Computer Systems

Naming

- **Name:** identifier used to refer to an object in a system.
- **Binding:** association from name to object.
- **Context:** set of bindings.
- **Resolution:** find the object given a context and a name.
- **Dictionary:** object which act as a context. Table of bindings between name and object.
- **Naming network:** directed graph where nodes are either contexts or objects and edges are bindings from a context to another context.
- **Pathname:** path in a naming network.
- **Naming hierarchy:** naming network which is a tree.
- **Indirect entry:** name which is bound to another pathname. The pathname is resolved relative to the same context of the indirect entry.
- **Pure name:** name which does not include any useful information about the object it refers to.
- **Address:** name which encodes some information about the location of the object it refers to.
- **Search path:** ordered list of contexts treated as a single context.
- **Synonyms:** different names for the same object.
- **Homonyms:** same name bound to different objects.

Kernel

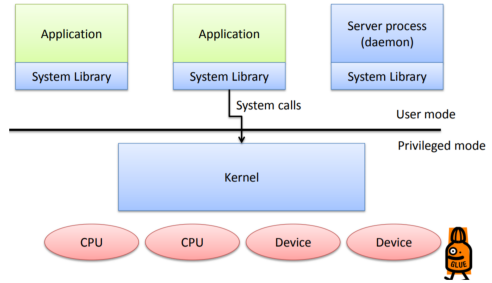
The OS is the part of software running on a machine which fulfils three roles:

- **Referee:** the OS multiplexes the hardware of the machine (CPU, cores, memory, ...) among users and programs. The OS also protects these principals from each other: they can't read each other's data (memory, file, ...) and using each other resources. The OS is a **resource manager**. Examples: scheduling, memory allocation, memory protection.
- **Illusionist:** the OS provides illusions of what the real hardware really is (e.g. the programmer assumes a very large memory). A very important tool to achieve this is virtualization. Examples: paging, virtual machine.
- **Glue:** the OS hides details of the hardware implementation to allow program portability. Examples: device access, program execution.

The OS has three key components:

- **Kernel:** part of the OS which operates in privileged mode. It is a computer program that responds to system calls, hardware interrupts and program traps.
- **System libraries:** special libraries which allows program running on the system to communicate with the kernel.
- **Daemon:** user-space process running as part of the OS.

General OS Structure

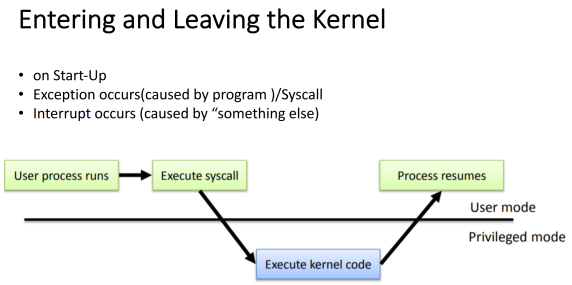


There are several ideal architectural models for kernels:

- **Monolithic kernel:** implements most of the OS functionalities inside the kernel (e.g. UNIX kernel).
- **Micro-kernel:** implements minimal functionality in the kernel (only memory protection, context switching and inter-process communication). The other OS functionalities (device drivers, file system pagers, ...) are executed in user-space processes. More robust to bugs and failures, difficult to evaluate performance.
- **Exokernel:** more functionalities on system libraries instead than user-space.
- **Multi-kernel:** runs different kernels on different cores of the machine.

Entering and leaving the kernel: mode transfer is the process of switching between kernel mode (privileged) and user mode (unprivileged). The kernel is entered from the user-space as a result of a processor exception (either synchronous trap or asynchronous fault) and the main goal is to protect the kernel from a potentially malicious user process. As with faults, when a user process executes a system call the processor starts executing in the kernel at a fixed point and the kernel has to figure out what happened (the kernel is an event driven program). Transferring from kernel mode to user mode happens when returning from a system call, when creating a new process or when switching from a process to another one. A system call is a trap (synchronous exception) deliberately invoked by a user program to request a service from the kernel.

Soel Micheletti



Execution state: every process has an execution state, i.e. the current values of user mode processor registers, stack pointer, program counter and page table address. When entering the kernel from user mode, the kernel saves the execution state of the currently running process so it can resume it later.
Bootstrapping: the process of starting the OS when the machine is powered on or reset up. When a processor is reset or first powered on it executes instructions in a fixed address in memory. The first part of code which is executed is the BIOS, which sets up a standard execution environment for the boot loader, a program which finds the OS kernel, puts it in memory and starts executing it. The OS kernel then starts the first process (called init in UNIX).

