Lecture Summary

Table of Contents

Info

Study Part

1 Introduction[1]

The OS has three different roles: it is a referee, an illusionist, and a glue. While this is heavily simplified, it's a very good way to think about the OS!

Services provided by the OS are:

- Program execution
- Access to I/O devices
- Protection and access control
- Error detection and reporting
- Accounting and auditing

Roles of the OS

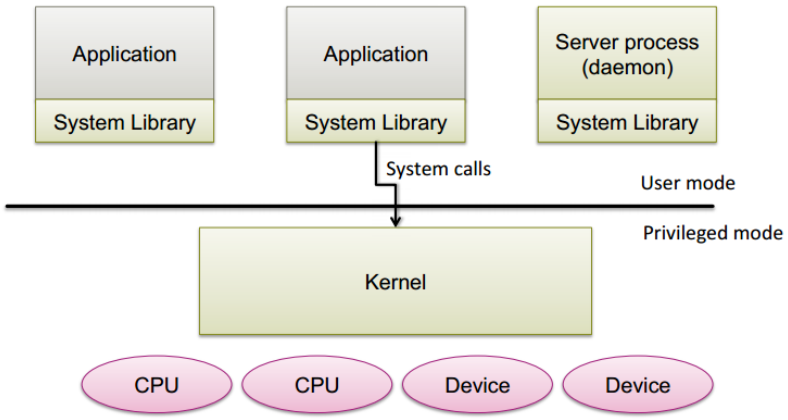
The **referee** manages the access of various applications to the hardware (the applications use system calls). It has to share and protect the resources but also shield applications from one another while still allowing communication between applications. Managing resources should be fair, efficient, and predictable – but this is in mutual contradiction. An example are threads.

When the OS behaves as an **illusionist**, it presents a physical resources as a perfect-world virtual resource. This is done by multiplexing (divide resources), emulation (perfect-world), and aggregation (join multiple resources to create a new one). The reasons behind this are sharing, sandboxing, decoupling (not tying a client to particular instance of a resource, e.g. CPU core), and abstraction (ease of use). Examples are (paged) virtual memory, virtual machines, file system, databases, windows, virtual circuits, VLANs.

Eventually, the **glue** abstracts the OS as much as possible, by providing high-level abstractions (which are easier to program to and also provide shared functionality), extending the hardware with added functionality (no direct hardware programming needed), and also hides details of the hardware.

Structure of an OS

The general structure of an OS (*pictured*) can be described as the privileged mode containing the kernel and the hardware and the user mode with applications and daemons.

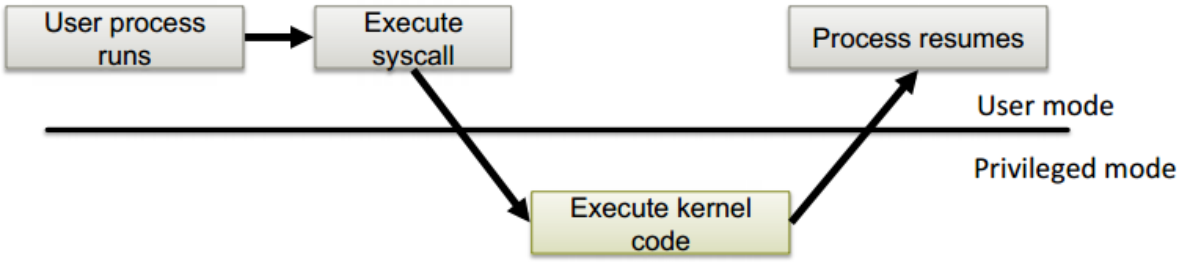


The **privileged** mode is used to protect the applications from the OS and vice versa. It contains the kernel[2] which is the part of the OS running in privileged mode and is nothing but a (special) computer program, typically an event-driven server responding to (= when the kernel is entered):

1. Startup

[1] The heading numbers [of level 2] correspond to the chapter/lecture numbers by Prof. Hoefler and the subheadings correspond to the different topics outlined in each lecture.
[2] In Unix and Windows, the kernel is pretty big, but microkernels exist and some embedded systems don't even have one.

2. System calls, are *the* way a program requests services from the kernel. Implementation varies a lot, arguments can be passed in processor registers, stored in memory or pushed on the stack



   - Examples: fork, read
3. Interrupts
   - Examples: hardware interrupt by the keyboard, incoming network packet
4. Program traps/exceptions
   - Examples: division by zero, segmentation violation

The kernel is exited when:

- A new process is created (including startup)
- A process is resumed after a trap
- User-level upcall (similar to an interrupt, but to user level)
- Switching to another process

System libraries (which are in user mode/land) provide convenience functions and also system call wrappers, allowing high-level languages to create and execute syscalls.

A **daemon** is a process which is part of the OS. The advantage of it being in user land is modularity and fault tolerance and it's also easier to schedule.[3]

2 Processes

"The execution of a program with restricted rights"

Process concepts and lifecycle

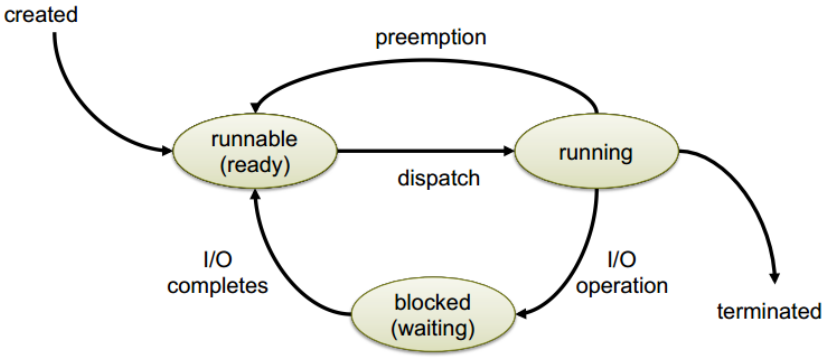
A process is comparable to (very small) virtual machine. Older systems[4] provided a single dedicated processor, had a single address space, and used syscalls for OS functions. The ingredients of a process are:

- Virtual processor: address space, registers, instruction pointer / program counter
- Program text (object code)
- Program data (static, heap, stack)
- OS stuff: open files, sockets, CPU share, security rights …

[3] In microkernels, most of the OS is a daemon and in Linux more and more is being outsourced as a daemon.
[4] In software a computer system is kernel + processes

Context switching

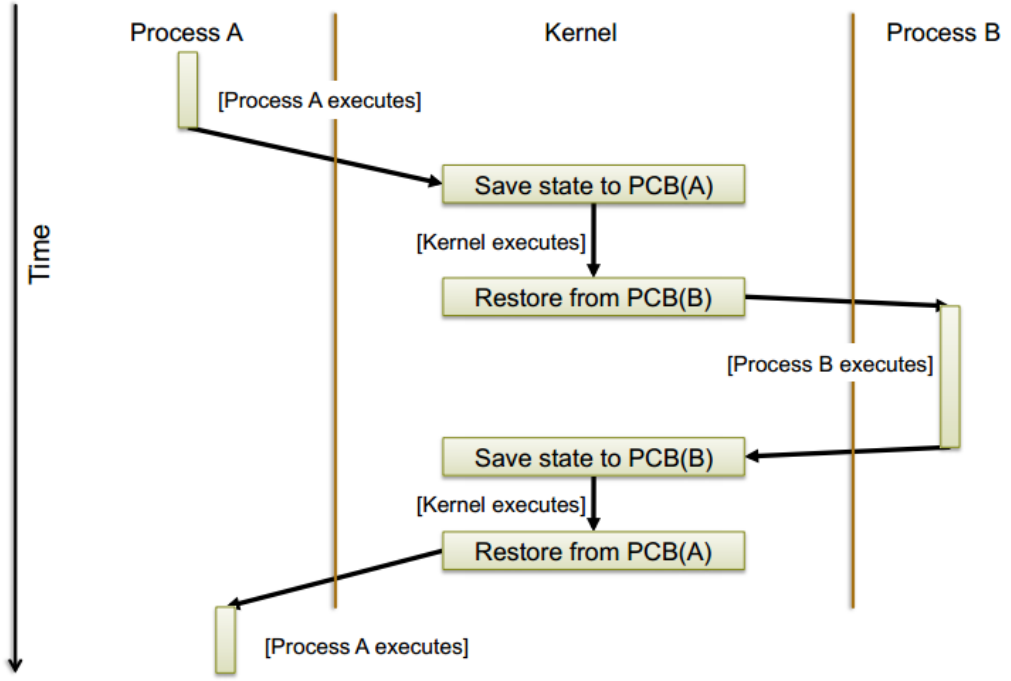
The **lifecycle of a process** (*pictured*) is vital to understand why context switches happen. The different states are:



- Created: it's runnable: can run, but not running
- Runnable: possible to be executed
- Running: actually running
- Dispatch: the OS prepares for it to run
- Blocked/waiting: file system read is slow; other processes run when waiting for I/O

Operating systems use time-division multiplexing on processes (or space-divion on multiprocessors). Each process has a **process control block (PCB)** which uses an in-kernel data stricture and holds all virtual processor state which includes: identifier/name, registers, memory use, pointer to page tables, files and open sockets etc.

Switching between processes means the PCBs are switched out, as illustrated in the following diagram.



Process creation

When a process is being created, a few things are needed: code, memory, basic I/O, and a way to refer to the new process. Typically, the syscalls "spawn" takes care of this and takes enough arguments to start a new process from scratch. Unix uses "fork" and "exec" syscalls which simplify process creation a lot. *fork()* creates a child copy of the calling process and *exec()* replaces the text of the calling process with a new program. This makes a function like createProcess() obsolete.

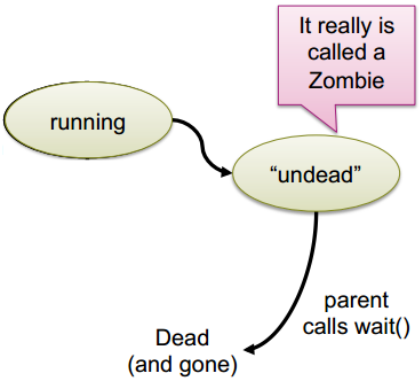
Unix is entirely constructed as a family tree of such processes. Please refer to the following listing[5] as to how a new process is created in Unix[6].

```
#include <sys/types.h> /* pid_t */
#include <sys/wait.h>  /* waitpid */
#include <stdio.h>     /* printf, perror */
#include <stdlib.h>    /* exit */
#include <unistd.h>    /* _exit, fork */

int main(void)
{
   pid_t pid = fork();

   if (pid == -1) {
      // When fork() returns -1, an error happened.
      perror("fork failed");
      exit(EXIT_FAILURE);
   }
   else if (pid == 0) {
      // When fork() returns 0, we are in the child process.
      printf("Hello from the child process!\n");
      _exit(EXIT_SUCCESS);  // exit() is unreliable here, so _exit must be used
   }
   else {
      // When fork() returns a positive number, we are in the parent process
      // and the return value is the PID of the newly created child process.
      int status;
      (void)waitpid(pid, &status, 0);
   }
   return EXIT_SUCCESS;
}
```

This property of Unix having child processes, adds another state to the process lifecycle diagram: the "zombie" state (*excerpt pictured*). A zombie is due to the waiting for a child. If the parent process doesn't wait for a child, the child becomes a zombie. A zombie occupies kernel, process block, use address space etc. A zombie can be killed by killing the (badly-behaving) parent. The "init" process adopts such killed processes and then takes care of zombie elimination.



Kernel threads[7]

There are different types of threads:

- Kernel threads
- One-to-one user-space threads
- Many-to-one
- Many-to-many

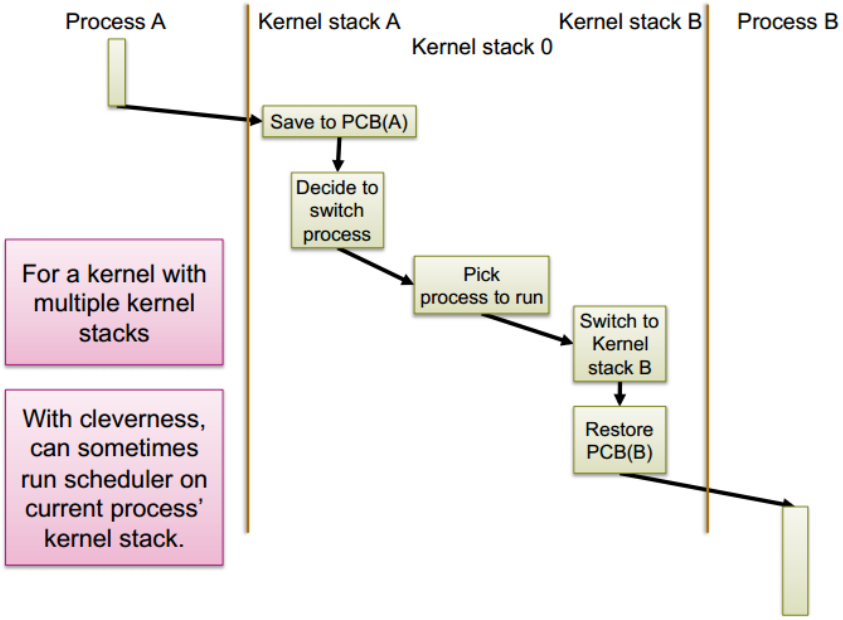
[5] By personal preference, I use Source Code Pro as monospace font. In the final PDF the fonts are (or at least should be) embedded. If you have problems viewing this file, please install the font by downloading it for free from Adobe on GitHub: github.com/adobe-fonts/source-code-pro

[6] Taken from Wikipedia's article on the fork syscall

[7] Not to be confused with hardware threads/SMT/Hyperthreading; in these, the CPU offers more physical resources for threads

Kernel architecture

Kernels can, and sometimes do, implement threads. This allows for multiple execution contexts inside the kernel, comparable to a JVM. However, this doesn't say anything about user space – context switches are still required to/from user processes. The basic question is how many kernel stacks there are. The 6th edition of Unix has one kernel stack per process (and the thread scheduler runs on thread #1, essentially making every context switch to switches). There are OSes which have only one kernel stack per CPU, e.g. Barrelfish, which requires less code, is more efficient, and the kernel must purely be event-driven. This extends the process switching diagram:



System calls in more detail

User space

1. Marshall the arguments somewhere safe
2. Saves registers
3. Loads system call number
4. Executes SYSCALL instruction (or SYSENTER, or INT 0x80 …)
5. … the kernel is entered at a fixed address and privileged mode is set

Kernel

6. Save user stack pointer and return address In the Process Control Block
7. Load SP for this process' kernel stack
8. Create a C stack frame on the kernel stack
9. Look up the syscall number in a jump table
10. Call the function (e.g. read(), getpid(), open(), etc.)

11. … the function returns …

12. Load the user space stack pointer
13. Adjust the return address to point to: return path in user space back from the call, OR loop to retry system call if necessary
14. Execute "syscall return" instruction

15. Result is execution back in user space, on user stack

The precise details of a syscalls vary a lot by OS and architecture.

User-space threads

There are three different options:

1. Implement threads within a process
2. Multiple kernel threads in a process
3. Some combination of the above

Assumption (from now on):

- Multiple kernel threads per CPU
- Efficient kernel context switching

Many-to-one user threads (many user-space threads map to one kernel thread) were used in early thread libraries like the original JVM and is also a typical student exercise. These threads are sometimes called "pure user-level threads" since they don't require kernel support. In terms of address space layout, there is simply some space allocated on the heap for each thread.

- Cheap to create and
- destroy
- Fast to context switch
- Can block entire process
- Not just on system calls

One-to-one user threads (every user thread is/has a kernel thread) are equivalent to multiple processes sharing an address space, except for the naming difference. This is used in most modern OS thread packages. In terms of address space layout, space is allocated on the stack for each thread.

- Memory usage (kernel
- stack)
- Slow to switch
- Easier to schedule
- Nicely handles blocking

Many-to-many threads multiplex user-level threads over several kernel-level threads and has become the way to go for a multiprocessor. This allows a user thread to be pinned to a kernel thread for performance and predictability.

3 Scheduling

Scheduling deals with the challenge of allocating a single resource among multiple clients for *what amount of time* and *in what order*. Usually the resource is the CPU and the necessary decisions are which task[8] is next, how long a given task should run, and on which CPU a task should run. There are different **metrics** to be optimized such as fairness, (enforcement of) policy, balance/utilization, power/energy usage. The metrics can also be dependent of workload (batch, interactive, realtime, multimedia) or architecture (SMP[9], SMT[10], NUMA[11], multi-node).

