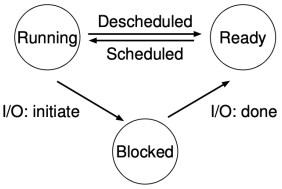
Chapter: **Title:**

How to provide the Illusion of many CPUs?
Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

| | |
|---|---|
| **Time sharing:** | By running one process, then stopping it and running another, and so forth |
| **Space sharing:** | Resources are divided among all who want to use them |
| **Context switch:** | Gives the OS the ability to stop one and run another program on the CPU |
| **Scheduling policy:** | what program to run? |
| **Machine state:** | what a program can read or update when it is running. what parts of the machine are important to the execution of this program? |

**Process API:**

| | |
|---|---|
| Create | create new process to run a Programm |
| Destroy | kill process currently running |
| Wait | wait for a process to stop running |
| Status | get status information about a process |
| Misc. Control | sometimes other controls besides wait and destroy are needed e.g.: |
| resume | |




| Time | $Process_0$ | $Process_1$ | Notes |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | $Process_0$ initiates I/O |
| 4 | Blocked | Running | $Process_0$ is blocked, |
| 5 | Blocked | Running | so $Process_1$ runs |
| 6 | Blocked | Running | |
| 7 | Ready | Running | I/O done |
| 8 | Ready | Running | $Process_1$ now done |
| 9 | Running | – | |
| 10 | Running | – | $Process_0$ now done |

**Process creation:**
Load program to mem in parts(lazy loading),
allocate memory on stack and heap to be used by the program.
The allocated mem might get bigger when the program needs more spac
initialise I/O (Input, output, error),
run main( );

**Process States:**
Running: The processor is executing instructions
Ready: The process is ready to run bit the OS is not running it atm
Blocked: not ready to run for some reasons (e.g. initiates an I/O)

**Process list/task list:**
A data structure to keep track of all programs currently running
Each entry is called process control block (PCB) and contains all necessary information

**Proc States:**
Unused:
Embryo:
Sleeping:
Runnable: Ready to run
Running: Running
Zombie: Exited but not cleaned up

ASIDE: KEY PROCESS TERMS

- The **process** is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its **address space**, the contents of CPU registers (including the **program counter** and **stack pointer**, among others), and information about I/O (such as open files which can be read or written).
- The **process API** consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.
- Processes exist in one of many different **process states**, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.
- A **process list** contains information about all processes in the system. Each entry is found in what is sometimes called a **process control block** (**PCB**), which is really just a structure that contains information about a specific process.

Chapter: **Title:**

**fork( ):** if you call fork you will get a child process exactly the same as the parrent process:

```
int main( ){
   Int id = fork( );
   printf("Hello world");
   return 0;
}
```

Hello world is printed 2 times since the fork and the parent process print it. Processid(PID) is different but the process itself is the exact same

Multiple forks: total number of processes 2^n (i is the number of fork calls)

**exec( ):**

doesn't create a new process but replaces the current with another one. Thus the processid(PID) doesn't change

**wait( ):**

waits for a process to finish (return 0;) use id to check for the child

**pipe( ):**

a pipe is used to send data between children or between parent and child. A pipe has 2 ends, a writing and a receiving end. To use the pipe you should always close the end you don't use.

ASIDE: KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID** (**PID**).
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent**; the newly created process is called the **child**. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses `fork()`, `wait()`, and `exec()` to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changing anything about the programs being run.
- Process control is available in the form of **signals**, which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user**; the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.

**How can the process perform restricted operations without allowing the process full control over the system?**

**User mode:** Limited mode that has no right to do syscalls directly. When in user mode the programm has to execute a special **trap()** instruction that lets the instruction jump to the **kernel mode** and raise the privilege level to kernel mode.

**Kernel mode:** now with more rights the syscalls (when allowed) will be executed. After finishing the **return-from-trap()** function is called to lower the privileges back to **user mode.**

**!!** Some registers have to be saved on the **kernel stack** when switching between user and kernel mode to be able to return correctly. **!!**

**Virtualisation Challenge: High performance while keeping control over the system.**

**Limited direct execution:**

Running the program directly on the CPU to get best timing.

Make sure the running program doesn't do things we don't like while being efficient.

Make sure the running program can always be stopped and switch to another one (time sharing).

The **trap table** is used to tell the hardware what code to run when the trap instruction is called (disk interrupt, keyboard interrupt, syscall). It's set up when booted (in kernel mode).

The hardware gets informed about the trap handlers and remembers the location

**Regaining control over the system:**

1. **Cooperative**: The OS trusts the process and the process periodically gives up the CPU to let the OS decide who is allowed to use the CPU. This can be done via a **yield** syscall to transfer the Controll, or by doing something illegal (e.g. devide by 0) to generate a **trap**.
2. **Non-Cooperative: (timer interrupt**) the timer interrupt just interrupts the process after a pedefined time. Then an interrupt handler runs and the OS has the full control over the system again.

**Saving and restoring Context:**

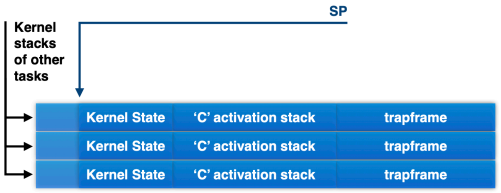
**Scheduler:** decides what program is running and manages to switch to other programs

**Context switch:** save current registers (e.g kernel stack), restore some registers for the new program.

1. Push a **tapframe**(exception frame(includes user-level PC and SP)) on the K-Stack.
2. Call the 'C' code to execute syscall, exception or interrupt (Solution is in the **activation stack**)
3. Kernel chooses a target task and pushes the remaining kernel context to the K-Stack (**Kernel state**)
4. Now another task on the kernel stack is selected and everything before now runs in reverse.

**Interrupt during interrupt handling**

**Just disable interrupts while handling. (More about that in part 2).**




ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

- The CPU should support at least two modes of execution: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- Typical user applications run in user mode, and use a **system call** to **trap** into the kernel to request operating system services.
- The trap instruction saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table**.
- When the OS finishes servicing a system call, it returns to the user program via another special **return-from-trap** instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.
- The trap tables must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs. All of this is part of the **limited direct execution** protocol which runs programs efficiently but without loss of OS control.
- Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**. This approach is a **non-cooperative** approach to CPU scheduling.
- Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch**.

How to define what syscall is allowed??

Is the OS always running on the kernel stack or only when called trap is called.

KERNEL STACK - what does the c activation staack do?
Variables of the c code executed in kernel mode.

How to syscall: write the corresponding number of the syscall In the correct register.

syscall - Schnittstelle zwischen

Warum braucht jede task einen eigenen Kernal stack

Rekursion im Kernel ist verboten!!

clock get time - Chris

**Scheduling: Introduction**

**Process assumptions:**
1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O) 5. The run-time of each job is known.

**Scheduling metrics:**

**Turnaround time**: (completion time) - (time of arrival)

**Response time:** (time of first run) - (time of arrival)

**Fairness**: some scheduling might be the most efficient but not the fairest when other processes have to wait.

**First In, First Out (FIFO):**

The first job that comes in is executed until finished.

*Problem:* A big task can block the CPU for a long time thus resulting in a bad response time.

Solution: SJF

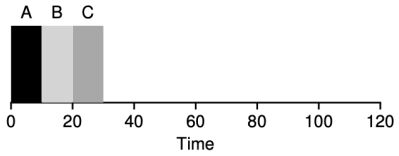



Figure 7.1: **FIFO Simple Example**

**Shortest Job First (SJF)**

*Problem:*

If the tasks don't arrive all at the same time you can't tell what the shortest task is and might have to run a longer task first.

Solution: STCF

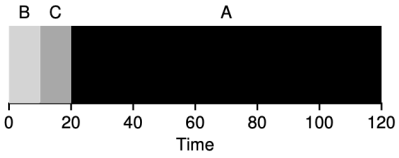



Figure 7.3: **SJF Simple Example**

**Shortest Time To Completion First (STCF)**

Every new task will be analysed to determine the time. Then the shortest task will be chosen to run by the scheduler. When a longer task is already running, it is stopped and the short one is started.

Problem:

If multiple tasks come at the same time the last task has to wait for all other tasks to finish before it can run thus the response time is bad.

Solution: RR

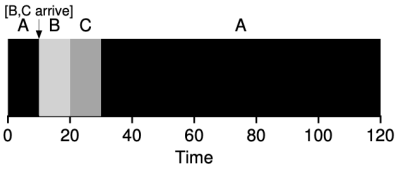



Figure 7.5: **STCF Simple Example**

**Round Robin(RR):**

Each task is only run for a "time slice" (or scheduling quantum) and then switches to another task.

