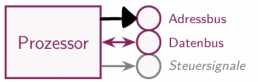
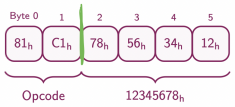
1 BSYS1 - Marco Agostini

1.1 Computermodell



Computermodell
AB: Adresse
DB: Daten
SS: Instruktion



Jede Instruktion hat eine eigenen Codierung bestehend aus dem Opcode und den Daten. (Achtung Little Endian!)

1.2 Assembler

Der Assembler ist ein Compiler, der textuelle Befehle in Maschienencode übersetzt. (Assembler/Assembly language) **Byteweise Adressierung** Intel-Prozessoren addressieren einzelne Bytes. NASM übersetzt jede Anweisung direkt in Binärzahlen und schreibt die Bytesequenz in die Datei.

Endianes/Byte order
Folgende Daten können spezifiziert werden.

i: Adressen der Bytes
b_1 b_2: Stellenwerte
Word 2Byte, Doubleword 4Byte
Quadword 8Byte, D. Quadword 16Byte

Offset in Byte Sequenzen: Jedes Byte in der Bytesequenz erhält einen Offset (Adresse/Index). **Labels:** Intern assoziiert der Assembler das Label mit dem Offset des nachfolgenden Befehls. (Wird nicht in Bytecode übersetzt) **Flat-Form Binaries (Plain,Pure,Raw)** Bytesequenz analog zum Quelltext. **Object-Files** Enthalten neben der Bytesequenz auch noch weiter Informationen: Symboltabelle. Die Symboltabelle assoziiert die Labels mit dem Offset.

Logische Operationen Assembler

```
not rax
and rax, rbx
or rax, rbx
xor rax, rbx
```

Flags Eigenständige Bits, die eine Bedeutung haben. Liegen im gemeinsamen Register RFLAGS, das nicht direkt verwendet werden kann.

CF = Carry Flags	Überlauf unsigned Arithmetik
OF = Overflow Flag	Überlauf signed Arithmetik
PF = Parity Flag	Niederwertigstes Byte, gerade Zahl gesetzter Bits
SF = Sign Flag	Entspricht dem höchstwertigsten Bit
ZF = Zero Flag	Wird gesetzt, wenn das Resultat 0 ist

Condition Codes Definieren eine Bedingung für einen Befehle. Der Befehl wird ausgeführt, wenn der Condition Code TRUE ist.

```
mov rax, [p]          Sets ZF = 1 if [p] == 0
cmovnz rax, [q]       Sets rax =[q] if [p] != 0
```

Compare Das gleiche wie sub, setzt aber nur die flags. *Test* Ist ein and ohne Operation und wird verwendet die Register auf 0 zu setzen.

```
mov rax, [uy]
mov rbx, [ux]
cmp rax, rbx          test rax, rbx
cmove rax, 5          cmovz rax, rbx
```

CC	Name	Flags
A	Above	CF = 0 und ZF = 0
AE	Above or Equal	CF = 0
B	Below	CF = 1
BE	Below or Equal	CF = 1 oder ZF = 1
E	Equal	ZF = 1
G	Greater	ZF = 0 and SF = OF
GE	Greater or Equal	SF = OF
L	Less	SF ≠ OF
LE	Less or Equal	ZF = 1 und SF ≠ OF
PE	Parity Even	PF = 1
PO	Parity Odd	PF = 0

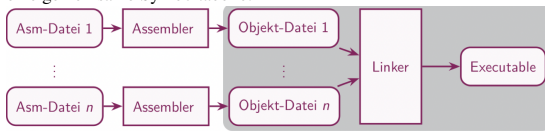
Signed: L / G
Unsigned: B / A

Conditional Jumps Benötigen einen Condition Code und springen nur wenn dieser erfüllt ist. Vergleich C und Assembly.

```
                      mov eax, [ux]
                      cmp max, 2
if(ux < 3 ) {         ja else_body
 /* if-body */       jmp after_if
 } else {             else_body:
 /* else-body */      :else-body
 }                    after_if:
```

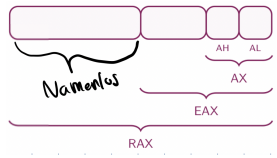
1.3 Linker

Programme werden aus mehreren Assemblerdateien erstellt. Diese Dateien müssen zu einem Executable gelinkt werden. Diese Dateien erhalte eine gemeinsame Symboltabelle.