One big challenge is the **complexity of scheduling algorithms** since the time spent to compute the schedule is wasted and thus should be minimized in order to maximize the utilization of the CPU. A lower overhead however, is no good if the scheduler produces a bad schedule. This leads to a trade-off between scheduler complexity/overhead and optimality of the schedule.

Another challenge is the **frequency of scheduling decisions**. An increased scheduling frequency not only increases the chance if running something different, it also leads to higher context switching rates and thus to lower throughput.[12]

[8] A task can be a process, thread, domain, dispatcher …
[9] Symmetric multiprocessing
[10] Symmetric multithreading
[11] Non-uniform memory access
[12] When switching the context, the pipeline is flushed, the register state reloaded, the TLB might be flushed, just like the caches, and locality is reduced.

Scheduling assumptions and definitions

- Only one processor
- Processor runs at fixed speed: realtime is the same as CPU time[13]
- Only work-conserving scheduling is conserved where no processor is idle if there is a runnable task
- The system can always preempt a task[14]

The different **points in time** where scheduling can occur are:

1. A running process bocks (imitates blocking IO, waits on a child)
2. A blocked process unblocks (IO completes)
3. A running or waiting process terminates
4. A interrupt occurs (IO, timer)

Of the above, in cases 2 and four, **preemption** can be involved. Non-preemptive scheduling requires each process to explicit give up the scheduler (e.g. by making a yield() call). With preemptive scheduling process are dispatched and descheduled without warning (e.g. timer interrupt, page fault etc.). This is usually the case for common OSes and in soft realtime systems (but not in hard realtime!)

When talking about **overhead**, the dispatch latency is time taken to dispatch a runnable process and the scheduling cost is twice the time for half a context switch plus the scheduling time. Time slice allocated to a process should be significantly more than scheduling overhead.

Batch-oriented scheduling

Properties:

- "Run this job to completion and tell me when you're done"
- Typical for mainframe / supercomputer / large clusters

Goals:

- Throughput (jobs/hour)
- Wait time (time to execution=
- Turnaround time (submission to termination)
- Utilization (don't waste resources)

Even though mainframes are a bit old, most systems have batch-like background tasks (including phones) and CPU bursts can be modeled as batch jobs.

The simplest algorithm is the **first-come first served** algorithm which depends on the arrival order and is unpredictable. This leads to the so-called convoy phenomenon where short processes back up behind long-running processes which is while undesirable well-known and widely seen e.g. in databases with disk IO (and also a simple form of self-synchronization.).[15]

The **shortest-job first** algorithm always runs the process with the shortest execution time first. It is optimal since it minimizes waiting time and thus also turnaround time. When adding preemption to SJF, the problem is jobs arrive all the time. The strategy is "shortest remaining time next" has the consequence of new, shorts jobs preempting longer jobs which are already running. This is still not ideal for dynamic, unpredictable, and especially interactive workloads.

[13] This is not true in reality for power reasons (DVFS: Dynamic Voltage and Frequency Scaling)
[14] Not true for very small embedded systems, hard realtime and early Windows/Mac OS
[15] `memcached` makes use of this

To measure *optimality* consider n jobs executed in sequence, each with processing time $t_i, 0 \leq i < n$. The mean turnaround time is $\frac{1}{n}\sum_{i=0}^{n-1}(n-i)\cdot t_i$. This is minimized when the shortest job is executed first. When estimating *execution time* the first problem is to actually define what it is. For non-batch workloads, CPU burst times can be used. Another metric can be the size of the requested web page.

Scheduling interactive loads

Properties:

- "Wait for external events, and react before the user gets annoyed"
- Used for word processing, browsing, …
- Common for PCs, phones …

Goals:

- Response time: how quickly does something happen?
- Proportionality: some things should be quicker

The simplest interactive algorithm is the **round-robin** which runs all runnable tasks for a fixed quantum in turn. It's easy to implement, understand, and analyze, and while it has a higher turnaround time than SJF, it has a better response. The problems being it treats all tasks the same and is rarely what you actually want.

A general class of scheduling algorithms are **priority** algorithms which assign every task a priority and dispatch tasks accordingly. The priorities can dynamically be changed and processes with the same priority are then scheduled using round robin, FCFS etc. **Multi-level** queues, which ideally generalizes to hierarchical scheduling, use for batch, background, low-priority tasks FCFS and for interactive, high-priority tasks round robin. The problem of starvation, where strict priority schemes do not guarantee progress for all tasks, can be solved with ageing where:

- Tasks which have waited a long time are gradually increased in priority
- Eventually, any starving task ends up with the highest priority
- Reset priority when quantum is used up

Multilevel feedback queues are designed to penalize CPU-bound tasks to benefit I/O bound tasks by reducing priority for processes which consume their entire quantum and eventually re-promote the process. I/O bound tasks tend to block before using their quantum and thereby remain at high priority. An example is the **Linux o(1)** scheduler which use 140 levels (0-99: high priority, static, fixed, "realtime", FCFS/RR; 100-193: user tasks, dynamic, RR per level, priority ageing for interactive (I/O intensive) tasks) where the complexity of scheduling is independent of the number tasks by using two array queues, "runnable" and "waiting" which are switched when the "runnable" array is empty. Another example is the **completely fair scheduler** where the task's priority is equvialnct to how little progress is has made. When implemented, a red-block tree with a sorted list of tasks is used. Essentially, this is the old idea of "fair queuing" from packet networks, and is also called "generalized processor scheduling". This ensures guaranteed service rate for all processes. CFS does not, however, expose (or maintain) these guarantees.

Unix scheduling has a few problems. Unix conflates protection domain and resource principal which has the effect of priorities and scheduling decisions being per-process. If one wants to allocate resources across processes, or separate resource allocation within a process (which is typical for a web server structure), whoever runs more compiler jobs, gets more CPU time. Additionally, in-kernel processing is accounted to nobody.

Resource containers, which are based on a 1999 paper emerging from above problems, are an OS abstraction for explicit resource management, separate from process structure.

- Operations to create/destroy, manage hierarchy, and associate threads or sockets with containers
- Independent of scheduling algorithms used
- All kernel operations and resource usage accounted to a resource container

This allows for explicit and fin-grained control over resource usage and the most obvious modern form are virtual machines.

[Soft] Real Time[16]

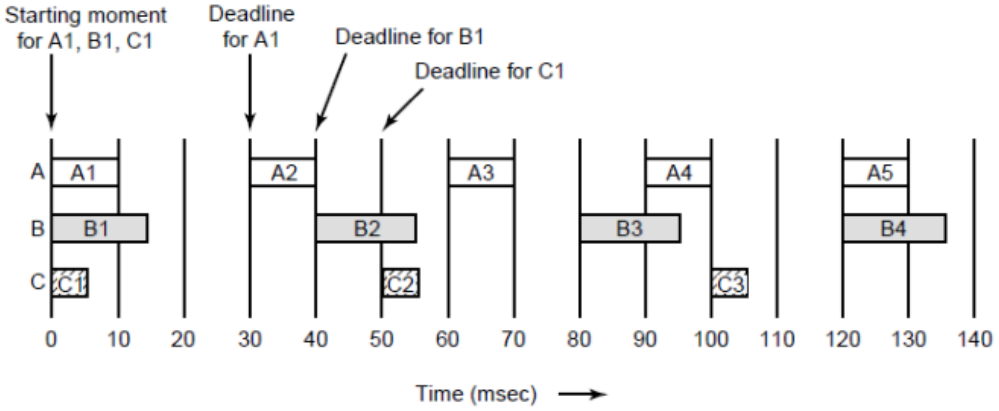
Properties:

- "This task must complete in less than 50ms" or "This program must get 10ms CPU every 50ms"
- Used for data acquisition, I/O processing
- Multimedia applications (audio and video)

Goals:

- Deadlines
- Guarantees
- Predictability (real time ≠ fast!)

In realtime scheduling the problems is to give real time-based guarantees to tasks, which can appear at any time, possibly have deadlines, generally known execution time, and be period or aperiodic. The possibility to *reject tasks* which are unschedulable or which would result in no feasibly schedule has to exist. (*pictured: example*).



Rate-monotonic scheduling works for realtime scenarios. Periodic tasks are scheduled by always running the task with the shortest period first. This implies the algorithm being static/offline. Suppose there are m tasks, C_i is the execution time of the i[th] task, and P_i is the period of the i[th] task, then RMS will find a feasible schedule if $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m\left(2^{1/m} - 1\right)$.

[16] When referring to realtime workloads, soft realtime is meant. Hard realtime workloads are not covered in this course. Examples for hard realtime workloads are: "Ensure the plane's control surfaces move correctly in response to the pilot's actions" or "Fire the spark plugs in the car's engine at the right time" (mission-critical, extremely time-sensitive control applications).

Earliest deadline first scheduling is more complex ($O(n)$), but is dynamic/online and tasks don't actually have to be periodic. Assuming zero context switch time, EDF is able to find a feasible schedule if $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$.

EDF can be used to guarantee a rate of progress for a long-running task, i.e. **guaranteeing processor rate**. This is done by breaking the task into periodic jobs with period p and time s. Then a task arrives at the start of a period and the deadline is then end of the period. This provided a reservation scheduler which ensures a task gets s seconds of time every p seconds and approximates weighted fair queuing.

Multiprocessor Scheduling

Challenge 1: Sequential Programs on Multiprocessors

Scheduling on a multiprocessor is much more complex than on a uniprocessor and especially is harder to analyze, even though in theory it seems straightforward. Also there's the overhead of locking and sharing the queue which is an example for a classic case of a bottleneck in OS design. The solution is **per-processor scheduling queues**.

Since threads can be arbitrarily allocated to cores, they tend to move between cores and thus also between caches. This leads to bad locality and hence performance. The solution is **affinity scheduling** where each thread is kept on one core most of the time is periodically rebalanced across cores. This is non-work-conserving. The alternative is hierarchical scheduling.

Challenge 2: Parallel Applications

Parallel applications often have global barriers and, as a corollary of Amdahl's Law, one slow thread has a huge impact in performance. While multiple threads would benefit from cache sharing, different applications pollute each other's caches. This leads to the concept of **co-scheduling** where the scheduler tries to scheduler all threads of an application together. This is critically dependent on synchronization concepts.

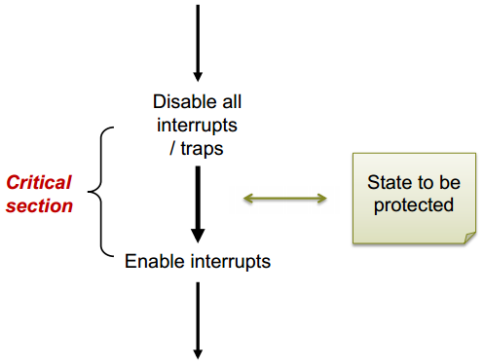
Multicore scheduling is two-dimensional – on one hand there's the question of when to schedule a task and on the other hand where/which core to schedule on. The problem is NP hard and to make it more difficult not all cores are equal and since no process should be able to hold a lock to sleep since other running tasks might be spinning on it.

Little's Law states "The average number of active tasks in a system is equal to the average arrival rate multiplied by the average time a task spends in a system".

4 Synchronization

Recap: Hardware support for synchronization

During the critical section, **interrupts need to be disabled** (*pictured*) since the program when in the critical section cant be rescheduled and data can't be altered by anything else, *as long as it's on a uniprocessor*.

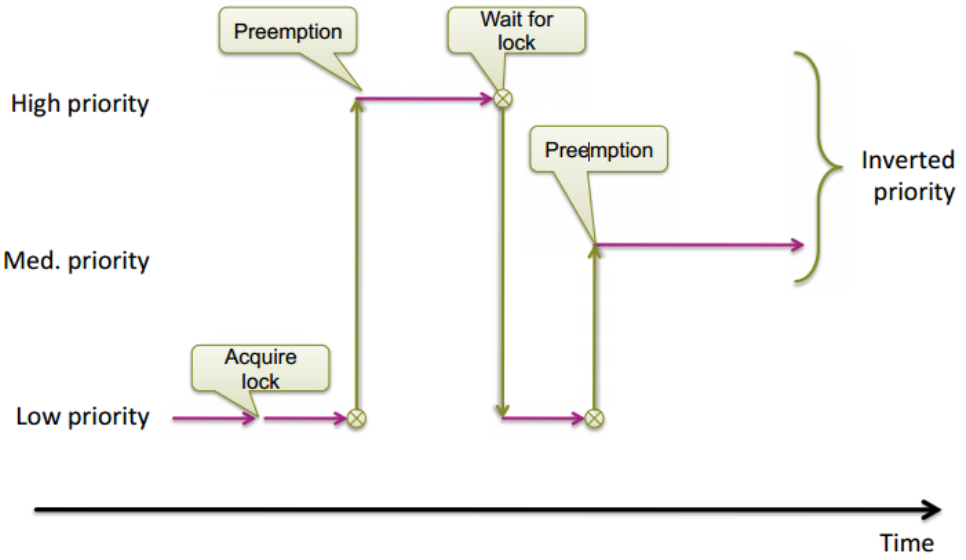


When this is to be implemented, there are a few helpers (which are often hardware-dependent):

- **Test-and-set**: atomically read the value of a memory location and set the location to 1
- **Compare-and-swap**
- **Load-Link**: load from a location and mark as "owned"
- **Store-Conditional**: atomically: 1) store only if already marked by this processor, 2) clear any marks set by other processors, 3) return whether it worked.
- **Spinning** on a multiprocessor[17] isn't possible forever and while another spin is always cheap, block a thread and rescheduling is expensive. Additionally, spinning only works if lock holder is running on another core.
- **Competitive spinning** is within a factor of 2 of an optimal, offline (i.e. impossible) algorithm. A good approach is to spin for the context switch time. In the best case the context switch is avoided entirely and in the worst case it's twice as bad as simply rescheduling

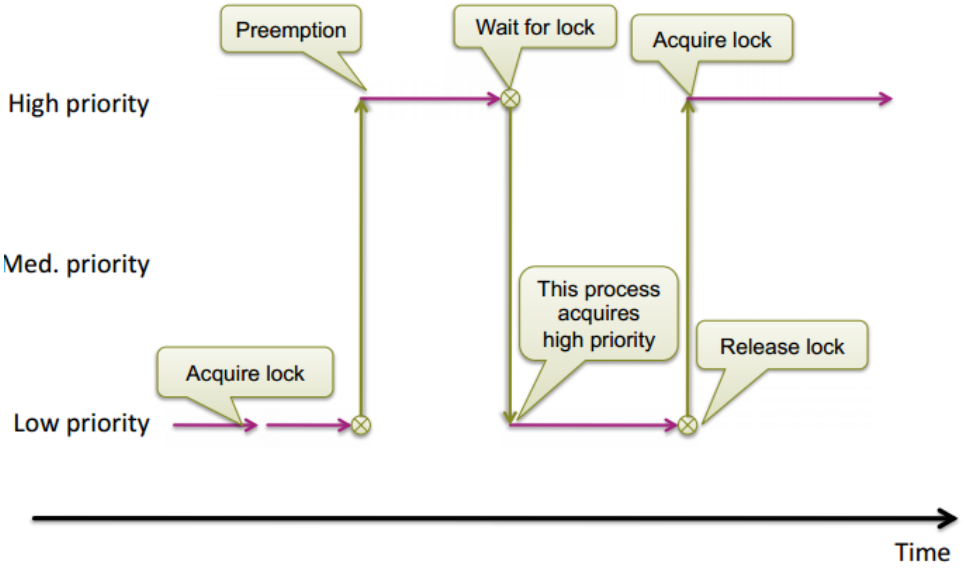
IPC[18] with shared memory[19]

This section focusses on the interaction with scheduling. One such example is **priority inversion** (*pictured*).



[17] Spinning on a uniprocessor is not really spinning since, save for an interrupt, not much will/can happen.
[18] IPC refers to "inter-process communication".
[19] The following techniques have already been introduced in previous lectures: semaphores, murexes, condition variables, monitors.