Processes

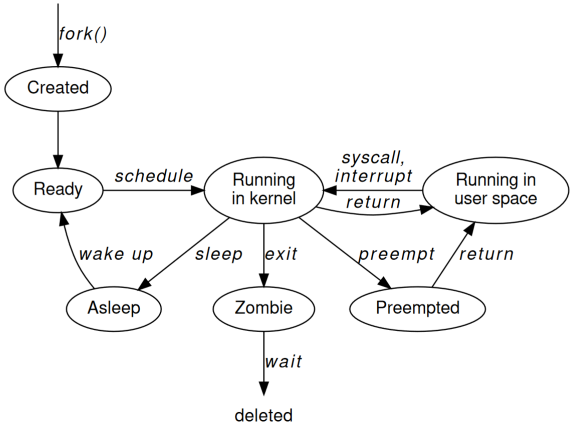
A **process** in an instance of a program. The OS protects processes from other processes in the computer. A process in a resource principal: it bundles a set of hardware and software together: some memory, some CPU cycles, ... A process can be thought of as a virtualized machine executing a program (however it would be a machine very different from the underlying hardware). Processes are identified with PID. The complete software running on the machine can be thought as the set of running processes plus the kernel. The ingredients of a process are the virtual processor (address space, registers, program counter, instruction pointer), the program text (code), the program data (heap and stack) and the OS stuff (open files, sockets, CPU shares, ...) The **execution environment** of a process is the virtual platform on which it executes: the virtual address space, available system calls...

When a process creates a new process we speak about parent and child processes. This creates a process tree. There are two main models to create processes:

- An OS spawns a child process by creating from scratch in a single operation, with a program specified by the parent.
- A fork operation, which creates a new child process as an exact copy of the parent. Child returns value of fork()=0 and parent returns value of fork()=PID of the child.

A process is in a given state. There are several states for processes:

- **Running:** the process is executing code, either in user or kernel mode.
- **Runnable:** the process can execute, but is not currently running.
- **Blocked:** the process is waiting for an event to occur before it can run.



Threads are a language abstraction which enable programmers to express parallelism in their programs by exploiting parallel hardware such as multiple cores. There are two types of threads:

- **User threads:** implemented in a user process. On a multiprocessor they can be multiplexed across multiple kernel threads to give a process true parallelism.
- **Kernel threads:** are implemented by the OS kernel directly and appear as different virtual processors to the user process.

Inter-process communication

Processes can communicate in two ways: shared memory or message passing.

- **Shared memory:**
  - Can be implemented with semaphores, locks and spinlocks (which are implemented with synchronization instructions). This is a pessimistic view: here is assumed that processes will probably have a conflict on read/ write to memory and hence only a single process can access the critical section at a time. This is done in order to make race conditions and inconsistencies impossible, but this gives bad performance in some cases.
  - Can be implemented in a more optimistic way: transactional memory. Here is assumed that conflicts are rare, so every process access the memory when it wants. When there is a conflict the transaction has to be undone and tried again.

  In order to coordinate communication with **shared memory**, there are some very important synchronization instructions.

Algorithm 5.2 Test-And-Set

```
1: inputs
2:     p {Pointer to a word in memory}
3: outputs
4:     v {Flag indicating if the test was successful}
5: do atomically:
6:     v ← *p
7:     *p ← 1
8: end do atomically
9: return v
```

Algorithm 5.3 Compare-and-Swap

```
1:  inputs
2:    p {Pointer to a word in memory}
3:    v1 {Comparison value}
4:    v2 {New value}
5:  outputs
6:    {Original value}
7:  do atomically:
8:    if *p = v1 then
9:      *p ← v2
10:     return v1
11:   else
12:     return *p
13:   end if
14: end do atomically
```

CAS is very powerful: every atomic operation can be simulated with CAS (but not with TAS).

Algorithm 5.4 Load-linked / Store Conditional (LL/SC)

Load-linked

```
1:  inputs
2:    p {Pointer to a word in memory}
3:  outputs
4:    v {Value read from memory}
5:  do atomically:
6:    v ← *p
7:    mark p as "locked"
8:  end do atomically
9:  return v
```

Store-conditional

```
10: inputs
11:   p {Pointer to a word in memory}
12:   v {Value to store to memory}
13: outputs
14:   r {Result of store}
15: do atomically:
16:   if *p has been updated since load-linked then
17:     r ← 1
18:   else
19:     *p ← v
20:     r ← 0
21:   end if
22: end do atomically
23: return r
```

Algorithm 5.8 TAS-based spinlock

```
1:  inputs
2:    p is an address of a word in memory
```

Acquire the lock

```
3:  repeat
4:    v ← TAS(*p)
5:  until v = 0
6:  ...
```

Release the lock

```
7:  *p ← 0
```

- **Message passing:** instead of sharing memory, processes can exchange information with each other by sending messages. This can be done in several ways, as shown in the figure below.

Asynchronous IPC is an example of asynchronous communication between processes. The sender does not block, the send operation returns immediately. If the receiver is not waiting for the message, the message is buffered until the receive call is made. The receive call blocks if there are no messages available. In **synchronous IPC** both the sender and the receiver may block until both are ready to exchange data.