1.4 Intel 64 Architektur

Ursprünglich hatten die Intel-Prozessoren 16-Bit Register. Auf Intel-64 werden 64 Bit-Register verwendet.



RAX: Accumulator
RCX: Counter
RDX: Pointer für I/O
RBX: Basepointer
RSI,RDI: Quell und Zielindizes- für Stringopeationen
RSP, Stackpointer, Adresse Stack
RSB Basepointer, Addresse in Stack
R8-R15: Zusätzliche Register

Länge der Instruktionen: 1 - 15 Byte **Grösse Operanden** Können 8,16,32,64 Bit betragen, müssen aber gleich gross sein (In der gleichen Registergrösse).

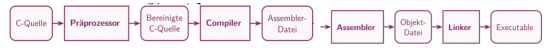
Grundegende Operationen in Assembler

`mov rax, rbx`	Setze Inhalt von rax gleich rbx
`mov rax, 0x8000`	Setze rax gleich 8000_h
`mov rax, [0x8000]`	Setze rax gleich Speicher 8000_h - 8007_h

Adresierungsmodi

`mov rax, [0x8000]`	**Displacement** Adresse
`mov rbx, 0x8000`	**Base**
`mov rax, [rbx]`	Adresse steht in einem Register
`mov Rex, 0x1000`	**Scaled Index**
`mov rax, [rcx * 8]`	Skalierte Adresse

1.5 C-Toolchain



C-Sprachebenen *Präprozessor:* Definiert Direktiven, die später als Tokenersetzung durchgeführt werden. *Basiskonstrukte:* Grundgerüst des Programmes: Variablen, Schliefen, Verzweigungen *Standardbibliotheken:* Stellen Funktionen und Typen bereit, die die Basis-Funktionalität enthalten.

Präprozessor
1. Entfernt Kommentare und wandelt forgesetzte Zeilen in eine Zeile.
2. Teilt den gesamten Code in Tokens ein (Tokenization). Der Präpozessor versucht immer die gröstmögliche Tokens zu erstellen.
3. Führt Präprozessor-Direktiven aus, ersetzt Makros durch ihre Expansion

Token-Klassen Bezeichner (Identifier), Präpozessor-Zahlen,String ("Hello") und Char-Literale ('x'), Operatoren und Satzzeichen (punctuators), Sonstige. (Nachfolgend Operatoren und Satzzeichen)

```
( ) [ ] { } . , : ; ? ... -> # ##
= + - * / % & | ^ ~ ! << >> == != < > <= >= && ||
*= /= %= += -= ++ -- <<= >>= &= |= ^=
```

Bezeichner (Identifier) sind das Gegenstück zu den Labels in Assembler. Bezeichner können deklariert und definiert werden.

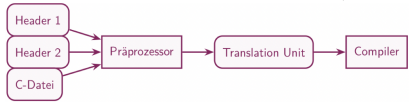
Präprozessor-Direktiven Ist das Token auf einer Zeile #, wird das nächste Token als Direktive interpretiert. Beide Token werden entfernt und die entsprechende Direktive ausgeführt. (#include, #define, #if, #else, #endif)

#Include Präpozessor öffnet die Datei anhand des nächsten Tokens und führt Durchläufe 1 - 3 für hile.h durch. (<file.h>Suche in Systemverzeichnis, "file.h"sucht im aktuellen Verzeichnis + Systemverzeichnis)

Objektartige Makros (#define XYZ 123) Der Präprozessor ersetzt im Programmcode nach der Definition des Makros jedes Token, das dem Makronamen entspricht.

Die *Wiederholte Expansion* ist, wenn der Präprozessor die Ersetzung auf weitere Token prüft und diese ersetzt.

C Translation Unit Der Präprozessor kann mehrere Dateien zu einer Translation Unit zusammenführen. (Vorbereitetes Sourcefile).