Each time slice has to be a **multiple of the timer interrupt**.

The shorter the time slice the better the performance of RR.

If the time slice is too short the context switch is the bottleneck for the performance.

*Problem:*

Each task is stretched  to the absolut maximum and the turnaround time is awful.

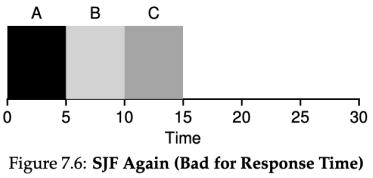



Figure 7.6: **SJF Again (Bad for Response Time)**

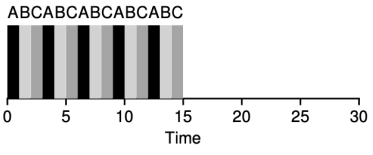



Figure 7.7: **Round Robin (Good For Response Time)**

**Minimize Waiting Time**

Do not want to spend much time in Ready queue

**Maximize Throughput**

Want many jobs to complete per unit of time

**Maximize Resource Utilization**

Keep expensive devices busy

**Minimize Overhead**

Reduce number of context switches

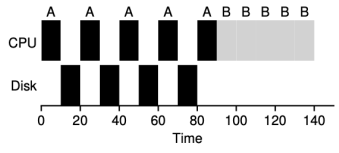



Figure 7.8: **Poor Use Of Resources**

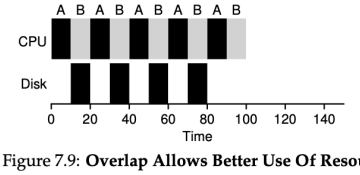



Figure 7.9: **Overlap Allows Better Use Of Resources**

**How to schedule without knowing all parameters? (e.g. runtime)**
Learn from history and keep track what a program did earlier and decide on this data

MLFQ Rules:
MLFQ has multiple distinct queues with different priorities within those queues the priority is the same.

**Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
**Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
**Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
**Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
**Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

**Problems:**
**Starvation:**
With too many interactive jobs the cpu is completely used by the fast jobs and the long ones never get done
**Behavioural change:**
At first the program needs a lot of CPU time and later changes to a lot of I/O but is already stuck in the lowest priority.
**Solution:** Implement rule 5. (Issue is to set the **voo-doo constant** S correctly)

**Game the scheduler:**
Rewrite your code so it issues a useless I/O just before a time slice is done just to stay in the same priority.
Solution: Rewrite rule 4
**Old:**
**-a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
-**b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

**How to parameterize the MLFQ scheduler:**
How many queues?
How big is a time slice per queue?
How often to priority boost?

**Answer:** Experience

First iteration of **multiprogramming** was by switching to another task while one is using the I/O. This soon developed **time sharing**, by running a code, stopping it and save all of it's state to a disk (REALLY slow).

Problem:

The process had **all** rights in the memory, thus was able to read or write to memory of another process.

Solution: **address space**

The address space consists of 3 parts: **code**, **stack** and **heap** virtualised within the memory (mng. The virtualised memory is just a small part of the actual physical memory).

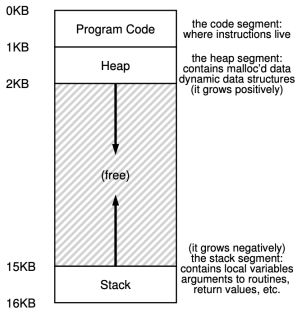


Figure 13.3: **An Example Address Space**

**How does the OS translate a call of a virtual address to a physical address?**

-> MMU (Memory management unit) Hardware component in the processor. It maps the memory between physical address and virtual address in the **page table**.

When a Programm requests something from the memory the MMU finds the corresponding physical address. If the virtual address doesn't correspond with any physical address a page fault exception is thrown and the OS is doing some allocation.

**Transparency:**

It should be invisible for the running program to see that its memory is virtualised and not really the entire physical memory. The OS is responsible to make the virtualised memory appear like the Physical memory.

**Efficiency:**

The os has to virtualise as efficient as possible to ensure good timing and good memory management.

**Protection:**

The OS has to protect (**isolate**) processes from each other as well as itself from the processes. Sometimes even the Kernel is split in multiple isolated parts (**microkernels**) to provide greater reliability.

Chapter: 14 Title:

Type to enter text

Type to enter text

Type to enter text

Type to enter text

Questions:

The virtualisation of memory is necessary to ensure that every program has its own memory and address space. Furthermore the virtualisation of memory ensures that the program can only access memory its supposed to access thus creating a security feature.

The technique to virtualise is called **hardware-based address translation** (aka. Address translation). To do this the Hardware is interposing on each memory access and translates the virtual address into a Physical address.

Old concept: **static relocation:**

A piece of software always just adds the address offset to the current offset.

Issue: without bounds you could access memory of other Programms thus creating a security issue.

**Base and bounds (dynamic relocation):**

We need 2 hardware registers in each CPU called **base** and **bound** registers (or limit register), which declare the start and end of the physical memory. The address space can be placed freely in between those 2 registers.

When we want to access memory, the base is added to the current address. When the Programm tries to access a memory of another program a **memory out of bounds exception** is thrown. The base and bounds registers can move while running the program thus making the memory **dynamic.**

The translation hardware is called the **MMU** (Memory Management Unit) and is part of each cpu (core).

Bound registers are either defined by the **size** of the Address space, and the hardware checks the virtual address against the registers before adding the base

or:

The bound register holds the last address of the address space and the hardware is adding the base first and then checks if it is within bounds.

The os must keep track of which parts of the memory are used and which are free. The easiest way to do this is by using a **free list** which contains a lit of ranges of used memory.

**Issues with dynamic relocation:**

The OS needs to look into the free list to find room to assign memory to a process and mark it used.

The OS needs to reclaim memory of a terminated process and mark it unused.

When a context switch occurs the OS needs to save and restore base and bounds registers. Those registers are saved in the memory in per-process structures as the **process structure** or the **process control block,** or set from memory to CPU when a process resumes. (Page 151, Fig. 15.5)

The OS must provide exception handlers to manage if a process is doing something it shouldn't (e.g. try to access memory out of bounds) in most cases the OS just terminates the process.

Questions:

Has each programm its own base and bounds register?

Where are bounds saved?

What happens when trying to access memory out of bounds

| Hardware Requirements | Notes |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | *Need pair of registers per CPU to support address translation and bounds checks* |
| Ability to translate virtual addresses and check if within bounds | *Circuitry to do translations and check limits; in this case, quite simple* |
| Privileged instruction(s) to update base/bounds | *OS must be able to set these values before letting a user program run* |
| Privileged instruction(s) to register exception handlers | *OS must be able to tell hardware what code to run if exception occurs* |
| Ability to raise exceptions | *When processes try to access privileged instructions or out-of-bounds memory* |

**The Issue with memory relocation:**

When using the base and bounds registers to mark the size of the address space a lot of unused space between stack and heap uses physical memory. This is pretty wasteful.

**Solution: Segmentation**

Instead of 1 base and bounds register there are multiple for each logical segment of the address space. One for the **Heap**, **Stack** and **Code** segment.

**How do we know the offset of a segment?**

**Explicit approach:**

We take the **first 2 bits** of the Virtual address and compare it. (00 = code segment, 01 heap segment) the remaining 12 bits now show the offset of the address.

**Implicit approach:**

How was the address generated.

If the address was generated by the PC (only when IF was called) the Address has to be in the code segment.

If the address is based off the SP or base pointer, the address is in the stack.

Every other address has to be on the heap.

**What about the stack?**

Issue: The stack grows in the other direction (addresses get smaller when the stack is filling). Thus the hardware needs a new bit that tells if the segment grows in positiv or negativ direction.

**Memory sharing:**

Why?: Sharing parts of the memory can save memory. A common practice is **code sharing**.

Solution: More hardware support in form of **protection bits**. Those bits indicate if a program is allowed to **read**, **write** or even **execute** code in a certain segment. Thus a single piece of physical memory can be mapped to multiple virtual addresses. With this hardware based memory sharing **isolation** is still ensured.

**In addition:**

The hardware has to check if an address is within the bounds AND if the protection is correct (e.g. don't execute memory thats read/write only).

