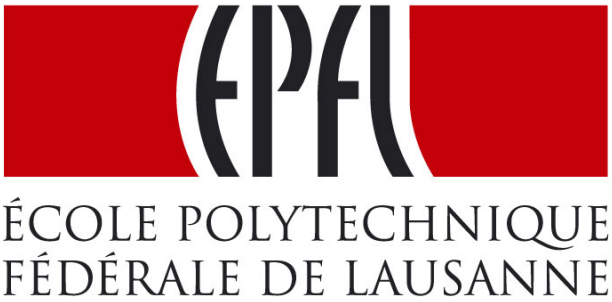
Ecole Polytechnique Fédérale de Lausanne

Spring 2017

Operating Systems



Olivier Cloux

Spring 2017

Contents

1 Intro

The very first pieces of OS were libraries to to I/O. **The OS makes hardware easier to use, using abstraction**

Example : let's write some Photoshop application. It's much easier to deal with a file that deals with photos, instead of data location. Similarly, for a web server, easier to deal with packets rather that with Network Interface Card

Our OS allocates hardware resources between programs, to allow multiple users/programs at the same time. It will share memory, disk space, CPU time,... In other words, it does the abstraction (hardware easier to sue) and resource management at the same time. Web browser does only abstraction (so not part of OS), so does graphic library. But device driver and printer server do both (so they are part of the OS)

The CPU can be in either kernel or user mode (separated by a bit). User mode can do really few things. The Kernel mode can use privileged instructions (set mode bit,...), has direct access to all of memory and devices.

This separation allows OS to protect itself and manage applications/devices (as the OS runs in Kernel mode). Going from kernel to user mode is easy : OS sets the mode bit itself. But from user to kernel, it's harder : a device generates an interrupt, or a program executes a trap or a system call.

Basically, the system call is the only way to go from user to kernel mode. Some example system calls are process management, memory management,... System

call implementation is architecture-specific. Each system call has a unique number. To perform a system call, put a system call number in $\%eax$. The parameters (architecture specitif), we put in designated registers, or on the stack. For Linux/x86, we put call number in $\%eax$ register, parameters in other registers. If more parameters, use registers as pointers.

We always call OS but never system call. We use Kernel API. We are given a number of functions, that wrap the whole operation. There is an additional layer, libc. This library calls itself the kernel API, that then calls system call interface.

Important to note : libc makes system call look like a function call, but not ! It's a user-kernel transition, and it's much expensive.

1.1 Traps

Trap is generated by CPU as result of an error (divide by zero, execute sudo as user,...). Works like an involuntary system call. Identified by a trap number. Interrupts are generated by a device that needs attention(packet from network, disk I/O completed,..). Identified by an interrupt number.

1.2 Control Flow

On a system call i, the hardware puts the machien in kernel mode, sets the $PC = SystemCallVector[i]$. SystemCallVector is a predefined location. Same for trap, with predefined location TrapVector (and InterruptVector)

2 Week 02 : Process management

Linux is composed of essentially processes. A process is a program in execution. In itself it is a piece of executable code. When it starts executing it becomes a process, and is represented in memory.

A process can do about anything. A process can be a shell, compiler, editor, browser,... It is always identified by a **pid**. We can create a process, or terminate by by normal exit, an error or terminated by another process.

Linux process are primitives, and we can do most by understanding 4 essential : fork, exec, exit and wait. All of these are system calls, so the OS is aware of the waiting and so.

FOrk : when executing pid = fork(), we create an identical copy of parent. In parent, returns pid of child, whild in child we return 0.

Exec takes a filename in argument, and loads executable from the file. The point is that it blows up anything that was being done at the moment, and executes the file. Everything else (in the process) is lost. There is no such thing as a "return" from the exec.

Wait only waits for one of its children to terminate.

Exit is the way to kill itself : terminate the process.

Form the OS point of view, a process either does computation (CPU usage) or I/O. With our current model (single process system), we have 2 issues : the first one is that it is highly inefficient. When a process requires I/O, the CPU is idle, and lot of computation time is lost. So we prefer multiprocess systems. Only one uses the CPU, and when one does I/O it lets the CPU idle.

2.1 Process switch and scheduling

In multiprocess systems, we need the CPU to switch process, as a process does not require CPU time anymore. When going in a I/O mode, we give CPU time to another process. This is a switch. The implementation requires registers, stack or more. It is an expensive operation, as it requires saving and restoring lots of stuff. Has to be implemented very efficiently, and used with care.

Furthermore, at a certain point we may have many processes ready at once. The scheduler must pick one. It will maintain a swt of queues, and remember running processes. We also implement a timer, that will kill a too long process. But now we need to be smart in the scheduling. So we have a scheduling algorithm. It ain't clear what makes a good scheduling algorithm.