Globale Variablen Haben immer einen Typ und einen Bezeichner und werden vor dem Main und ausserhalb von Funktionen definiert. Umfasst einen Speicherbereich mit einem Label darauf. Mit einem Header kann sichergestellt werden, dass mehrere C-Dateien dieselbe Deklaration sehen. Wird eine Globale Variable nicht definiert erhält sie Wert 0.

```
                global x
int x = 12;     x: dd 15
```

Definition einer globalen Variable in C und Assembler.

Lokale Variablen Leben innerhalb einer Funktion und müssen bei jedem Funktionsaufruf neu angelegt werden. **Objekte** sind ein zusammenhängender Speicherbereich.

Variablen Jede Variable ist ein Objekt, aber nicht jedes Objekt muss einer Variable entsprechen. Rückgabewerte von Funktionen werden in Objekten zurückgegeben.

Typen C kennt verschiedene Arten von Typen: Basistypen, Abgeleitete Typen, Enumerationen. Diese hängen von der verwendeten Maschine sowie Compiler ab.

```
sizeof(T)
sizeof(char) = 1;
```

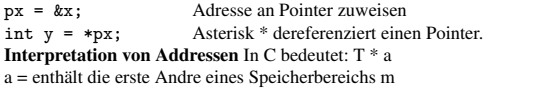
Die Grösse von Typen wird in vielfachen von char angegeben. Auf Intel 64 1 Byte.

`signed char`	≥ 8 Bit
`short int`	≥ 16 Bit
`int`	≥ 16 Bit und ≥ `short int`, soll der «natürlichen» Grösse der Architektur der Ausführungsumgebung entsprechen
`long int`	≥ 32 Bit und ≥ `int`
`long long int`	≥ 64 Bit und ≥ `long`

Ausdrücke C übernimmt die Verwaltung der Register. Stattdessen schreiben wir Formel-ähnliche Ausdrücke. Jeder Ausdruck hat einen Typen, den der Compiler aus den Operanden ableitet (Konstanten, Literale, Bezeichner).

Pointer Auf Maschienencodeebene gibt es keine Variablen, sondern nur Adressen. Die Adresse eines Objektes, dessen Typ nicht bekannt ist, ist vom Typ (void *). Die Addition eines integers zu einem Pointer berücksichtigt sizeof(T)

Die Adresse eines Objektes vom Typ T ist vom Typ *.




```
int x = 5;          & erzeugt die Adresse eines Ausdruckes.
int *px = &x;       Pointer syntax [Datentyp *Bezeichner]
px = &x;            Adresse an Pointer zuweisen
int y = *px;        Asterisk * dereferenziert einen Pointer.
```

Interpretation von Addressen In C bedeutet: T * a
a = enthält die erste Andre eines Speicherbereichs m
m = umfasst sizeof(T) Maschienenbytes. (a bis a + sizeof(T)-1)

Index-Operatoren a[b] ist definiert als *(a + b)
Ein Operand miss eine typisierte Andre sein (a) Der andere Operand (b) muss ein Integer sein
Das *Asterisk* * hat mehrere Bedeutungen. Es bezeichnet den Typ Pointer. Als Operator dereferenziert er eine Adresse. Im Arithmetischen beutetet er multiplizieren.

Arrays

```
T var[n]            Definition
int32_t a[10]
a[1] = 0x42;        Labels eines Arrays als Pointer
```

Die Grösse eines Arrays gibt die Anzahl Maschienebytes zurück. Die Anzahl Elemente erhält man (Arraygrösse / Elementgrösse)

size_t Unsigned Datentyp, der gross genus ist, um Grösse beliebiger Objekte zu halten. (Auch Iteration über Arrays)

Null-terminierte Strings char-Array, in welchem das letze Element \0 (ASCII 0) ist.

String-Literale Entsprechen einer Sequenz von char und enden mit einer impliziten \0. Werden an einem speziellen Ort gespeichert.

```
char * s = "Hai"; // same as following lines:

char c[] = {'H', 'a', 'i', '\0'};
char * s = c;
```

Const (Konstanten) Definiert einen Wert, der nach der Initialisierung nicht mehr geändert werden kann. Der Wert kann sich durch äussere Einflüsse ändern.

