2017-1 Operating System

42205 characters in 6612 words on 1173 lines

Florian Moser

February 20, 2018

1 INTRODUCTION

1.1 what makes a good OS

reliability
does it keep working

security
is it compromised

portability
how easy can it be retarget

performance
how fast/cheap is it

adoptation
will people use it

1.2 roles

1.2.1 referee

resource manager

1.2.1.1 sharing

multiplex hardware among application, application are not aware of each other

1.2.1.2 protection

ensure one application can now read write data of another, and cannot use other's resources

1.2.1.3 communication

protected application must still be able to communicate

1.2.1.4 goals

fairness
no starvation

efficiency
best use of maschine resources

predictability
guarantee real-time performance

1.2.1.5 examples

scheduling algorythmus (batch, interactive, realtime)

1.2.2 illusionist

virtualization
create illusion of a real resource, however simplified view of it

1.2.2.1 multiplexing

divide resources among clients

1.2.2.2 emulation

create illusion of a resource

1.2.2.3 aggregation

join multiple resources together to create a new one

1.2.2.4 goals

sharing
enable multiple clients of single resource

sandboxing
prevent client from accessing other's resources

decoupling
avoid tying client to particular instance of resource

abstraction
make resources easier to use

1.2.2.5 example

resource abstractions (memory, CPU, virtual memory, virtual machines, files, virtual circuits)

1.2.3 glue

os as abstract machine

1.2.3.1 provides high level abstraction of hardware

easier to programm to, contains shared functionality, ties together

1.2.3.2 extends hardware with added functionality

direct programming of hardware unnecessary

1.2.3.3 hides details of hardware

application is decoupled from specific device

1.2.3.4 goals

abstraction
decouple application from hardware

1.2.3.5 example

syscalls, services, driver interface

1.3 services provided by OS

programm execution
load programm, execute it on CPU

access to IO
network, HDD, SSD

protection & access control
for files, connections

error detection & reporting
trap handling

accounting & auditing
stats, billing, etc

1.4 general OS structure

1.4.1 user mode

contains applications with system library, and server processes (deamons)

1.4.2 privileged mode

contains the kernel & has access to resources

1.4.3 kernel

runs privileged programs

event driven server, handles
system calls, hardware interrupts, program traps

1.4.4 system call wrapper

convenience functions, function which then call the kernel

1.4.5 deamons

processes which are part of OS
easier to schedule & fault tolerance, better than placing it in the kernel

1.4.6 kernel enter

occurrs when
startup, interrup (caused by something else), exception/trap (cased by user program), system call

the way programs request services from the kernel

1.4.7 kernel exit

creating a new process
resuming process after trap
user-level upcall
switching to another process

2 PROCESSES

2.1 process

Each process provides the resources needed to execute a program.
A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution.
Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.
process is the execution of a program with restricted rights
"virtual maschine"

ingredients
virtual processor (with address space, registers, Instruction pointer)
program text (object code (at the bottom of stack))
program data (static, heap, stack (from top to bottom))
OS stuff (open files, sockets, security rights)

2.2 Thread

A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources.
In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled.
The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.
Threads can also have their own security context, which can be used for impersonating clients.

2.3 process livecycle

runnable
dispatch to running

blocked
IO completed to runnable

running
IO operation to blocked, preemption to runnable, terminate

OS time-division / space-division processes

Process Control Block PCB
in-kernel data structure
holds all info about process (identifier, registers, memory used, pointers, address space, files & sockets open, ...)

switching
enter kernel
save PCB(A)
restore from PCB(B)
exit kernel

2.4 creation

2.4.1 need

code to run, memory to run it in, basic IO setup, identification

2.4.2 windows

what to run, what rights, environement (folder etc)

2.4.3 unix

fork
creates child copy of calling process, if returns 0 → in child, if > 0 → in parent

exec
replaces text of calling programm with the specified

wait
need to be called by parent to clean up child processes → child processes enter "undead"/zombie state till this happens

2.5 kernel architecture

unix
one kernel stack per process, thread scheduler runs on thread #1, so each switch are actually two!

barrelfish
one kernel stack per core, more efficient, more complicated

2.6 perform system call

marshall arguments somewhere safe
save registers
load system call number
execute SYSCALL instruction
→ kernel entered at fixed address
save user stack pointer & return address in PCB
load SP for this process' kernel stack
create C stack frame on kernel stack
load syscall number from jumptable
→ execute function
load user space stack pointer
adjust return address to point to user space, or to retry syscall
execute syscall return instruction

2.7 user space threads

user thread advantages
easy to create & destroy
cheap to context switch

kernel thread advantages
blocking can be handled nicely
easier scheduling

many to one
early thread libraries
pure user-level threads, tasks, lightweight processes
no kernel support required
many user level threads belong to one kernel thread
inactive thread stacks are allocated on heap (bottom) active one on top
cheap to create / destroy, fast context switch
one thread can block entire process

one to one
every user thread has kernel thread
multiple process share address space (but process now refered to as group of threads)
each thread gets portion of whole address space
slow to switch but easy scheduling, nice handling of blocking

many to many
multiplex user-level threads over multiple kernel threads
the way to go for multicore
can pin user thread to specific kernel thread