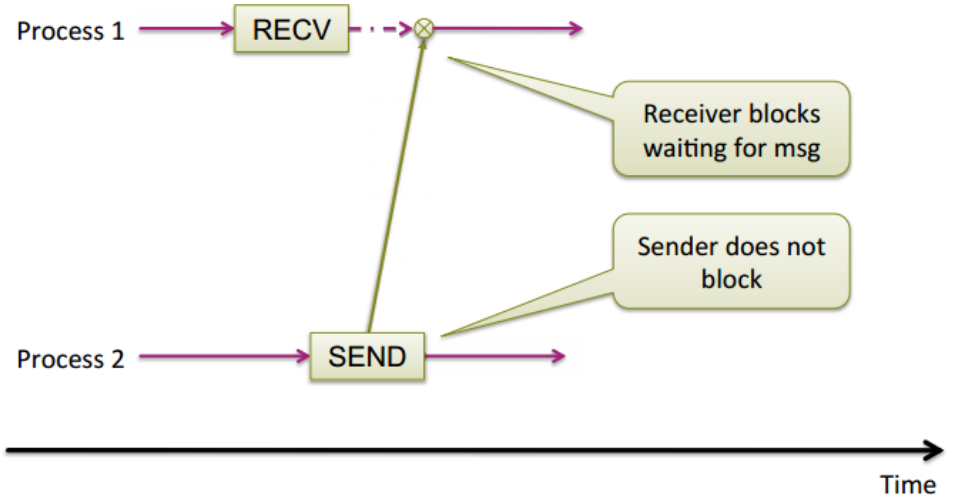
A solution to this problem is called **priority inheritance** (*pictured*) where the process holding lock inherits the priority of highest priority process that is waiting for the lock. When the lock is released the priority returns to previous value and forward progress is ensured. An alternative is **priority ceiling** where the process holding the lock acquires the priority of highest-priority process that can ever hold lock. This requires static analysis and is used in embedded RT systems.

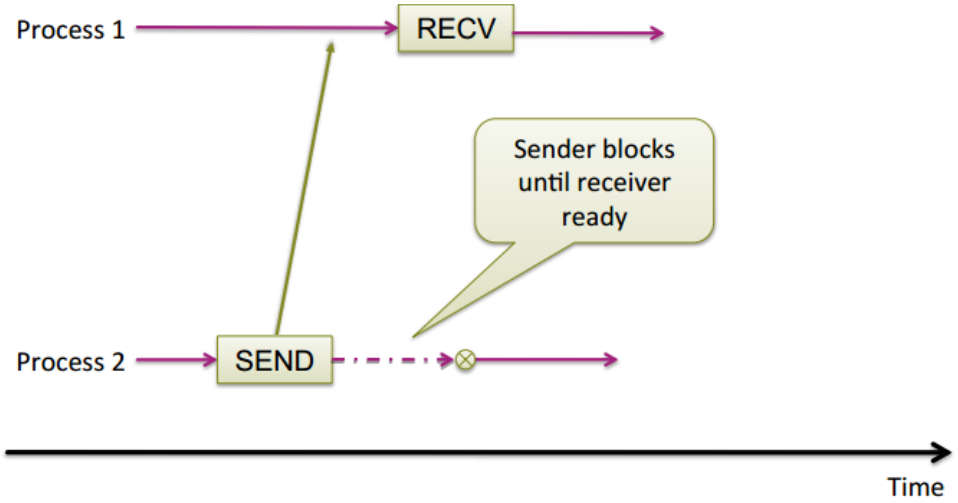


IPC without shared memory

There are two types of IPC – asynchronous (buffered; *picture 1*) and synchronous (unbuffered; *picture 2*). There is a famous claim referring to the deaulity of messages and shared-memory by Lauer and Needham:

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





A very basic Unix IPC mechanism are **pipes** which are unidirectional, buffered communication channel between two processes. To set up a pipe between two processes, you first create the pipe and then fork it (*see listing below; taken from* `man 2 pipe`).

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
   int pipefd[2];
   pid_t cpid;
   char buf;

   if (argc != 2) {
    fprintf(stderr, "Usage: %s <string>\n", argv[0]);
    exit(EXIT_FAILURE);
   }

   if (pipe(pipefd) == -1) {
       perror("pipe");
       exit(EXIT_FAILURE);
   }

   cpid = fork();
   if (cpid == -1) {
       perror("fork");
       exit(EXIT_FAILURE);
   }

   if (cpid == 0) {    /* Child reads from pipe */
       close(pipefd[1]);          /* Close unused write end */

       while (read(pipefd[0], &buf, 1) > 0)
           write(STDOUT_FILENO, &buf, 1);

       write(STDOUT_FILENO, "\n", 1);
       close(pipefd[0]);
       _exit(EXIT_SUCCESS);
```

```
    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

When pipes are used in the shell, each element of the pipeline[20] is forked and connected via pipes, connecting stdout of process i to stdin of process i+1 and each process then exec()s the appropriate command.

Pipes are normally only named by their descriptors and the namespace is local to the process. By making it a named pipe however, it is put into the global namespace with a special file type, the "pipe" aka FIFO.

Messaging system may have different requirements:

- End-points may or may not know each other's names
- Messages might need to be sent to more than one destination
- Multiple arriving messages might need to be demultiplexed
- Can't wait forever for one particular message

One such an example are ports[21] which allow for naming of different endpoints with a process, provide demultiplexing of messages, and can wait selectively for different kinds of messages.

Another message passing system is **local remote procedure call**. An RPC can be used locally to define a procedural interface in an interface description language, or compile/link stubs, or make transparent procedure calls over messages. The naïve implementation is slow.

Another very simple yet powerful messaging interface are **Unix signals** which are asynchronous notifications from the kernel, i.e. the receiver doesn't wait, the signal just happens. A signal interrupts the process and can kill, stop/freeze it, or do something else (*table below*).

NAME	DESCRIPTION / MEANING	DEFAULT ACTION	ORIGIN
SIGHUP	Hangup / death of controlling process	Terminate process	The "TTY Subsystem"
SIGINT	Interrupt character typed (CTRL-C)	Terminate process	The "TTY Subsystem"
SIGQUIT	Quit character typed (CTRL-\)	Core dump	The "TTY Subsystem"
SIGKILL	`kill -9 <process id>`	Terminate process	Other user processes
SIGSEGV	Segfault (invalid memory reference)	Core dump	Memory management subsystem
SIGPIPE	Write on pipe with no reader	Terminate process	IPC system
SIGALRM	alarm() goes off	Terminate process	
SIGCHLD	Child process stopped or terminated	Ignored	

[20] Get it?!
[21] Which are analogusto sockets and TCP ports in IPv4

SIGSTOP	Stop process	Stop	Other user processes
SIGCONT	Continue process	Continue	Other user processes
SIGUSR1,2	User-defined signals	Terminate process	Other user processes

To send a signal to a process, you can use `$ kill –NAME PID` in a shell[22] or `#include <signal.h>; int kill(pid_t pid, int signo);` in a C program.

Signal handlers allow to change what happens when a signal is delivered, this can be the default action, to ignore the signal, or to call a user-defined function in the process, the so-called signal handlers. This allows signals to be used as "user-space traps". The related C function is called signal() and takes two arguments: an integer (the signal type, e.g. SIGPIPE) and a pointer to a handler function and returns a pointer to a handler function (the previous handler). Signal handlers can be called at any time and executes on the current user stack by default. If process is in kernel, it may need to retry current system call. The user process is in *undefined* state when signal delivered. There is very little you can safely do in a signal handler:

- Can't safely access program global or static variables
- Some system calls are re-entrant, and can be called
- Many C library calls cannot be called (including _r variants!)
- Can sometimes execute a `longjmp` if you are careful
- With `signal`, cannot safely change signal handlers

If multiple signals of the same type are to be delivered, all but one are discarded. If multiple signals of different types are to be delivered, they are delivered in any order. POSIX provides an improved version of signal(), sigaction().

Upcalls

Signals ca be used as (a particularly specialized (and complex) form of) upcalls (kernel RPC to user process). Other OSes than Unix use upcalls much more heavily (e.g. Barrelfish) e.g. for "scheduler activations": dispatch every process using an upcall instead of return. Upcalls are a very important structuring concept for systems.

5 Memory Management

Goals of Memory Management

- Allocate physical memory to applications
- Protect an application's memory from others
- Allow applications to share areas of memory, data, code, etc.
- Create illusion of a whole address space
- Virtualization of the physical address space
- Create illusion of more memory than you have

Terminology:

- Physical address: address as seen by the memory unit
- Virtual or logical address: address issued by the processor

[22] I prefix shell (e.g. Bash) commands with a dollar sign ("$") to distinguish them from e.g. C code. The dollar sign is *not* part of the command.

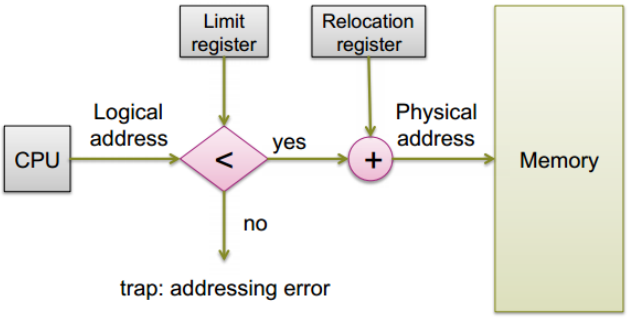
Basically, memory management works as follows (functions 2 & 3 usually involve the hardware Memory Management Unit (MMU)):

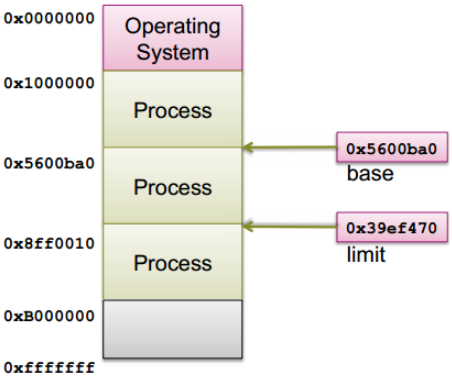
1. Allocating physical addresses to applications
2. Managing the name translation of virtual addresses to physical addresses
3. Performing access control on memory access

Simple scheme: partitioned memory

There's a pair of **base and limit registers** (*pictured*) which define the logical address space. The base address isn't known until load time which forces the code to be either completely independent of the position or a register relocation maps compiled address to dynamic address.

In **contiguous allocation** the main memory is usually divided into two partitions: one for the OS (lower memory addresses) and user processes (higher memory addresses). Relocation registers protect user processes not only form each other but also prevent them from changing OS code and data. The two involved registers are the base register which contains the value of the smallest physical address and the limit register which contains the range of logical addresses (and each logical address must be less than the limit register). The MMU then maps logical addresses dynamically to physical addresses. *(pictured: hardware support).*



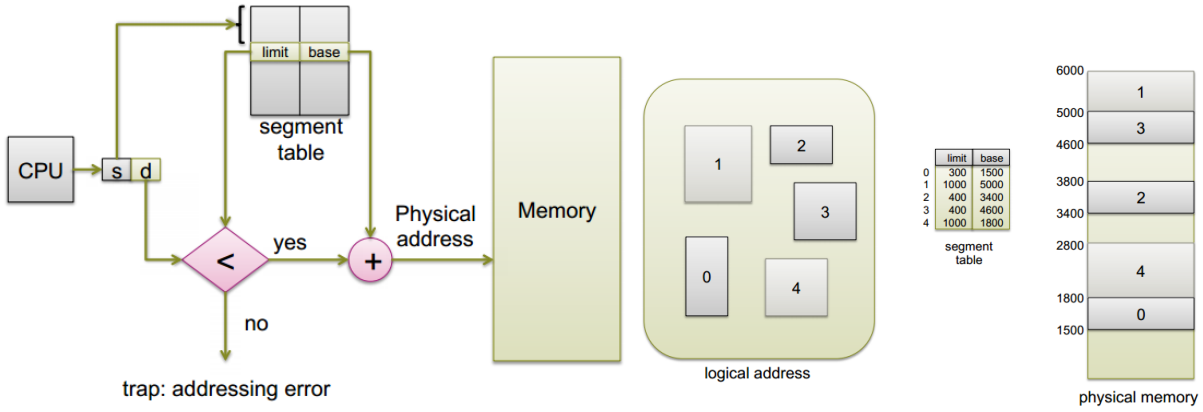


Base & Limit summary

- Simple to implement (addition & compare)
- Physical memory fragmentation
- Only a single contiguous address range
- How to share data between applications?
- How to share program text (code)?
- How to load code dynamically?
- Total logical address space ≤ physical memory

Segmentation

Segmentation generalizes base/limit registers by dividing physical memory into segments and making the logical address into a (segment id, offset) tuple. The segment identifier is supplied by explicit instruction reference, explicit processor segment register, or implicit instruction or process state. (*pictured right: different chunks of memory; pcitured left: segmentation hardware*)



Segmentation requires some **architecture**, most importantly a segment table in which each entry base, the starting physical address of the segment, and a limit, the length of nthe segment. Addtionally there's a segment-table base register (STBR), which stores the current segment table location in memory, and a segment-table length register (STLR), which stores the current size of segment table. A given segment number s is legal if $s < \text{STLR}$.

Segmentation Summary

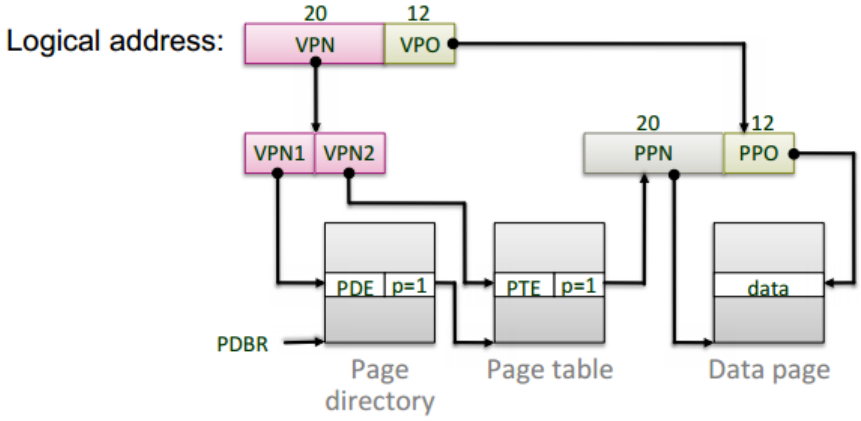
- Fast context switch: simply reload STBR/STLR
- Fast translation: 2 loads, 2 compares; segment table can be cached
- Segments can easily be shared: segments can appear in multiple segment tables
- Physical layout must still be contiguous: (external) fragmentation still a problem

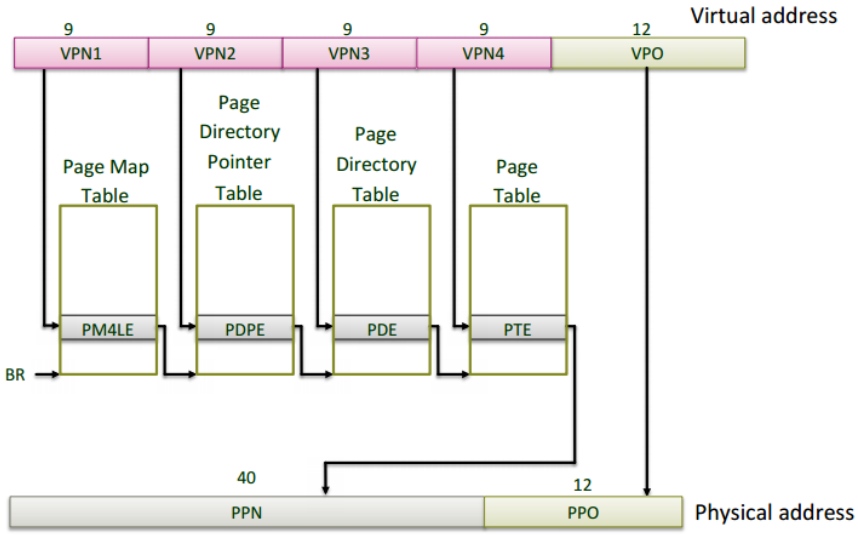
Paging

Paging solves the contiguous physical memory problem and thereby making it always possible for a process to fit as long as there's available free memory. This is done by dividing physical memory into *frames* which are the size of a power of two, e.g. 4096 bytes. The logical memory is divided into *pages* of the same size as the frames. Say a program is of n pages in size: n frames are found and allocated, then the program is loaded, and eventually the *page table* is set up to translate logical pages into physical frames.

- **Frame**: physical memory
- **Page**: logical memory
- **VPN**: virtual page number (upper bits of virtual/logical address)
- **PFN**: page frame number (upper bits of physical/logical address)
- **Page table**: translation from logical pages / VPN to physical frames / PFN
- **PTE**: page table entry
- **TLB**: translation lookaside buffer
- **TLBT/TLBI**: TLB tag / index

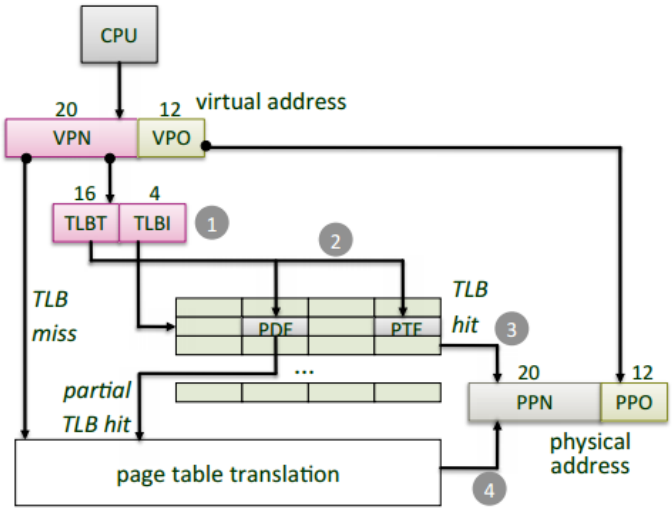
P6 page tables (*pictured; 1*) are used in 32 bit systems and pages, page directories, and page tables are all 4 KB. x86-64 uses a different paging architecture (*pictured; 2*).