Blocking is an orthogonal concept: blocking communication operations may block the calling thread; non blocking variants returns immediately with an indication that the operation should be retried.

There are three main message passing methods:

- **Unix pipes:** a fundamental IPC mechanism. A pipe is a unidirectional, buffered communication channel between two processes.
- **Upcalls:** so far every communication operation has involved the sender or receiver calling down the OS to perform an operation. An upcall is an invocation by the OS (usually the kernel) of a function inside a user process. This is basically the inverse of a system call.
- **Client server and RPC:** the client-server paradigm in distributed computing is composed by a server which offers a service and potentially multiple clients who connect to the server in order to invoke the service. An RPC is a programming technique that implements interaction from the original process to other processes as a simple procedure call. This procedure call (which is very useful to the programmer) is compiled in a particular way by the stub compiler, which automatically include the parts which are necessary to perform the communication.

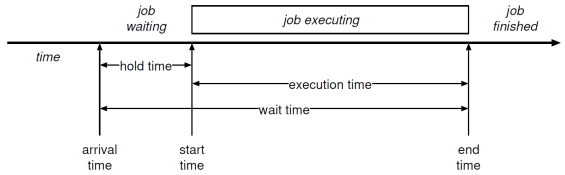
CPU SCHEDULING

CPU scheduling is the problem of deciding at any point in time and for every core, which process or thread is currently executing on that core. **Dispatching** refers to the mechanism of starting a particular process or thread on a particular core. Scheduling algorithms can be of different types:

- **Non-preemptive:** scheduler always allows a job to run to completion once it has started.
- **Preemptive:** processes are dispatched and descheduled without warning.

Let's consider first the task of scheduling tasks on a single processor core. We distinguish between different kinds of workloads (with different priorities and hence different scheduling algorithms):

- **Batch workload**: of a set of batch jobs, each of which runs for a finite length of time and then terminates. Batch scheduling is the problem of scheduling a set of batch jobs, which appear according to some arrival process. Main goals are throughput maximization and overhead minimization.



In this case, in order to estimate the performance of a scheduler we consider:

- **Throughput:** number of jobs the scheduler completes per unit time.

- **Overhead:** proportion of CPU time spent running the scheduler itself, as opposed to a client job. Overhead is the sum of the context switch time and the scheduling cost.

For batch workload we present the following algorithms (classified as preemptive or non-preemptive):

- Non-preemptive algorithms: a first straightforward scheduling algorithm is the following:

Algorithm 6.16 First-come-first-served (FCFS) scheduling

1: Assume each job P_i arrives at time t_i

When the scheduler is entered:

2: Dispatch the job P_j with the earliest arrival time t_j

This algorithm can lead to a **convoy phenomenon** when many short processes back up behind long running processes. We can remove this phenomenon with the next scheduling algorithm:

Algorithm 6.19 Shortest-Job First (SJF) scheduling

1: Assume each job P_i has an execution time of t_i seconds

When the scheduler is entered:

2: Dispatch the job P_j with the shortest execution time t_j

This algorithm is optimal in the sense that if all jobs are released at the same than the average waiting time for all jobs is minimized. A disadvantage of this algorithm is that it introduces some overhead (e.g. for sorting) and requires prior knowledge of the execution time of the job.

- Preemptive algorithms: the following algorithm is a slight modification of the shortest job first which takes advantage of the possibility of switching between processes.

Algorithm 6.22 SJF with preemption

When a new job enters the system or the running job terminates:

1: Preempt and suspend the currently running job (if there is one)
2: Dispatch (start or resume) the job P_j with the shortest execution time

The previous algorithm is however still problematic: new short jobs may extend the wait time of an already running longer job in an unacceptable way.

- **Interactive workloads:** wait for an external event and react before the user gets annoyed. Main goals: response time.

The most simple algorithm to manage interactive workload is the round robin scheduling.

Algorithm 6.25 Round-robin (RR) scheduling

1: Let R be a double-ended queue of runnable processes
2: Let q be the scheduling quantum (a fixed time period)

When the scheduler is entered:

3: Push the previously-running job on the tail of R
4: Set an interval timer for an interrupt q seconds in the future
5: Dispatch the job at the head of R

The scheduling algorithms we have seen until now use a FIFO queue. Instead we could have used a priority queue in order to get a **priority-based scheduling**. In this case processes with the same priority can still be scheduled with one of the previous approaches. Strict priority scheduling can lead to problems such as starvation (i.e. a low-priority process may starve because there are too many processes with higher priority). Instead of strict priority, most schedulers use **dynamic priority scheduling**, i.e. the priorities of tasks change over time in response to system events and application behaviour. A solution to starvation could be aging, i.e. tasks which are waiting gradually increase their priority until they get to the head of the queue. **Priority inversion** happens when a low priority process P_1 holds a lock for an high priority process P_h. In this case it can happen that P_h is descheduled and instead a medium priority process P_m begins running. There are two possible solutions to this issue:

- **Priority inheritance:** the low-priority process holding the lock required by the high priority process inherits its priority. This solves the priority inversion problem, but introduces overhead since the scheduler must know the lock graph of all the processes.