**Fine-grained vs. coarse-grained segmentation:**

We currently only used **coarse-grained** segmentation, meaning we have only a couple of bigger pieces of memory. But you could also do a **fine-grained** segmentation and segmentate a lot more to get better memory usage and a more flexible memory overall.

**OS Support:**

Context switching: Segment registers need to be saved and restored

Managing free mem: **External fragmentation** (small holes in the memory when mem is freed) makes it difficult for the OS to allocate new mem.

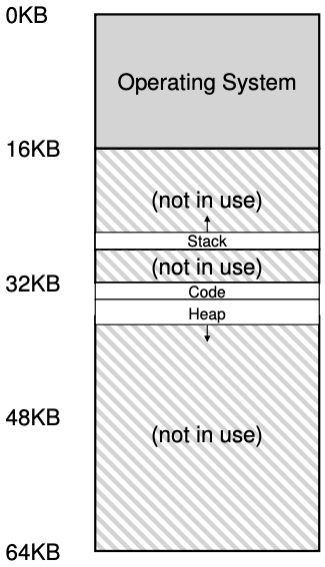
Solution 1: **compacting** the memory by rearranging the existing segments into one counties block. This solution is pretty expensive since you need to copy and paste a lot of memory.

Solution 2: A **free-list management algorithm** is trying to keep big segments of memory free for big allocations. (**best-fit**, **worst-fit**, **first-fit**, **buddy-algorithm**).

| Segment | Base | Size (max 4K) | Grows Positive? | Protection |
|---|---|---|---|---|
| $Code_{00}$ | 32K | 2K | 1 | Read-Execute |
| $Heap_{01}$ | 34K | 3K | 1 | Read-Write |
| $Stack_{11}$ | 28K | 2K | 0 | Read-Write |

Questions:

Why do we hold on to the idea of stack and heap growing towards each other if we segment them anyways?



**Problem: external fragmentation**

How can you manage your free space the most efficient way and minimise fragmentation?

**Assumptions:**

We use **free** and **malloc**, the heap is structured by the **free-list**, our biggest concern ist **external fragmentation** and we don't really focus on internal fragmentation, a allocated memory **can't be relocated** (no compaction of free space) and we **can't grow the heap**.

### Splitting and Coalescing


fig. 17.1.1

Splitting (fig. 17.1.2): If we call malloc to allocate less than the block of free space, a piece of the free memory would be split off and could be used. The start address of a free space is increased by the amount called in malloc byte and the length is reduced by the same amount. Thus we keep as much free space as possible.

Coalescing(fig. 17.1.2): If we free a chunk of memory the start address and length will get added back into the free-list. To be able to allocate the maximum possible space the free fragmented chunks need to be **coalesced** to be able to allocate bigger pieces of memory.

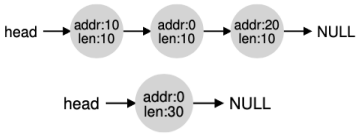



fig. 17.1.1

### Tracking The Size Of Allocated Regions

Before each chunk of allocated memory a block called the **header** contains information about the size of the allocated memory, and can also contain informations like additional pointers for speedup or a magic number for integrity checking.

After the user called free(ptr) the library can calculate where the header begins and if the magic number matches the expected. After that the lib will calculate the freed chunk by adding the header size to the allocated size. This means the library actually searches for a chunk of mem thats *N bytes + size of header* big.

### Embedding A Free List

The library calls the syscall mmap() to get a piece of memory that can be used as a heap and sets a header with just a size and a next field set to NULL (there is no other entry). If the user calls malloc now the only possible chunk (there is only one) will be chosen and split to get the desired allocation (Be careful, the actually allocated memory is 8 bytes bigger to have space for the header).

When calling free the allocated memory is returned and immediately added back into the free list. After freeing we need to coalesce the memory by merging neighbour chunks of free memory to get one big memory piece and don't have fragmentation issues.

### Growing The Heap

The easiest would be to just return NULL if the heap runs out of space for more allocations but there are better solutions. By executing a syscall like sbkr the heap can be extended and can grow. The OS will look for free space, map it to the requesting address space and returns the value of the end of the new heap.

### Basic Strategies to manage free space:

The quest is, to find an allocator method that is fast and minimises fragmentation.

### Best fit

Iterate through the free list and find a chunk of mem that's as big or bigger then the requested memory. After that the smallest in the group of candidates is chosen.

Issue: Bad implementations pay a heavy performance penalty.

### Worst fit

It's the exact opposite of the best fit method and has the same flaws. Return the largest chunk of memory and split it to fit, thus keeping the chunks as big as possible.

### First fit

Find the first block of memory that is big enough and return. This is fast since we don't need to iterate over the entire free list.

Issue: The beginning gets polluted by small objects and the free list management can become an issue.

Solution: **address based ordering.** By keeping the list ordered by address of the free space coalescing becomes easier and fragmentation is reduced.

**Next fit**

Its pretty similar to first fit, but a new pointer is implemented to keep track of where we were already looking for space and continues where it left of last time thus reducing answer time

**Segregated lists**

The idea is to keep a separate list for objects of a popular size. This reduces fragmentation a lot and speeds up the allocation time by a lot.

**Slab allocator:**

Allocates object caches for predictable kernel objects. The allocator keeps free objects on the list in a pre-initialised state. This means if an object gets allocated it already contains some default values or is zeroed-out. This can drastically reduce overhead (makes it faster due to the reduced workload)

**Buddy Allocation**

The binary buddy allocator uses a unique way to make coalescing easier. By always dividing the 2^N big space by 2 until it is the smallest size the block can fit into. With this method we always get a block size that's a multiply of 2. If we now free an allocated block there will always be another block the same size. If both blocks are free they get coalesced. The now new block has double the size and will get coalesced with ITS buddy if it is free. Like this you can coalesce the memory recursively until 1 buddy is in use (fig. 17.2)

With this approach the buddies addresses only differ by 1 bit which makes it extremely easy to find the corresponding free memory and coalesce it.

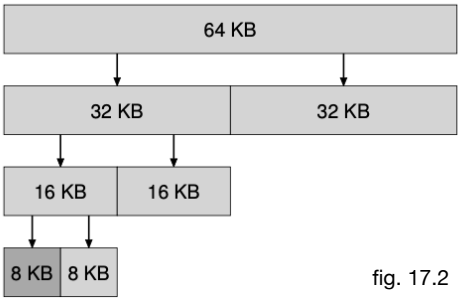



fig. 17.2

Questions:

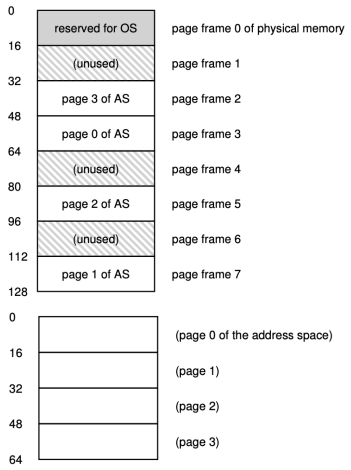
What happens with next fit if memory gets freed at a position before the pointer?

Slab allocator anschauen

**Paging the basic approach:**

Instead of splitting the address space into a number of different sized logical segments (e.g. stack, heap, code), the OS is using fixed sized blocks of memory called **pages**. The Physical memory also consists of an array with fixed sized slots called the **page frames**. Each frame can contain a single virtual-memory page.

**Issue:** How do we avoid (internal?) segmentation?




**Overview:**

If we have a physical memory of 128 bytes and a virtual address space of 64 bytes with a page size of 16 bytes we have 4 pages of virtual memory and 8 pages of physical memory.

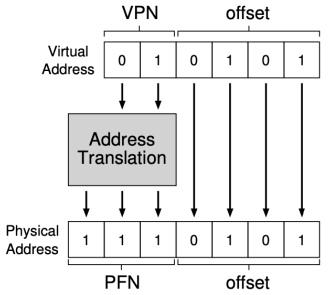
If the OS now places our virtual memory into the physical memory, the OS can just assign any free pages of the physical memory to the virtual memory. The free list meanwhile keeps track of free pages in the physical memory. (fig. 18.1)

The biggest improvements are **flexibility** and **simplicity,** which result in an overall better time and space overhead.

Within the, for the OS reserved part of memory, the **page table** is placed. The page table keeps track of which page frame has been assigned to which virtual page. The page table is a per-process structure which means, each process has its own page table.

**Address translation:**

To translate from virtual to Physical addresses we need to split the address into two pieces. The **virtual page number (VPN)** and the **offset.**




Example: In our 64 bit address space we need 6 bits to map our entire address space. Since we have a page size of 16 bytes we need to reach 64/16 = 4 pages thus needing 2 bits for page addressation. The remaining 4 bits tell us which byte of the page we want to target called the offset.