3 SCHEDULING

3.1 scheduling

how to allocate a specific resource for multiple clients (how long & in what order)

usually refers to CPU scheduling
on which CPU, how long, which task

optimize
fairness, policy, utilization, power usage

objectives
depend on workload; batch jobs, interactive, realtime, multimedia

complexity of scheduling algos
scheduling complexity vs quality of scheduling

frequency of scheduling
higher context switch rates decrease throughput (flush TLB, caches, pipeline; reduces locality)

implementation
A timer interrupt arrives (hardware feauture)
Processor switches to kernel mode and executes the interrupt handler (IH)
IH saves the registers in processes user-mode stack (calculate address)
IH alls scheduler to determine the next process
IH load registers from the new processes user-mode stack
Switch to the new process

3.2 workloads

3.2.1 wait time

time spend waiting to be scheduled
minimized by SJF

3.2.2 turnaround time

time from submission to termination (total time)

3.2.3 response time

till first time scheduled
minimized by RR

3.2.4 throughput

jobs/time

3.2.5 utilization

CPU used for processes (not scheduling)

3.2.6 batch processing

run job to complextion and tell when its done
typical usecase in supercomputer

goals
throughput, wait time, turnaround time, utilization

3.2.7 interactive workloads

wait for external events & react in reasonable time
word processing, mouse movements

goals
response time, proportionality (some things should be quicker)

3.2.8 soft realtime workloads

this task must complete in 50ms / this task runs all 50ms for 10ms
data aquisition, IO processing, multimedia applications

goals
deadlines, guarantees, predictability

3.2.9 hard readtime workloads

execute actions at very specific points of time
plane, car automation

3.3 assumptions

CPU-bound task
long streams of CPU usage

IO bound task
multiple small streams of CPU usage

3.4 when to schedule

running process blocks (initiates IO or waits for child) or calls yield()
blocked process unblocks (IO finishes)
running or waiting process terminates
interrup occurs (time or IO)

3.5 preemption

non-preemptive
each process needs to explicitly give up the scheduler

preemptive
dispatched / descheduled without warning (common case, in soft-realtime)

3.6 scheduling overhead

dispatch latency
time taken to dispatch a runnable process

scheduling cost
2x half context switch + scheduling time

maximum response time
max time till process is scheduled

3.7 batch scheduling

3.7.1 first-come first-served

first process in queue is completed first

convoy phenomen
short processes back up behind long ones

3.7.2 shortes job first

minimizes waiting / turnaround time, always executes the shortes job in queue

better
"shortest remaining time next" because new jobs always arrive

3.7.3 round robin

runs all tasks for a fixed timeframe in turn
good response time
but treats all tasks the same

3.7.4 priority

assign every task a priority, highest priority gets dispatched
priorities can be dynamically changed
same priority tasks can be scheduled with RR, FCFS...

3.7.5 multi level queues

executes tasks with same priorities different depending on high (with RR) / low (with FCFS)

ageing (avoid starvation)
threads who waited a long time have increases priority, reset priority after executes once

penalize CPU tasks
penalize tasks which use entire quantum → IO tasks are not penalized because they block before

3.7.6 example

linux o(1)
multilevel feedback queue, 0-99 high priority with RR, 100-139 for user threads with ageing, two arrays; runnable & waiting; switch if all array is empty

linux completely fair scheduler o(logn)
priority = how little progress has been made, bonus if tasks yields early, fudge factors adjustments over time, "generalized processor scheduling", guarantees service rates

problems with unix
conflates (vermischt) protection domain & resoure principal → simply create more processes to get more resource

3.8 resource containers

OS abstraction
operations to create/destroy, assosiate threads, sockets with containers
independent of scheduling algo
→ all operations are accounted for by container

forms
virtual maschine, containers

3.9 real time scheduling

problem
give real time guarantees to tasks (can appear at any time, can have deadlines, execution time is known, periodic or aperiodic); reject tasks which can not fit schedule

rate monotonic scheduling
schedule periodic tasks by always scheduling shortest task first
m tasks, Ci execution time, Pi period → will find solution if SUM(Ci/Pi) ≤ m(2^(1/m) - 1)

earliest deadline first
schedule tasks which has the earliest deadline (more complex for decisions)
will find feasable solution if possible
can guaranteee rate of progress for longrunning task

3.10 multiprocessor scheduler

one queue not possible because of locking → one queue for each processor core

affinity scheduling
keep same thread at same core, rebalance cores at larger time steps

parallel applications
try to schedule threads of same application together (avoid cache pollution, avoid one slow thread, enable cache sharing)

problem is NP hard
where & when to schedule, need to have locks released before scheduling, cores are not equal

little's law
average number of active tasks is equal to average arrival & average execution time