3 Week 3

3.1 Recap of week 2

Linux processes are managed by fork, exec, wait and exit. They are used in the shell below init. Multiprocessing has the advantage of loweing response time when I/O (not CPU idle during long I/O). That is done using process switch (change process using the CPU). Save and restore registers and other info. Finally, we looked at scheduler, that decides which process to run next. We need to be careful with scheduler, as it may take a lot of time : running it for 1msec every 10msec, we lose 10% of the machine.

3.2 Application Multiprocess structuting and interprocess communication

So far for us, one program = one process. Shell, compiler... are mono-process. But in real world, many programs consume multiple processes. For example, a web server takes multiple processes ! Basically, it will (forever) wait for an incoming request, read file from disk and send response. TheWe have the problem that if we have two successive request, one is filled at a time. And until the request is done, the disk and CPU will both have idle time ! We could use this for the second request. That means we only treat one request at a time, but the treating takes very few ressources...

Better idea : Wait for incoming request, and then create a new "worker" that will do the task. That worker will do the real work. The two will have a better interpolation. But what work is needed ? We need to receie the network packet, run the listener, create the worker, read the requested file and send a response (network packet). Except the disk reading (because independent of CPU), the heaviest part is the creation of worker. We can dodge the problem by crating in advance a pool of workers (processes) during initialization.

3.2.1 Interprocess communication

We look again at our web server. We might need interprocess communication when receiving or sending packet. This is a form of communication between client's and server's processes. But also when we send the request to the worker, and the receiving of it. Two processes are communicating. The communication happens between memory allocation. Allocate space for the message, fill it with your message, send the address. Receiver will allocate memory for the message. We send the message **by value** (not by address, one can't access the others' memory). Kernel does the sending. Actually, we have 2 messages : one from the kernel giving the size of the incoming message, so the receiver can allocate enough space.

The implementation is through a queue. the proctable has a queue of incoming message for each process. Sending puts message at the end of the queue, and receiver will read the head of the proctable.

The addressing is either symmetric (seldom) or asymmetric. Symmetric will have a receive method from a particular pid, when asymmetric is ready to receive message from any process.

Send is either blocking or non-blocking (common). The blocking way : sender blocks until message is delivered. non-blocking will returns as soon as

message is sent. Similar way for sending.

Our version of client-server looks like a function call : call something, wait for an answer. Callee is invoked, executes and returns. This is called **Remote Procedure Call** (RPC)

explain stub

We know our call message contains argument. It also needs to include which procedure is called.

3.2.2 Application multithreading and synchronization

In our worker example, we still have a performance problem : the disk access is really expensive. We would prefer to first check the presence of the file in a cache, and if not read from disk and put in cache. That creates a huge problem : synchronization. At a request, we read file from disk and put in cache (actually it's in the heap). But if after finished another worker wants the same file, it won't find the file ! It's in the cache (heap) of worker 1. So we will have 2 cached copies, one in each worker. So basically, the root of the problem is that the workers don't share memory, so the effectiveness of cache is much reduced. This is called **multithreading**. It's a thread like a process, but does not have its own heap and globals (but its own PC, registers and stack). It shares globals and heap with other threads in same process.

The main difference between processes and thead is separation : processes provide memory separation, and are suitable for coarse-grain interactions, when threads share memory and data, so suitable for tighter integration.

Having shared data has a pros and cons. In its cons, it can lead to data race. **Data race** is an unexpected/unwanted access to shared data : two threads try to access/modify one value at the same time, so their access will interleave and cause strange results.

We have troubles with globals only. So we define *criticals*, that assure some part of the code will be atomic. It avoids data races.

provide better explanation

Notes on Pthreads ?

4 Week 4

4.1 Recap week 3

Multiprocess strcturing : one application is structured as multiple processes (we server). We want to overlap computation with I/O to have good response time. Classical example of web server : one listener and many workers. Careful

: processes have no memory sharing, so no pointer. Only passage by value. Processes communicate through remote procedure call.

4.2 Multithreading vs Multiprocessing

Threads lie in a process, and may share code, globals and heap. Their stack, registers and are not shared. That's the main difference : processes don't share memory when threads do. ??

Problem : if a process crashes, only it crashed (other not affected). But if a thread crashes, the entire process and other threads crash too.

Coming back to the Web server : serving static content can be considered bug-free so done in a multithreaded process. On the other hand, serving dynamic content (third-party) might be buggy so we want to keep those in different processes.

Sharing data can be dangerous : it's a great advantage but threads might suffer **data races**. We have interleaving of threads while they access a shared memory location ; e.g. one thread tries to read an index and increment it, while the other one tries to decrement it.

- Divide work among multiple threads
- Identify shared data complete

By putting shared data in critical section, no other threads can access it at the time, so we don't have any problem.

