Nico Fehr | Ostschweizer Fachhochschule

Rechenhilfe

B	H	D
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

		Präfix
2^{10}	K	Kilo
2^{20}	M	Mega
2^{30}	G	Giga
2^{40}	T	Tera
2^{50}	P	Peta
2^{60}	E	Exa

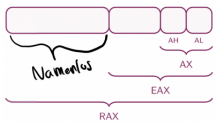
$2^0 = 1$	$2^5 = 32$
$2^1 = 2$	$2^6 = 64$
$2^2 = 4$	$2^7 = 128$
$2^3 = 8$	$2^8 = 256$
$2^4 = 16$	$2^9 = 512$

Little-Endian & Big-Endian (16 Bit Beispiele)

BE	$CAFE_h$
LE	$FECA_h$
BE	0065_h
LE	6500_h

Zweierkomplement:
$N(0101\ 1101_b)$

1. Invertieren: $1010\ 0010_b$
2. Eine 1 addieren: $0000\ 0001_b$
3. Resultat: $1010\ 0011_b$



Register

Es gibt 8 (AH, AL), 16 (AX), 32 (EAX), 64 (RAX) Bit Register

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

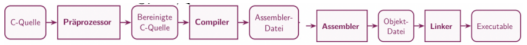
Assembler: Grundlegende Operationen

`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

Assembler: Addressierung

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

C-Toolchain



Präprozessor

1. **Durchlauf:** Entfernen von Kommentaren
2. **Durchlauf:** Tokenization von gesamten Sourcecode in Bezeichner (identifiers, variablen, typennamen), Präprozessor Zahlen, String und Char Literals, Operatoren und Satzzeichen (punctuators)
3. **Durchlauf:** Token Liste wird mehrmals durchlaufen und führt Prozessor-Direktiven aus und ersetzt Makros. Es gibt objektartige und funktionsartige Makros.

Makro wird wie folgt definiert

```
#define Name Tokenliste
#define XYZ 123
```

Beispiel:

```
#define ABC XYZ
#define XYZ 123
int y = ABC;

-------
int y = XYZ
int y = 123
```

Typen in C

- Typen beschreiben Objekte, bzw. deren Eigenschaften
- Jeder Typ hat feste Objektgrösse sizeof(T)
- Typinformationen existieren nur auf Sprachlevel C
- Durch Typen kann Compiler die richtige Grösse für Objekte bestimmen

Basistypen (Typ | Grösse | Suffix)

Unsigned Integer Basistypen

(char | >= 8Bit |), (unsigned char | >= 8Bit | u), (unsigned short int | >= 16Bit | u), (unsigned int | >= 16 Bit und >= short "natürlicher"Grösse Architektur der Ausführungsumgebung | u), (unsigned long int | >= 32 Bit und >= int | ul (kleines L)), (unsigned long long int | >= 64 Bit und >= long | ull).

Signed Integer Basistypen

(signed char | >= 8Bit |), (short int* | >= 16Bit |), (int* | >= 16 Bit und >= short "natürlicher"Grösse Architektur der Ausführungsumgebung |), (long int* | >= 32 Bit und >= int | l (kleines L)), (long long int* | >= 64 Bit und >= long | ll).

- mit * markiert sind implizit signed.
- Signed kann optional davor geschrieben werden.
- Bool kann werte 0 und 1 annehmen
- void hat keine Werte und wird verwendet, wenn keine Werte zulässig
  - wenn Funktion keinen Rückgabewert hat
  - wenn Funktion keine Argumente hat
  - untypisierte Pointer (cannot be dereferenziert)

Ermitteln von Bit size eines Typen

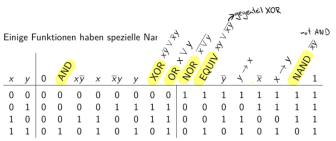
- `sizeof(T)` wird in Vielfachen von `char` angegeben
- `sizeof(char) 1`
- genaue Grösse in `CHAR_BIT` in `<limits.h>`
- Bits eines Typen `T = sizeof(T) * CHAR_BIT`

Pointer

`int x = 5;`	& erzeugt die Addresse eines Ausdruckes.
`int *px = &px;`	Pointer syntax
`px = &x;`	Adresse an Pointer zuweisen
`int y = *px;`	Asterisk * dereferenziert Pointer