The problem with the P6 architecture is performance: every logical memory access needs more than two physical memory accesses – one to load the page table entry (→ PFN) and one to load the desired location. This cuts the performance in half when compared to a system without translation. The solution, as ever so often, is to cache the PTEs. In a P6 architecture, translation with a TLB works as follows:

1. Partition VPN into TLBT and TLBI
2. Is the PTE for VPN cached in set TLBI?
3. Yes: Check permissions, build physical address
4. No: Read PTE (and PDE if not cached) from memory and build physical address



The previously discussed segments are still of use in e.g. x86 architecture where segments are used for thread-local state, sandboxing (e.g. Google NativeClient), or virtual machine monitor (e.g. Xen).

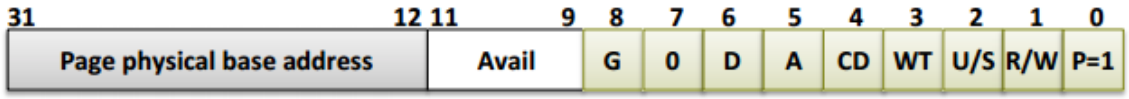
The **effective access time** for a single-level page table is calculated as

$$(1+\varepsilon)\cdot\alpha+(2+\varepsilon)\cdot(1-\alpha)=2+\varepsilon-\alpha$$

Where an associative lookup costs ε time units, the memory cycle time is assumed to be 1 time unit and the hit ratio α is the percentage of the time a page number is found in the TLB (which depends on locality and TLB entries (→coverage)).

Page Protection

Page protection is important and different strategies exist: protection can be associated with each frame, a valid/invalid-bit could be used (legal/not part of address space), or when requesting an invalid address a page fault happens. Or the built-in P6 PTE fields are used. The P bit can be used to trap on any access (read or write).



FIELD	DESCRIPTION
PAGE BASE ADDRESS	20 most significant bits of physical page address (forces pages to be 4 KB aligned)[23]
AVAIL	Available for system programmers
G	Global page (don't evict from TLB on task switch)
D	Dirty (set by MMU on writes)
A	Accessed (set by MMU on reads and writes)
CD	Cache disabled or enabled
WT	Write-through or write-back cache policy for this page[24]
U/S	user/supervisor
R/W	read/write
P	Page is present in physical memory (1) or not (0)

Protection information typically includes: readable, writeable, executable (can fetch to instruction cache), and reference bits used for demand paging.

Page sharing

Shared code allows for only one copy of read-only code to be stored and then shared among processes. Shared code appears in same location in the logical address space of all processes. While the data segment is not shared (and different for each process) it is still mapped at same address (so code can find it). **Private** code and data on the other hand allows code to be relocated anywhere in the address space.

Per-process protection stores protection bits in the page table (the PTE has plenty of bits available). This is independent of frames themselves and allows different processes to share pages. Each page can have different protection to different processes (this can be used for e.g., debugging, communication, copy-on-write etc.).

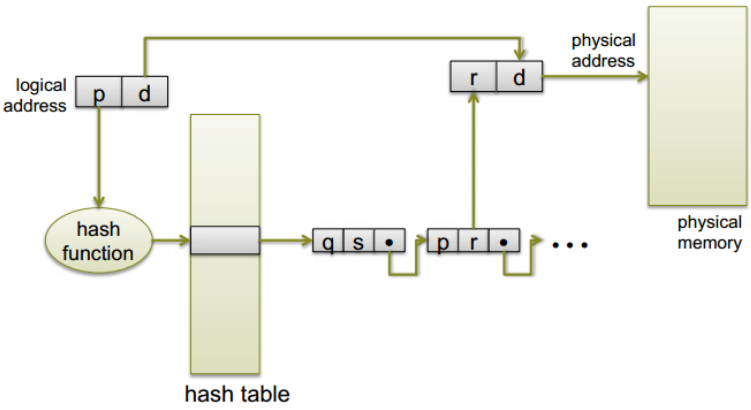
[23] Recall P6 is for 32-bit systems. $2^{32-20}=2^{12}=4096\equiv 4$ KB

[24] (Wikipedia) write-through: write is done synchronously both to the cache and to the backing store. write-back: initially, writing is done only to the cache. The write to the backing store is postponed until the cache blocks containing the data are about to be modified/replaced by new content.

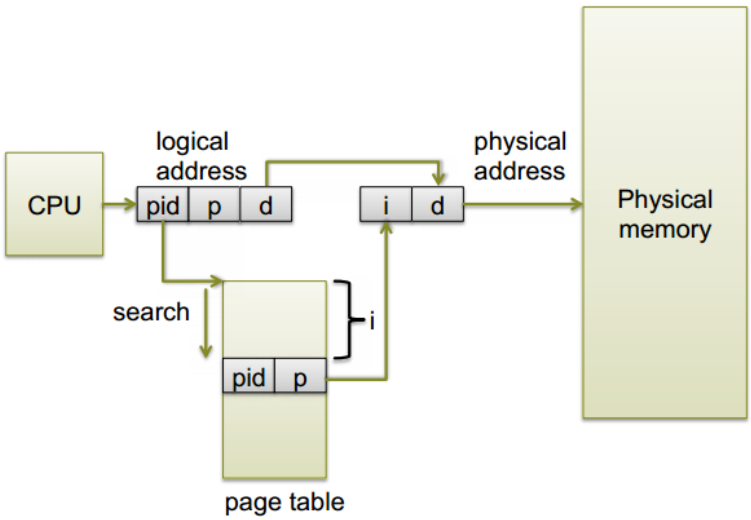
Page Table Structures

Since a simple linear page table is too big, there are a few solutions available: hierarchical PTs, virtual memory PTs, hashed PTs, and inverted PTs.[25]

In **hashed page tables** (*pictured*) the VPN is hashed into a table and the hash bucket has a chain of logical-to-physical page mappings, and the chain is traversed to find a match. While it can be fast, it is unpredictable. It is often used for portability and in software-loaded TLBs (e.g. MIPS).



Inverted page tables (*pictured*) are system-wide PFN-to-VPN mappings and contain one entry, which contains the VPN and the owning process, for each real page of memory. This bounds the total size of all page information on a machine and hashing is used to locate an entry efficiently. Examples where inverted PTs are used include PowerPC, ia64, and UltraSPARC.



Most operating systems keep their own translation information[26] for portability, to track memory objects, to have software virtual-to-physical address translation, and for physical-to-virtual address translation.

TLB shootdown

Since a TLB is nothing more than a cache and machines have many MMUs on many cores (and thus many TLBs), TLBs need to be kept coherent as a security measure (e.g. when memory is reused). To ensure consistency, there are various approaches.

Hardware TLB coherence (rarely implemented) integrates TLB management with cache coherence and invalidates TLB entry when PTE memory changes.

In **virtual caches** a required cache flush / invalidate will take care of the TLB. This leads to a high context switch cost and most processors use physical caches.

---

[25] Hierarchical and virtual memory PTs were covered in the last semester. Thus only hashed and inverted PTs are covered here.

[26] Per-process hierarchical page table (Linux) or system wide inverted page table (Mach, MacOS)

A **software TLB shootdown** is most common. The OS on one core notifies all other cores[27] typically by using an IPI and then each core provides local invalidation.

Another approach are **hardware shootdown instructions** which broadcast special address access on the bus which is then interpreted as TLB shootdown rather than a cache coherence message (used e.g. in the PowerPC architecture).

6 Demand Paging

Demand paging turns RAM into a cache for processes on disk.

Demand paging refers to the functionality of bring a page into memory only when it is actually needed. This reduces IO and memory usage, allows for more users, and provides a faster response.

Uses for virtual memory

Virtual memory, which virtualizes physical memory using hardware and software has a variety of use cases (below). Further advantages, apart from the logical address space possibly being larger than the physical address space, are more efficient process creation and the need for only a port of the program being in the RAM for execution.

- Process isolation
- IPC
- Shared code segments
- Program initialization
- Efficient dynamic memory allocation
- Cache management
- Program debugging
- Efficient I/O
- Memory mapped files
- Virtual memory
- Checkpoint and restart
- Persistent data structures
- Process migration
- Information flow control
- Distributed shared memory

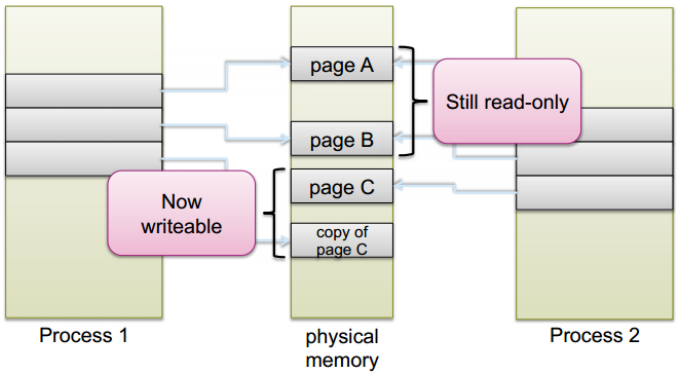
Copy-on-write (COW)

Recall fork(): It's very expensive to create a complete copy of the process' address space (especially when just calling exec() afterwards). That's where vfork() comes into play, which uses a shared address space, which while fast is also dangerous since there are two pointers to the heap afterwards. From this problem, the need for a solution which only copies something when something gets written arises.

Copy-on-write allows both parent and child process to initially share the same pages in memory and *if* either process modifies a shared page, only then the page is copied. This allows for more efficient process creation (since only modified pages are copied) and free pages are allocated from a **pool** of zeroed-out pages. The virtual addresses, unlike with a fork() call, aren't necessarily the same.

[27] Typically using an IPI (inter-processor interrupt)

This works by initially marking all pages as read-only and triggering a **page fault** when either process writes. The page fault handler then allocates a new frame and makes a copy of the page in that new frame and marks each copy as writeable for the respective process. By only copying modified pages, less memory is used (by sharing more) but the cost is to have a page fault for each mutated page. (*pictured: after process 1 has written to page C*)



The general principle, is to mark a VPN as invalid or readonly (which implies a trap indicates an attempt to read or write). When a page fault happens, the mappings are changed in a certain way and the instruction is restarted as if nothing happened. This allows for **emulation** and **multiplexing** of memory which can be very useful.

Demand Paging

If a page is needed, a reference (load or store) to it is created. Of the reference is invalid, the process is aborted and if the page is not in memory, the page is brought into memory. A **lazy swapper**[28] never swaps a page into memory unless the page will be needed. While it is possible with segments, it's much more complex. **Strict demand paging** only pages in when the page is referenced.

The **performance** depends on the page fault rate p, $0 \leq p \leq 1$ where $p = 0$ means no page faults and $p = 1$ means every reference is a fault. The effective access time (EAT) is calculated as:

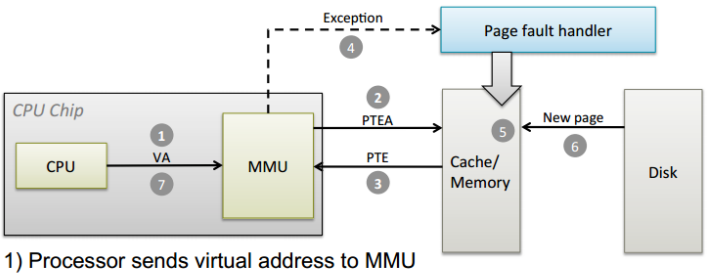
$$\text{EAT} = (1 - p) \times \text{memory access} + p \cdot (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$$

With an example of memory access time 200ns and an average page-fault service time of 8ms and 1 in every 1,000 accesses cause a page fault, the EAT is 8.2ms which is a slowdown by a factor of 40.

Page fault handling

A **page fault** is a trap triggered by the first reference to a page. This is handled as follows (*pictured: hardware*):

1. The operating system looks at another table to decide: if the reference is invalid, the process is aborted, otherwise the page is just not in memory
2. Get empty frame
3. Swap page into frame



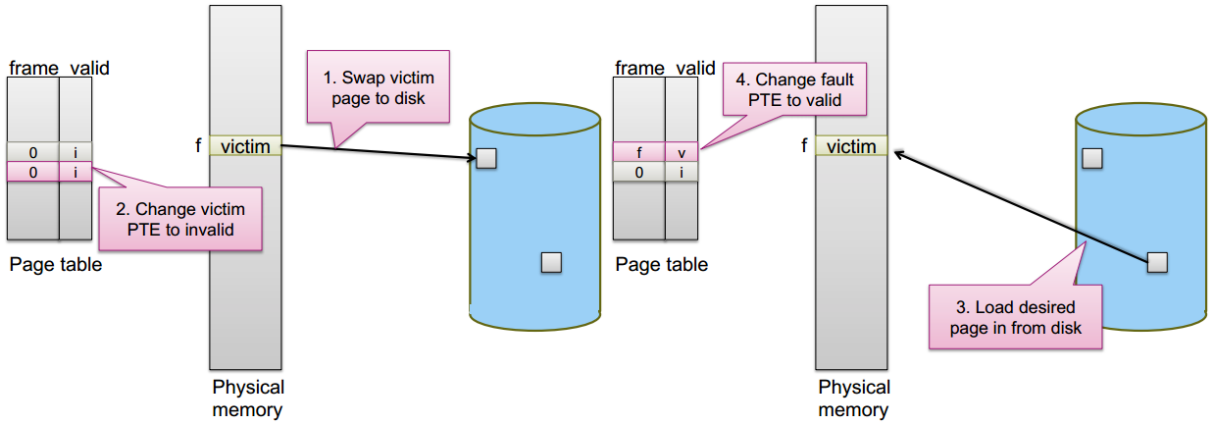
[28] A swapper that deals with pages is a pager

4. Reset tables
5. Set valid bit v
6. Restart the instruction that caused the page fault

Page replacement algorithms

If no free frame is available, an existing page has to be replaced, which should be "little used" (by some mean of measurement) and is then either discarded or written to disk. This page is called **victim page**. An algorithm ideally reduces the number of page faults, and also considers the same page may be brought into memory several times. The algorithm is evaluated by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

While there are various heuristics to pick a victim page which won't be referenced in the future it's ultimately a guess. The PTE uses a "modify" bit which prevent clean/unmodified pages from written to disk and clean pages are preferred over dirty ones to save a disk write. (*pictured: page replacement process*)



By running a simple FIFO algorithm on a given reference string of length 12[29], there are 10 page faults using 3 frames, and 10 page faults using 4 frames. This is described by **Belady's Anomaly**: more frames result in more page faults.

An **optimal** algorithm always replaces the page that will not be used for the longest period of time. Using the same reference string as above and 4 frames, only 6 page faults are needed. The problem however is, this is algorithm is impossible, it's only good to compare to an another algorithm.

An **LRU** (least recently used) algorithm can be implemented using a counter or a stack.
Counter: Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter. When a page needs to be changed, look at the counters to determine which are to change
Stack: A stack of page numbers in a double-linked form is stored. When a page is referenced, it is moved to the top. This makes the search for a replacement obsolete, only pointers are needed. Additionally, this has the property of adding frames always reduces page faults (no Belady's anomaly).

[29] Example taken from the slides

There are also **LRU approximation** algorithms. One of them uses the **reference bit**, which is initially 0 and set to 1 when a page is referenced. Replaced pages have a reference bit of 0 (if such a page exists), but the order is unclear. The **second chance** algorithm uses the reference bit as well. If the page to be replaced (in clock order) has a reference bit of 1, then the reference bit is set to 0 and the page is left in memory and the next page (in clock order) is inspected using the same rules.

Frame allocation policies

Each process needs a minimum number of pages. There are two major allocation schemes: fixed and priority.

Fixed allocation uses either equal allocation where all process get an equal share, or proportional allocation where pages are allocated according to the size of the process.

Priority allocation uses a proportional allocation scheme but with priorities instead of size. If a process generates a page fault, one of its frames or a frame from a process with lower priority is selected.

Additionally, there's a distinction between **global and local** replacement. Using global replacement the process selects a replacement frame from the set of all frames which implies one process can take a frame from another. Using local replacement each process selects from only its own set of allocated frames.