- **Priority ceiling:** a process holding a lock runs at the priority of the highest priority process which can ever hold the lock, until it releases the lock.

An important class of schedulers for interactive workload is called **multilevel feedback queue scheduler**: they aim to deliver good response for interactive jobs plus good throughput for background tasks. They penalize CPU bound tasks in favour of I/O tasks.

- **Soft real-time workloads:** this task should complete in less than a given time or must get a given fraction of CPU work. This is a priority, but the correctness does not depend on these timing constraint. Main goals: deadlines, guarantees, predictability.

- **Hard real-time workloads:** the correctness depends not only on the output, but also on the execution time (e.g. airplanes processes). Very difficult to achieve.

In the general case, hard real-time workloads is impossible. In practice the execution time of each task and the deadlines are known in advance and the system performs an admission control, where tasks for which a feasible schedule is not possible are discarded. A first (offline) approach is called **rate monotonic schedule** (it schedules periodic tasks by always running the task with shortest period first). This approach finds a schedule if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{\frac{1}{m}} - 1)$$

This approach is very efficient (it can lead to up 69% of utilization), but it does not work if we don't know the job mix in advance. An alternative solution is called **earliest deadline first** and it schedules sorted tasks by deadline (i.e. always runs the first deadline first). This is a dynamic approach and is online. It always finds a feasible schedule if:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

This approach can use potentially 100% of a processor, but it is unstable because its behaviour is unpredictable.

Until now we have considered a single processor and its workload. If we want to manage more than one processor things get more complicated (fully general multiprocessor scheduling is NP-hard). In the short overview of the problem in this course we consider that the system can always preempt a task and no processor is ever idle

when there is a runnable task. A simple solution would be to have a global queue and, whenever a processor is available it picks a task from the queue. To remove the bottleneck of a single run queue and to improve cache locality of running processes, affinity-job scheduling tries to keep jobs on one core as much as possible. In general things get much more complex when we consider jobs not as single threads, but as collection of parallel threads which coordinate among themselves (e.g. barriers present a significant challenge).

Input/ Output

Every OS has an **I/ O subsystem**, which handles all interaction between the machine and the outside world. The I/O subsystem presents a more or less uniform interface to integrate devices with the rest of the system (e.g. via low-level code to interface with individual hardware devices).

To an OS programmer, a **device** is a piece of hardware visible from software. It occupies some space on a **bus** and exposes a set of hardware registers which are either memory mapped or in I/ O space. A device is usually a source of interrupts and may initiate Direct Memory Access (DMA) transfers.

The **device driver** is the software part in the OS which *talks* to a particular device and abstracts properties of the device to the rest of the OS.

A **device register** is a physical address location which is used for communicating with a device using reads and writes.

Programmed I/O consists of causing input/ output to occur by:

- Writing data values from software to hardware registers.
- Reading values from hardware registers into CPU registers.

Algorithm 7.6 Programmed I/O input

```
1:  inputs
2:      l: number of words to read from input
3:      d: buffer of size l
4:  d ← empty buffer
5:  while length(d) < l do
6:      repeat
7:          s ← read from status register
8:      until s indicates data ready
9:      w ← read from data register
10:     d.append(w)
11: end while
12: return
```

An **interrupt** is a signal from a device to a CPU which causes the latter to take an exception and execute an interrupt handler.

Algorithm 7.8 Interrupt-driven I/O cycle

1 Initiating (software)

1: Process A performs a blocking I/O operation
2: OS initiates an I/O operation with the device
3: Scheduler blocks Process A, runs another process

2 Processing (hardware)

4: Device performs the I/O operation
5: Raises device interrupt when complete, or an error occurs

3 Termination (software)

6: Currently running process is interrupted
7: Interrupt handler runs, processes any input data
8: Scheduler makes Process A runnable again

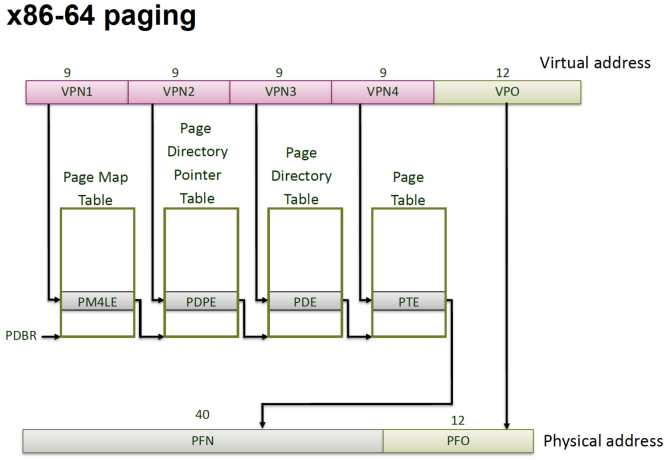
4 Resume (software)

9: Process A restarts execution.