3.11 hardware support for synchronization / critical sections

disable all interrupts (to avoid interruption inside critical section) → does not work in multiprocessor, instead use TAS possibilities

3.11.1 atomic operations

TAS
read value of location, set to one

CAS
compare value with expected old one, replace with new one if true

LL
Load-Link, load from location and mark as owned

SC
Store-Conditional: store if marked by same processor, clear marks set by other processors, return result

3.11.2 spinning

only makes sense on multiprocessor, spinning cheaper than resheduling, but only makes sense if lock owner is active (on another core)

competitive spinning
spin for context switch time → worst case twice as bad as resheduling, best case do not need to reshedule at all

3.11.3 IPC

mutexes
aquire / release (classic lock, in C#: can be used for interprocess locking)

semaphores
wait / signal (generalized mutex, has atomic counter which allows a specific count of clients in critical section)

condition variables
wait / notify / notifyAll (implemented with semaphore)

monitors
enter / exit (implemented with semaphore, in C#: more lightweight than mutexes)

3.12 scheduling with locks

priority inversion
if lock held by low priority thread, but wanted by high priority one

priority inheritance
process holding lock inherits priority of process waiting on that lock, afterwards priority resets → ensure progress

priority ceiling
process holding lock aquires highest priority of all processes that can hold the lock

3.13 IPC without shared memory

asynchronous
receiver blocks, sender fire&forget

synchronous
sender blocks till receiver is ready

duality of messages & shared memory
Any shared-memory system (e.g., one based on monitors and condition variables) is equivalent to a non-shared-memory system (based on messages)

3.14 unix pipes

IPC message passing

3.14.1 code

int pipef[2];
pipe(pepif); //if -1 returned → error;
pid_t id = fork(); //if -1 returned → error;

if (id == 0) { //child reads from pipe
close(pipef[1]); //close write end
read(pipef[0], &buf, 1); //read from pipe
close(pipef[0]; //close again, can exit now

} else { //parent sends to pipe
close(pipef[0]); //close read end
write(pipef[1], "hi", strlen("hi")); //write to pipe
close(pipef[1]); //close write end
wait(NULL); //wait for child process

}

3.14.2 shell pipes

with | operator

3.15 messaging systems

endpoint may not know each other
messages may need to be sent to multiple receipients
multiple arriving messages might need to be demultiplexed
timeouts

port
naming different endpoints in process

3.16 naming pipes

to put pipe in global namespace

3.16.1 code

console 1
mkfifo /etc/hi
echo "hi mom" > /etc/hi

console 2
cat /etc/hi

3.17 local remote procedure call

define procedural interface in Intermedial Definition Language
compile / link stubs
transparent procedure calls implemented with messages

3.18 unix signals

asynchronous notification form the kernel, but receiver does not wait
interrupt process, and kill/freeze it, continue with another

3.18.1 some types

SIGSEGV
core dump → from memory management subsystem

SIGPIPE
no reader from pipe which has been written to → from IPC system

SIGKILL, SIGSTOP, SIGCONT
kill, stop, continue process → from other user processes

SIGFPE
kernel trap handler

SIGNINT, SIGQUIT, SIGHUP
ctrl-c, ctrl-\, handup → from TTY subsystem (console)

3.18.2 signal handler

overrides default action of OS if a signal occurres
of the form void my_handler(int)
very litle capabilities; no programm variables, many C stuff not possible

same signal multiple times
deliver just one

multiple signals
deliver them in order

3.18.3 are a form of upcalls, kernel to user process

4 MEMORY MANAGEMENT

4.1 terminology

physical address
address as seen by memory unit

virtual / logical address
address issued by the processor

4.2 memory management

allocating physical addresses to applications
managing the translating virtual to physical addresses (MMU)
performing access control (MMU)

4.3 base & limit registers

defined logical address space
base higher up (but lower number), limit down below (but higher number)

4.3.1 address binding variants

base address is not known till runtime → compiled code must be position-independent
relocation register maps compiled to physical addresses (in MMU)

4.3.2 contiguos allocation

main memory in two partitions
OS in low memory with interrupt vector, User Process in higher memory

relocation register protects each other & changing OS data

4.3.3 procedure

CPU with logical address asks
Limit Register (which contains max logical address); if smaller than max got to
Relocation register (which adds the base register value) then do the memory access

4.3.4 bad

memory fragmentation, sharing complicated, total logical space ≤ physical memory, how to share code / load code dynamically?