Strict (Strukturierte Variablen) Strukturierte globale Variablen mit verschiedenen Datentypen.

vollständiger Struct: Compiler hat genug Informationen um die Grösse eines Objektes zu bestimmen. *unvollständige Typen* werden durch Forward-Deklaration eingeführt.

Grundbegriffe der Logik

Logische Funktion Funktion von n Bits auf 1 Bit. *Parameter* Variable zur Übergabe von Werten an eine Funktion. *Argument* Wert eines Parameters bei konkreter Verwendung der Funktion.

Arität von Funktionen: Es gibt vier unäre Funktionen (False(x), True(x), id(x), not(x).

Es gibt nulläre (null Parameter), Unäre, Binäre, Tender und n-Äre Funktionen. XOR, NAND - wichtige binäre Funktionen Grössere Zahlen als Argumente - Funktionen müssen nicht von allen Bits abhängig sein. Bitwise AND, OR, NOR

Bitwise und Logische Operatoren in C

```
not: z = ~q             Logische Operatoren:
and: z = q & p          z = !q
or: z = q | p           z = q && a
xor: z = q ^ p          z = q || a
```

Funktionen Shifts in C

```
z = q << p
z = q >> p
```

Beim Rechts-Shift wählt der Compiler die Instruktion abhängig vom Typ. (Unsigned:shr/Signed:sar) Einen Signed Links-Shift gibt es nicht.

Funktionen Aller ausführbarer Code in C muss in Funktionen stehen. *Funktionsdeklaration:* Typ des Rückgabewertes, Bezeichner, Parameterliste in Klammern. Funktionen mit dem Typ void sind leer, aber Funktion mit () kann die Parameterliste irgendwas sein. Jede Funktion hat eine Adresse somit kann diese als Variable gespeichert werden.

Funktion printf
Dient der Ausgabe auf stdout und in stdio.h definiert.

```
int i = 20;
printf("Integer = %d \n", i);
```

Die Format-Zeichen für die Funktion printf.

Jede Konvertierungs-Spezifikation beginnt mit % gefolgt von einem Format-Zeichen:

- `%i`: Die nächsten `sizeof(int)` Bytes vom Stack als **signed Dezimalzahl**
- `%u`: Die nächsten `sizeof(int)` Bytes vom Stack als **unsigned Dezimalzahl**
- `%x, %X`: Die nächsten `sizeof(int)` Bytes vom Stack als **Hexadezimalzahl**
- `%li`: Die nächsten `sizeof(long)` Bytes vom Stack als signed Dezimalzahl
- `%lli`: Die nächsten `sizeof(long long)` Bytes vom Stack als signed Dezimalzahl
- `%p`: Die nächsten `sizeof(void *)` Bytes vom Stack als Pointer (hexadezimal)
- `%s`: Interpretiert die nächsten `sizeof(char *)` Bytes als Pointer auf einen **null-terminierten String**

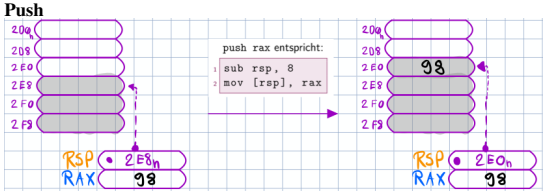
Datentypen Existieren nicht auf Maschinen-Code-Ebene. *Maschinen-Byte:* kleinste Menge an Bits mit eigener Adresse (Intel64: 8). *Maschinen-Wort:* grösste Menge an Bits, die ein Prozessor in einem Zyklus bearbeiten kann.

Typen-Alias Mit typedef kann ein bestehender Typ einen weiteren Namen (Alias) erhalten. Der Compiler behandelt alle Aliase gleich.Es gibt vordefinierte Aliase.

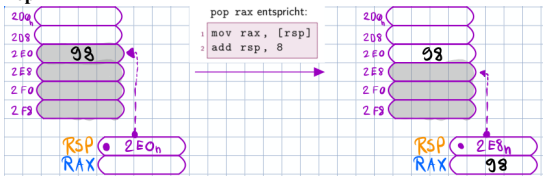
1.6 Stack

Umfasst einen Stackpointer, Operationen: push, pop. *Intel Stack*: Hat 8 Byte grosse Elemente, Oberste E. liegt an niedrigster Adresse, Der Stack wächst in Richtung der niedrigsten Adresse. Der Stackpointer (RSP) zeigt immer auf das zuletzt abgelegte Objekt.