%i bedeutet: Die nächsten sizeof(int) Bytes vom Stack werden interpretiert als signed dezimalzahl (int, hex, octal) %d sizeof(int) signed dezimalzahl (int) %u sizeof(int), unsigned dezimalzahl %x, %X sizeof(int) Hexadezimalzahl %li sizeof(long) signed Dezimalzahl %lli sizeof(long long) signed Dezimalzahl %p sizeof(void *) Pointer (Hexadezimalzahl) %s sizeof(char *) Pointer auf einen null-terminierten String

Assembler Instruktionen Hilfstabelle



Data Movement

mov, push, pop

Unary Operations

inc, dec, neg (Zweierkomplement), not

Binary Operations

add, sub (dest - src), imul, xor (dest by src), or (dest by src)
Shift Operations: sal / shl (left shift by bits), sar (arithmetic right shift, signed), shr (logical right shift unsigned), rol (left rotate), ror (right rotate) → Rotates füllen mit ursprünglichen Bits auf (statt 0 auf 1).

Setzt nur Flags

cmp (compares, performs sub but only sets flags), test (logical and, but only set flags)
Conditional Operations (XX = Condition Code): cmovXX, jXX (jmp), setXX

Condition Codes (siehe Flags)

ZE = not equal
Z = equals zero
→ Der Befehl wird nur dann ausgeführt, wenn in diesem Moment die Flags genau so gesetzt sind, wie der Condition Code angibt (siehe oben **Setzt nur Flags**).

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

```
dec rax             ; sets ZF = 1 if rax == 0
cmovz rax, [q]      ; sets rax = [q] if rax == 0
```

Bitweise Operatoren in C

(**not:** ~ tilde), (**and:** &), (**or:** |), (**xor:** ∧) Bitweises NAND: `z = ~(x & y)`
Logisches NAND: `z = !(x && y)`

Shift in C (mit gleichen Operatoren wie Java)

→ Zweierpotenzen als Shift: 2^n entspricht $1 \ll n$
→ Beim Rechtsshift wählt der Compiler die Instruktion abhängig vom Typ aus (unsigned shr / signed = sar).
Folgende Operationen interpretieren ihren Operanden als signed oder unsigned:

- Multiplikation: `mul(u) / imul(s)`
- Division: `div (u) / idiv(s)`
- Rechts-Shift: `shr (u) / sar(s)`

Flags

Flags liegen im gemeinsamen Register RFLAGS

Carry Flag (CF)

- Gibt überlauf in unsigned Arithmetik an
- Entspricht Übertrag bei MSB

$0001 + 1111 = (1)0000$	$\Leftrightarrow$	$1 + 15 = 0$	$\Rightarrow$	$CF = 1$
$0001 + 0111 = 1000$	$\Leftrightarrow$	$1 + 7 = 8$	$\Rightarrow$	$CF = 0$
$0000 - 0001 = (1)1111$	$\Leftrightarrow$	$0 - 1 = 15$	$\Rightarrow$	$CF = 1$
$1000 - 0001 = 0111$	$\Leftrightarrow$	$8 - 1 = 7$	$\Rightarrow$	$CF = 0$

Overflow Flag (OF)

- Gibt Überlauf bei signed Arithmetik an
- Wird gesetzt, wenn Vorzeichen nicht stimmen
- Prozessor weiss nicht, ob bearbeitete Zahl signed oder unsigned ist
- Flags OF und CF werden immer bestimmt
- Programm muss richtiges Flag auswerten

$(+x) + (+y) = (-z)$	$0111 + 0001 = 1000$	$7 + 1 = 8$
$(-x) + (-y) = (+z)$	$1111 + 1000 = (1)0111$	$-1 + (-8) = 7$
$(+x) - (-y) = (-z)$	$0111 - 1000 = (1)1111$	$7 - (-8) = -1$
$(-x) - (+y) = (+z)$	$1000 - 0111 = 0001$	$(-8) - 7 = 1$

Zero Flag (ZF)

Wird gesetzt wenn $Resultat = 0$

Sign Flag (SF)

Entspricht höchstwertigem Bit des Resultats

Parity Flag (PF)

Wird immer gesetzt, wenn das niedrigste Byte des Resultats eine gerade Anzahl an gesetzten Bits enthält.