The virtual page number now gets translated into the **physical page number (PPN)** and the offset is attached. The offset still stays the same since it tells us the byte within the page we want. (fig. 18.2)

**Issue: Where are page tables stored**

With a typical 32 bit system and 4KB of page size we would have 20 bits for VPN and 12 bits for offset. This would mean we need 2^20 address translations which would be about 4 MB in size. This amount would even be multiplied by the number of running processes.

Thus the page tables can't be stored in any special hardware in the MMU and just sits somewhere in the physical memory that the OS can manage.

**Whats inside the page table?**

A **valid bit (V),** to determine if a particular translation is valid (e.g. if a program starts running it has a code and a heap at the top and a stack on the bottom. All unused space between the stack and heap is marked invalid). When trying to access invalid memory a trap to the OS will be generated.

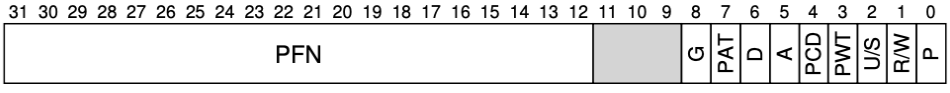
Some **protection bits (R/W),** to determine if a page can be read, written to or executed with a trap if violated.

A **present bit (P),** to determine if a page is in physical memory or on the disc.

A **dirty bit (D)**, to determine if a page has been modified.

A **reference bit (A)** (access bit) to track if a page has been used. This is useful to find popular pages that should stay in the memory.

There are even more bits as seen in the image but those are the most important for now. (fig. 18.3)




**Paging is also to slow**

To translate a virtual address to a physical we need to do the following steps:

1. Shift the virtual address to get the VPN.
2. Translate the VPN into a PFN by using the VPN as index of the page table.
3. Concat the PFN with the offset via a left shift and a bitwise OR.
4. Fetch data from the physical memory.

In comparison to the segmentation we need to perform an additional memory reference which takes a lot of time (slow down by a factor of 2 or more!)

=> We need to optimise hardware and software towards performance to reduce time.

Issue: How can we seed up address translation?

Each address translation needs to look into physical memory to determine the translated address.

Solution: A **translation-lookaside buffer (TBL)** is a part of the MMU and acts like a cache for popular address translations. When translating an addreess the hardware first looks into the TLB to check if the address can be converted quickly. If not the hardware needs to look into the page table.

**TLB hit, miss rate**

If you access continuous data the TLB becomes really efficient since not each access, of an array for example, needs to be looked up in the page table since the desired page is already loaded into the TLB after the first access thus we only have 1 miss per page. If you access this memory frequently you can often get even no miss since it is still in the TLB

**Who handles a TLB miss?**

Hardware: In older systems the TLB misses are managed by the hardware which has some complex instruction sets (**CISC**) to translate addresses from the page table. If a the hardware is used for this the page table always needs to be at the same location addressed by the **page-table base register** an example for a hardware based TLB miss handler is an Intel x86 system.

Software: In modern systems (MIPS, SPARC) which use **RISC** (reduced instruction set computer) the software handles the TLB. On a miss an exception is raised, privilege level is raised to kernel mode and a **trap handler** is called which updates the TLB. After that a **return from trap** is used and the hardware retries to hit in the TLB.
To make this trap and return from trap call work, the OS needs to save the PC differently when the trap was called from a TLB miss.

**TLB issue: context switch**

When switching contexts the TLB needs to know which translation is part of which process. It would be fatal if a different process could access memory of another process.

Solution: Just always flush the TLB by setting all valid bits to zero. BUT if a the OS switches between processes frequently the TLB becomes useless since we need to look into the page-table anyways each time. To encounter this problem we can add an **address space identifier (ASID)** field in the TLB (much like the PID)

**Replacement policy**

Two common approaches are **LRU** and **random** both have their benefits. LRU behaves really well if you have a few translations which repeat but behaves horribly if the program loops over n +1 pages. With this corner case the LRU behaves like we don't even have a TLB. Random may have a worse best case scenario but will handle corner cases way better.

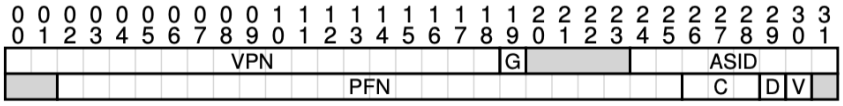
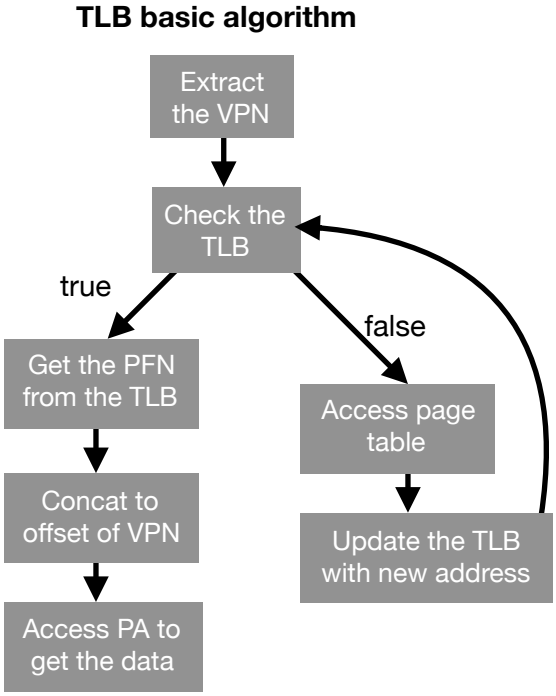



Figure 19.4: **A MIPS TLB Entry**




Questions:

Why don't we build just a lot of small chaches that have small caches which reference them (Binary tree) to get a big O notation of log(n)

What the fuck is this magic number?!?!?

Issue: Page tables are too big.

Solution 1: **Make the pages bigger.** The problem with this approach however is that it leads to more internal fragmentation.

Solution 2: **Hybrid approach: paging and segments**

To realise this approach we use 3 page tables. One for the stack, heap and code segment which each use the **base** and **bound** register from segmentation to indicate the physical address of the page table and the end of it. On a **context switch** all 3 pairs of base and bound registers need to be saved and changed. On a **TLB miss** the hardware uses the segment bits to determine in which page table it has to look. Then the hardware takes the physical address within the table and combines it with the VPN to get the PTE.

Example: If the code segment is only using the first 3 pages the page table of the code segment will only have 3 entries and does not have to contain all the invalid addresses between the stack and heap.

Issue: Segmentation is not flexible since it assumes a basic pattern of the address space (Stack, Heap, Code) this approach could also lead to external fragmentation.

### Multi-level page tables

The idea is to chop up the page table in page sized pieces and if an entire page of page table entries is invalid it doesn't get allocated at all. To keep track of all the valid page-table-pages we use a structure Called the **page directory**. The only thin the page directory can tell us is, where a page-table-page (PTP) is or that the entire PTP is invalid. The Page directory consists of a number of **page directory entries** (PDE) which minimally contain a **valid bit** and the **page frame number.**

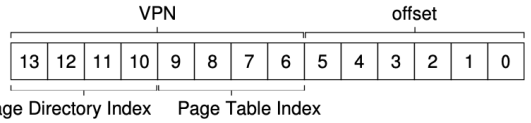
Advantages: You are only using up page table size when we actually have valid data. We also have the freedom to place PTPs where ever we want in the physical memory.

Trade-off: On a TLB miss we have to do 2 loads from memory instead of one (PDE access and PTP access) which result in a worse time.

### Multi level page table access

Given: 16KB Address space size; 64B page size; P 1,2 Code; P 4,5 Heap; P 254, 255 Stack; PTE size = 4B

1. 256*4 = 1024 = 1KB (Linear page table size)
2. 1024 / 64 = 16 (Number of PTEs per page table page)
3. log2(16) = 4 (Bits needed for the **page directory index (PDIndex)**)
4. PDEAddress = PageDirBase + (PDIndex * sizeof(PDE)) (Page directory entry address)
5. => PFN = 55 = 11 0111 << SHIFT (offset amount) + offset
6. 00 1101 1100 0000




### More than two levels

If a page table directory gets to big we can implement a new level to our page directory.