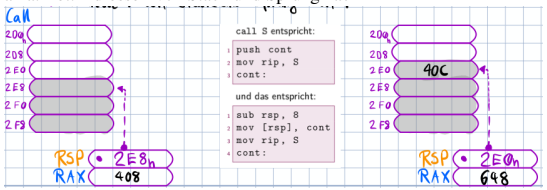
Push



Pop



Call und RET Call a: Nachfolgende Adresse r auf den stack und Sprung an a. Ret: Adresse r von Stack und Sprung nach r



Frame Pointer
Die Adressen des Stacks sind relativ zum Frame Pointer. Dafür wird für den Stack ein Prolog und Epilog benötigt.

```
push rbp                mov rsp, rbp
mov rbp, rsp            pop rbp
```

Calling Convention Vereinbarung zwischen Caller und Calle, welche Register die Funktion verändern darf und in welchen Registern die Argumente der Funktion übergeben werden.

1.7 Mathematische Grundlagen

Addititon Funktioniert wie bei den Dezimalzahlen.

Subtraktion Ist der Miuend kleiner als der Subtrahend, erhöht man den Minuenden um 2ñ auf eine (n+1)-stellige Zahl.

Vorzeichenbehaftete Ganzzahlen
Unsigned Integer: Wertebereich: 0 bis $2^n - 1$
Signed Integer: Wertebereich: $-(2^{n-1})$ bis $-(2^{n-1}) - 1$

Einer- und Zweierkomplement
Sind in der läge auch negative Zahlenwerte darzustellen. Im Einerkomplement wird das MSB als Vorzeichen interpretiert. Problem - zwei mögliche Nullen. Das Zweierkomplement behebt dies.
Inversionsverfahren 1: b - 1, invertieren
Inversionsverfahren 2: b invertieren, b + 1

Arithmetische Operationen in Assembler und C

`add z, q`	$z \leftarrow z+q$		$z = q + p$	$z \leftarrow q+p$
`sub z, q`	$z \leftarrow z-q$		$z = q - p$	$z \leftarrow q-p$
`adc z, q`	$z \leftarrow z+q+c$		$z = -q$	$z \leftarrow 0-q$
`sbb z, q`	$z \leftarrow z-q-c$		$z = q\ ++$	$z \leftarrow q+1$ und $q \leftarrow q+1$, «Erst erhöhen, dann auslesen»
`neg z`	$z \leftarrow 0-z$	Zweierkomplement	$z = q\ ++$	$z \leftarrow q$ und $q \leftarrow q+1$, «Erst auslesen, dann erhöhen»
`inc z`	$z \leftarrow z+1$	Inkrement	$z = --q$	$z \leftarrow q-1$ und $q \leftarrow q-1$
`dec z`	$z \leftarrow z-1$	Dekrement	$z = q\ --$	$z \leftarrow q$ und $q \leftarrow q-1$

Links- und Rechts-Shift
Links-Shift um n Bits: $b * 2^n$, *Rechts-Shift um n Bits:* $b/2^n$
Sign-Extension Wenn eine n-Bit Zahl auf eine n+m-Speicherstelle kopiert wird, werden die oberen m Bits auf 0 gesetzt. Dies ändert die Bedeutung der Zahl (wenn sie im Zweierkomplement geschrieben ist). Die Sign-Extension füllt die oberen m Bits mit 1-en auf.
Shifts und Rotate in Assembler Rotates füllen statt mit 0 oder 1 mit den ursprünglichen Bits auf.

`shl z, i`	$z \leftarrow z \cdot 2^i$	
`shr z, i`	$z \leftarrow z \cdot 2^{-i}$	z unsigned
`sar z, i`	$z \leftarrow z \cdot 2^{-i}$	z signed
`rol z, i`	Left-Rotate i Bits	
`ror z, i`	Right-Rotate i Bits	

Signed Multiplikation in Assembler

```
imul r, z
```

`mul r, z` *imul* ist das signed Äquivalent zu *mul* auf Assembler.