Thrashing and working set

When a process is busy swapping pages in and out, it is **thrashing**. If a process does not have "enough" pages, the page fault rate is very high. This leads to low CPU utilization and the operating system thinking it needs to increase the degree of multiprogramming by adding another process to the system.

Demand paging is based on the **locality model** where a process migrates from one locality to another and localities may overlap. If the sum of localities is greater than the total memory size, thrashing occurs.

In the **working-set model** Δ is called the working-set window and is a fixed number of page references. The WSS_i (the working set of process p_i) is the total number of pages referenced in the most recent Δ (this value varies over time).

- If Δ is too small it will not encompass the entire locality
- If Δ is too large, it will encompass several localities
- If $\Delta = \infty$, it will encompass the entire program

$D = \Sigma\, WSS_i$ is the number of total demanded frames (which intuitively translates to how much space is really needed). If D is greater than m, thrashing occurs, thus as a policy, some process is suspended if that condition holds.

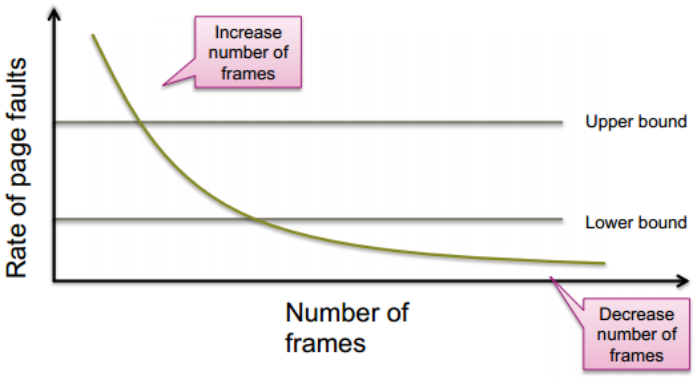
To keep track of the working set, it is approximated using an interval timer and a reference bit. In an example situation where $\Delta = 10{,}000$, this looks as follows[30]:

- Timer interrupts after every 5000 time units

[30] This is not completely accurate: improvement = 10 bits and interrupt every 1000 time units

- Keep in memory 2 bits for each page
- Whenever a timer interrupts shift + copy and sets the values of all reference bits to 0
- If one of the bits in memory = 1 ⇒ page in working set

In a **page-fault frequency scheme** an "acceptable" page fault rate is established. When the actual rate is too low, the process loses frames and consequently when the actual rate is too high, the process gains frames (*pictured*).



7 File System Abstractions

The file system (FS) is a virtualization of the disk. While the disk uses blocks to store data, the programmer's abstraction is files. The FS combines multiplexing and emulation and is general a part of the OS' core. It needs to provide:

GOAL	PHYSICAL CHARACTERISTIC	DESIGN IMPLICATION
HIGH PERFORMANCE	High cost of I/O access	Organize placement: access data in large, sequential units Use caching to reduce I/O
NAMED DATA	Large capacity, persistent across crashes, shared between programs	Support files and directories with meaningful names
CONTROLLED SHARING	Device stores many users' data	Include access control metadata with files
RELIABLE STORAGE	Crashes occur during update	Transactions to make set of updates atomic
	Storage devices fail	Redundancy to detect and correct failures
	Flash memory wears out	Wear-levelling to prolong life

Filing System Interface

A very important property of a file for the FS is the **metadata** which is data *about* an object and not the object *itself*. Such metadata includes: name, location on disk, times of creation, last change, last access, ownership, access control rights, file type, file structure, arbitrary descriptive data (used for searching).

Naming[31]

Naming in computer systems in general is complex yet fundamental and can be seen anywhere (virtual memory, fs, internet …). Naming provides **indirection** which is fundamental in computer science as this well-known quote by David Wheeler states:

> *"All problems in computer science can be solved by another level of indirection"*[32]

[31] This is not constrained to OS but to CS in general.
[32] Which can be rephrased as "Any problem in computer science can be recast as a sufficiently complex naming problem"

The association between a name and a value is called a **binding** and bindings are often not immediately visible, since people miss it / don't know it exists or is conflated with creating the value itself. And sometimes bindings are explicit and are objects themselves.

A General Naming Model

A designer creates a **naming scheme** which is also known as a resolver and requires a **context**. A context makes a name useful or clear[33] and any naming scheme has to have at least one context.

1. Name space: what names are valid?
2. Universe of values: what values are valid?
3. Name mapping algorithm: what is the association of names to values?

An example naming scheme for the virtual address space consists of:

- Name space — virtual memory addresses (e.g., 64-bit numbers)
- Universe of values — physical memory addresses (e.g., 64-bit numbers)
- Mapping algorithm — translation via a page table
- Context — page table root

Globally-routable IPv4 addresses have only a single context[34] and ATM virtual circuit/path identifiers are in a local context (and thus only valid on a particular link/port) hence have many contexts.

Naming operations

As outlined in the previous section, **resolving** a name is a function of the name and the context. A resolution example is how a computer A can determine B's MAC address[35] given B's IP address:

- Name space — IP addresses
- Universe of values — Ethernet MAC addresses
- Mapping algorithm — ARP: the Address Resolution protocol

Managing **bindings** involves two typical operations, bind and unbind. While both need a name and a context parameter, unbind may not need the value. These operations may fail according to naming scheme rules. Such an example (which will be discussed further on a later section) is the Unix pair `$ ln` and `$ rm`.

Enumeration, which is not always available, returns all bindings or names in a context as a list. Such an example is `$ ls`[36].

Comparison, which compares two names, is in general impossible, also because it requires a definition of equality on objects. Equality may refer to the names themselves, whether they are bound to the same object, or whether they refer identical copies of one thing.

---

[33] E.g. "you" or "here"
[34] For the sake of the argument, assume this is the case.
[35] MAC, short for Media Access Control, will be discussed further in the networking summary.
[36] In Windows this is the `dir` command, but for the sake of simplicity I omit Windows commands also because Linux is just better "suited" for shell usage (I'm a Windows user myself…).

Naming policy alternatives

Assuming a single context, the question of how many values a name can have arises. If there is only one value for a name, the mapping is injective or 1-to-1 like license plates or virtual memory addresses. Otherwise there are multiple values for a name like in a phone book (people can have more than one number) or DNS names (which can return multiple A records)[37].

Conversely, there's also the question of many names for a value. There can be either only one name for each value (e.g. names of models of a car or IP protocol identifiers) or multiple names for the same value (e.g. a phone book where people share a phone line or URLs where multiple links lead to the same page).

Unique identifier spaces and stable bindings bind at most one value to a name and once created, bindings can never be changed. This is very useful since it always possible to determine the identity of two objects. Examples include SSN[38] and MAC addresses.

Types of lookup

The simplest type of lookup is a **table lookup** which is like a phone book. This is used in a lot of places:

- Processor registers are named by small integers.
- Memory cells are named by numbers.
- Ethernet interfaces are named by MAC addresses
- Unix accounts are named by small (16bit) numbers (userids)
- Unix userids are named by short strings
- Unix sockets are named by small integers

Alternatives are **recursive** (for pathnames) and **multiple** (search paths) lookups.

Default and explicit contexts, qualified names

The default/implicit context is supplied by the resolver and can either be constant/built-in or variable, depending on the current environment/state. If the context is explicit, it's supplied by the object and is either per object or per [qualified] name.

In an explicit per-name context, each names comes with its context (actually the *name* of the context) and the context and the name together are called a qualified name. The resolution process is recursive: the context is resolved to a context object and the name is resoled relative to the resulting context.

A few examples:

- **Constant default** DNS, where the context is the DNS root server. And the hosts file. And the nsswitch.conf file. And the WINS resolver. And …
- **Variable default** the current working directory (`$ pwd`)
- **Explicit per-object** (note: context reference is a name, called base name)
  `$ ssh –l htor spcl.inf.ethz.ch` or `$ dig @8.8.8.8 –q a spcl.inf.ethz.ch`
- **Explicit per-name** me@example.com, /var/log/syslog

---

[37] Please see the summary of the networking part for more information. Simply put, an A record is the master record of a DNS entry.
[38] Social security numbers (USA) or AHV (Switzerland)

Path names, naming networks, recursive resolution

As aforementioned, **recursive** resolution is used for pathnames. Pathnames can be written forwards (e.g. /home/lm/…) or backwards (lm.pikapp.ch). The recursion has to terminate either at a fixed, known context reference (the root) or at another name which names a default context (used for relative pathnames). The syntax gives clues, e.g. a leading "/" or a trailing ".".

So far names resolve to values (and value usually are names in a different naming scheme), but **soft links** are different since they resolve to other names in the *same* scheme. This is used for shortcuts (Windows), symbolic links (`$ ln-s`), and forwarding addresses (Internet, e-mail, snail mail).

Multiple lookup

Multiple lookup, also referred to as "search path", is a lookup where several context are tried in order. Examples include binary directories in Unix, resolving symbols in link libraries, and the `$PATH`.

Name Discovery

There are many options how to find a name in the first place, and they often reduce to another name lookup. Options include:

- Well-known
- Broadcast the name
- Query (Google search)
- Broadcast the query
- Resolve some other name to a name space
- Introduction
- Physical rendezvous

File System Introduction

The FS is a naming scheme providing directory (which is the namespace) operations: link, unlink, rename, and list entries.

Acyclic-graph directories allows for aliasing (two different names):

If a nodes deletes a list, there is a dangling pointer

Solutions:

- Backpointers, so we can delete all pointers (variable size records can be a problem)
- Backpointers using a daisy chain organization
- Entry-hold-count solution

This requires a new directory entry type

- Link – another name (pointer) to an existing file
- Resolve the link – follow pointer to locate the file

In a **general graph directory** there has to be a guarantee for no cycles. This can be enforced by:

- Allow only links to files and not directories
- Garbage collection (with cycle collector)
- Check for cycles when every new link is added
- Restrict directory links to parents: e.g., "." and "..", all cycles are therefore trivial

Access Control

The owner/creator of a file should be able to control what can be done by whom. Different types of access are: read, write, execute, append, delete, list. This produces a matrix of principals (columns) and rights (rows) which is difficult to store.

Row-wise: ACLs

- Access Control Lists: for each right, list the principals & store with the file
- Good: Easy to change rights quickly and scales to large numbers of files
- Bad: doesn't scale to large numbers of principals

Column-wise: Capabilities

- Each principal with a right on a file holds a capability for that right which is stored with principal, not object (file) and cannot be forged or (sometimes) copied
- Good: very flexible, highly scalable in principals and access control resources charged to principal
- Bad: revocation: hard to change access rights (need to keep track of who has what capabilities)

POSIX (Unix) Access Control simplifies ACL by assigning each file identifies 3 principals:

- Owner (a single user)
- Group (a collection of users, defined elsewhere)
- The World (everyone)

For each principal, file defines 3 rights:

- Read (or traverse, if a directory)
- Write (or create a file, if a directory)
- Execute (or list, if a directory)

Actually POSIX support full ACLs but this rarely used. Furthermore, Windows has very powerful ACL support including arbitrary groups as principals, modification rights, and delegation rights.

Concurrency

When concurrency is present in the system, the OS must ensure that, regardless of concurrent access, file system integrity is ensured. This requires careful design of file system structures, internal locking in the file system, and ordering of writes to disk to provide transactions. Additionally, mechanisms for users to avoid conflicts themselves have to provided, which are advisory locks (separate locking facility) and mandatory locks (write/read operations will fail). The granularity of locking can extend to the whole file, byte/records ranges, or write-protecting executing binaries.

Contrary to the file system, databases have a way better notions of locking between concurrent users and durability in the event of crashes. Records and indexed files have largely disappeared in favor of databases, yet file systems remain much easier to use (and are much faster).

8 File System Implementations

File Types

The answer to the question **whether a directory is a file** is both yes and no. Yes, since it's allocated on the file just like a file, and it has entries in other directories like a file. No, because users

can't be allowed to read/write to it (→corrupt file system data structures, bypass security mechanisms), and the FS provides a special interface (opendir, readdir …).

Directories can be implemented in different ways:

- **Linear list:** uses (file name, block pointer) pairs, is simple to program but lookup is slow for lots of files (due to linear scan)
- **Hash table:** uses a linear list with closed hashing allowing for fast name lookup, but there may be collisions and the table has a fixed size
- **B-tree:** uses a name index and stores the block pointers in the leaves. While complex to maintain, it scales well and is increasingly common.

In addition to directories, other file types are treated in a special way by the OS, such as executables, symbolic links, and file system data. Furthermore, text and binary can be distinguished and for a user-friendly implementation, there are e.g. "document" types which are then used to select default applications.

Unix also uses the file namespace for naming IO devices (/dev), named pipes (FIFOs), domain sockets, process control (/proc), and OS configuration and status (/proc, /sys) – (almost) everything is a file.

Executables are automatically recognized by most OSes and will then be loaded, linked dynamically, and executed in a process. Other files, e.g. script in Unix, can use "#!" to be registered as executable.

File system operations are (Unix-specific):

- Create and variants: mknod, mkfifo, ln –s …
- Change access control: chmod, chgrp, chown, setfacl …
- Read metadata: stat, fstat …
- Open: Operation: file → open file handle

As it is hinted above, contrary to the typical file operations (rename, stat, create, delete …) "open" creates an open file handle which is a different class of object and allows for reading and writing of file data.

Open File Interface

There are three different kind of files: byte sequence (the average-Joe file), record sequence (fixed (at creation time) record; mainframes/minicomputers some decades ago), and key-based/tree structured (mainframe, nope superseded by databased; moved to libraries).

A **byte sequence file** is a vector of files which can be appended to, truncated, updated in place, but typically not inserted to. Access used to be sequential, nowadays it's random. **Random access** supports read, write, seek (absolute or relative to current position), and tell (returns current index). The state it keeps is the current position in file (the unit of the index for byte sequence files is bytes).

In **record sequence files** the vector allows for the same operations as in a byte sequence file, but the vector is of fixed-size. Record size (and perhaps format) are fixed at creation time. Read/write/seek operations take records and record offsets instead of byte addresses.

Memory-mapped files use the virtual memory system to cache files by mapping the file content into virtual address space. The backing store of region is set to the file and the file can now be accessed using load/store. When memory is paged out, updates go back to the file instead of swap space.

On-disk data structures

Disk addressing isn't an easy task – disks have tracks, sectors, spindles, and a lot more and also bad sector maps. This is made much more convenient by the use of **logical block addresses** which teat disk as compact linear array of usable blocks (the block size typically 512 bytes) and, except for performance, ignore geometry. This abstractions is also used for flash drives, storage-are networks, and virtual disks (RAM, RAID). For implementation, a few aspects have to be considered:

- Directories and indexes where on the disk is the data for each file?
- Index granularity what is the unit of allocation for files?
- Free space maps how to allocate more sectors on the disk?
- Locality optimizations how to make it go fast in the common case

An overview of different file system implementations:

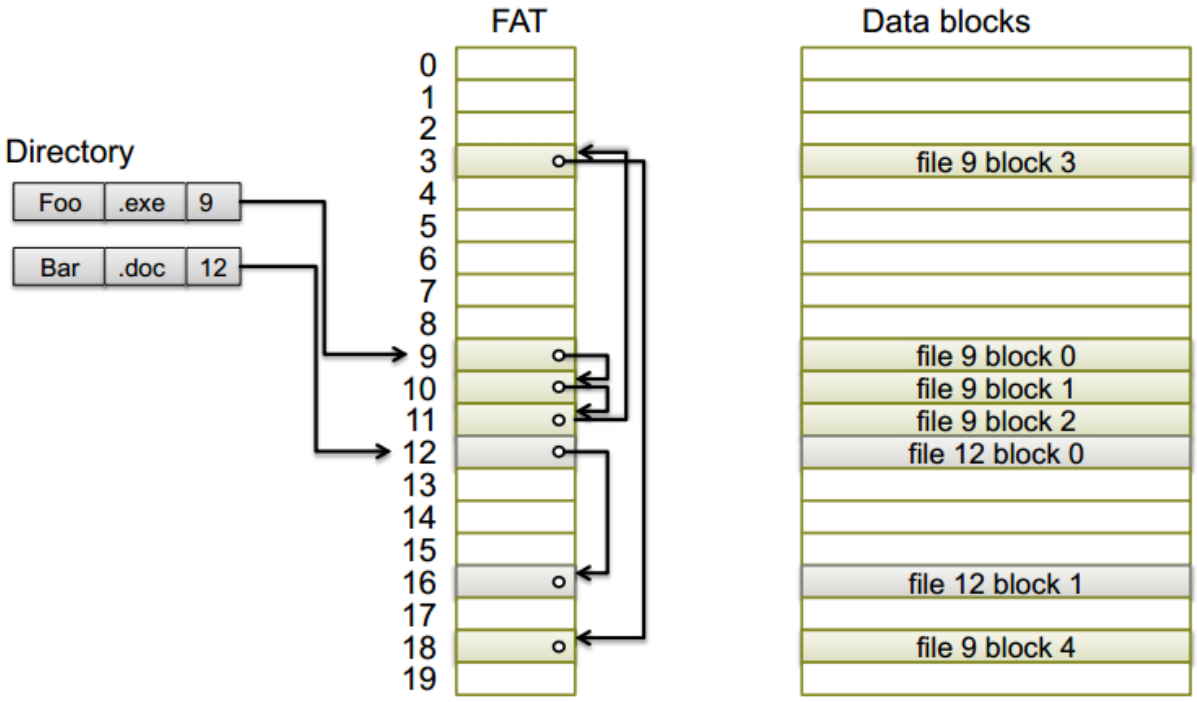
	FAT	FFS	NTFS	ZFS[39]
INDEX STRUCTURE	Linked list	Fixed, asymmetric tree	Dynamic tree	Dynamic COW tree
INDEX GRANULARITY	Block	Block	Extent	Block
FREE SPACE MANAGEMENT	FAT Array	Fixed bitmap	Bitmap in file	Log-structured space map
LOCALITY HEURISTICS	Defragmentation	Block groups, Reserve space	Best fit, Defragmentation	Write anywhere, Block groups

FAT-32

FAT, which stands for **File Allocation Table**, is very old, has no access control, very little metadata, limited volume size, no support for hard link, yet is still extensively used.