Using **DMA**, a device can be given a pointer to buffers in main memory and transfer data to and from those buffers without further involvement from the CPU.

Memory management

A **base and limit register pair** is a couple of hardware registers containing two addresses B and L. A CPU access to an address a is permitted iff $B \leq a \leq L$. The base address is not known at compile time, hence programs had to be compiled to position-independent code with the use of relocation registers (i.e. an address a is relocated to $B + a$ and allowed iff $B \leq B + a \leq L$). This approach still has some disadvantages: there is no memory sharing between processes and there can be external fragmentation. A **segment** is a triple (I, B_I, L_I) where I is an identifier, while B_I and L_I are base and limit of segment I. A **segment table** is an in-memory array of base and limit values indexed by segment identifier. The MMU in a segmentation system holds the location and size of its table in a **segment table base register** and **segment table limit register**. Logical memory access causes the MMU to look up the segment id in this table to obtain the physical address and protection information. Here we allow sharing (e.g. if segments are allowed to overlap). The principal downside of this approach is that segments are still contiguous in memory and hence we can have a problem of external fragmentation.
Paging divides physical memory into fixed size frames and virtual memory into pages of the same size. The page table is used to map from virtual to physical memory (the results are first cached in TLB in order to improve performance). Page tables can be linear or hierarchical as shown in the next figure.



Paging solves the external fragmentation problem: process can always fit if there is available free memory. Each process has its own page table. At a high level, all operating systems provide three basic operations on page mappings:

- **A.map(v, p):** takes a virtual page number v and a physical page number p and create a mapping from v to p in the address space A.
- **A.unmap(v):** takes a virtual page number v and removes any mapping from v in the address space A.

- **A.protect(v, rights):** takes a virtual page number v and changes the page protection on the page. Examples of rights are **Readable**, WRITEABLE and EXECUTABLE.

When we call fork we create a child process identical to the parent. A naive idea would be to copy the whole address space of the parent such that both the parent and the child can work independently. However there is a cheaper way to fork a process. The parent and child process share the address space, which get READABLE protection. Whenever one of the two process want to read an address a protection fault is triggered. When this happens the single address is duplicated and gets WRITEABLE protection.

Algorithm 8.18 Copy-On-Write

inputs
  A_p {Parent address space}
  A_c {Child address space}
  $\{(v_i, p_i), i = 1 \dots n\}$ {A set of virtual to physical mappings in A_p}

Setup

for $i = 1 \dots n$ **do**
  A_p.protect(v_i, READONLY)
  A_c.map($v_i \rightarrow p_i$)
  A_c.protect(v_i, READONLY)
end for

Page fault in child

inputs
  V {Faulting virtual address }
$n \leftarrow$ VPN(V)
$p' \leftarrow$ AllocateNewPhysicalPage()
CopyPageContents($p' \leftarrow p_n$)
A_c.map($v_n \rightarrow p'$)
A_p.protect(v_n, WRITEABLE)
return

Let's see some software operations the OS can do on caches:

- **Invalidate:** marks the contents of the cache (or line) as invalid. This operation generally doesn't care whether there is dirty data in the cache (line): it just throws away everything.
- **Clean:** writes any dirty data held in the (write-back) cache to memory.
- **Flush:** writes back any dirty data from the cache and then invalidates the line.

In caches, there can be problems of homonyms and synonyms.

- Imagine two processes which have the same virtual address space, but the virtual addresses map to different physical addresses. If the cache is virtually indexed, if the word at address one is in the cache and then we switch to the other process, it is possible that if we want to read/ write the the words at address one for the second process, we refer to the copy in the cache (which refers to the first process). This is of course a problem and introduces inconsistencies. Two possible solutions are: flushing the cache on process switch; using ASIDs (i.e. the address space becomes part of the tag).
- Imagine two virtual addresses that map to the same physical address. The data contained in those address can be both cached at the same time in different parts of the cache. If we update the content of one of the two virtual addresses the cache becomes inconsistent (i.e. if we read the wrong copy in the cache the old value is returned). There are also problems if the content of the cache is written back to memory.

Some cache types are:

- **Virtually indexed, virtually tagged:** suffers from homonyms, because same virtual address mean different physical addresses in different processes. The advantage is that this approach is fast.
- **Physically indexed, physically tagged:** it's good for context switches, but it is slow since address needs to be translated before lookup.
- **Virtually indexed, physically tagged:** it is good if it works, but it is also complicated.
- **Physically indexed, virtually tagged:** not really useful.

DEMAND PAGING

Demand paging uses page faults to exchange virtual pages on demand between physical pages in main memory and locations in a larger page file held on cheaper persistent storage. In practice, the access time to fetch a virtual page from disk is much higher and hence demand paging is lazy: a page is loaded into memory only when a virtual address in the page has been touched by the processor.