5 Week 05

5.1 Recap week 04

Multithreading : one process with many threads, threads share a bit (memory...). Processes don't share anything. To derive multithread : divide work, locate shared data, synchronize with big lock, optimize with fine-grain locks, maybe introduce shared data. Consider using Pthreads.

5.2 Memory management : VIrtual memory

As we know, the memory is hierarchized : processor, L1 cache, L2 cache, L3, Main memory, Disk (from smallest to biggest). For our class, the caches are invisible, so we only consider the processor, main memory and the disk. Just

for this week, we don't consider the disk (program is either fully in memory or not), so only consider interactions between processor and memory.

The OS has to allocate the memory, protect it and be transparent. When allocating the main memory, we need to consider : where . Because of its importance, kernel is almost always in low memory (first Kbytes). To provide good security, we need to check the CPU's accesses to memory (in hardware, or too slow). If the process asking for a certain address is allowed to, it will access. But it must not be allowed to access anything.

complete

We also want to provide transparency : programmer should not worry about where he's working, or where are other programs. For that we create the MMU (memory management unit), to provide at the same time security (check of bounds) and translation from virtual to physical. All programs believe they are located at 0 in memory, and know their limits. MMU translates to correct physical address.

Our virtual addresses are limited by the address size of CPU : typically 32 or 64 bits. The physical address is limited by the size of memory. There are various schemes for the MMU :

- Bounds : literally translate 0-¿bound, max-¿bound+max. Almost not used, only in high end supercomputer (want to keep translation simple)
- Segmentation : virtual and physical are shred in segments, each segment is translated using simple bounds.Segments can be whatever you want it ot be : part of a program, the main, a subroutine, the stack,...
- Paging (simplified) : we have pages (portion in virtual) and frames (portion in physical). Their size are identical, of usually 2k-8k (always power of 2). Virtual pages are linear, from 0 to a multiple of page size. Physical frames a non-continuous set of frames. We have one frame per page.
- Segmentation with paging : think of it as segmentation, but evey segment is paged.One segment table per process, one page table per valid segment.

Page tables are in the kernel's memory. We still have a problem : we know the format of stack and heap. But if we try to access to some unallocated address in-between, no error will emerge. To solve that we add a valid bit in each page table entry, set to valid for used portions of address space.

We saw that kernel goes in low memory. How to find space (and allocate it) for processes ? We need to keep in memory the occupied spaces and the "holes". And how to find the best one ?

- First fit : find first big enough

- Best fit : find the smallest of sufficiently big holes.
- Worst-fit : take always the biggest hole.

6 Week 06

6.1 Recap week 05

Machine is separated between virtual and physical address space. Virtual = what programmer think is its memory, physical = where the program actually is. MMU does the mapping from virtual to physical. There are various way to do the mapping : "base and bounds", "segmentation", "paging", "segmentation with paging". Each possesses a particular address format.

We also consider the allocation of main memory. First, the kernel is placed in low memory. We need to consider where to put processes. The complexity depends on the physical address space. We want to put as many processes in memory as possible, to enable fast switches. Considering the properties, main memory allocation is easiest done with paging (with or without segmentation).

6.2 Complement to week 05, not yet seen

We may run out of memory at some point. We then need to get rid of one or more processes, and store them temporarily on disk. This is swapping (either in swapfile or dedicated partition). But swapping is not ideal. Latency can be very high !

We want also a fine-grain protection, so different protection for each part of memory. We can add a valid bit in page table, and more for

We earlier said that processes never share memory. That's not quite true : actually they occasionnaly share memory. With base&bounds, it's not possible. But with segmentation we create a shared segment and put an entry if segment table of both processes that point to shared segment. Similarly, with paging, we put entries in page table of both processes that point to shared pages. Sharing is easier with segmentation.

In modern systems, segmentation was abandoned (too complex for not enough gain). While Base&Bounds is reserved for particular niches (supercomputer), paging is now universal in most systems.

We have a serious problem with address translation. As page table is in memory, each virtual address requires 2 physical memory accesses. It would reduce performance by factor of 2 ! A solution to that is to use a TLB (trans-

lation lookaside buffer). TLB is a small fast cache of maps from page-number to frame-number. As a cache, we first look in the TLB for the frame page number. If found, extract frame number. If not, look in page table and cache result in TLB.

6.3 Virtual Memory : Implications and demand paging

Modern machines are addressed using 64 bits. If each page is 4KiB (12 bits), we still have 52 bits for addressing. We would require 2^{52} page entries. If each is 4 bytes, we have 2^{54} bytes in main memory. Too huge !

To counter that, we use 2-level page tables. First part of address yields entry in first level, that yields a second page table. Second part of address yields entry in second page, that is a real page. Finally we use the offset directly.

complete explanation

6.4 Review of process switching

We have a problem with process switches : if we do nothing, the virtual access of second process may access physical memory of first one ! We could either flush the TLB at each process switch (very inefficient), or add a field in the TLB

complete first slides

7 Week 07

7.1 Recap week 06

Multilevel is great for sparse address space.