4.4 segmentation

generaliation of base & limit
physical memory divided into segments

logical address= (segement Id, offset)

implemented with segment table, which maps identifier to base & holds limit for this specific base

segment table
entries with (base; starting physical address, limit; length of segment)

Segment Table Base Regsiter STBR
current sgement table location in memory

Segment Table Length Register STLR
current size of segment table (segement number s legal if s < STLR)

fast context switch
load STBR, STLR

fast translations
2 loads, 2 compares, caching possible

segments easely shareable

but
physical layout must still be continuous → external fragmentation still a problem

4.5 paging

solves contiguous memory problem
physical memory divided into frames
logical memory divided into pages

4.5.1 for programm of n size

find n frames & load program
setup page table to translate logical to physical frames

4.5.2 page table

maps VPN (Virtual Page Number) to PFN (Physical Frame Number)
split VPN, each part defines entry in a page table → put them togther to get full PFN
can combine page tables → each lookup produces the STBR of the next page table, you can cache page tables now

4.5.2.1 performance problem

use TLB, split VPN in TLBT (16, tag) + TLBI (4) (index) → only if not there use page tables

4.5.2.2 encoding of one PTE

page phyiscal base address (20), avail (3),

bits
G global page (don't evict form tlb on switch), D dirty (MMU on write), A accessed (MMU on r/w)
CD cache enabled, WT write-through or write-back policy, U/S user/supervisor, R/W read/write, P page is present in physical

4.5.3 x86

combines segmentation & paging: cpu → segementation → paging → physical

4.5.4 effective access time

given e=associative lookup, a=hit ratio: (1+e)a + (2+e)(1-a) for single level

4.6 page protection

page table entry has valid bit (P)
requesting an invalid page causes page fault → then can get additional info as executable, on-demand paging

4.7 page sharing

shared code
read-only, all processes, same location of VPN for all processes (often...) → data segment not shared, but still same VPN so code can find it

private code
can be everywhere

4.8 per process protection

protection bits are stored in page table
each process can have different bits set

4.9 page table structures

linear table is too slow, so use hierarchical, virtual, hashed or inverted

hashed
VPN is hashed, lookup → fast but unpredictable, often used in software TLB

inverted
system-wide table maps PFN → VPN, one entry for each real page, use hash table for efficient lookup → bounds size of table

most OS keep own structure
portability, tracking, software virtual → physical & physical → virtual

4.10 TLB shootdown

multiple TLB → but must be coherent, else security issues

4.10.1 keep consistent

hardware TLB coherence
integrate TLB with cache coherence, invalidate if PTE memory changes

virtal caches
required cache flush will take care of this, but expensive!

software TLB shootdown
OS notifies other cores with IPI (most common)

hardware shootdown instructions
special messages for this

4.11 address translation

process isolation
IPC
shared code segements
program initialization
efficient dynamic memory allocation
cache mangement
program debugging
efficient IO
memory mapped files
virtual memory
checkpoint & restart
persistent data structures
process migration
information flow control
distributed shared memory

4.12 COW

copy on write

problem
fork is expensive, can call vfork() with shared address space but this is dangerous

solution
copy only when something is being written → child & parent share pages, modified pages are copied (which are allocated from a pool of zeroed pages)

how
mark all pages as read only, if one process writes → page fault. now copy

page & map into resp. process, restart operation from before

4.13 demand paging

read in page into memory only when it is needed
can now cache for processes for disk

lazy swapper
swap page into memory only when it is needed → called pager

strict demand paging
only page in when referenced

performance
1 out of 1000 → 80x slow down!

4.14 page fault

first reference to page will cause page fault → OS kicks in

procedure page fault
processor sends VPN to MMU
MMU can't find it, asks cache/main memory
main memory delivers PTE or not
page fault occurrs, OS kicks in
OS chooses frame to use
OS loads page into this frame
OS create PTE, sets valid bit
instruction restarted

4.15 page replacement

find little used page to write to disk

4.15.1 choose page

will not be used anymore, is clean (no write needed)

4.15.2 FIFO

first-in first -out

4.15.3 Belady's Anomaly

more frames → more page faults (mabye; bigger cache does not always mean less page fault)

4.15.4 optimal algorythms

replace page that will not used for the longest time; is used as a measurment for real algos

4.15.5 LRU

4.15.5.1 Least-Recently-Used

save clock of last access to pages, replace the oldest one

4.15.5.2 implementaion with stack, most recent one on top

4.15.5.3 no Belady's Anomaly

4.15.5.4 can be implemented with reference bit "second chance algorythm"

initially 0, set to 1 if used

if page needs to be replaced
if clock-order page has bit=1, set bit=0 and move on. if page has 0, then replace it

4.16 allocation of pages

minimum amount of pages needed to function (for example move instruction needs 6)

4.16.1 fixed allocation

equal
all processeds get same number

proportional
number according to size of process

4.16.2 priority allocation

use priorities rather than size → on page fault select one of own pages or grab one from process with lower priority

4.16.3 global vs local replacement

select from all available frames vs only from own

4.17 thrashing

if process does not have enough pages page fault rate very high

issues
low CPU ussage, OS thinks it needs to increase degreee of multiprogramming (because of low CPU ussage) → another process added

occurrs when
size of locality > total memory size

working set model
working set is page references used by process in the last k instructions
choose working set window W (carefully, if too big will encompass several localities, if too small not entire)
count accessed frames in time window, and sum them for all processes → if too large then suspend some

page fault frequency scheme
choose acceptable page fault rate
if too high, process gains frame, else loses