- Free space: linear search through FAT
- Slow random access: need to traverse linked list for file block
- Very little support for reliability: lose the FAT and it's game over
- Poor locality: files can end up fragmented on disk

[39] Not covered in class.



FFS

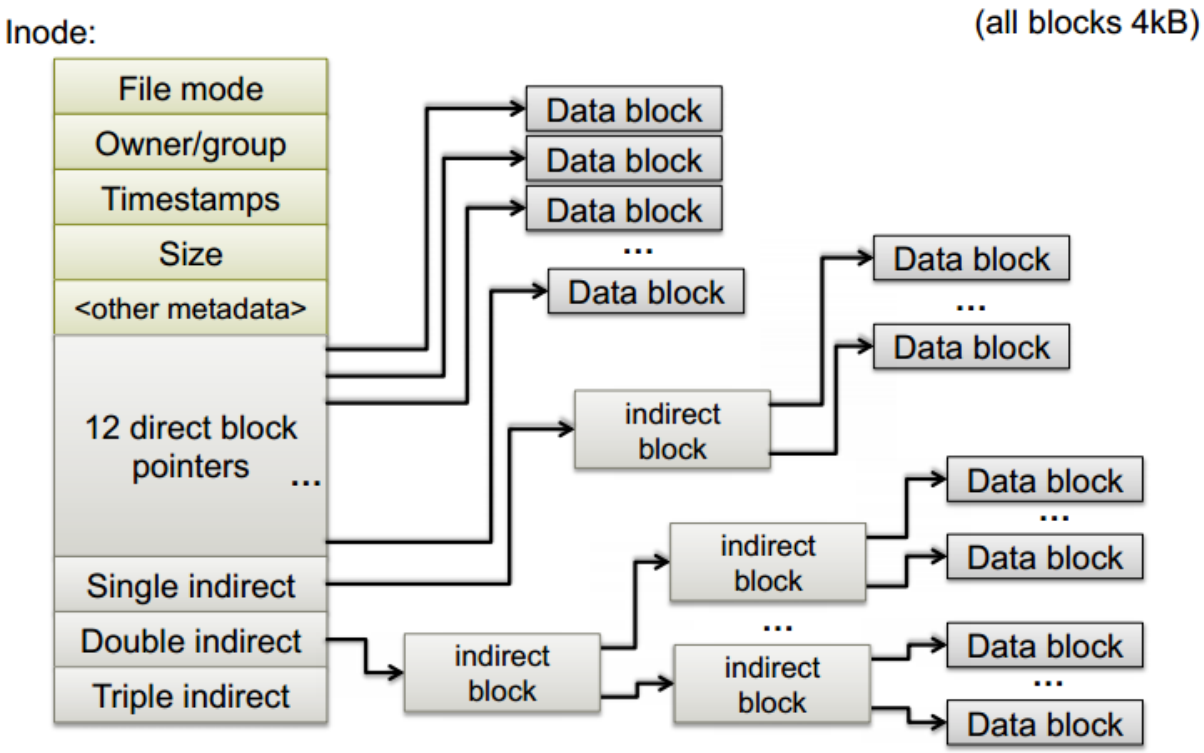
FFS, which stands for Unix **Fast File System**, uses indexed allocation. Every file is repsented by an index block or **inode** which contains the file metadata, a list of blocks for each part of file, and a directory contains pointers to inodes. Inodes and file sizes are related. Example: the inode is 1 block = 4,096 bytes, the block addresses are 8 bytes, and the inode metadata is 512 bytes. Hence there are $(4096 - 512)/8 = 448$ block pointers and $448 \cdot 4096 = 1792$ kB is the maximum file size. The file size can be extended using indirect blocks.

- Very small files: fit data straight into inode in place of pointers
- Very fast random access for files which fit in a single inode
- Very large files: tree keeps random access efficient

The free space map in FFS is a simple bitmap which is initialized when the file system is created and contains one bit per disk (file system) block. Allocation is reasonably fast, can scan through lots of bits at a time, and the bitmap can be cached in memory.

Block groups:

1. Optimize disk performance by keeping together related: files, metadata (inodes), free space map, directories
2. Use first-fit allocation within a block group to improve disk locality
3. Layout and block groups defined in the superblock; Replicated several times.



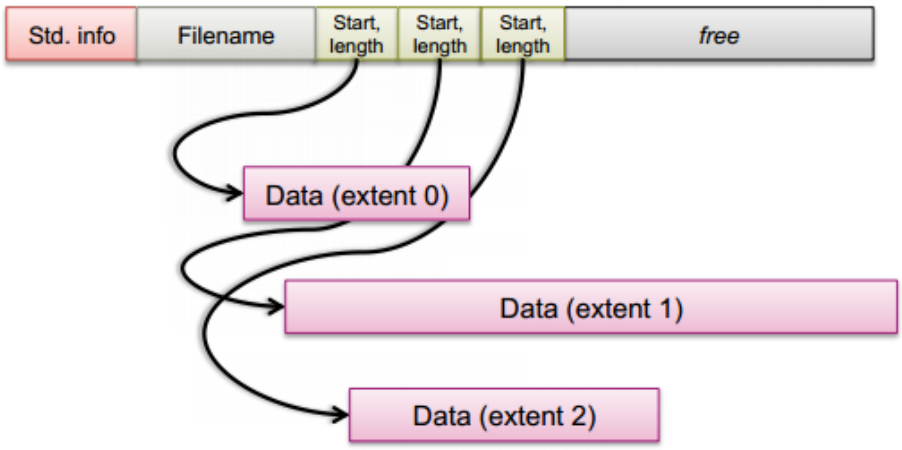
NTFS

The master file table contains a lot of MFT records which are of 1 kB fixes size, with standard info at the beginning, followed attributes, data, and metadata (and there are lots of options for what goes in there)

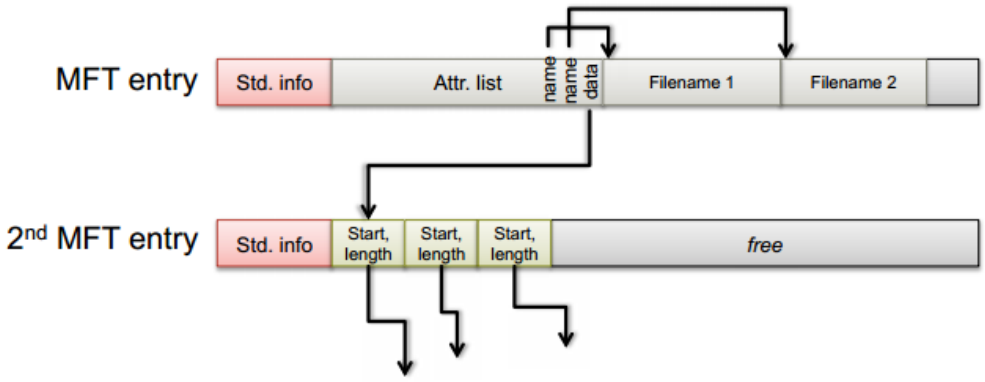
- Small file fits into MFT record

- Hard links (multiple names) stored in MFT

- MFT holds list of extents



- Attribute list holds list of attribute locations



Metadata in NTFS is held in files. The following table provides some insight:

FILE NUM.	NAME	DESCRIPTION
0	$MFT	Master file table[40]
1	$MFTirr	Copy of first 4 MFT entries
2	$Logfile	Transaction log of FS changes
3	$Volume	Volume information & metadata
4	$AttrDef	Table mapping numeric IDs to attributes
5	.	Root directory
6	$Bitmap	Free space bitmap
7	$Boot	Volume boot record
8	$BadClus	Bad cluster map
9	$Secure	Access control list database
10	$UpCase	Filename mappings to DOS
11	$Extend	Extra file system attributes (e.g. quota)

In-memory data structures

When opening a file, directories are translated into kernel data structures on demand. When reading and writing, there's a per-process open file table (which stores indexes) and the system open file table which caches inodes.

The efficiency is dependent on disk allocation, directory algorithms, and the types of data kept in file's directory entry. The performance depends on:

- disk cache – separate section of main memory for frequently used blocks
- free-behind and read-ahead – techniques to optimize sequential access
- improve PC performance by dedicating section of memory as virtual disk, or RAM disk

A **page cache** caches pages rather than disk blocks using virtual memory techniques. Memory-mapped I/O uses a page cache while routine I/O through the file system uses the buffer (disk) cache. This leads to two different caches which are then merged into a unified cache.

Recovery can work in different ways:

[40] The first sector of the volume points to the first block of the MFT

- Consistency checking: compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Disks, Partitions and Logical Volumes

Partitions are used to multiplex the disk among more than one FS. Each FS has a contiguous range of blocks assigned.

A **logical volume** emulates one virtual disk from more than one physical disks and uses a single FS spanning over all physical disks.

Different file systems use different naming for files, e.g. Windows uses letters on the top-level which is problematic. A more flexible approach are mount points (e.g. in Unix).

Virtual File Systems (VFS) provide an object-oriented way of implementing file systems and allows for the same system call interface (the API) to be used for different types of file systems. The API is to the VFS interface, rather than any specific type of file system.

9 I/O Subsystem I

What does a device look like?

To an OS programmer, a device is a piece of hardware visible from software occupying some location on a **bus**. It also has a set of **registers** (which are memory mapped or in IO space) and is a source of **interrupts**. It also may initate **Direct Memoy Access** transfers.

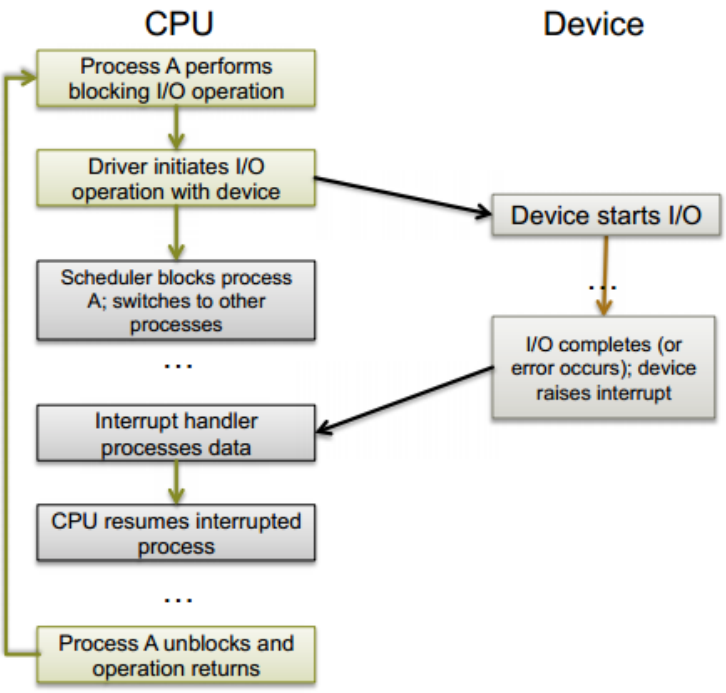
The details of **registers** are given in chip “datasheets” or “data books” (this information is rarely trusted by OS programmers). A very simple UART (Universal asynchronous receiver/transmitter) driver might be using programmed IO (PIO):

- CPU explicitly reads and writes all values to and from registers
- All data must pass through CPU registers

And uses polling:

- CPU polls device register waiting before send/receive
- Can’t do anything else in the meantime
- Without CPU polling, no I/O can occur

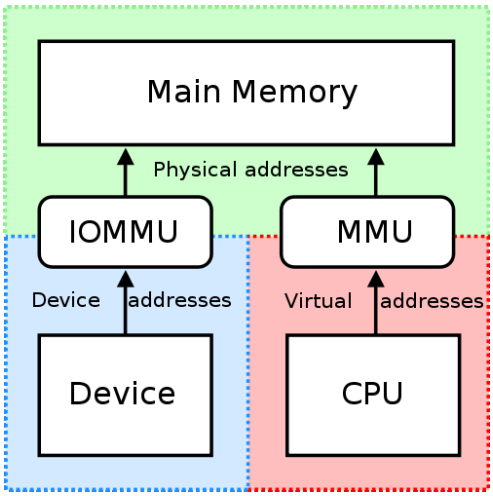
The CPU **interrupt**-request line is triggered by I/O device (*pictured: Interrupt-Driven I/O Cycle*) and the interrupt handler receives interrupts (which is maskable to ignore or delay some interrupts). The interrupt vector is used to dispatch interrupt to correct handler (based on priority and might be nonmaskable). The interrupt mechanism also used for exceptions.



Direct Memory Access is used to avoid programmed I/O for lots of data (e.g. fast network or disk interfaces). This requires a DMA controller (which is generally built-in) and this bypasses the CPU to transfer data directly between I/O device and memory thus not taking up CPU time and might save memory bandwidth and there's only one interrupt per transfer.

I/O Protection is important since I/O operations can be dangerous to normal system operation. Dedicated I/O instructions are usually privileged and in case I/O performed via system calls, register locations must be protected. Furthermore, DMA transfers must be carefully checked since they might bypass memory protection. This can happen when multiple operating systems are on the same machine (e.g., virtualized). This is where IOMMUs come into play.

IOMMU does the same for the I/O devices as MMU does for the CPU, it translates device addresses (so called DVAs) into physical ones by using an IOTLB and it works for DMA-capable devices (*pictured)*. They also add security features for VMs by allowing to assign different devices to different address domains and using address remapping these domains can be isolated from one another, thus sandboxing untrusted devices. IOMMUs were designed for enhancing virtualization and the remapping & security features can be applied to guest virtual machines providing better performance than software-based I/O virtualization.



OMMUs take as the input request the following IDs:

- Bus ID, stored in root tables (support for multiple buses),
- Device ID, stored in context tables (support for multiple devices within each bus)
- Function ID, also stored in context tables (support for multiple func. within each device)

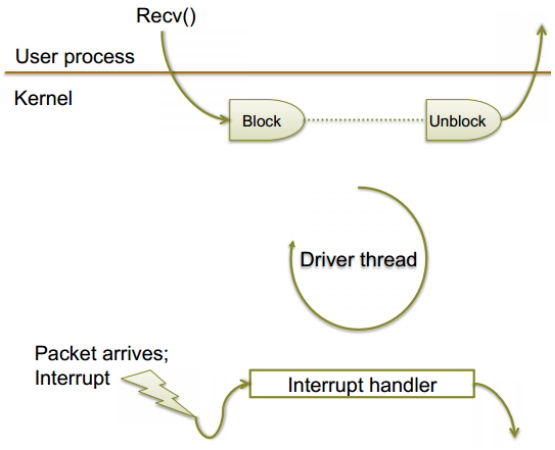
A different page table is used per I/O device and IOMMUs support page remapping. IOMMU Page Tables are similar to "normal" multi-level page tables including write-only/read-only bits and

support for huge pages. At the moment however, there's no support for more extended features (e.g., reference bits).