Stack

Register Basepointer (rbp): Beginn des Stacks (an höheren Addresse)
Register Basepointer (rbp): Aktuelle Addresse im Stack → Beim Rücksprung aus Funktion ist stack und base pointer wieder derselbe

Push und Pop

→ **push** fügt Element dem Stack hinzu, **pop** nimmt das oberste Element (an niedrigster Addresse) wieder runter.

`push rax` entspricht:

```
sub rsp, 8
mov [rsp], rax
```

`pop rax` entspricht:

```
mov rax, [rsp]
add rsp, 8
```

Call und Ret

call s: Nachfolgende Addresse r auf den Stack und Sprung an S.
ret: Addresse r von Stack entfernen und Sprung nach r

`call S` entspricht:

```
push cont
mov rip, S
cont:
```

und das entspricht:

```
sub rsp, 8
mov [rsp], cont
mov rip, S
cont:
```

Speicher

- Physikalische Grenze: Elektrische Energie 2/3 der Lichtgeschwindigkeit
- Eine 3 GHz CPU kann pro Takt nur 10cm weit kommunizieren
- CMOS-Logik verbraucht Energie, wenn Transistoren schalten

Speicherhierarchien

Hochwertiger Speicher (Flüchtiger Speicher)

(Ram, Cache, Register): niedrige Distanz.
CPU, Hohe Geschwindigkeiten, Geringe Fehleranfälligkeit, Geringe Sicherheitsanforderungen

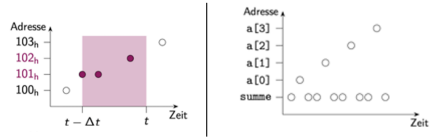
Niederwertiger Speicher (Nichtflüchtiger Speicher)

(Tape, HDD, SSD): Preis, Kapazität, Transfereinheiten.

Merksatz

Moderne Computer benötigen aus physikalischen, logischen und ökonomischen Gründen eine Speicherhierarchie, in der verschiedene Sorten Speicher eingesetzt werden.

Lokalitätsprinzip



Alle relevanten Informationen zum Verständnis eines Zusammenhangs sollen an einer Stelle des Systems lokalisiert und möglichst ohne Kenntnis des Kontext nachvollziehbar sein.

Arbeitsbereich

$W(t, T)$ sind alle Speicherstellen, die im Zeitintervall $[t - T, t]$ referenziert wurden.

Räumliche Lokalität

Zugriff auf benachbarte Speicherstellen. (Wenn Speicherstelle A benötigt wird, dann demnächst vermutlich auch A+1, A+2...) → array

Zeitliche Lokalität

wiederholter Zugriff auf die gleiche Speicherstelle. Wenn A jetzt benötigt wird, dann später vermutlich auch.

Cache

Im Cache werden Hauptspeicheradressen explizit mit den Daten gespeichert. Üblicherweise in Cachezeilen aufgeteilt. In Realität mehrstufige Caches mit verschiedenen Cache-Typen.

Cache-Grösse	Anzahl *Nutz*-Bytes (Payload)
	Platz für Adressen kommt hinzu, aber nicht exp
Cache-Hit	Gesuchte Adresse ist im Cache
Cache-Miss	Gesuchte Adresse ist nicht im Cache
T_C	Zugriffszeit auf den Cache
T_M	Zugriffszeit auf den Hauptspeicher
p_C	Wahrscheinlichkeit eines Cache-Hits (wegen Lokalitätseffekt oft > 0.9)

Durchschnittliche Zugriffszeit: Erwartungswert $E(T) = pc*Tc+(1-pc)*Tm$

Fully-Associative Cache FAC

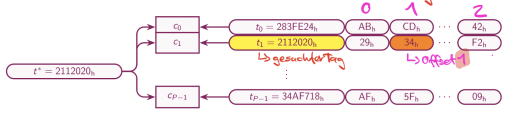
Cachezeilen: üblicherweise 64 Byte. Somit jede Cachezeile Adressen $c_k + 0$ bis $c_k + 63$. Die Adresse besteht aus Tag und Offset. Das Tag der Zeile sind die oberen $n - 6$ Bits $t_k = c_k/26$ Offset = untere 6 Bits. Nutzt Lokalitätsprinzip bestmöglich aus, teuer in HW implementieren

- Gesucht sei der Wert an der (Hauptspeicher-)Adresse $a^* = 84480801_h$

⇒ Tag von 84480801_h ist $t^* = 84480801_h / 2^6 = 8448080_h / 2^2 = \mathbf{2112020_h}$