5 FILE SYSTEM

5.1 general

abstraction from blocks (disks) to files (programmer)

5.1.1 goals

high performance
high cost of IO → organize placement, access data in large, use caching

named data
large capacity, persistent, shared → support files & directories

controlled sharing
device stores multiple users's data → access control metadata

reliability
crash may occurr at any time → transactions to make updates atomic

durability
storage devices may fail, flash memory wears out → redundandancy & wear-leveling to prolong live

5.1.2 architecture

application
on top

library
copy / cut etc

file system
NTFS, FAT

block cache
served in blocks of data

block device interface
communication with driver

device driver
connects OS & device

IO, DMA, Interrups
used to control physical device

5.2 file system interface

5.2.1 file is

size
named

metadata
data about object, not object itself
size, location, name, last access, last written, creation date, ownership, type, file structure, descriptive data (for search)

5.3 naming

provides

5.3.1 Indirection

All problems in computer science can be solved by another layer of indirection → rename to understand what it is

5.3.2 Binding

association between name & value

5.3.3 model of naming

naming scheme
what are valid names (namespace), what are valid values (universe of values), name mapping algorythm (resolver, mapper)

context for resolver required

5.3.4 example

Virtual Address Space

Name Space
virtual memory addresses

Universe of values
physical memory addresses

mapping algorythm
translation with page table

context
page table root

5.3.5 operations

ln, rm, ls, compare (define if for name or data or both)

5.3.6 naming policy alternatives

injective if only one value (if key is unique identifier)

stable bindings
binding cannot be changed (primary key)

5.3.7 lookup types

5.3.7.1 table lookup

ethernet interface, memory cells, processor registers

5.3.7.1.1 context

default (implicit)
supplied by resolver → constant/builtin (DNS) or from current environment (working directory)

explicit per object
supplied by object (specify DNS server)

explicit per name
each name comes with context (me@famoser.ch), first resolve context, then name

5.3.7.2 recursive lookup

path names, emails
recursion must terminate
syntax gives glue

soft links
names resolve to other names in same scheme

5.3.7.3 multiple lookups

search paths
try several contexts in orders

5.3.8 name discovery

well-known, braodcast, query, introduction

5.3.9 name model is good servant but bad master

5.4 file system

5.4.1 directory operations

link, unlink, rename, ls

5.4.2 acyclic graph

on delete dangling pointer problem
use backpointers & clean up, reference counting

5.4.3 guarantee no cycles options

allow only links to files (no directories)
garbage collection (cycle collector)
check for cycles with each new link
restrict directory links to parent

5.5 access control

file owner can control what / who changes

5.5.1 type of access

read, write, execute, remove, append, list

5.5.2 Access Control List ACL

definies who can access what

row-wise
for each file & right list the principals (for each right list the principals (easy to add rights, scalable with files, but bad with alot of principals)

colum-wise
for each principal & right list the files (scalable with principals but hard to change rights (hard to revocate))

5.5.3 POSIX (unix)

ACL row-wise simplified, 3 principals (Owner, Group, Everyone) with 3 rights (Read, Write, Execute)

5.5.4 Windows

full ACL SOOO POWARFUL

5.6 file types

directory is file too; but cannot be accessed by user (corrupt data structures, bypass security)

5.6.1 implementations

linear list
(file name, block pointer) → lookup slow

hash table
linear list with closed hashing → fast name lookup, collisions, fixed size

B-tree
name index, leaves are block pointers → complex but scalable

5.6.2 executable files

recognised by most OS, magic number first two bytes, or #!, windows locks currently executed file

5.6.3 symbolic links

5.6.4 unix

uses sockets, OI devices, pipes, process control, OS configuration, etc too as file

5.7 open file

byte sequence
file is a vector of bytes, can be edited → sequential & random access (read write seek tell)

record sequence
file is a vector of fixed size records, can be edited → random access at record level

key-based, tree-structured
database feature, now libraries

memory mapped files
cache file in main memory, but use same file operations as usual

5.8 on disk data structures

treat disk as compact linear array (in reality has sectors, tracks, spindels, ...)

5.8.1 implemention aspects

index structure
locate data on disk

index granularity
unit of allocation for files

free space management
to allocate more sectors on disk

locality heuristics
make it fast in common case

5.8.2 implementations

5.8.2.1 FAT

very old, no access control, little metadata, limited volume size, no hard links
linked list / block / FAT array / defragmentation
FAT table which specifies for each block the next block. each entry in FAT corresponds to data block on disk
needs lookup for filename to first FAT table entry (provided by the directory entry)

free space found by traversing FAT table linearly

implications
slow random access, lose FAT and goodbye, files can end up fragmented on disk