7.2 Demand paging

We now don't suppose anymore that a program can fit entirely in the memory. Mainly because we offer 64 bits address to the program (16 exabytes). We don't have that much memory. We need to use disk too. Also great because we might not need all of the program (e.g. error handling), so load only minimal pages in memory and start faster.

The program is entirely on disk, and part of it are copied in the memory. This is slightly different than swapping. We can't access pages that are on disk, as CPU can't directly read the disk, only memory. The usual workflow is : program runs ; if needs page that is not in memory, stop program ; OS runs and loads page in memory ; program continues.

Needing a page not in memory is called **page fault**, and loading page in memory is called **page fault handling**. Main issues with

The first issue to tackle is to discover the page fault : for that, we use the valid bit in page table. For the bit to be set to 1, the page needs to be correct AND to be in memory. Otherwise, bit is 0. In this latter case, we have a page fault.

Second issue : stop the faulty process. The valid bit does the job (accessing it generates a trap).

Third : get page from disk. Allocate a frame, find page in disk and copy it. While we are busy copying, call the scheduler to schedule a new process. And when done, suspend running process and get back to handling. But to allocate a frame, we need to have one available. We first pick a frame to be replaced, invalidate that page table entry (set valid bit in page table and TLB). But to invalidate, we might need to write that frame to disk first. Note that the page table entry has a *modified* bit, set by hardware if page is modified. IF set, write out page to disk, wither proceed with page fault handling.

But how to find a page to replace ? We need to be careful, as a normal memory access takes nanoseconds but a disk I/O takes milliseconds. We always prefer to replacing clean over dirty. There are many policies for that :

- FIFO : self explicit. Replace oldest page. Bring new to the tail, empty the head.
- OPT : the "optimal algorithm". We replace the page that will be referenced in the furthest future. This is not really implementable as we need to look forward in future. This is only used for reference, as the optimality for the reference string. No policy can achieve better result.
- LRU : least recently used. Idea : if page was not used for a long time, you have least chance to call it soon. As we cannot look into the future, you try to predict using past. Similar to FIFO, but when calling a page, bring it to tail. **Problem :** you would need to manipulate the queue at each memory access, which would heavily impact performances. We need to time-stamp every memory access, which is too expensive. But we can approximate it with the following : we conserve a reference bit. understand

  Better approximation : understand and complete
- FIFO with second chance : Similar to standard FIFO. But at replacement, look at head : if reference bit os 0, replace ; otherwise, put at head of queue and set reference bit to 0, and look at new head to replace.

- Clock : not seen during course, in slides.

In general, to evaluate performances, we pick a "reference string", and simulate the policy with this one. Then count page faulty while simulating.

But we have multiple programs. How many processes to keep in memory ?

complete

There is a close link between degree of multiprocessing and page fault rate. If we give each process frames for about all of its pages, we have low multiprocessing but few page faults. Will be slow when switching on i/o. On the other hand, giving each process 1 frame, we have high degree of multiprocessing, but many page faults. Will be quick when switching on i/o. To optimize it, we define a **working set of a process** . We would like to give each process enough frames to maintain its working set in memory. If $\sum$ of all working sets $>$ memory, swap out one or more processes. If $<$, swap in one or more processes. To predict the working set for the next N refs, we allocate the working set for last N refs.

complete

7.3 Optimization

Prepaging : So far, we paged in 1 page at a time. Prepaging consists of paging in multiple pages at a time, with the idea that we will likely need next one too. So we page in surrounding the faulty page. We rely on locality of virtual memory access. We avoid page faults and process switched, and can also et better disk performances. But we may bring in unnecessary pages.

Cleaning : So far, we only replaced "clean" pages. Cleaning writes out "dirty" pages when disk is idle.

Free frame pool : We try to keep some frames unused. Page fault handling is quick. Reduces effective main memory size.

Copy-on-write : Clever trick for sharing pages between processes : that are initially the same, that are likely to be read-only, but that may (unlikely) be modified by a process.

8 Week 8

8.1 Recap Week 7 : Demand Paging

How to handle page fault. Which page to pick when replacing a page. How many frames to allocate to a process ? concept of working set. Interaction between page replacement and frame allocation (global vs local replacement).

Finally a few optimizations. One important is Copy-on-write sharing : we make the page table entry point to same frame. We set a read-only bit so we generate a trap if process writes ; the subsequently fault is not treated as illegal memory access. Instead : create separate frame for faulting process, insert tuple (pageno, frameno) in page table, set read-only bit to off, and copy the page into that frame. The further access won't generate a page fault.

