TCB as small as possible



CFG - Control Flow Graph
node ~ Basic block
code sequence
with 1 branch in/out

Code Reuse - adding path to CFG
- limited to available nodes
- needs static analysis of attacked code

Hybrid - e.g. use Library code to copy
Malware into area, then making it executable

Code Injektion - adding new node to CFG
+ arbitrary code (remote console, exploit OS)

but
DEP - Data Execution Prevention

Return-into-libc Attacks Solar Designer 1997
- relies on library (no changes)
- not directly arbitrary exec
redirect execution to security critical func in shared libaries
e.g. libc fn with open() malloc(), printf(), system(), execve()
e.g system("/bin/sh"), exit())

Cache-based Side-Channel Attacks
Prime + Probe - measures if victim overade cache line
└ overrides all cache lines
- Noise e.g durch scheduling

e.g Buffer Overflow
└ protection: strncpy(), strncat(), snprintf()
instead strcpy(), strcat(), sprintf()
manually check bounds
ASLR - Address Space Layout Randomization
Stack Canaries
Data-Execution Prevention
Compiler protection: non-executable stack

e.g AES uses substitution box
key ~ S-box access
- know code S-box (open)
- know memory mapping
- trigger single encryption

[column: least significant nible of key
[row: most ~

Virtualisation

Pros:
- Fault tolerance if system crashes
- Isolation of sensitive data
- no dedicated hardware
- Deployability: all dependencies, right versions
  - allows freezing and starting from new location
  - virtual appliance - package app in VM
- Reliability
  - hypervisor way smaller than OS

Kernel Kernel
VM1 VM2 ...
virtual machine manager
hardware

Requirements
Safety - hypervisor full control of virtualised resources
Fidelity - behavior VM = as if hardware
Efficiency - little hypervisor intervention

Sensitive instr ⊆ Privileged instr (Popek & Goldberg 1972)
- exec diff in user/kernel mode
- trap in user mode

Trap-and-Emulate
Guest OS thinks it's in kernel mode (virtual kernel mode)
privileged op => trap to hypervisor
- instr by guestOS: carry out instr
- process: emulate HW response

Binary Translation for sensitive ops that don't support trapping
rewrite Basic Code Blocks before exec
- sensitive op → call to hypervisor
  - final branch instr.

Guest OS in ring 1 to protect kernel Mem from user processes

Older Processors: e.g. popf instr. loads diff flags from stack in kernel+user mode
- designed for calculator
- backwards compatibl

Current ~: 2005: Intel VT, ~~AMD SVM~~ AMD SVM (Technology, Secure)
Containers for guest OSs
hardware bitmap controls trapped ops
~~Trap-and~~

Paravirtualisation ↔ full virtualisation
not transparent to guestOS
Machine-like interface: hypercalls for sensitive op's
+ simpler, faster impl.

Type 1 ↔ Type 2 hypervisor
- Type 1: on hardware
- Type 2: process on host OS

BRUNNEN