5.8.2.2 FFS

Unix Fast File System
fixed, asymetric tree / block / fixed bitmap / block groups, reserve space
Inode which specifies metadata (file mode, owner, timestamps, size, other), 12 direct block pointers (4kb), single indirect (to block with 4k/8=512 pointers), double indirect, triple
very small files placed directly in inode
directory entry maps filename to inode pointer
free space found by simple bitmap (1 bit for each block of memory, intialized with filesystem creation)

block groups for optimization
keeping together files, metadata, directories, free space map
use first-fit allocation to improve locality
layout & free space bitmap defined in superblocks

5.8.2.3 NTFS

New Technology File System
dynamic tree / extend / bitmap in file / best fit, defragmentation

Master File Table MFT with entry for each file
standard info, metadata, free (fixed 1kb)

very small files directly inside MFT entry, hard links too

MFT entry holds list of extends (start, length)

fixed entries with info about system (file 0 - 11 reserved); example
free space map, volume infos, master file table (fixed at first sector of volume)

5.8.2.4 ZFS

dynamic COW tree / block / log-structures space map / write anywhere, block groups

5.9 in memory data structures

opening file
directory structure translated into kernel data structure on demand

read & write
per process open file table, cache of inodes at system wide table

efficieny
disk allocation, directory algorythms, type of data kept in file's directory entry

performance
disk cache (for frequently used files), free-behind, read-ahead,

page cache
caching whole pages

architecture
file system → buffer cache → a) page cache → memory mapped cache or b) file access with read() write()

5.10 concurrency

provide mechanisms for users to not contradict themselves with advisory(users do it) & mandatory(OS does it) locks (granularity: while-file, byte-ranges, write protect executing binaries)

file system integrity must be ensured
careful design, locking, order or writes to provide transactions

5.11 recovery

backup / revert

consistency checking
compares entries in directory structure with actual data blocks on disk

5.12 disk partitions

partition table specifies dimensions of file systems
can have one filesystem over multiple disks

multiple filesystems
A/B/C in windows, mounted in linux

5.13 virtual file system

use same APi for multiple types of file systems

5.14 IO hardware stuff

device: hardware, has location in bus, set of registers, source of interrupts, may uses DMA

registers
can be read out by OS to get infos about device, documentation sometimes baaad

driver
reads our registers & acts accordingly

programmed IO
all data must pass trough CPU registers, is explicity read & written by CPU

polling
spinloop waiting for change

interrupts
triggered by IO device, goes to CPU, handler called (which can ignore or mask some interrupts with interrupt vector), exceptions

IO cycle
process A starts IO operation, driver initiates action, device completes IO & interrupts, scheduler continues execution of A

DMA
has direct memory access, avoids programmed IO, only one interrupt for whole process

IO protection
DMA needs to be checked (with IOMMU), IO performed with syscalls because instructions are priviledged

5.15 IOMMU

like the MMU for CPU
translates Device Virtal Adresses (DVA) to physical ones, IOTLB, guarantees security for VM (better performance than with software)
has page table per IO device, identifies device by Bus ID, Device ID & Function ID
page tables similar to multi-page tables, has bits for r/w etc
interrupt remapping

5.16 device drivers

software object with abstract device
between hardware & OS
understands registers, interrupts, DMA

5.16.1 structure

hardware is interrupt driven
system must respond to events

application is always blocking
must wait for specific event to occurr

considerable processing in between
TCP/IP processing, retries, file access, locking

5.16.2 three-layer model

interrupt handler
driver thread
user process
does mediation between interrupt-driven hardware and blocking user processes.

5.16.3 architecture

user thread does syscall
IO subsystem sends request to driver & blocks user process
device driver issues commands to device, blocks
device issues interrupt when completed
interrupt handler handles interrupt, signals device driver
device driver processes, determines source of request
IO subsystem copies data to user space, returns completion code
user thread continues

5.16.4 solution using drivers (FLIH)

interrupt handler
masks interrupts, does minimal processing, unblocks driver thread (Linux calls this the "Top-Half", in contrast to good OS's)

driver thread

performs package processing, unblocks user process, unmaks interrupt

user process
per-process handling (different all the time), copies packet to user space, returns from kernel

5.16.5 solution using deferred procedure calls DPC (SLIH)

interrupt handler enqueues DPC (DPC called 2nd level/soft/slow interrupt handler, or bottom-half in linx (retard alert))
once user process continues,
executes all pending DPC on next process being dispatched (while still in kernel) → save context switch, but execution time can't be accounted to correct process

5.16.6 Bottom-Half

FLIH + SLIH (first + second level interrupt handler)

5.16.7 Top-Half

called from user space

5.16.8 in short

move data from & to IO devices, abstract hardware, manage async

5.16.9 example (UDP packet receive)

User process blocks (system call), waits
NIC transfers packet to kernel memory via DMA, usually using a ring buffer
NIC sends interrupt to OS → Mask interrupts, checkes mac & IP, unblocks driver thread
driver thread → packet processing, demultiplex packets based on ip & port, unblocks processes, unmasks interrups
data is copied from kernel memory to user memory
blocking syscall returns