8.2 Week 8 : Introduction to File Systems

8.2.1 Notion of "permanent" storage

What does permanent mean ? Should it be across program invocations ? Across login (/tmp)? Across machine failures/restarts (most computers) ? Across disk failures (server) ? Across multiple disk failures (data center) ? We will in this course concentrate on permanent accress a computer shut-down/reboot.

We could keep everything in memory, but we would need it to be battery-backed (not suitable). We could also use nonvolative memory (flash, other tech), Most storage are disks, and older systems still use tapes. We will study disks and flash memory.

8.2.2 File system interface

For all we know, a file is an un-interpreted collection of objects (file system does not know meaning, only program does). Those objects are essentially bytes, could also be records,... Files can be typed (= FS knows what object means). Good, because we can set default program, prevent error,... Sometimes better to have it untyped (more flexible, and less code as less types). We will look at the latter. Careful : filename extension is not equal to type (in linux), extension is just a helpful reminder for user.

Access primitives : read creates empty file and returns UID (not filename). Delete(uid) deletes file with uid. Read and write as we know them. We go from file to buffer, or inverse. They are usually sequential (here presented as random). That means we keep a file pointer, and when reading we read and move the pointer as much as we read. We can build sequential on top of random (the "from" argument would be a custom fp, and we manually increase it). But we can't do reverse, as we would need the fp. For that, FS provides a seek() method, to move the cursor.

Concurrency : What if two processes want to access the same file ? What will become of *fp* ? For that, we don't let anyone use the uid and fp. We

create an *open()* method, that will return a *tid* (temporary). We can now write and read using this "public" tid. Be default, Linux has a completely shared copy/write. That means that writing to a document is immediately available to others.

Naming : name is a mapping from human-readable to a uid. compléter

FS tree is not really a Tree : it's a directed acyclic graph (so we can share two UIDs under different name). We can make links, hard or soft complete hard/soft explanation

Why keep the graph acyclic ? Many operations will result in looping or completely lost space. Softlink can't create cycle, while hardlinks can. That's why hardlink can only be done to a file (leaf) and not a directory.

8.2.3 Disk management

Disk still represent 99% of storage, flash (SSD) are still very low compared to servers. Disk are quite simple in the mechanical sense. Main terminology to remember : sector is a block on a platter, of the size of the read/write head. All secors on a platter of the same diameter are a track. All tracks with same diameters on all platters are a cylinder. The disk in solely accessible by sector : writing a byte is equivalent to writing whole sector. We apply read/write_sector to a buffer. The FS main task is to translate from user interface methods (read) to disk interface method (read_sector).

A disk access is fragmented in multiple operations : Select the platter (head selection), Move arm over cylinder (seek), move head over sector (rotational latency), read from sector (transfer time). Head selection is very fast (few nanoseconds). Seek requires mechanical movements, so we have a time linear in the number of cylinders (few miliseconds). The rotational latency is no faster : at 4'500 - 15'000 RPM, we need between 2 and 7 milliseconds. Finally the reading is quite fast (around 1 Gbps), so less than a millisecond for one byte. It's important to realize that disk access time if 10'000 to 100'000 times slower than memory access time, and that seek time dominates (followed by rotational latency). That's why we implement a cache to avoid disk access. Compléter

We know how caches work. Reading is easy : if in cache, read, if not find free space, read from disk and write in cache. Writing is more complicated : we have 2 policies, write-through (when writing, write to cache as well as to disk) and write-behind (write to disk at specific time). Write-behind is more efficient, but in case of a crash we lose everything on memory not written to disk. But we still prefer write-behind with periodic flush. As crash are rare, we gain a lot of time.

Another optimization is to read more than necessary (read-ahead, prefetch).

Only for sequential access. As most access is sequential, we do that.Third optimization : minimize seeks, by doing either clever disk allocation (locate related date on same cylinder), or clever scheduling (reorder request to seek as little as possible). There exist also multiple policies for clever scheduling : first come-first served, Shortest-seek-time-First, SCAN (continuously move from 0 to end, and pick the required sectors the closest to you in the good way), C-SCAN, LOOK, C-LOOK.

9 Week 9

9.1 Recap week 08

File system interface : File are uninterpreted (the OS can't interpret it on his own) un-typed sequence of bytes. Identified by a globally unique *uid*. Some file system primitives, as create(), delete(), read(), write(). Access are generally sequential, and we use Seek().

The directories structure is called the *tree* (but really an acyclic graph, because of hard/soft links). For files, we first create (with *Creat()*, then open, then read requested number of bytes and finally close.

A lot of terminology. Basically we can only modify sectors, by only moving the arm. We can't touch "bytes" or anything else, and we have control on the arm only. We use *Read_sector* and *Write_sector* to modify tracks. Accessing the disk is really really heavy and long. This long operation is first because of the seek, then because of rotational latency, and finally actual reading. This is optimizable with caching (keep read data, avoid disk access), read-ahead (read further, avoir waitin for disk), disk allocation (allocate disk in a clever way) and disk scheduling (reorder accesses to disk smartly so head must move as little as possible, avoid seek).