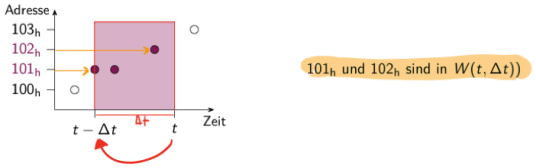
Signed Divison in Assembler

```
idiv r, z
```

`div r, z` *idiv* ist das signed Äquivalent zu *div* auf Assembler.

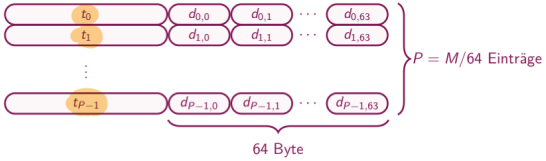
1.8 Hauptspeicher und Cache

Physikalische Grenzen der Computertechnik Elektrische Energie 2/3 der Lichtgeschwindigkeit. Somit kann eine 3GHz CPU pro Takt nur 10cm weit kommunizieren. CMOS-Logik verbraucht Energie, wenn Transistoren schalten.
Speicherhierarchien Ein Computer Verwendet verschiedene Arten von Speicher. Höherwertiger Speicher: (Ram, Cache): niedrige Distanz CPU, Hohe Geschwindigkeiten, Geringe Fehleranfälligkeit, Geringe Sicherheitsanforderungen, Niederwertiger Speicher: (Tape, HDD): Preis, Kapazität, Transfereinheiten.
Lokalitätsdiagramm Speicherzugriff im zeitlichen Verlauf. Arbeitsbereich W(t, delta t). *Räumliche Lokalität:* Arbeitsbereich enthält nahe beieinander liegende Speicherstellen. *Zeitliche Lokalität:* Auf Speicher, auf den kürzlich zugegriffen wurde, wird bald wieder zugegriffen.

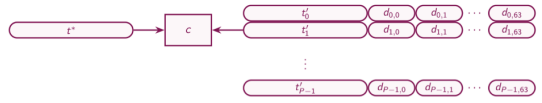


Cache ist ein Zwischenspeicher, der kleiner und schneller ist. *Nutzen:* Ergibt sich aus dem Lokalitätsprinzip.
Durchschnittliche Zugriffszeit
E(T) = p_c * T_c + (1 - p_c) * T_m
T_c = Zugriffszeit auf den Cache
T_m = Zugriffszeit auf den Hauptspeicher
p_c = Wahrscheinlichkeit eines Cache-Hits (oft > 0.9)

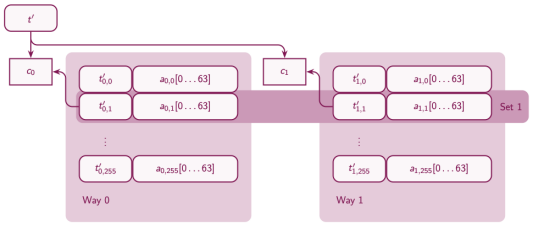
Fully-Associate-Caches
Chachezeilen: besteht aus 64 Byte. Somit umfasst jede Cachezeile die Adressen c_k+0 bis c_k+63.
Die Adresse Besteht aus Tag und Offset. Das Tag der Zeile sind die oberen (n-6) bits $t_k = c_k / 2^6$, Offset = untere 6 Bits.
Nutzen den Lokalitätseffekt best-möglich aus, sind aber teuer zu implementieren.



Direct-Mapped Caches
Der DMC ist sehr einfach zu implementierender Cache mit einem schnellen Lookup. Hat aber viele Kollisionen.
Die Adresse besteht aus Tag, Index, Offset.



k-way Set-Associative Cache
Eine parallele Verwendung von k Direct-Mapped Caches mit jeweils P / k Einträgen. Jede Chachezeile kann ich k verschiedenen Cacheeinträgen gespeichert werden. (Set) Der SAC ist ein Kompromiss zwischen FAC und DMC: weniger komplex als FAC, weniger Kollisionen als DMC, genauso schnell wie FAC und DMC. Eine stelle teilt sich ($HS * ways$ / Cachegrösse mit WAY-stellen den Platz).