5.17 IO subsystem

caching
fast memory holding copy, key to performance

spooling
hold output for device (if device can serve only one at a time)

scheduling
request ordering, trying fairness

buffering
store data in memory when transfering device / memory: different speed, transfer size mismatch

discovery of devices
hotplug, unplug events, discovery

match driver to device
device has model identifiers; OS calls each driver and asks if it can handle it

naming device driver instances
creates identifier for bock & caracter devices, using class of device (major number) & more specifics (minor number)

block devices
structured IO, transfers whole blocks, look like files, use shared buffer cache, filesystem uses block devices →harddisks

caracter devices
unstructured IO, bytestreams, single caracter / bytes, buffering with libraries → keyboards etc

naming outside kernel
put into /dev, indode (type, major num, minor num)

pseudo devices
devices with no hardware, examples include dev/null, dev/random

old unix
all drivers in kernel compiled, driver probes for supported devices, sysadmin populates /dev

new linux
devices inside fake filesys /sys, user deamon populates /dev with infos, drivers loaded dynamically at boottime

5.18 network IO

software routing
rotuing protocols in user space (easier to change, non-cirtical), forwarding information in kernel (needs to be fast, part of protocol stack)

5.19 network stack

5.19.1 NIC

network interface card

5.19.2 receive

interrupt
allocate buffer, enqueue packet, post s/w interrupt

s/w interrupt
high priority, any process context, defragmentation, tcp processing, enqueue on socket

application
copy buffer to user space, application context

5.19.3 send

application
copy from user to buffer, call TCP code & process, enqueue on socket

s/w interrupt
any process context, remaining tcp processing, IP processing, enqueue on i/f queue

interrupt
send packet, free buffer

5.19.4 TCP

needs to handle additionaly
congestion control state
flow control window
retransmission timeouts

state transmission triggers
timer expires, packet arrives, user request

actions
enqueue packet on transmit or socket, configure a timer, manage tcp control block

5.19.5 protocol graph

how to handle protocol in OS

per-connection
initiated dynamically

per-protocol
handle all flows, works with demux tags in packets

5.19.6 memory management

structure which can add/remove headers, avoids copying, fragment dataset into smaller units, combine half-defined packets → use linked list of buffer structures

buffer strucure (sk_buffs)
next, offset, length, type, data (112 bytes), next object

5.19.7 implement own protocol

register receive hook
fill packet_type struct with .type, .func (receive function) → get hook which is called on every arriving packet

interact with applications
implement handlers for connect(),... register protocol family

process SKB fields

5.19.8 preformance issues

1GB → 700'000 ether net packes → process packet in under 3000 cycles for 2Ghz processor (forget it)

5.19.9 performance fixes

TCP offload (TOE)
tcp processing on card

buffering
transfer lots of packets in single transaction

interrupt coalescing
dont interrupt at each packet / don't interrupt if load is high

receive-side scaling
parallelize: direct interrupts & data to different cores

5.19.10 NAPI (Linux New API)

can change if interrupts happen or CPU polling

5.19.11 producer-consumer descriptor rings

two pointers; one consumer, one producer; in same direction; behind consumer blanc

state maschine
running, running (host blocked), idle; dont forget that "nearly full" stays inside blocked state

descriptor format
phsical address, size, flags

used by most aplications (USB, SATA)

DMA used twice (one to write actual data, one to write descriptor ring)

variations with complex descriptors possible, as descriptors only define ower of data (can be subsets, out-of-order)

5.19.12 receive side scaling

handle multiple flows on multiple cores (one ring buffer per core)
→ get flow ID with packet header → assign correct ring buffer

6 Virtual Maschine Monitors

6.1 what

virtualizes an entire hardware maschine, therefore full OS required on top

6.2 why

server consolidation
each maschine is mostly ide

performance isolation
so one application does not starve another

backwards compatability
so old programs can still be executed

cloud computing (selling cycles)
decouple alocation of resources (VM's) from provisioning (physical maschine, power)

OS development
build & testing new OS

more
Tracing, Debugging, Live-Migration, rollback

some control under the OS

6.3 hypervisor

OS with scheduling, multiplexing, virtual memory, device drivers,...
but provides illusion of hardware

6.4 virtual maschine monitor

we don't distinguish to hypervisor

hosted
upon host operating system, like VMware, Hyper-V

hypervisor-based
on real hardware, like Xen, VMware ESX

6.5 how to virtualize

6.5.1 CPU

6.5.1.1 strictly virtualizable

if all non-privileged instruction execute natively → privileged are a trap caught by VMM which emulates it → x86 is not!

6.5.1.2 non-virtalizable

x86, example: pushf, popf from code register, which includes interrupt flags → but this is info the VMM needs!