9.2 Week 9 : Basic file system implementation

We implement 2 main functionalities : naming/directory (use file to store directory) and storage (either on-disk data structures, and in-memory data structures). The main task of our FS is to translate from user interface methods (such as *Read(uid, buffer, bytes)* to disk interface methods (such as *ReadSector(logical_sector_number, buffer*). Main reaction : user read allows arbitrary number of bytes. Easy to translate. But for simplicity, we consider we will only read a block. We also simplify for presentation : block size = sector size /usually sector = 512 bytes, block = 4096 bytes). For terminology : when talk-

ing about disks, the word pointer is often used as a disk address, so a logical sector number. Not a memory address. Also, pictures often contain "arrays", even they are not so on disk.

Golden rule : if not on disk during a crash, it's gone. It must be on disk if we need to recover after a crash. We have 4 main data structures on disk : boot block, device directory, the user data and free space.

Boot block : at fixed location on disk (usually sector 0), it contains the boot loader. Immediately read on machine boot. Never look at it, it's terribly ugly.

Device directory fixed, reserved area on disk. It contains an array of records, (each is a device directory entry, or DDE). This array is indexed by *uid*, and a record contains : in-use bits (reference count), size of file, some other administrative info (right, owner,...), and disk address(es) pointing to file data. User data allocation can be done in various ways :

- Contiguous : disk data blocks contiguous on disk, need only 1 pointer in device directory entry. This is highly efficient, as data are close physically. But as we need to find room for a specific data, we will have **disk fragmentation**.
- Linked list : we allocate blocks, and each points to next one. Good for sequential, but for random access it's horrible. Other problem : as we have pointer to next block, we have to ignore it and it's a pain in the ass.
- Indexed : we have N pointers in device directory entry, that point to data blocks on the file. It's good for random access, but for sequential access we constantly need to come back to DDE to read next address and move to next block.
  - Better way : cache DDE and only move between blocks. Even better : we store them almost sequentially, so we don't even have to move arm. Problem here : DDE can contain about 15 pointers, so we are limited in file size with this system.
  - With indirect blocks : we have N pointers in DDE. First M point to M first data blocks, when M+1 to N point to indirect blocks. Indirect blocks do not contain data, but point to subsequent data blocks. Double indirect blocks are also possible. This of course has a cost, as to access last blocks we need to read 1 or 2 additional pointers. But this occurs rarely, as files are usually small.

- Extent-based allocation : we split our file in N contiguous spaces (composed of multiple blocks). Then each pointer in DDE points to the beginning of such a subspace.

9.2.1 In-Memory Data structures

- cache : fixed contiguous area of kernel memory. Size = max number of cache blocks times blocks size. A large chunk of memory of the machine.
- Cache directory : usually a hash table. The index is the hash of the disk address. With an overflow list in case of collision. It usually has a dirty bit, to tell if modified from what is on disk.
- (System-wide) Active File Table : We have one array for the entire system, and one entry per *open file*. Each of these entries contains the DDE of the file, and some additional info, such as the refcount of number of file opens.
- (Per process) Open File Tables. We have one array per process, and one entry per *file open*. Indexed by file descriptor *fd*. Each entry contains a pointer to corresponding entry in active file table, the file pointer *fp* and additional info.

9.2.2 Putting all together

1. Creation of file : $uid = Create()$. Find a free uid (refcount = 0), set refcount to 1, fill in additional info, write back to cache (and to disk). Device directory is cached in memory. Usually easy o find free uid.
2. Deletion : $delete(uid)$. Find inode, decrement refcount. If refcount == 0, free all data blocks and indirect blocks, and set entries in free space bitmal to 0. Finally write back to cache, and to disk.
3. Open : $tid = Open(uid)$. First check in Active File Table if uid is already open. If so : increment refcount in AFT, allocate ??? if not, find free entry in AFT, read inode and copy in AFT entry, set refcount to 1, and then allocate entry in OFT, point to entey in AFT and set fp to 0.
4. read : find fp in OFT and increment. From here, compute what block needs to be read, follow pointer to find disk address in inode in AFT. Lookup in cache directory (disk address). If present return, if not go read on disk. read slides...

10 Week 10

10.1 Recap week 09

Main task of file system if to translate user interface methods to disk interface methods. Disk is structured with specific data structures : Allocation is indexed with indirect and/or double-indirect blocks. Cache is also structured with specific data structures (open/active file table, queue, cache,...).

10.2 Week 10 : Dealing with crashes

Depending on what we are doing, we have problems. Particularly, if we crash between 2 write, or just before the close. To avoid half-written file, we need to have atomicity (all or nothing, all updates on disk or no update on disk). But how can we implement atomicity with open/close in FS ?