Algorithm 9.2 Demand paging: page fault handling

On a page fault with faulting VPN v_{fault}:

if there are free physical pages **then**
  $p \leftarrow$ get_new_pfn()
else
  $p \leftarrow$ get_victim_pfn()
  $v_{old} \leftarrow$ VPN mapped to p
  invalidate all TLB entries and page table mappings to p
  if p is dirty (modified) **then**
    write contents of p into v_{old}'s area in storage
  end if
end if
read page v_{fault} in from disk into physical page p
install mapping from v_{fault} to p
return

In order to have good performance, the goal is to minimize page faults. The critical part of the previous algorithm is selecting a *good* page to evict: the page is chosen by the **page replacement algorithm**. The **page replacement policy** of a demand paging system is the algorithm which determines which physical page (the **victim page**) will be used when paging a virtual page in from storage.

The **effective access time** is the average time taken to access memory and is given by:

$$EAT = ((1 - p)m) + p(o + m)$$

where p is the probability of having a page fault, m is the memory access latency and o is the paging overhead (which considers factors such as page fault overhead, the cost of swapping a dirty virtual page out, the cost of swapping the required virtual page in and the cost of restarting the instruction).

In general we are seeking the optimal page replacement, i.e. we want to evict the page that will not be referenced again for the longest period of time.

A first replacement algorithm uses a simple FIFO queue:

Algorithm 9.8 FIFO page replacement

Return a new victim page on a page fault

```
1: inputs
2:    pq: a FIFO queue of all used PFNs in the system.
3: p ← pq.pop_head()
4: pq.push_tail(p)
5: return p
```

This algorithm exhibits the **Belady's anomaly**, i.e. it can happen that increasing the size of the cache actually reduces the hit rate for some reference strings.

Another famous algorithm (which does not present the Belady's anomaly) is LRU, which can be implemented with a stack:

Algorithm 9.11 A Least Recently Used (LRU) page replacement implementation

Initialization

```
1: inputs
2:    S: Stack of all physical pages p_i, 0 ≤ i < N
3: for all p_i do
4:    p_i.referenced ← False
5:    S.push(p_i)
6: end for
```

When a page p_r is referenced

```
7: S.remove(p_r)
8: S.push(p_r)
```

When a victim page is needed

```
9: return Return S.remove_from_bottom()
```

LRU needs detailed information about page references. The OS sees every page fault, but in general does not get informed about every memory access a program makes. In order to get this information we use the *flag* mechanism of the following algorithm:

Algorithm 9.12 2nd-chance page replacement using reference bits

Initialization

```
1: inputs
2:    F: FIFO queue of all physical pages p_i, 0 ≤ i < N
3: for all p_i do
4:    p_i.referenced ← False
5:    F.add_to_tail(p_i)
6: end for
```

When a physical page p_r is referenced

```
7: p_r.referenced ← True
```

What a victim page is needed

```
8: repeat
9:    p_h ← F.remove_from_head()
10:   if p_h.referenced = True then
11:      p_h.referenced ← False
12:      F.add_to_tail(p_h)
13:   end if
14: until p_h.referenced = False
15: return p_h
```

All the schemes seen so far operate on a single pool of physical pages and don't differentiate between processes or applications. In **global physical page allocation** the OS selects a replacement physical page from the set of all such pages in the system. Instead, with **local physical page allocation** a replacement physical page is allocated from the set of physical pages currently held by the faulting process.

The **working set** $W(t, \tau)$ of a process at time t is the set of virtual pages referenced by the process during the previous time interval $(t - \tau, t)$. The working set size is the cardinality of the working set: this value is a good approximation to how many pages of physical memory a process needs to avoid excessively paging pages to and from storage. The following algorithm is a way to estimate the working set size (note that for each table entry we keep a hardware-provided accessed bit u_0 and $K-1$ software maintained use bits; moreover an interval timer is programmed to raise an interrupt every σ time of units):

Algorithm 9.17 Estimating the working set by sampling

On an interval timer every σ time units

```
1: WS ← {}
2: for all page table entries do
3:    for all use bits u_i, 0 < i < K do
4:       u_i ← u_{i-1} {Shift all use bits right}
5:    end for
6:    u_0 ← 0
7:    U ← ⨁_{i=0}^{K-1} u_i {Logical-or all the use bits together}
8:    if U = 1 then
9:       WS.add( page )
10:   end if
11: end for
12: return WS {The working set W(t, σK)}
```

A paging system is **trashing** when the total working set significantly exceeds the available physical memory, resulting in almost all application memory access triggering a page fault. In those cases performance falls near to zero.

File System Abstractions

The **Filing system** is the functionality in an operating system which provides the abstractions of files in some system-wide namespace. It comprises the core functionality for accessing files, together with a set of additional maintenance utility programs which are usually not needed by regular users.