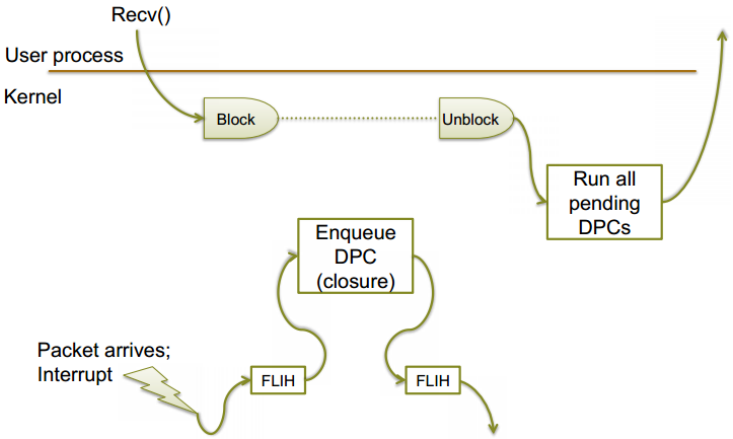
Device drivers

A **device driver** is a software object abstracting a device. It sits between hardware and rest of OS, it understands device registers, DMA, and interrupts, and it presents uniform interface to the rest of the OS. The abstraction (i.e. driver models) may vary a lot. The reason is the following problem: Hardware is *interrupt driven* (i.e. the system must respond to unpredictable I/O events or events it is expecting, but doesn't know when) while application oftentimes are *blocking* (i.e. process is waiting for a specific I/O event to occur), and often there's considerable processing (e.g. TCP/IP processing, retries, etc. or file system processing, blocks, locking, etc.) *in between*. The solution are **driver threads** (*pictured*).

1. **Interrupt handler**
   i. Masks interrupt
   ii. Does minimal processing
   iii. Unblocks driver thread
2. **Thread**
   i. Performs all necessary packet processing
   ii. Unblocks user processes
   iii. Unmasks interrupt
3. **User process**
   i. Per-process handling
   ii. Copies packet to user space
   iii. Returns from kernel



Another solution are **deferred procedure calls** (*pictured*) which instead of using a thread, execute on the next process to be dispatched before it leaves the kernel. This solution is used in most versions of Unix, it doesn't need kernel threads, saves a context switch, but can't account processing time to the right process.

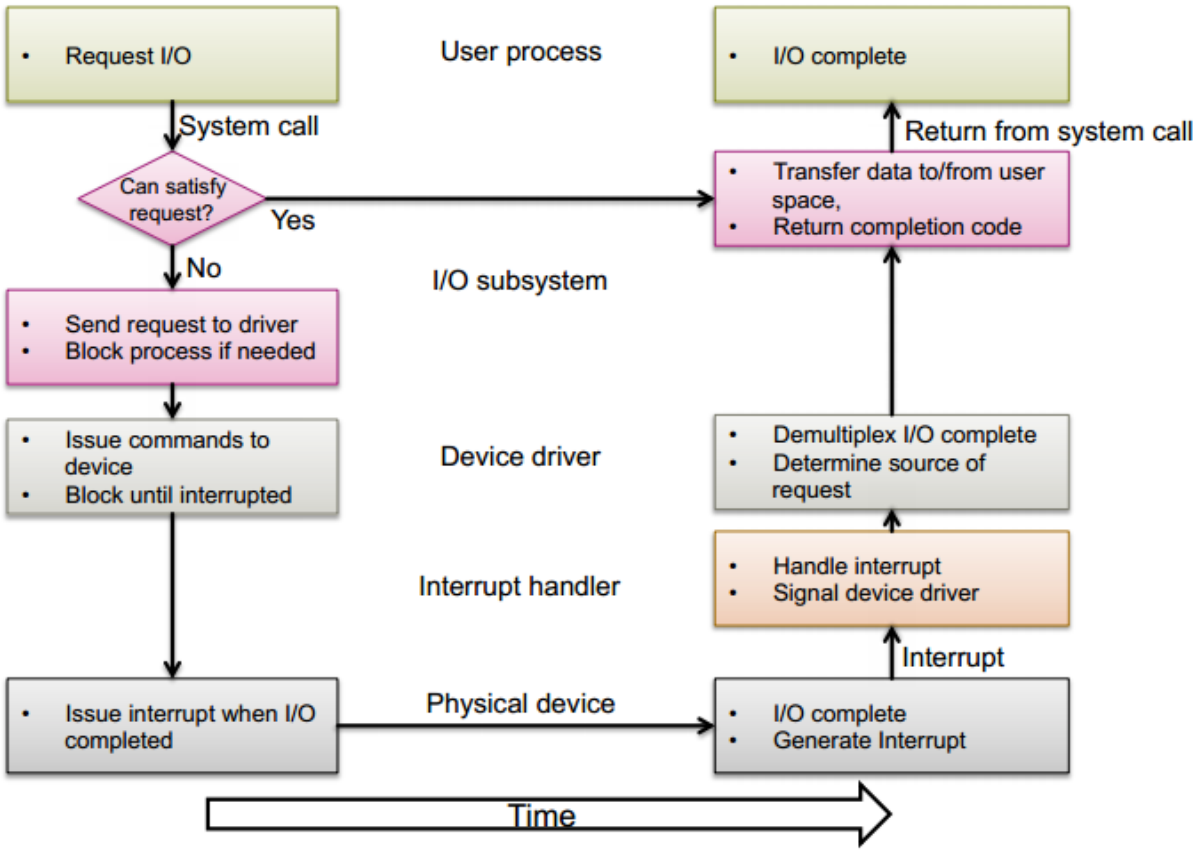


Some **terminology**:

- 1st-level Interrupt Handler (FLIH): Linux calls this the "top half"[41]
- DPCs are also known as: 2nd-level interrupt handlers, soft interrupt handlers, and slow interrupt handlers; in Linux ONLY: bottom-half handlers
- Any non-Linux OS (the way to think about it): bottom-half = FLIH + SLIH, called from "below", top-half = Called from user space (syscalls etc.), "above"

[41] In contrast to every other OS on the planet.

The lifecycle of an IO request:



The I/O subsystem

Essentially, device drivers move data to and from IO devices, abstract the hardware, and manage asynchrony. The IO subsystem includes generic functions for dealing with this data:

- **Caching:** fast memory holding copy of data which is always just a copy and the key to performance
- **Spooling:** hold output for a device if device can serve only one request at a time (e.g. printer)
- **Scheduling: s**ome I/O request ordering via per-device queue – some OSs try fairness
- **Buffering**: store data in memory while transferring between devices or memory. This is to cope with device speed mismatch, with device transfer size mismatch, and to maintain "copy semantics"

In terms of **naming and discovery**, the OS needs to manage discovery (bus enumeration), hotplug/unplug events, and resource allocation. Since a driver instance doesn't equal a driver module (one driver typically manages many models of devices), devices have to be named. To do so, devices have a unique (model) identifier and drivers recognize particular identifiers. The kernel then offers a device ton each driver and the driver can claim a device it can handle and creates a driver instance. To name devices *in the kernel* (or rather driver instances) in Unix, the kernel creates identifiers for block, character, and network devices with a major and a minor device number, where the major number is the class of device (disk, CD, keyboard …) and the major number specifies a device within a class.

Block devices are used for "structured I/O" which deal in large "blocks" of data at a time and often look like files (seekable, mappable; often use Unix' shared buffer cache), and are also mountable (file systems are implemented above block devices).

Character devices are used for "unstructured I/O" using a byte-stream interface (no block boundaries) and only consist of a single character or short strings. Buffering implemented by libraries. Examples: keyboards, serial lines, mice.

Deices *outside the kernel* are represented as a special file type, created with mknod() (the inode stores the type, and the major and minor numbers), and are typically stored in /dev. Unix also supports pseudo-devices which have no hardware but have major/minor device numbers such as /dev/null.

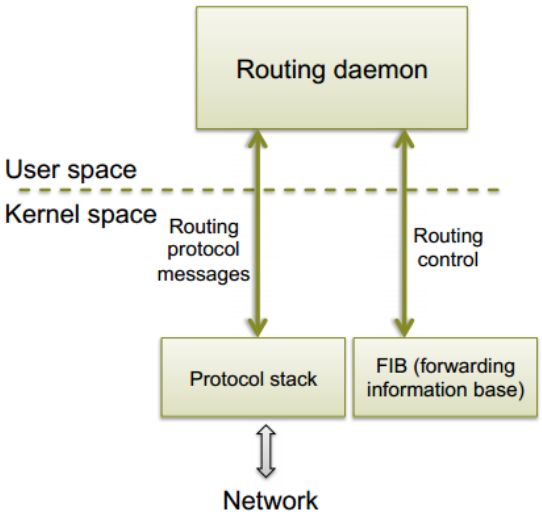
Old-style Unix device configuration
- All drivers compiled into the kernel
- Each driver probes for any supported devices
- System administrator populates /dev
- Manually types mknod when a new device is purchased!
- Pseudo devices similarly hard-wired in kernel

Linux device configuration today
- Physical hardware configuration readable from /sys using a special fake file system: sysfs and plug events delivered by a special socket
- Drivers dynamically loaded as kernel modules (the initial list given at boot time) and a user-space daemon can load more if required
- /dev populated dynamically by udev and the user-space daemon which polls /sys

10 I/O Subsystem II

Interface to network I/O

The OS protocol stacks include routing functionality. The routing protocols are typically in a user-space daemon since they are non-critical and easier to change that way. Forwarding information typically happens in the kernel because it needs to be fast and is integrated into protocol stack. (*pictured*)



Network stack implementation

Receiving and sending a packet in BSD

RECEIVING	SENDING
1. Interrupt	1. Application
- Allocate buffer	- Copy from user space to buffer
- Enqueue packet	- Call TCP code and process
- Post S/W interrupt	- Possible enqueue on socket queue
2. S/W Interrupt	2. S/W Interrupt
- High priority	- Any process context
- Any process context	- Remaining TCP processing
- Defragmentation	- IP processing
- TCP processing	- Enqueue on i/f queue
- Enqueue on socket	
3. Application	3. Interrupt
- Copy buffer to userspace	- Send packet
- Application process context	- Free buffer

While the TCP state machine is already pretty complex, the OS TCP state machine is even more complex because it also needs to handle: congestion control state (window, slow start, etc.), flow control window, retransmission timeouts etc. State transitions are triggered when (left column) and actions include (right column):

TRIGGERS	ACTIONS
- User request: send, recv, connect, close - Packet arrives - Timer expires	- Set or cancel a timer - Enqueue a packet on the transmit queue - Enqueue a packet on the socket receive queue - Create or destroy a TCP control block

In protocol graphs, nodes can be per-protocol (which handle all flows) or per-connection (instantiated dynamically) which support multiple interfaces (in addition to multiple connections) such as bridging (Ethernet ↔ Ethernet) or IP routing (IP ↔ IP).

Memory management

To ship a packet data around a structure is needed which can: (1) easily add and remove headers, (2) avoid copying lots of payload, (3) uniformly refer to half-defined packets, and (4) fragment larger datasets into smaller units. The solution is to hold data in a linked list of **buffer** structures.

Linux 3.x implements a simple protocol over Ethernet which allows to play with networking equipment and, allows for developing specialized protocols.[42]

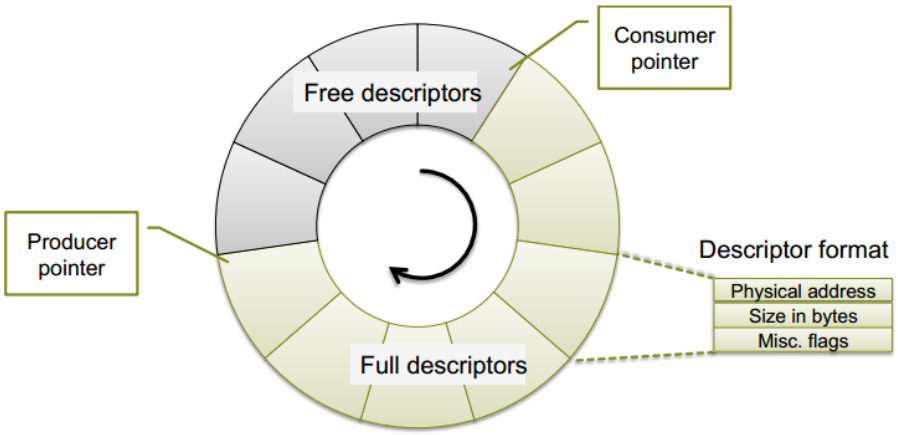
Performance issues

If the networking card has say a 10Gb port, this equals to roughly 1GB per second which is around 700k full-size Ethernet frames per second. If the processor was clocking at 2GHz, it has to process a packet in less than 3000 cycles, and that includes TCP/IP checksum, TCP windows calculations, TCP flow control, and copying the packet to user space. With a typical processor factoring in cache misses and interrupt latency, this will be an issue. Considering said card is full duplex i.e. also sends at 10Gb/s (i.e. two ports), there is barely any time left to do something useful with the packets. But wait, there's more: a 100 Gb/s card… which won't work without advanced features, such as:

- **TCP offload (TOE):** put TCP processing into hardware on the card
- **Buffering**: transfer lots of packets in a single transaction
- **Interrupt coalescing / throttling**: don't interrupt on every packet and don't interrupt at all if load is very high
- **Receive-side scaling**: parallelize: direct interrupts and data to different cores

The **Linux New API (NAPI)** mitigates interrupt pressure by switching between each packet interrupting the CPU and the CPU polling the driver. NAPI-compliant drivers offer a poll() function which calls back into the receive path. At that point, **buffering** comes into play by decoupling sending and receiving (neither side should wait for the other) by only using interrupts to unblock the host *and* batching together requests to spread the cost of transferring over several packets.

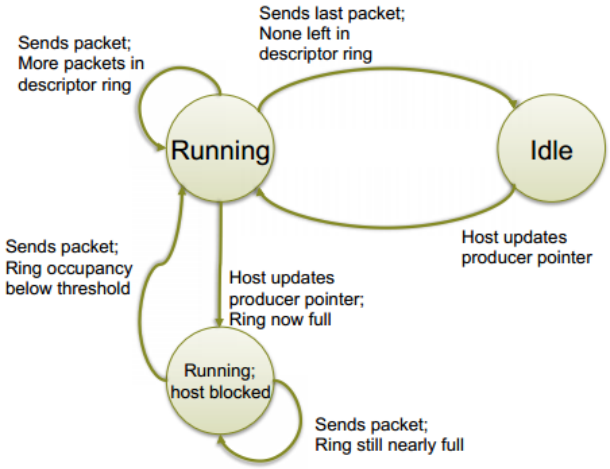
[42] Please refer to lecture slides 37 to 42 for more information.



When using buffering for network cards producer, consumer pointers are NIC[43] registers.

Transmit path
- Host updates producer pointer, adds packets to ring
- Device updates consumer pointer

Receive path
- Host updates consumer pointer, adds empty buffers to ring
- Device updates producer pointer, fills buffers with received packets.



The transmit interrupts are:

Ring empty
- all packets sent
- device going idle

Ring occupancy drops
- host can now send again
- device continues running

Summarizing buffering:

- DMA used twice: data transfer and reading/writing descriptors
- Similar schemes used for any fast DMA device: SATA/SAS interfaces (such as AHCI), USB2/USB3 controllers etc.
- Descriptors send ownership of memory regions
- Flexible – many variations possible:
  - Host can send lots of regions in advance
  - Device might allocate out of regions, send back subsets
  - Buffers might be used out-of-order
- Particularly powerful with multiple send and receive queues

Receive-side scaling is used when there's too much traffic for one core to handle[44]. They key idea is to handle different flows on different cores by demultiplexing on the NIC. This is done by using

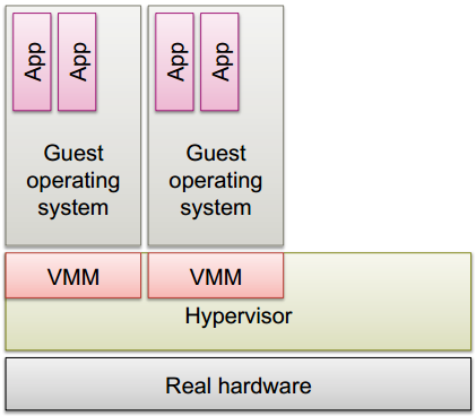
[43] Network interface controller
[44] And cores aren't getting any faster thus parallelization has to be used

DMA on packet to per-flow buffers/queues and send interrupting only the core handling the flow. This allows for balancing flows across cores[45] by assuming n cores processing m flows is faster than one core thus the network stack and protocol graph must scale on a multiprocessor.

11 Virtual Machine Monitors

Basic definitions

A **virtual machine monitor** (VMM)[46] virtualizes an entire (hardware) machine and provides an illusion of real hardware. The applications running on such monitors are called **guest operating systems**.



Why would you want one?