1.9 Dynamischer Speicher

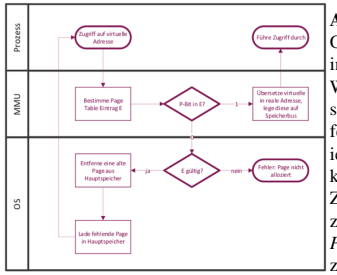
Der Heap ist ein Speicherbereich für vollständig dynamischen Speicher. Dieser wird vom OS verwaltet.
Implizite Speicherfreigabe
Virtuelle Laufzeitumgebungen (Java, .NET) schränken Pointer stark ein. Dafür gibt es keine Speicherlecks. Keine Kontrolle über Zeitpunkt Speicherfreigabe, Zeitverhalten inderterministisch, Zeit- und Speicheroverhead zur Laufzeit.
Explizite Speicherfreigabe
Programmiere bestimmt Zeitpunkte für Speicherfreigabe und Reservationen explizit. (In C malloc und free). Beachten das die Dualität beibehalten wird. **Interne Fragmentierung** Wenn eine Heap-Implementierung einen grösseren Speicherblock reserviert als benötigt. *Mögliche Lösung:* Programm übernimmt selbst feingranulare Verwaltung (Java, .NET, DBMS) **Externe Fragmentierung** Programm reserviert immer wieder Speicher und gibt ihn unregelmässig frei. Programmierer können das Problem umgehen. (Object Pools, Komposition statt Aggregation).
Suchalgorithmen
First Fit: Wählt erste passende Lücke am Anfang. *Next Fit:* Wählt erste passende Lücke nach zuletzt reservierten Bereich. *Best Fit:* Durchsucht alle Lücken und wählt beste aus. *Worst Fit:* Durchsucht alle Lücken und wählt grösste aus. *Quick Fit:* Wählt erstes passendes Element aus der Liste.
Buddy-System Variante des Verfahrens Grössenklassen mit Zweierpotenzen von 2^m bis 2^n. (2^m kleinste Speicherbereichsgrösse, 2^n gesamter Speicher) Wird ein 2^k-Bereich in zwei 2^{k-1}-Buddies geteilt, müssen deren Startadressen die untersten k-1 Bits = 0.
Object Pools Ein Speicherbereich fester Grösse (Page) wird in kleinere Bereiche mit gleicher Grösse unterteil (Objekte).

1.10 Virtueller Speicher HW

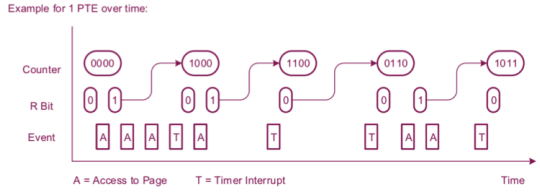
Virtuelle Adressen Das OS gibt dem Prozess keine reale Adressen, sondern virtuelle. Die MMU übersetzt virtuelle in reale Adressen. (Physische = reale Adresse, logische = virtuelle = lineare Adresse)
Gültiger Zugriff: Prozess will auf gültige Adresse zugreifen, MMU findet mapping in Mappingtabelle, MMU legt reale Adresse auf den Speicherbus, Prozessor lies/schreibt Daten von/auf Speicherbus.
Ungültiger Zugriff: Prozessor will auf ungültige Adresse zugreifen, MMU stellt fehlendes Mapping fest, MMU signalisiert Fault-Interrupt, CPU ruft OS-Interrupt-Handler auf, OS Memory Manager übernimmt.
Effekte: Mehr Speicher pro Prozess (Prozess hat mehr Speicher als Hauptspeicher vorhanden), Mehr Speicher als reale Adressbits ermöglichen.
Monoprogrammierung: Der Prozess kennt keine anderen Prozesse
Seitenbasierter Virtueller Speicher Verwaltungseinheit: Seite bzw. Page, typischerweise 4KB (entspricht 12 Bit offsets). Der Hauptspeicher besteht aus Page Frames. Die Page repräsentiert die Daten, Page ist kein Speicher, sondern benötigt einen Page Frame. Page Number = Startadresse der Page ohne Offsetbits. Das OS entscheidet, welche Pages wann wo liegen müssen. MMU kennt nur den Hauptspeicher und kann nur das Fehlen einer Page bemerken.