Access control is about deciding which access rights a given principal has on a given object. A **principal** is the entity to which particular access rights are ascribed, which grant to ability to access particular objects (e.g. users). The **protection domain** of a principal is the complete set of objects that the principal has some rights over. The **access control matrix** is a table whose rows correspond to principals and whose columns correspond to objects. Each element of the matrix is a list of the rights that the corresponding principal has over the corresponding object. An **access control list** is a compact list of non-zero entries for a column of the access control matrix, i.e. an encoding of the set of principals that have any rights over the object, together with what those rights are. This kind of lists make it easy to change rights on an object quickly.

A **file** is an abstraction created by the OS that corresponds to a set of data items with associated metadata.

The **metadata** of a file is additional information about a file which is separate from its actual contents (e.g. size, file type, access control information, time of creation, time of last update). It also includes where the file's data is located on the storage devices.
A **filename** is a name which is bound to a file in the namespace implemented by the filing system. More than one name can refer to the same file.
An **open file** is an object which represents the context for reading from, writing to and performing other operations on a file. An **access method** defines how the contents of a file are read and written. For example: direct access is an access method where any part of the file can be accessed by specifying its offset; sequential access is an access method under which the file is read strictly in sequence from start to end.
An **executable file** is one which the OS can use to create a process.

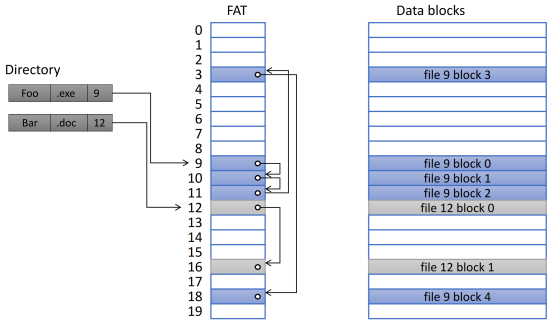
File System Implementation

A **volume** is the generic name for a storage device and consists of a set of fixed size blocks which can be read or written. Blocks in a volume are accessed with logical block address (LBA), i.e. the volume is considered as a linear array of blocks. It is possible to divide a single machine in more volumes or save a single volume on multiple machines (tradeoff durability vs. performance). **A file system** is an abstract data type which fill a volume and collectively provide file storage, naming and protection. File systems are classes which implements the interface of a file system. A **mount point** in a hierarchically named OS (like Unix), is a directory which is mounted on a complete other file name system. One file system, the root file system, sits at the top and all other file systems accessible in the name space are mounted in directories below this.
File system goals are:

- **Performance:** minimize time to open, read and write a file. Since many operations with file involve operations with data which are stored on disk (and doing operations with the disk is high latency), file system should aim to maximize locality in order to have a lot of sequential I/O operations and improve performance.
- **Reliability:** the key challenges are consistency and durability. Approaches are atomic operations, transactions with a log file...

FAT file system

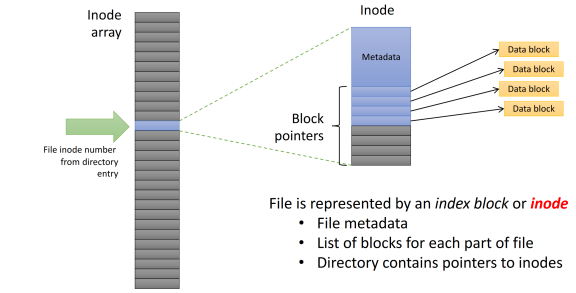
FAT file system

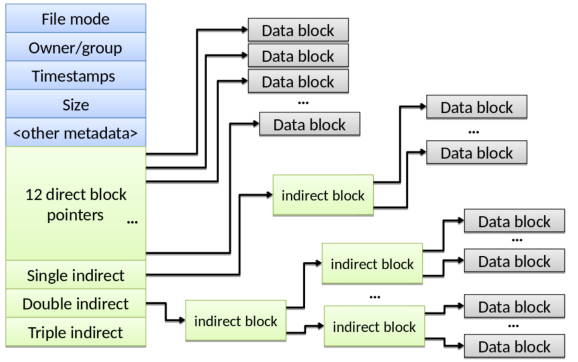


- No access control
- Very little metadata
- Limited volume size
- Does not support multiple names for the same file
- Used in flash devices, cameras, phones
- Slow random access: need to traverse the linked list for file block
- Very little support for reliability: lose the FAT and the game is over
- Poor locality: files can end up fragmented on disk
- Slow to allocate a new block (worst case $\mathcal{O}(n)$
- A directory is a simple table of entries, which give name, metadata and an index into the FAT.
- Each FAT entry marks the corresponding block on the volume as free or used.