We assume a single sector disk write is atomic (meaning it's instantaneous). This assumption is not entirely true, but with very high probability yes (disk vendors try to ensure that). To achieve so, we make sure to have old copy AND new copy on disk. Never erase file before being sure the new copy is complete. Once this is done, switch atomically between the two.

With write-through cache, we first open (read DDE into AFT), then write (allocate new blocks on disk, fill in address of new blocks in incode DDE, write to cache and disk), and finally close (write incore DDE to disk DDE). Slightly different with write-behind : we write all cached blocks to new disk blocks when closing.

If all goes smooth, the old blocks are put back in free list of disk, if crash before we can free them, they will be at reboot when $fsck$.

10.3 Log-Structured File System

This is an alternative way of structuring FS. It takes idea of log-write to the extreme.

Rationale for LFS : If we have large memories, we have large buffer caches. Most reads are served from cache, and most disk traffic is write traffic. Now, how do we optimize disk I/O ? We optimize disk writes, and we optimize disk writes by writing sequentially to disk. The key idea is to consider the log as a append-only data structure (on disk). *All* writes go to log, including data and inode modifications.

Writing is then done as follow : first go into cache (write-behind, both inodes and data), and also into (in-memory) buffer. When the buffer is full,

we append to log (this is called a **segment** of log. This implies we don't need to seek on writes !

trous...

11 Week 11

11.1 Recap week 10

We mainly fear crashes. To avoid data partly written, we try to have atomicity (either all data on disk, either nothing). We assumed single disk sector write is atomix, but multiple disk sector writes are not. We achieve atomicity by keeping new and old copies at any time during write, and then switch atomically by writing a single switch sector in inode (this is **shadow paging**). We can also use **intentions log**, where we write data and inodes to a log and copy data in-place later.

LFS : log-structured file system. We have bigger and bigger memories, leading to large buffer caches. That means most of reads are done to cache, and most of disk traffic is composed of write. That means we optimize disk-write (by writing sequentially). The key idea is that we write everything to log, including data and inode (everything except checkpoints).

On disk, we have checkpoints and the log. Checkpoint region is a fixed location on disk. Log used the remainder of disk

[...]

On memory, we have cache (regular write-behind buffer) and segment buffer (segment being written). Other data structure : Inode map. This is an array indexed by uid that point to last-written inode for uid.

[...]

With this sequential method, disk is quickly full : we need to clean disk, and so remove old data (there is a newer write in log). We inspect a segment, read a block (particularly its uid), look into inode_map, retrieve most recent uid. If analysed and most recent uid are different, we can clean. But this is really heavy, so Linux did not implement that.

11.2 Alternative storage media : RAID and SSD

Disk evolved enormously. From slow, large, expensive they have become fast, small and cheap. We are facing 2 issues : bandwidth (for servers, big data computation) and response time (for desktop and laptops, or transaction systems). RAID answer bandwidth, SSD answer response time (and bandwidth,

for a much higher cost).

11.2.1 RAID

Redundant Array of Independent Disks. With that, we try to optimize I/O bandwidth through parallel I/O, meaning accessing multiple disks at once. Here, rather than putting a file on a disk, we stripe it across multiple disks. That multiplies bandwidth by number of disks (not infinitely, at some point there are other limiting factors).

As disks are cheap and small, we can place them in a sort of RAID box. To OS, this RAID box looks like one disk.

Problem : MTBF[1] becomes drastically smaller too. If one drive fails, all data is lost. At the opposite, without RAID, only data on failed disk is lost.

Solution : redundancy. We store redundant data on different disks. No redundancy is RAID-0 (no redundancy). 1 is mirroring, 4 is parity disk, 5 is distributed parity.

- RAID-0 : no redundancy, best bandwidth, highest risk.
- RAID-1 : mirrored disks (we use half of disks as redundancy). We write to both disks, and read from either. If a disk fails, we have a complete copy. But for same number of disks as RAID-0, we have only half storage. But this is reliable.
- RAID-2/3 : not covered.
- RAID-4 : We use 1 disk for parity for N data disks. The parity is an algorithm of error detection and recovery. For example, for 4 data bits, we define the parity as XOR of all bits. If we lose one, we can do inverse : XOR of remaining bits and parity. Loosing a disk can be recovered. Read are done from data disks, write to data disks and parity is corrected. In case of crash, recover from data AND parity disk. Still an issue : all writes need to pass through parity, so no real gain.
- RAID-5 : this addresses the RAID-4 problem : we spread the parity between all disks. We still "lose" 1 disk, but now bandwidth is distributed between disks (potentially idle), so we remove the bottleneck.