6.5.1.3 solutions to non-virtual

full virtualization
emulate all kernel-mode code in software → very slow for IO

hardware assisted emulation
type of full virtualization where the hardware can create virtual devices

paravirtualization
guest OS has replaced evil instructions by explicit trap instruction when building its kernel, it realizes its being emulated → used by Xen

binary rewriting
protect kernel instruction pages; on first read trap to VMM, replace evil instructions → used in VMware

hardware support
extra processor mode causes it to trap

6.5.2 MMU

VMM can't let guest install mappings

6.5.2.1 MA

maschine address: real physical

6.5.2.2 PA

Logical/Physical address

6.5.2.3 VA

Virtual Address

6.5.2.4 virtualizing memory (dump approach)

each guest OS has VA → Guest OS Address mapping
VMM does Guest OS mapping → MA
→ extremely costly

6.5.2.5 direct "writable" pagetables

require paravirtualization;
guest OS creates those;
host OS validate all updates, batch updates to avoid trap overhead

6.5.2.6 shadow pagetables

guest OS has page tables in memory (which are not used by hardware, contains VA → PA)
VMM has shadow pagetabes for each guest OS (which tracks PA → MA)
VMM manages real pagetables (which are used by hardware)
on read of new entry or on write of an entry of guest page tables the VMM traps and updates shadow page tables & responds with correct MA

6.5.2.7 hardware-assisted paging

hardware knows hypervisors PA → MA & guests VA → PA, there is a new kind of TLB to reflect this knowledge

6.5.2.8 Ballooning

reclaim memory from guest system; modified driver inside guest system which communicated with hypervisor; and claims RAM if necessary, or gives free

6.5.2.9 Virtualizing devices

trap-and-emulate, interrups to upcalls conversion, copy data into guest private address space to emulate DMA

6.5.2.10 Paravirtualizing devices

faster than virtualizing, communicate with VMM via hypercalls

6.5.2.11 Networking

virtual network device, entire virtual IP/Ethernet network

6.5.2.12 real drivers

in hypervisor
need to rewrite device drivers VMware ESX

in the console OS
export virtual devices to other VM

driver domains
device passhthrough; run trusted OS only for that task, use IOMMU

self-virtualizing devices
PCI devices can add copies of itselves, virtual copies are passed directory to guest OS

6.6 reliable storage

6.6.1 reliability

continue to store data & be able to read & write it

6.6.2 available

responds to requests

6.6.3 thing that go wrong

operating interruption (crash/interruption)
transactions

loss of data (media faillures)
redundancy

6.7 sector & page faillures

disk keep working, but sector / page broken

error correcting codes
internally in drive
encode data with redundancy to recover

remapping
externally in the OS / internally too
identify bad sectors & avoid using them

caveats
significant for nonrecoverable
not constant (age, workload)
not independet (time & space correleation)
not uniform (different model, different behaviours)

6.8 device faillure

just stops working
faillure more explicit

6.8.1 MTTF

mean time to faillure

6.8.2 Annual Faillure Rate

1/MTTF

6.8.3 caveat

advertised faillures can be misleading

faillures correleated
same rack, production

not constant faillure rates

6.8.4 bathtub curve

children immortality → advertised → disk wears out

6.9 approaches

6.9.1 RAID 1

simple mirroring (2 disks)
write go to both disks
reads from either disk

6.9.2 parity disk

4 real disks, one parity disk,
faillures always discovered
writes to block; then parity must be updated → two writes necessary

RAID 5
distribute parity so parity disk is not accessed 5times as often, in strips for sequential access efficiency

6.9.3 atomic updates → what if system crashes?

use non-volatile write buffer
transactional update to blocks

recovery scan
remap bad sectors& reconstruct content from stripes / parity, replace disk & reconstruct data

do nothing

6.9.4 RAID 5 data loss

1
two full disk faillures

2
one disk faillure, sector faillure on another disk

3
overlapping sector faillures on two disks

MTTR
Meat Time To Repair

MTTDL
Mean Time To Data Loss (till 1,2,3 happens)

solutions
more redundant disks
scrubbing (read entire disks for sectore failures)
more quality disks (disks with less error rate)
hot spaces (disks already in rack, which can be used after disk failures to reduce repair time)

7 HARDWARE TRENDS

7.1 ausblick

more cores

7.1.1 NUMA

non uniform memory access; some memory closer to some core

numa heuristics
allocate memory in node of processor, scheduling, replicate hot OS structures,

7.1.2 OS for high performance computing

basically only hypervisor

7.2 abstractions / mechanisms

7.2.1 IPC / communication

A
sockets, channels, read/write

M
network devices, packets, protocols

7.2.2 memory protection

A
access control

M
paging, protection rings, MMU

7.2.3 paging/segmentation

A
infinite memory, performance

M
Caching, TLB, replacement Algos, tables

7.2.4 naming

A
hierachical name spaces

M
DNS, name lookup, directories

7.2.5 file system

A
files, directories, links

M
block allocation, inodes, tables

7.2.6 IO

A
device services (music, pictures)

M
registers, PIO, interrupts, DMA

7.2.7 reliability

A
reliable hardware

M
checksum, transaction, RAID 1/5

7.2.8 virtualization

A
virtualized x86 CPU; memory

M
paravirtualization, rewriting, hardware extensions; shadow pages, writable pages, IOMMU