FFS file system

FFS uses indexed allocation

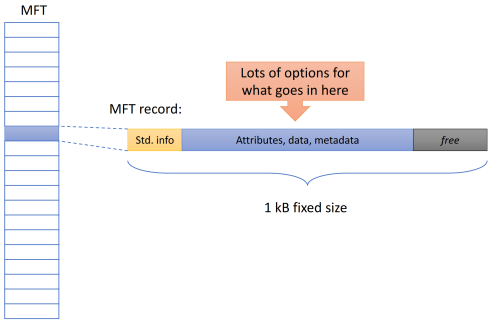




- An inode contains region of data on a disk that contains all necessary metadata for file. All names for a given file points to the right inode and all data blocks comprising the file are listed by the inode.
- FFS has an area of disk called the inode array which holds a table of complete inodes, indexed by number.
- Directories in FFS are lists of file names and corresponding inode numbers.
- An inode is typically 4096B large and metadata are 512B.
- There are some special indirects block which are used to store very large files.
- Free space on the volume is managed using a bitmap (one per disk block) that can be cached in memory.

NTFS file sytem

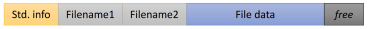
NFTS Master file table



NTFS small files

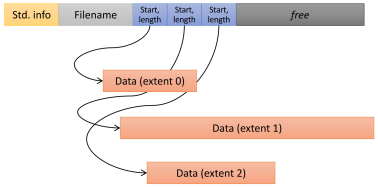
- Small file fits into MFT record:

- Hard links (multiple names) stored in MFT:



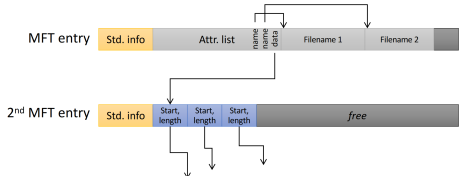
NTFS normal files

- MFT holds list of *extents*:



Too many attributes?

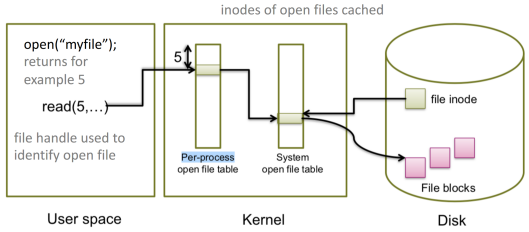
- Attribute list holds list of attribute locations



File system implementations

	FAT	FFS	NTFS
Index structure	Linked list	Fixed, assymetric tree	Dynamic tree
Index granularity:	Block	Block	Extent
Free space management	FAT Array	Fixed bitmap	Bitmap in file
Locality heuristics	Defragmentation	Block groups, Reserve space	Best fit, Defragmentation

Open Files



Network Stack

The **network stack** is the component of the OS which handles all network I/O, including packet transmit and receive, multiplexing, demultiplexing and other kind of protocol processing. Not all network protocols are handled by the OS: some part (e.g. HTTP) is handled as part of the application. The division of responsibility for a complete network communication channel between application and OS network stack is usually at the transport layer. OS network stack:

- NIC, network hardware
- First level interrupt handlers for the NIC
- Rest of the NIC driver bottom half
- Libraries linked with the application

Multiplexing is the process of sending packets from multiple connections at one layer in the protocol stack down a single connection at a lower layer. **Demultiplexing** is the inverse process: taking packets received on a connection at one layer and directing each of them to the appropriate channel on the upper layer. **Encapsulation** is the mechanism by which a packet is transformed for the lower layer by adding a header. **De-capsulation** is the reverse: interpreting a packet as the result of the union of a header and a payload.

The networking stack needs to do more than simply move packets between the network interface and main memory. In addition it must maintain and execute **state machines** for some network protocols.

The **header space** is an abstract vector space which represents the set of all possible headers of a packet. The **protocol graph** of a network stack is a directed graph representation of the forwarding and multiplexing rules at any point in time in the OS. Nodes in the protocol graph represent a protocol acting on a communications channel and perform encapsulation and de-capsulation.

A NIC operates in the following way: received packets are copied into buffers supplied by the OS and enqueued onto a descriptor queue. These filled buffers are then returned to the OS using the same, or possibly a different, queue.

Packet descriptors are data structures which describe an area of memory holding network packet data.

Packet **forwarding** is the process of deciding, based on a packet and the interface on which the packet was received, which interface to send the packet out on. Packet **routing** is the process of calculating rules to determine how all possible packets are to be forwarded.

Glossary

- **Flash memory:** non-volatile memory chips used for storage and for transferring data between a computer and digital devices. Often found in USB flash drives, MP3 players, ...
- **Hard vs soft links:** an hard link is an additional name for an existing file; a soft link is a file which points to another file.

Soel Micheletti

- **Non volatile memory:** computer memory that can retrieve stored information even after having been power cycled.
- **NIC:** computer hardware that connects a computer to a computer network.
- **Volatile memory:** computer memory that requires power to maintain stored information. Example RAM.