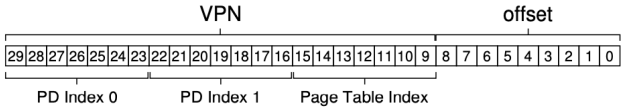
Example: 30-bit Virtual address size; 512-byte pages; 4-byte PTE size **=>** 21-bit VPN, 9-bit offset.

512/4 = 128 PTEs per page

log2(128) = 7 bits of VPN as page table index (least significant)

Check the upper level page directory with PD-Index 0

```
if(valid)
    Combine the PFN with PD-Index 1
        if(valid)
            Combine the PFN with the page table index to get the PTE
```




### Inverted page tables

Instead of having a seperate page table for each process we have a big page table that has an entry for each physical page. This entry tells us some kind of PID and which virtual page of this process is mapped to which physical address in the memory

### Swapping the page tables to disk

If a page table becomes too big, it may be useful to place move the page table out of the kernel virtual memory and into the hard drive, allowing the system to swap some of the pages in and out.

How can the OS make use of the hard drive to provide the illusion of a larger virtual address space?

**Swap space**

We can look at swap space like a piece of the hard drive the OS can use to swap pages to and from. To do this the OS needs to remember the **disk address** of each page.

**The present bit**

When we do a memory reference and the TLB has a hit it stays as it was and the swap space isn't needed. On a TLB miss and a miss in the page table however, the OS needs to have a new Bit. The **present bit** tells the OS if the desired page is present (in physical memory) or on the hard drive. The latter case is called a **page fault,** which is dealt with by the OS which is invoking a piece of code called the **page-fault-handler.**

**The page fault**

In nearly all operating systems a page fault is handled by the software even with a hardware TLB. When a page fault arrises the OS looks in the page table into the PTE to get the desired hard drive address and issues a request to fetch the page into memory.

After the disk I/O is done the OS will update the page tables present bit and update the PFN field of the page-table entry to get the new in-memory location of the page, and retries to execute the instruction.

This time it will not generate a page fault but maybe a TLB miss which is handled like before.

Note: While we handle the page fault an I/O request is issued thus the process is in a **blocked** state.

**What if memory is full**

If the memory is full the OS needs to **page out** a page to the hard drive before being able to **page in** a new page into memory. To do this efficiently we need some kind of **page-replacement policy**. With a bad policy the entire system will slow down significantly and might even be as slow as the hard drive instead of being almost as fast as memory.

**When replacements really occur**

The OS will most likely keep a portion of the memory proactively free. Most OSs will have some kind of **high watermark** (HW) and **low watermark** (LW). When the OS notices that there are fewer pages then LW available it will run a background process to free some pages until we reach the HW and vice versa.

The background process is called **swap deamon** or **page deamon.**

By performing multiple replacements at once many OSs are able to group a number of pages and write them out at once. Thus increasing efficiency by reducing the needed I/O requests.

Questions:

At which point is the present bit checked if valid?

Issue: How does the OS decide which page to evict when we have **memory pressure**?

To measure how good or bad a policy is, we have a metric called **average memory access time (AMAT)**

$(AMAT = T_M + (P_{Miss} * T_D))$

$T_M$ = cost of a memory access

$P_{Miss}$ = probability of a cache miss

$T_D$ = cost of a disc access

Example:

10% miss rate, $T_M$ = 100ns $T_D$ = 10ms

$0{,}0001ms + (0{,}1 * 10ms) = 1{,}0001ms$

### The optimal replacement policy

The optimal replacement policy would be the one, that replaces the page accessed the furthest in the future. This will always be our comparison for all replacement policies. Comparing with this optimal policy is useful to determine how close your policy is to the optimal.

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 2 | 0, 1, 3 |
| 0 | Hit | | 0, 1, 3 |
| 3 | Hit | | 0, 1, 3 |
| 1 | Hit | | 0, 1, 3 |
| 2 | Miss | 3 | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |

$\frac{Hits}{Hits + Misses} = \frac{6}{6+5} = 54{,}5\,\%$ with **cold start misses**

$\frac{Hits}{Hits\,\%\,Misses} = \frac{6}{6\,\%\,5} = 85{,}7\,\%$ without cold start misses

Note that those are the optimal hit rates we can't achieve with any policy known to date

### A simple policy: FIFO

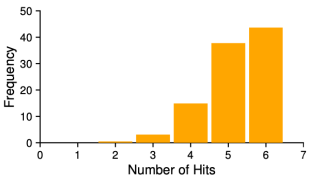
Fifo is really simple to implement, by just kicking out the page that was inserted first when new memory is needed. The issue is that FIFO can't determine how important a page is and thus can result in really bad hit rates (in this example 36,4%)

Another downside of FIFO is the **Belady's Anomaly**

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | First-in→ 0 |
| 1 | Miss | | First-in→ 0, 1 |
| 2 | Miss | | First-in→ 0, 1, 2 |
| 0 | Hit | | First-in→ 0, 1, 2 |
| 1 | Hit | | First-in→ 0, 1, 2 |
| 3 | Miss | 0 | First-in→ 1, 2, 3 |
| 0 | Miss | 1 | First-in→ 2, 3, 0 |
| 3 | Hit | | First-in→ 2, 3, 0 |
| 1 | Miss | 2 | First-in→ 3, 0, 1 |
| 2 | Miss | 3 | First-in→ 0, 1, 2 |
| 1 | Hit | | First-in→ 0, 1, 2 |

### Another simple policy: random

With random you just replace any random page and hope it is efficient. The performance of random solely relies on luck. It can be faster or slower, but with increasing cache capacity it gets better.




### Using history: LRU

A group of policies that are based on the **principle of locality** this means a program tends to access certain memory over and over again. Those pages should be kept in memory. To do this we can either look how **frequent** or **recent** a page was used and decide with this information, which page to page-out. Examples for these kinds of algorithms are **Least-Frequently-Used (LFU)** or **Last-Recently-Used (LRU)**

(in this example with LRU we have a hit rate of 85,7% which is optimal)

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | LRU→ 0 |
| 1 | Miss | | LRU→ 0, 1 |
| 2 | Miss | | LRU→ 0, 1, 2 |
| 0 | Hit | | LRU→ 1, 2, 0 |
| 1 | Hit | | LRU→ 2, 0, 1 |
| 3 | Miss | 2 | LRU→ 0, 1, 3 |
| 0 | Hit | | LRU→ 1, 3, 0 |
| 3 | Hit | | LRU→ 1, 0, 3 |
| 1 | Hit | | LRU→ 0, 3, 1 |
| 2 | Miss | 0 | LRU→ 3, 1, 2 |
| 1 | Hit | | LRU→ 3, 2, 1 |

### Types of locality

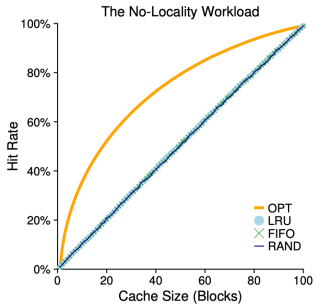
There are 2 types of locality that programs tend to exhibit, but keep in mind that these rules are more what you'd call 'guidelines' than actual rules.

**spacial locality:** if a page P is accessed it is likely that the next page or the page before is also accessed.

**temporal locality:** if a page P was accessed recently it is likely that the same page is accessed again soon.

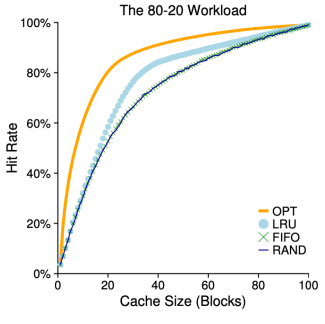
### Workload examples: no locality

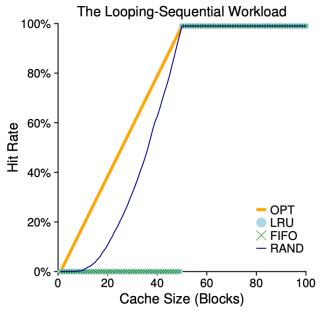
The Graph shows the 4 policies OPT, LRU, FIFO and RAND when they run optimally. As we can see all but OPT are performing the same with no Locality in the Workload, meaning we don't access pages that are next to each other in memory or access pages we used before.




### Workload examples: 80-20

This 80-20 Workload exhibits locality: 80% of the references are made to 20% of the pages and vice versa. As we can see in the graph, FIFO and RND perform reasonably well, but LRU is a bit better. Even though this seems minor, a slight improvement of the hit rate can result in way better times since misses are often really costly.