There are a lot of uses for a VMM:

- Server consolidation (program assumes it has its own machine, very useful for mostly idle machines)
- Performance isolation
- Backward compatibility
- Cloud computing (unit of selling cycles): hypervisors decouple allocation of resources (VM from provision of infrastructure (physical machines)
- OS development/testing: VMM often gives you more information about faults than real hardware anyway
- Something under the OS: replay, auditing, trusted computing, rootkits
- Running multiple OSes on one machine for application or backwards compatibility
- Etc.: tracing, debugging, execution replay, lock-step, execution, live migration, rollback, speculation …

Structure

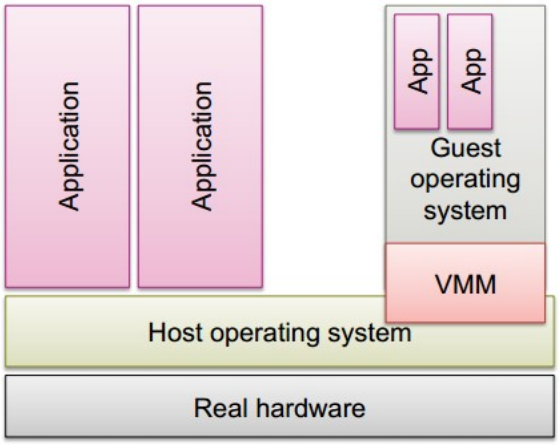
A hypervisor is basically an OS which has similarities in terms of multiplexing resource, scheduling, virtual memory, and device drivers but also differences since it creates the illusion of hardware and guest OSes are less flexible in resource requirements.

Hosted VMMs
- VMware workstation
- Linux KVM
- Microsoft Hyper-V
- VirtualBox

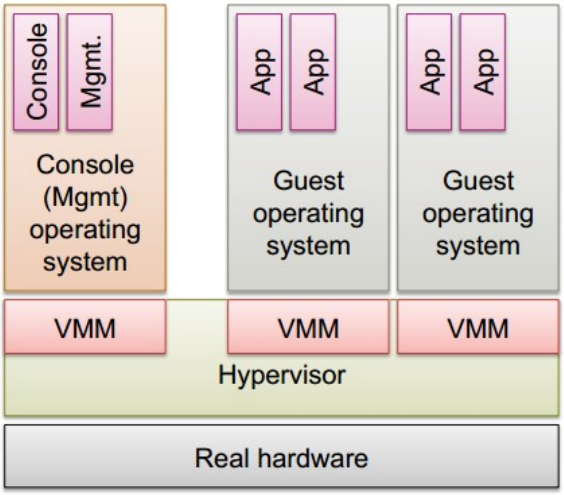
Hypervisor-based VMMs
- VMware ESX
- IBM VM/CMS
- Xen

[45] Note: this doesn't help with one big flow
[46] Sometimes a distinction between the VMM and the hypervisor is made.

Hosted VMMs



Hypervisor-based VMMs



Virtualizing the CPU

A CPU architecture is strictly virtualizable if it can be perfectly emulated over itself, with all non-privileged instructions executed natively. Privileged instructions lead to a trap which is then handled by the kernel (i.e. the VMM) handled and emulated – this implies the guest's kernel is actually in user mode. A strictly virtualizable processor can execute a complete native Guest OS where guest applications run in user mode as before and the guest kernel works exactly as before.

Unfortunately the x86 is not virtualizable, an example being then PUSHF/OPF instructions (push/pop condition code register; includes an interrupt enable flag IF). These instructions are unprivileged (hence fine in user space) but contrary to user mode, the IF isn't ignored when in kernel mode. Additionally, a VMM can't determine if the guest OS wants interrupts disabled and thus is unable to cause a trap on a privileged POPF and this prevents the guest OS from functioning correctly. The following solutions exist:

1. **Emulation**: emulate all kernel-mode code in software
   - Very slow – particularly for I/O intensive workloads
   - Used by, e.g., SoftPC
2. **Paravirtualization**: modify Guest OS kernel
   - Replace critical calls with explicit trap instruction to VMM
   - Also called a "HyperCall" (used for all kinds of things)
   - Used by, e.g., Xen
3. **Binary rewriting**
   - Very slow – particularly for I/O intensive workloads
   - Protect kernel instruction pages, trap to VMM on first IFetch
   - Scan page for POPF instructions and replace
   - Restart instruction in Guest OS and continue
   - Used by, e.g. VMware
4. **Hardware support**: Intel VT-x, AMD-V
   - Extra processor mode causes POPF to trap

Virtualizing the MMU

The hypervisor allocates memory to VMs and the guest then assumes control over all physical memory. The VMM can't let Guest OS to install mappings.

Terminology

- Virtual address: a virtual address in the guest
- Physical address: as seen by the guest
- Machine address: real physical address, as seen by the Hypervisor

MMU Virtualization is critical for performance, and also challenging to make it fast (especially for SMP). By hot-unplugging unnecessary virtual CPUs and using multicast TLB flush paravirtualizations etc. the MMU performance can be increased. Xen supports 3 MMU virtualization modes: (1) direct ("writable") page tables, (2) shadow page tables, and (3) Hardware Assisted Paging[47].

Using the **paravirtualization** approach, the guest OS creates page tables the hardware uses and the VMM then must validate all updates to page tables and modifications to the guest OS are required. Furthermore, the VMM must check *all* writes to PTEs by write-protecting all PTEs to the Guest kernel and adding a HyperCall to update PTEs. Batch updates are used to avoid trap overhead and the OS is now aware of machine addresses (huge overhead overall).

When **paravirtualizing** the MMU the guest OSes allocate and manage their own PTs and a Hypercall is available to change PT base. VMM must validate PT updates before use and this approach allows for incremental updates which avoids revalidation. The validation rules applied to each PTE:

1. Guest may only map pages it owns
2. Page table pages may only be mapped RO

The VMM traps PTE updates and emulates, or "unhooks" PTE page for bulk updates.

When **shadow page tables** are used, the guest OS sets up its own page tables (which aren't used by the hardware) and the VMM maintains shadow page tables. They map directly from guest VAs to Machine Addresses and the hardware switched whenever Guest reloads PTBR. The VMM must keep V→M table consistent with Guest V→P table and its own P→M table. To do so, the VMM write-protects all guest page tables and when a write happens, it traps and applies the write to shadow table as well (which is a significant overhead).

When relying on **hardware support** for nested page tables (which is relatively new in Intel/AMD) a two-level translation of addresses in the MMU is used where the hardware knows about: V→P tables (in the guest) and P→M tables (in the hypervisor). Tagged TLBs are used to avoid expensive flush on a VM entry/exit. While it's very nice and easy to code to, there's a significant performance overhead.

Virtualizing Memory

Allocating memory doesn't come without challenges such as the guest OS not expecting physical memory to change in size, or the hypervisor wanting to overcommit RAM, or the question of how to reallocate (machine) memory between VMs. A phenomenon called "Double Paging" can be observed where the hypervisor pages out memory, the guest OS decides to page out physical frame, and the (unwittingly) faults it in via the Hypervisor, only to write it out again.

Ballooning is a technique used to reclaim memory from a guest by installing a "balloon driver" in the guest kernel which, like any other part of the kernel, can allocate and free kernel physical memory and uses HyperCalls to return frames to the Hypervisor, and have them returned – and the guest OS is unware and simply allocates physical memory.

[47] OS paravirtualization compulsory for #1, optional (and very beneficial) for #2&3

Taking RAM away from a VM

1. VMM asks balloon driver for memory
2. Balloon driver ask guest OS kernel for more frames → "inflates the balloon"
3. Balloon driver sends physical frame numbers to VMM
4. VMM translates into machine addresses and claims the frames

Returning RAM to a VM

1. VMM converts machine address into a physical address previously allocated by the balloon driver
2. VMM hands PFN to balloon driver
3. Balloon driver frees physical frame back to guest OS kernel → "deflates the balloon"

Virtualizing Devices

Virtualizing devices also uses trap-and-emulate, protects memory and trap, and the "device model" is the software model of device in VMM. Interrupts cause upcalls to the guest OS and emulate interrupt controller (APIC) in the guest as well as emulate DMA with copy into Guest PAS.

For better performance, **paravirtualized devices** are used which are "fake" device drivers which communicate efficiently with VMM via hypercalls. This is mostly used for block devices like disk controllers or network interfaces.

Virtualizing the Network

To use networking in the guest OS, a virtual network device in the guest VM is used and the hypervisor implements a "soft switch" (entire virtual IP/Ethernet network on a machine). There are many different addressing options: separate IP addresses, separate MAC addresses, NAT etc.

There are four options where the real drivers are:

1. In the Hypervisor
   - Problem: need to rewrite device drivers (new OS)
   - E.g. VMware ESX
2. In the console OS
   - Export virtual devices to other VMs
3. In "driver domains"
   - Map hardware directly into a "trusted" VM
   - Device Passthrough
   - Run your favorite OS just for the device driver
   - Use IOMMU hardware to protect other memory from driver VM
4. Use "self-virtualizing devices"

The key idea behind **Single-Root I/O Virtualization is to** dynamically create new "PCIe devices" where VFs created/destroyed via PF registers. When used for networking the network card's resources are partitioned and using direct assignment passthrough can be implement.

Terminology

- Physical Function (PF): original device, full functionality
- Virtual Function (VF): extra "device", limited functionality

An example of a **self-virtualizing device** is that really fast network card from earlier[48] which can dynamically create up to 2048 distinct PCI devices on demand. The hypervisor can create a virtual NIC for each VM. The softswitch driver programs "master" NIC to demux packets to each virtual NIC and the PCI bus is virtualized in each VM. Each guest OS appears to have "real" NIC, talks direct to the real hardware.

[48] see page 41

12 Reliable Storage, NUMA & the Future

Reliable Storage

Two main things can go wrong: either the operation is interrupted in case of a crash or power failure. In that case transactions can be used to ensure data consistency, but this is not widely supported and requires very careful design of FS data structures. The other problem is data loss when a medium fails. In that case, redundancy can be used to tolerate the loss media, e.g. RAID.

One type of media failure is **sector and page failure** where the disk keeps working, but a sector doesn't, in which case writes don't work and reads are corrupted (page failure is the same just for flash memory). There are two approaches to this type of failure: (1) error correcting codes which encode data with redundancy to recover from errors (internal) or (2) remapping which identifies bad sectors and avoids them (internal or external).

Assume the nonrecoverable read errors per bits are 10^{-14}, then the chance a full 3TB disk could be read without errors is $(1 - 10^{-14})^{8 \cdot 3 \cdot 10^{12}} \approx 78.68\%$.

Another type of media failure is a **device failure** when the entire disk just stops working. This is always detected by the OS and since it's an explicit failure, less redundancy is required.

A bathtub curve can be observed: drives either fail very quickly (infant mortality) or after say 5 years when it simply wears out.[49]

RAID 1 is simple mirroring where data is written to both disks and read from either disk (which may be faster). If a sector or the whole disk fails, data can still be recovered.

In **RAID 5**[50] there's a parity block and striping is used for better performance and error recovery. At least three disks are required. Errors are always detected and parity allows for correction. In RAID 4 there's a

A storage system is:

Reliable if it continues to store data and can read and write it. Reliability is the probability it will be reliable for some period of time

Available if it responds to requests. Availability is the probability it is available at any given time

Caveats for sector/page failure

- Nonrecoverable error rates are significant
- Nonrecoverable error rates are not constant (affected by age, workload, etc.)
- Failures are not independent (correlation in time and space)
- Error rates are not uniform (different models of disk have different behavior over time)

Device failure is expressed as:

- Mean Time to Failure (**MTTF**) (expected time before disk fails)
- Annual Failure Rate = 1/MTTF (fraction of disks failing in a year)

Caveats for disk failure:

- Advertised failure rates can be misleading (depend on conditions, tests, definitions of failure…)
- Failures are not uncorrelated (disks of similar age, close together in a rack, etc.)
- MTTF is not useful life (annual failure rate only applies during design life)
- Failure rates are not constant (devices fail very quickly or last a long time)

[49] See https://www.backblaze.com/blog/ for great posts on hard drive reliability
[50] Note: in the lecture RAID 4 was covered which uses a designated parity disk; RAID 5 is much more used.

high overhead for small writes and the parity disk is accessed much more often.

Should a crash happen in the middle of updating data and parity, atomicity should be ensured. This can be achieved by (1) the use of a non-volatile write buffer, (2) transactional update to blocks, (3) a recovery scan, or (4) doing nothing.

Recovery in RAID 5 depends on the type of failure. If there's an unrecoverable read error on a sector, remap the bad sector and reconstruct its contents from stripe and parity. Should the whole disk fail, it needs to be replaced and is reconstructed from the other disks.

A RAID 5 can lose data in three ways:

1. Two full disk failures (second while the first is recovering)
2. Full disk failure and sector failure on another disk (most likely)
3. Overlapping sector failures on two disks

Terminology

- **MTTR** (mean time to repair): expected time from disk failure to when new disk is fully rewritten, often hours
- **MTTDL** (mean time to data loss): expected time until 1, 2 or 3 happens (on the left)

There are a few possible solutions:

- More redundant disks, erasure coding
- Scrubbing: regularly read the whole disk to catch UREs early
- Buy more expensive disks, i.e. disks with much lower error rates
- Hot spares: reduce time to plug/unplug disk

Hardware Trends

A very interesting thought is the following: assume a human can add two (implied: 64 bit) numbers in 1 second (compare: one core performs 8 floating point operations per cycle and a cycle takes 0.45 ns), then the following table shows how long other things take:

ITEM ACCESSED	COMPUTER TIME	HUMAN TIME
L1 CACHE	2.3 ns	5 s
L2 CACHE	10 ns	22 s
L3 CACHE	35 ns	78 s
LOCAL DRAM	70 ns	2.5 min
REMOTE CHIP	94 ns	3.5 min
REMOTE DRAM	107 ns	4 min
REMOTE NODE MEMORY	1 µs	37 min
SSD	100 µs	2.6 d
HDD	5 ms	8.3 m
INTERNET ZURICH – CHICAGO	150 ms	10.3 y
VMM OS REBOOT	4 s	277 y
PHYSICAL MACHINE REBOOT	30 s	2000 y

NUMA (non-uniform memory access) is becoming more and more important.[51] When implementing NUMA in OSes, memory is classified into NUMA nodes and create affinity to processors and devices since node-local accesses are fastest. The memory allocator and the scheduler should cooperate to schedule processes close to the NUMA node with their memory for increased performance. At the moment, different approaches can be observed:

[51] Even Windows 8 (and higher) supports NUMA nodes in the task manager

- Ignore it (no semantic difference)
- Striping in hardware (consecutive CLs come from different NUMA nodes): homogeneous performance, no support in OS needed
- Heuristics in NUMA-aware OS
  - "First touch" allocation policy: allocate memory in the node where the process is running (this can create big problems for parallel applications)
  - NUMA-aware scheduling: prefer CPUs in NUMA nodes where a process has memory
  - Replicate "hot" OS data structures: one copy per NUMA node
  - Some do page striping in software: allocate pages round robin (benefits unclear)
- Special NUMA control in OS
- Application control

How to compute fast? / Case study: OS for High-Performance Computing / IBM Blue Gene

Please refer to slides 46 – 62.

The key takeaway points are:

- Use a bypass for faster communication
- Reduce overhead

Super-short Summary[52]

Roles:
- Referee, Illusionist, Glue

Example: processes, threads, and scheduling
- R: Scheduling algorithms (batch, interactive, realtime)
- I: Resource abstractions (memory, CPU)
- G: Syscalls, services, driver interface

Slicing along another dimension:
- Abstractions
- Mechanisms

IPC and other communications
- A: Sockets, channels, read/write
- M: Network devices, packets, protocols

Memory Protection
- A: Access control
- M: Paging, protection rings, MMU

Paging/Segmentation
- A: Infinite memory, performance
- M: Caching, TLB, replacement algorithms, tables

Naming
- A: (hierarchical) name spaces
- M: DNS, name lookup, directories

File System
- A: Files, directories, links
- M: Block allocation, inodes, tables

I/O
- A: Device services (music, pictures)
- M: Registers, PIO, interrupts, DMA

Reliability:
- A: reliable hardware (storage)
- M: Checksums, transactions, raid 0/5

And everything can be virtualized!
- CPU, MMU, memory, devices, network
- A: virtualized x86 CPU
- M: paravirtualization, rewriting, hardware extensions
- A: virtualized memory protection/management
- M: writable pages, shadow pages, hw support, IOMMU

Sources

Unless otherwise noted: Lecture slides by Torsten Hoefler available on the course website accompanying the course 252-0062-00L taught in the spring semester 2015 at ETH Zürich. Simple definitions might be from Wikipedia.

[52] Taken from lecture 12