[1] mean time between failure

11.2.2 SSD

Solid State Drive. Not really a disk, this is purely electronics (NAND flash), with no moving parts. But it was made to look exactly like a disk (same physical and software interfaces). They are composed of pages of $4k$. A block is a set of pages (e.g. 64 pages). Huge difference : even though we can read a page simply, just as a sector, we can not overwrite a page (only write). We need to erase a whole block first, before being able to rewrite. But we can only perform about $100k$ erase before SSD failing.

To avoid nasty situations, we try to wear leveling : pick blocks to erase in such way to erase all at the same rate, avoiding one particular block dying too quick.

12 Week 12

12.1 Recap week 11

LFS data structures : on disk (lok or checkpoint) and in memory (inode map, segment buffer, both in addition to conventional data structures). Problem when cleaning (we never delete). Disk cleaning : we clean a segment at a time, and decice whether a block is old/new by inspecting inode. Cleaning is expensive and has limited adoptions on HDD.

RAID : as today, disk bandwidth is stagnating but disk size is improving fast, we decided to do parallel I/O to improve disk bandwidth. This also improves reliability (higher levers survive disk failures, using redundancy).

Better disks : SSD. We read one page at a time, and erase blocks. We can write a page only after erasing containing block, which reduces lifetime. This induced LFS on SSD, with wear-leveling.

12.2 Week 12 : Virtual machine

12.2.1 What is a Virtual Machine ?

A VM is a machine running on a machine. Usually, many machines run on a physical machine. The base is physical, and the ones on top are virtual. We can represent a common basis to all machines being the hardware and VMM (or hypervisor). This representation looks awfully like applications on OS : basis of hardware and OS, then VMs 'are' applications. But it is really different. It is similar as both OS and VMM provide resource management, but they are

also different as OF provides abstraction (processes, address spaces,...) while VMM does not provide it (only provide an identical copy on the machine).

12.2.2 What does a VMM do ?

It is a resource manager for VMs. It provides creation/destruction/scheduling of VMs, memory management, disk management, I/O management. There is a similarity with what OS does, with processes replaces with VM. But again, it does not provide abstraction !!

Typically, we run an OS in a VM, with applications on top. Cool thing : we don't need to port it, as VM provides machine interface.

12.2.3 Terminology

- Virtual Machine (VM) : efficient isolated duplicate of real machine
- (VMM) virtual machine monitor or hypervisor : provider of VMs. Is in control of the hardware, uses hardware to efficiently provide VMs.
- Guest OS : OS that runs in a VM.
- Native mode : application or OS running on real hardware
- Virtualized mode . application or OS running on virtual hardware

12.2.4 History

Developed by IBM in 70s. Their motivations was that computers were very expensive mainframes, so VMS allowed sharing by users running different OSes. In 80s-90s, we see the introduction of microprocessors. The original motivation for VMs disappeared, the concept fell out of flavor. In the late 90s to now, the concept became prominent again. The motivation is to allow Windows and Linux apps on same machine. Ease of debugging new OSes. We also craved server consolidation in data centers (most servers mostly idle, power consumption has become big cost factor, we can run multiple servers on one machien). Server consolidation is today the main motivation.

12.2.5 Technical issues

How to implement a VMM ? From first lectures, we are reminded we have kernel and user mode. We used privileged instructions and system calls. As we did earlier, OS and hardware are in kernel mode, applications in user mode.

Clearly, we can't allow anything from VM to be in kernel mode. If we allowed so, a VM could modify other VMs and host OS. Not good.

Problem with that : as whole VM is in user mode (including guest OS), when the emulated app uses a system call, it will want to go to kernel mode. But as guest OS is not in kernel mode, this will cause troubles. But main issue : OSes are written to run in kernel mode, but actually run in user mode. Resulting that : system calls do not work properly, applications could access guest OS memory and guest OS uses privileged instructions.

How to deal with system calls ? What if an application does a system call ? it want to perform in its guest OS. Syscall vectors for the physical machine in VMM, while syscall vectors for its OS in its guest OS. Hardware directs syscalls to physical syscall vectors. VMM must somehow forward system call. To solve that, at VMM boot time we forward system call : install physical machine system call.

At system call time, hardware puts machine into kernel mode and jumps to machine syscall handler. VMM sets PC to appropriate syscall vector in OS, and then returns to user mode.

How to implement virtual memory ? Process(or) issues virtual addresses and MMU produces phsyical address (either from TLB or from page table), then physical goes to memory. In netive mode, OS allocates physical to processes and installs page table and TLB entries to do so. In virtualized mode, VMM allocates memory between Vms and installs page table and TLB to do so. We then have a similar problem : guest OS thinks it is running on the real machine and that is has control over all memory of real machine. But no : running on virtual machine and has limited portion of memory (allocated by VMM). So, we have not the host and guest physical address (hPA and gPA).