1.11 Virtueller Speicher SW

Page-Table Jede virtuelle Adresse wird via Page-Table in reale Adresse übersetzt. (Single-Level, Mutei-Level, Inverted, Hashtable) Jeder Eintrag id der Page-Table enthält zusätzlich zwei Statusbits. *Translation Lookaside Buffer:* Cache für häufig benötigte Mappings (keine Daten).
Paging
Auf Intel x86 hat jeder Page-Table Eintrag 32 Bits. Das Unterste Bit ist das P-Bit (Present). Wenn dieses auf 1 steht, ist die Page im Hauptspeicher. Zusätzlich gibt es das A-Bit und D-Bit.



Ablauf im Interrupt
OS prüft, ob Lokalisierungsinformationen gültig sind. Wenn HS voll, OS entscheide, welche Page entfernt werden soll. OS initiiert Page-Transfer von sekundärem Speicher. In der Zwischenzeit laufen die Prozesse weiter.
Page geladen OS setzt Prozess mit Instruktion fort. test
Dreschen (Threshing)
Bschreibt das Problem von zu häufigem Pagen. (HS zu klein, zu viele Prozesse)
Teilstrategien Ladestrategien (fetching policies), Entladestrategien (cleaning policies), Verdrängungsstrategien (page replacement policies)
Ladestrategien *Demand Paging:* Laden auf Anfrage (minimaler Aufwand, lange Wartezeiten). *Prepaging:* Pages werden frühzeitig geladen durch statistische Systemanalyse (In der Praxis nicht anzutreffen). *Demand Paging mit Prepaging:* Laden auf Anfrage und benachbarte Pages werden mitgeladen. Pages werden in Clustern geladen (4-8). (Weniger Page-Faults, Bocktransfer, möglicherweise zu viele Pages)
Entladestrategien
Demand Cleaning: Entladen auf Nachfrage. Page wird zurückgeschrieben, wenn Frame wieder verwendet werden soll. (Minimaler Aufwand, Erhöhte Wartezeit) *Precleaning:* Modifizierte Pages werden frühzeitig in den sekundären Speicher geschrieben. (Reduzierte Wartezeit, Mehraufwand)
Veränderungsstrategien
Zahlreiche Varianten: FIFO, Second Chance, Clock. Massiver Einfluss auf die Systemperformance. MMU setzt die Statusbits und das OS löscht sie.
Optimum: Ersetzt die Pages, die am spätesten in der Zukunft gebraucht wird. *FIFO:* Ersetzt jeweils die älteste Seite. Benötigt keine Statusbits. Problem der Bélàdys Anomalie: Grösserer HS kann zu mehr Page Faults führen.
Second Chance: Erweiterung von FIFO. Prüft A-Bit der ältesten Page. Wenn A = 0 - Entladen der Page. Wenn A = 1, OS löscht A-Bit und schiebt Page ans ende der Linked List.
Clock: Linked-List wird zum Kreis (=Clock) und ein Pointer wird verwendet.
LRU - Last Recently Used: Ersetzt die am längsten unbenutzte Page. Bei jedem Zugriff notiert MMU den Zeitpunkt. (Grosse Aufwand in Hardware)
NFU - Not Frequently Used: Benötigt einen zusätzlichen Counter in der Page-Table. Wenn es einen oder mehere Zugriff*e gab, Counter erhöht. Ersetzt die Page mit dem niedrigstem Counter. Problem: alte pages können lange bleiben.
NFU mit Aging: Counter gewichtet nach Zeit. Pro Page ein n-Bit counter.



Working Set: Behalte Pages vom Working Set (mit Intervall T). Pro Page-Table-Eintrag ein Zeitstempel T. Statt Timer-Interrupt: Scanne alle Pages mit Fault-Interrupt. Wenn A = 1, setze t = now, und A = 0. Wenn A = 0, wenn now -t <T Page im WS und bleibt oder -t >= T Page nicht im WS und wird entfernt.