### Workload examples: looping sequencial

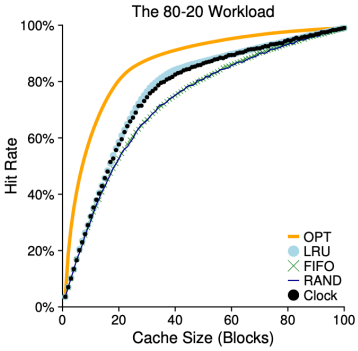
This looping workload is the worst case for FIFO and LRU, since those 2 policies throw out the exact wrong pages only if all looped pages fit into the cache the hit rate is above 0%. This time random actually has a benefit because of its nature it doesn't care about corner cases because there are no if you throw out random pages.

### Implementing historical algorithms

The idea of any historical algorithm like LRU is simple. The implementation however is quite a challenge. To implement such an algorithm by always saving the time of the page and walk through the list to check which is the oldest.

Problem: iterating over any kind of datatype is slow. Especially when you consider that we have pages of 4KB or bigger in modern systems, thus needing to check over 1 Million pages.

### Approximating LRU




The solution is approximating the LRU strategy by implementing some hardware support in the form of a **reference bit** or **use bit.** The Hardware just sets the use bit to 1 whenever a page is referenced. Resetting the use bit however is a task for the OS.

With a **clock** algorithm all pages in memory are arranged in a circular list and a ‘pointer’ that works like a clock hand. If the pointer is looking at a page with use bit 1 it will set it to 0 and set the pointer to the next page. If the use bit the pointer looks at is a 0 it has found its replacement candidate. If a page is used again the hardware will set the use bit to 1 again. As we can see is this approximation of LRU pretty close to the original.

### Considering dirty pages

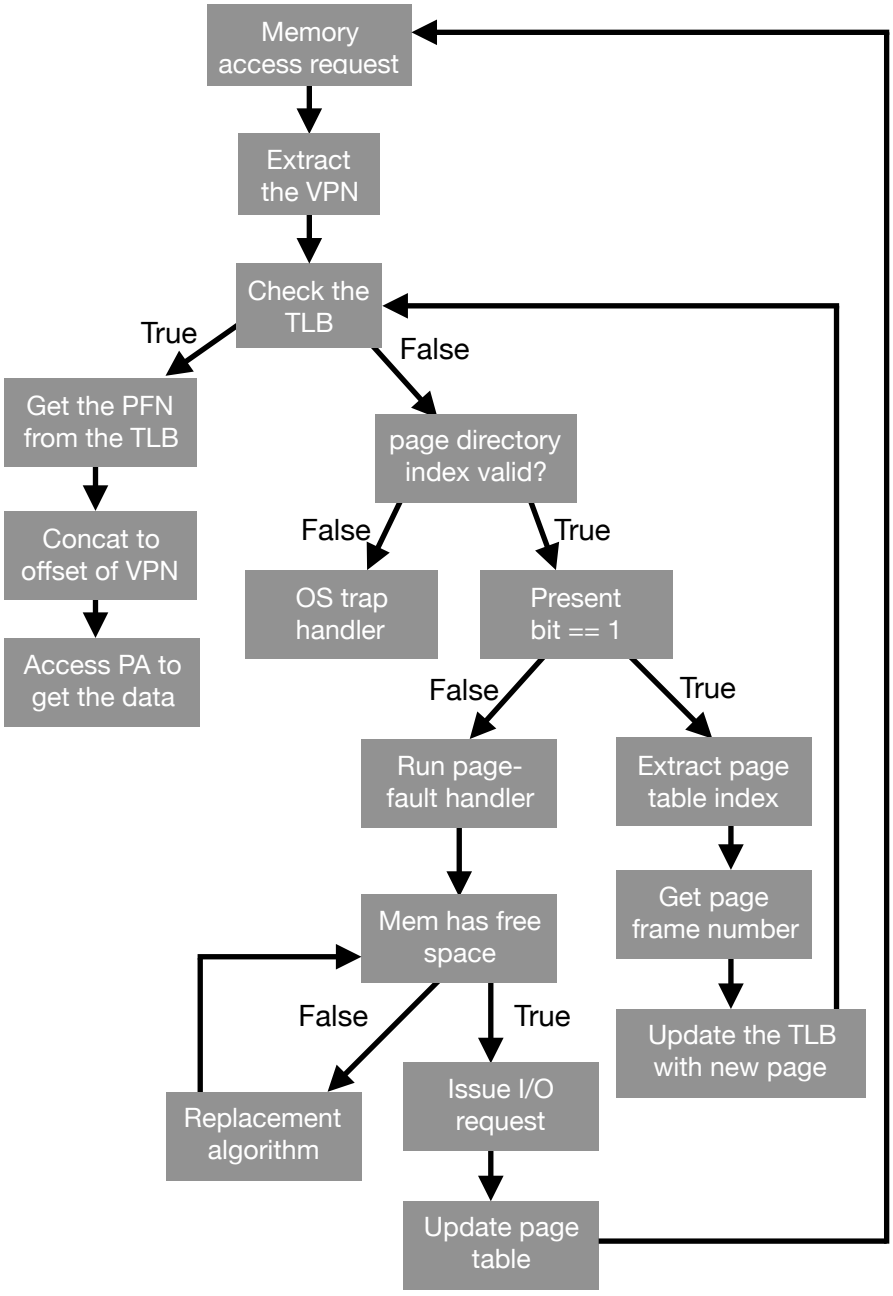
In addition to the clock algorithm we can consider to also check for dirty pages. If a page is dirty it has to be written back to the hard drive thus needing I/O. To prevent this we can implement hardware support that also checks the dirty bit and doesn't page-out a page if not necessary.

### Other VM policies

The OS has many other policies like **page selection** (decide when to bring a page in memory) examples for this are **demand paging** (put page in memory when demanded) or **prefetching** (put a page in memory when we know we need the page). The OS also needs a policy how to write pages out to the disk (**grouping** multiple pages to reduce I/O).

### Thrashing

We still run into some problems when the program wants more memory than we have physical memory available. In this case policies like **admission control** or **out-of-memory killer** come in handy.


Memory access request
Extract the VPN
Check the TLB
True
False
Get the PFN from the TLB
Concat to offset of VPN
Access PA to get the data
page directory index valid?
False
True
OS trap handler
Present bit == 1
False
True
Run page-fault handler
Extract page table index
Get page frame number
Mem has free space
False
True
Replacement algorithm
Issue I/O request
Update page table
Update the TLB with new page

We already have multiple **virtual CPUs** and the illusion of a large **private memory space.** Now we will look at a new abstraction called **threads**. Instead of using a single PC we can use more than one PC and make it a **multi-threaded** process. We can think of a thread like a separate process with the same address space as another thread. Each thread has its own registers that need to be saved and restored when a thread context switch happens.
A process uses a PCB to store the state of the registers, the thread uses a **TCB** for each thread to store the state of its registers.
One big difference is that we don't need to switch pages when a thread context switch occurs, since they have the same address space.

Each thread has its own stack (sometimes called **thread-local storage**) that resides within the virtual address space.

### Why use threads?

**Parallelism:** if we can utilise more than 1 processor we can use threads to let the processors run parallel thus saving time.

**Overlap:** If we have an I/O request on one thread we can use another thread to run other parts of the program thus making the I/O request less expensive, since we can still run the code when an I/O occurs.

### Shared data

If we have 2 threads writing to the same variable in a single-thread system we can not expect it will work like it should do on a multi-threaded system because of system interrupts (see fig.). We call it twice but because of unlucky interrupt timing we only increment once. A case like this is called **race condition** and occurs every time we have a bad timing on interrupts the results are **indeterminate**.

| OS | Thread 1 | Thread 2 | PC | eax (after instruction) | counter (after instruction) |
|---|---|---|---|---|---|
| | *before critical section* | | 100 | 0 | 50 |
| | `mov 8049a1c,%eax` | | 105 | **50** | 50 |
| | `add $0x1,%eax` | | 108 | **51** | 50 |
| **interrupt** | | | | | |
| *save T1* | | | | | |
| *restore T2* | | | 100 | 0 | 50 |
| | | `mov 8049a1c,%eax` | 105 | **50** | 50 |
| | | `add $0x1,%eax` | 108 | **51** | 50 |
| | | `mov %eax,8049a1c` | 113 | 51 | **51** |
| **interrupt** | | | | | |
| *save T2* | | | | | |
| *restore T1* | | | 108 | 51 | 51 |
| | `mov %eax,8049a1c` | | 113 | 51 | **51** |

The code responsible for such a shared variable is called a **critical section** and MUST not be concurrently executed by more than one thread at a time. If this requirement is met, we speak of **mutual exclusion**.

### We wish for atomicity

To remove this issue completely we want to execute the necessary instructions atomically. This means we want either all instructions run at once or none. It should not be possible to interrupt the instructions in the middle of a calculation.

To do this we need to use some instructions to build something called **synchronisation primitives** which allow us to access critical sections in a synchronised and controlled manner.

The Idea of atomic operations is really powerful since we are able to either do all or nothing which allows us to build deterministic code.

### Thread implementation in C

```c
#include <pthread.h>

//function to call in thread
void *myThreadFun(void *vargp) {
    printf("Hello World from thread \n");
    return NULL;
}

int main() {
    pthread_t thread_id_1, thread_id_2;
    pthread_create(&thread_id_1, NULL, &myThreadFun, NULL);
    pthread_create(&thread_id_2, NULL, &myThreadFun, NULL);
    pthread_create(
        thread, (pointer to thread_id)
        attributes (NULL for default)
        start_routine, (function to call in thread)
        arg) (arguments to start_routine)
        returns 0 on success and non-zero on failure
    pthread_join(thread_id_1, NULL);
    pthread_join(thread_id_2, NULL);
    int pthread_join(
        pthread_t thread, (thread to wait for)
        void **retval) (pointer to return value)
    exit(0);
}
```

### Locks

```
#include <pthread.h>

int nbr = 0;
pthread_mutex_t lock;


void *myThreadFun(void *arg) {
    Pthread_mutex_lock(&lock);
    nbr += 1; //critical section
    pthread_mutex_unlock(&lock);
    return NULL;
}

void Pthread_mutex_lock(pthread_mutex_t *mutex){
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}

int main() {
    pthread_t thread_id;
    int rc = pthread_mutex_init(&lock, NULL);
    assert(rc == 0);
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    exit(0);
}
```

With the pthread_mutex_lock function we can guarantee that a critical section is only used by one thread.

### Condition variables

If a thread needs to communicate with another thread (e.g. wait for a result of another thread), **condition variables** come in very handy.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;
int ready = 0;

int main() {
    pthread_t id_1, id_2;
    int rc = pthread_mutex_init(&lock, NULL);
    assert(rc == 0);
    pthread_create(&id_1, NULL, fun1, NULL);
    pthread_create(&id_2, NULL, fun1, NULL);
    pthread_join(id_1, NULL);
    pthread_join(id_2, NULL);
    exit(0);
}

void *fun1 (void *arg){
    Pthread_mutex_lock(&lock);
    while (ready == 0)
        Pthread_cond_wait(&cond, &lock);
    Pthread_mutex_unlock(&lock);
}

void *fun2 (void *arg) {
    Pthread_mutex_lock(&lock);
    ready = 1;
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&lock);
}
```

**The goal of our lock**

**Mutual exclusion**: Does the lock prevent other threads to access the code while locked?

**Fairness**: Does each thread get the same chance to lock a lock or can a thread **starve**?

**Performance**: How much time overhead do we get when using a lock?

**Building a lock**

Just using a lock variable to check if a thread is already executing a critical section is not safe, since a badly timed interrupt can still break the code and create a race condition. By just disabling the interrupt when a thread is in a critical section is also a really bad idea since this is a kernel level instruction and can lead big issues.

Solution 1 (**test and set**): We implement a new assembler instruction called “test and set” that tests and sets a variable atomically, thus ensuring no interrupt can break it. This implementation provides mutual exclusion thus making it a viable option.
Issue: It is not really fair since a single thread can keep a section locked indefinitely. It is not really performant if every other thread is waiting in a while loop for its turn to lock.

Solution 2 (**compare and swap**): This is also a hardware supported assembler instruction to compare a value at a ptr and update the memory at the ptr if the value is equal to the expected value. Otherwise do nothing. In both cases it will return the actual value of the memory at that location.

Solution 3 (**load-linked** and **store-conditional**): Load Linked and Store Conditional is a pair of instructions used in some architectures to implement spin locks. The **Load Linked** (LL) instruction loads the value of a shared variable and marks it as "linked," creating a reservation. The **Store Conditional** (SC) instruction attempts to store a new value to the shared variable only if it has not been modified by other threads since the LL instruction. If the SC succeeds, it implies that the lock has been acquired. The advantage of LL/SC is that it reduces contention by avoiding unnecessary retries. However, it requires special hardware support and is not available on all architectures.

Solution 3 (**Fetch and Add**): Fetch and Add is an atomic operation used in spin locks to increment a shared variable by a specified value and return its previous value. It is commonly used in lock-free algorithms to ensure exclusive access to resources. The advantage of FAA is that it provides a simple and efficient way to implement counters or queues. However, it may suffer from contention when multiple threads try to modify the same shared variable simultaneously.

Every lock we build thus far is a **spin lock** meaning: If a thread is trying to acquire a lock it will ‘spin’ or loop until the lock is free. As we can imagine this might not be the most efficient method but it is simple.

**Advantages of Spin Locks:**

Spin locks are typically faster than other synchronisations (e.g. mutex) in scenarios where the lock is held for a short time or when contention is low.

They are simple to implement.

Spin locks are usable in real-time systems where blocking operations (e.g., waiting on a mutex) are not desirable.

**Disadvantages of Spin Locks:**

Spin locks can waste CPU cycles when a thread is spinning to wait for another thread to unlock.

They are not suitable if high contention exists, as they can cause a significant performance degradation.

Spin locks can potentially lead to deadlocks if not used correctly, such as in scenarios where a thread holding a lock is preempted or interrupted indefinitely.

When a context switch happens while in a critical section the threads can start to spin endlessly

Spin locks should be avoided not only for performance reasons but also for correctness, specifically to prevent **priority inversion**.

Priority inversion occurs when a high-priority thread is blocked by a lower-priority thread holding a spin lock, causing the system to hang.

Even avoiding spin locks does not eliminate the issue of inversion, as higher-priority threads may still get stuck waiting for lower-priority threads.

To address priority inversion, techniques such as avoiding spin locks, implementing priority inheritance where higher-priority threads temporarily boost the priority of lower-priority threads, or ensuring all threads have the same priority can be used.

**How to build a lock that doesn't need to spin?**

To build a lock that doesn't spin we can use the yield() function, changing its own process state from running to ready to give up CPU time, effectively descheduling itself. One issue still remains:

Issue: How can we prevent starvation (a process is stuck in a yield() loop)?

The Solaris operating system provides two calls, park() and unpark(threadID), which can be used together to build a lock that puts a caller to sleep when attempting to acquire a held lock and wakes it when the lock is free. With this approach we still get some spinning but it is limited and we don't get as much overhead.

Additionally, we can use setpark() to avoid sleeping indefinitely and passing the lock directly from the releasing thread to the acquiring thread without setting the flag to 0 in between.

Futex

**TBC**

Type to enter text

Issue: How to add a lock to a data structure to make it thread safe?

**Concurrent counters**

By just adding a lock to the data structure and locking and unlocking each time we want to access the data structure we will get a concurrent counter:

```
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;
```

```
void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);}
```

This approach is perfectly fine but can get really slow. Thus we better use:

**Concurrent scalable counters**

To realise this approach we use a technique called **approximate counter**. We represent a single logical counter by several local counters (one per CPU) and a single global counter. Each thread runs its own private counter that gets updated to the global counter when a predefined threshold is reached. This is way more efficient but only an approximation.

**Concurrent linked lists (scalable)**

A normal concurrent linked list is pretty easy to implement with 2 structures for node and list. The node contains the value and the next pointer. The list contains the node and the mutex. The only thing we need to be careful about is, if malloc fails, since we need to unlock the mutex before handling the error.

When building a concurrent linked list that scales better with large amounts of data, we can use a strategy called **hand-over-hand locking** (lock coupling). With this approach each node has a lock instead of the entire list. When traversing the list we first grab the next lock before unlocking the current.

Type to enter text

Issue: How can we build a program where thread A always runs before Thread B

Solution: Condition Variables are ‘global’ variables used to indicate a state of a process.

We have 2 common approaches to make use of a condition variable. Either by **waiting** for it in a low priority queue or by **signaling** on the condition.

```
pthread_cond_t c;
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

**Producer–consumer problem (Bounded buffer problem)**

If a producer and a consumer have a shared variable the producer can write something in the buffer until it is full and needs to wait until the consumer has removed something from the buffer, or vice versa.

The issue: How can we ensure the producer never writes to a full buffer and the consumer never reads from an empty buffer? Also how can we ensure the producer and consumer never access the buffer simultaneously?

Solution: Condition Variables

The producer and consumer each get a condition variable assigned thus the producer can wake a consumer and the consumer can wake a producer. We wait for this variable with a while loop to eliminate the risk of accidentally letting multiple consumers access data only once can have.

Producer-consumer example code:

```
cond_t empty, fill;
mutex_t mutex;

void *producer (void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock (&mutex);
        while (count == MAX)
            Pthread_cond_wait (&empty, &mutex);
        put (i);
        Pthread_cond_signal (&fill);
        Pthread_mutex_unlock (&mutex);
    }
}
void *consumer (void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock (&mutex);
        while (count == 0)
            Pthread_cond_wait (&fill, &mutex);
        int tmp = get () ;
        Pthread_cond_signal (&empty);
        Pthread_mutex_unlock (&mutex);
        printf ("%d\n", tmp);
    }
}
```

A semaphore is a datatype that can be used as lock aswell as condition variable with 2 standard routines: `sem_wait()` and `sem_post` (aka. P() and V()). Semaphores have an integer value and thus have to be initialized before using.

```c
#include <semaphore.h>
sem_t sem;
// init with 1 and shared between threads
sem_init(&sem, 0, 1);
```

When sem_wait() is called, it is checked if the semaphore is <= 0 and the thread goes to sleep. When the semaphore becomes greater then 0 the thread can access the critical section and sem_wait() decrements the value atomically. When sem_post() is called the semaphore is incremented atomically, waiting threads will be woken and the lock is free again.

Producer-consumer lock with semaphore

```c
pthread_sem_t full, empty;
pthread_sem_init(&full, 0, 1);
pthread_sem_init(&empty, 0, 1);
pthread_mutex_t mutex;

void *producer (void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}

void *consumer (void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get () ;
        sem_post(&mutex);
        sem_post(&empty);
        printf ("sd\n", tmp);
    }
}
```

Reader-Writer lock with semaphore

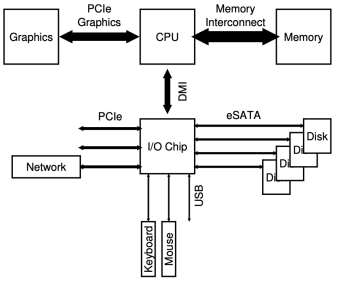
```c
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int readers;   // #readers in critical section
  rwlock_t;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Each I/O Device is connected to the CPU via different connections:

The fastest connection needs to be between memory and CPU since this is the bottleneck for the processor and between GPU and CPU.

PCIe (6.0) has a transfer rate of 64 GT/s which result in a maximum of 121.000 GB/s

In modern systems the CPU connects to the I/O chip, which is responsible for every other I/O, with DMI (theoretical max. Bandwidth of 7.86 GB/s)




### Communication Protocol

Each device has 3 registers: Status, Command and Data.

1. OS waits for the device to change the status register to ready (**polling**).
2. OS sends data to the data register.
3. OS sends a command to the command register to tell the device what to do with the data.
4. OS is polling to see if the device has finished or produced an error

### Polling is inefficient

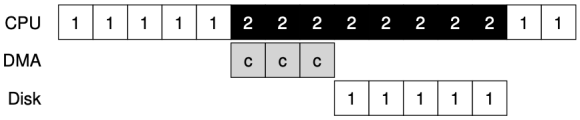
As we know each time an I/O request is issued we throw an interrupt to let the OS do different things while the I/O device is working. However this might not be the best in terms of performance, since an I/O device might be faster then the time it takes to perform a context switch.

To solve this a hybrid approach is the solution. The OS polls for a while and issues a context switch when it takes too long.

### Copying data is inefficient

When data is copied the CPU is occupied the entire time to copy. This is also really inefficient and the time would be better used by running another process.

To solve this, we introduce the DMA, which is a device that can transfer data between main memory and a device without help from the CPU. Thus the CPU can do other tasks while the DMA is copying data.

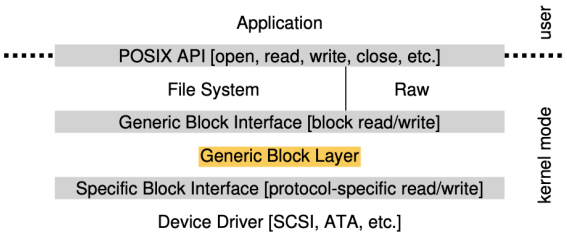



### How can a device communicate with the OS at all?

The OS needs to communicate with the device in some way.

The issue: Each device works differently and has different registers

Solution: The OS has a load of device drivers for different devices. (e.g. about 70% of the Linux kernel consists of drivers). To make sure the OS can always work efficiently we have an abstaction layer the OS can write to. These commands then get assigned to the correct driver. (This is the reason drivers are such a big security concern, since each driver lays deep in the kernel and has a lot of rights)




Questions:

A HDD consists of a large number of numbered sectors (512-byte blocks) acting like an array. Each sector can be written to atomically. Each disk is made out of aluminum coated with a layer of ferro magnetic material. The disk can be read and written by an arm, hovering over the disk and reading or changing the magnetic charges. The disk typically spins between 7200RPM and 15000RPM.

To read from or write to a disk, the disk needs to go through different phases:

1. Acceleration - the disk speeds up as the arm is searching
2. Coasting - the disk is moving at full speed and the arm is searching
3. Deceleration - the disk decelerates to get to the right sector
4. Settling - the disk settles over the correct sector

Setteling can be the most time consuming part, since it always needs to be accurate. This part can take 0.5ms up to 2ms.

**I/O Time: Doing the math**

Time to perform an I/O request:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

Rate of I/O:

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$

Average seek distance:

$$D_{Seek} = \frac{1}{3}N$$

**Disk scheduling**

Because I/O requests can be pretty costly the scheduling of the requests are important. Since the disc scheduler can estimate the time a specific request will take we can build some really nice schedulers:

SSTF (Shortest Seek Time First)

with this scheduling algorithm we can ensure the disk is only moving as much as needed since we always go to the nearest block (NBF). However if a lot of request happen to be to the inner tracks, requests for blocks in an outer track could **starve**.

Elevator (a.k.a SCAN or C-SCAN)

This scheduling algorithm moves the head back and forth between inner and outer tracks and ensures each track gets accessed. C-SCAN even goes a step further and only moves from outer to inner and then resets to the outer track, ensuring each track gets the same chances. However, this is still not the best algorithm

SPTF (Shortest Positioning Time First)

This algorithm is rather difficult to implement, since we need in depth information. If seek time is faster then rotation, a block on any track that is nearer in rotational direction would be faster to access. If rotation is faster then seek, any block anywhere on the same track would be faster to access.

**Other scheduling issues**

If the scheduler gets requests that are close to each other, even when another request is in between, it can **merge** multiple requests to a single one to improve time, since Sequential requests are always faster then random requests.

Another issue the OS has to encounter is, when to issue an I/O request to disk. If it works **work-conserving**, each request is serviced immediately. By working **non-work-conserving** we sometimes wait for a better request so we can get an overall better performance.

Type to enter text

Type to enter text

Type to enter text

Questions:

| Short | Translation |
|---|---|
| CPU | Core Processing Unit |
| GPU | Graphical Processing Unit |
| MMU | Memory Management Unit |
| TLB | Translation Look aside Buffer |
| VPN | Virtual Page Number |
| PFN | Physical Frame Number |
| PPN | Physical Page Number |
| PTE | Page Table Entry |
| PDE | Page Directory Entry |
| PDI, PDIndex | Page Directory Index |
| ASID | Address Space Identifier |
| DBMS | Database Management System |
| LW/HW | Low Watermark/ High Watermark |
| AMAT | Average Memory Access Time ($AMAT = T_M + (P_{Miss} * T_D)$) |
| LRU | Last Recently Used |
| FIFO | First-In First-Out |
| LFU | Last Frequently Used |
| PCB | Process Controll Block |
| TCB | Thread Control Block |
| USB | Universal Serial Bus |
| eSATA | External Serial Advanced Technology Attachment |
| PCIe | Peripheral Component Interconnect Express |
| DMI | Direct Media Interface |
| DMA | Direct Memory Access |
| HDD | Hard Disk Drive |
| RPM | Rotations Per Minute |
| SSTF | Shortest Seek Time First |
| SPTF | Shortest Positioning Time First |