Offset $j^* = \mathbf{1}$ (6 Bit verwerfen → Offset)

⇒ $t^* = t_1 \Rightarrow d_{1,j^*} = d_{1,1} = 34_h$ (Tag)



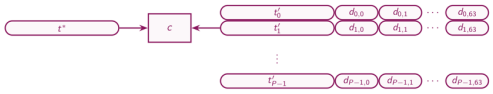
Direct-Mapped Cache DMC

DMC arbeitet mit reduzierten Tags. Zusätzlich zum Offset (6 Bit) bei FAC wird hier eine Anzahl p Bits abgezogen → es entsteht ein Reduziertes Tag t'. Bsp: Aus Tag 1234 wird Red. Tag 12. Offset für Byteermittlung dasselbe wie FAC (j)

- Gegebene Adresse a^* wird in t'^*, e^* und j^* aufgeteilt
- t'^* kann sich nur am Eintrag e^* befinden

⇒ Man benötigt nur einen einzigen Vergleichsbaustein

- Wenn $t'^* = t'_{e^*}$ dann wird $d_{e^*j^*} = [(t'_{e^*} \cdot 2^s + e^*) \cdot 2^6 + j^*]$ zurückgegeben



Problem: Kollisionen mit identischen t'

k-Way Set-Associative Cache (k = Anzahl Ways)

- Parallele Verwendung von mehreren DMCs
- Jeder DMC in SAC heisst Way
- Jede Cachezeile kann in k verschiedenen Cacheeinträgen gespeichert werden
- Way arbeitet autonom
- Set / Entry Nummer ist die Nummer des Eintrags in jedem Way
- Grösse des Way (1 DMC) = Gesamtgrösse des Caches / Anzahl Ways

Berechnung einzelner Bit-Grössen

Offset j = Exponent der Cachezeilenlänge $64B \rightarrow 2^6 \rightarrow 6Bit$
Addresse (aus Hauptspeicher) = Exponent der Grösse des Hauptspeichers
Zeilen im Cache = Nutzdaten / Cachezeilenlänge
Zeilen pro Way = Zeilen im Cache / Anzahl Ways
Eintrag/Entry e = Exponent der Anzahl Zeilen eines Ways
Red. Tag t' = Addresse (aus Hauptspeicher) - Entry - Offset
Overhead = Anzahl Bits pro red. Tag * Zeilen im Cache

Dynamischer Speicher (Heap)

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

Programmierer bestimmt Zeitpunkt für Speicherfreigabe und Reservation explizit. (In C malloc und free). Beachten das Dualität beibehalten wird.

Interne Fragmentierung

Wenn eine Heap Implementierung einen grösseren Speicherblock reserviert als benötigt (Weil z.B. nur vielfaches von 8 Byte alloziiert werden kann). 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 für freie Speicherbereiche

Suchalgorithmen um idealen freien Speicherblock zu finden, wenn neuer Heap alloziiert werden soll.

First Fit

Wählt erste passende unbelegte Speicherlücke am Anfang. (viele kleine Lücken)

Next Fit

Wählt passende Lücke nach zuletzt reserviertem Bereich. resultiert in vielen kl. Lücken, schlechter als First Fit)

Best Fit

Durchsucht alle Lücken nach kleinsten passenden. Grosser Speicherverschleiss, sehr langsam.

Worst Fit

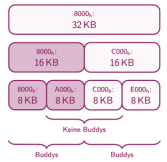
Durchsucht alle Lücken nach grössten passenden. Ebenso langsam, auch keine guten Ergebnisse.

Object Pools

Speicherbereich fester Grösse (Page) wird in kleinere Bereiche gleicher Grösse unterteilt (Objekte).

Buddy System

Eltern-Bereich ist 2^{k+1}-aligned an a:
⇒ In den Adressen beider 2^k-Buddys sind die untersten k Bits alle 0
⇒ Bit k ist 0 in a und 1 in $a + 2^k$
⇒ Zwei Bereiche sind genau dann Buddys, wenn
- sie die gleiche Grösse 2^k haben und
- sich ihre Startadressen genau im Bit k unterscheiden



Virtueller Speicher Hardware

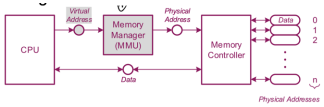
→ OS weist Prozessen Hauptspeicher zu
→ OS schützt Hauptspeicher der Prozesse gegeneinander
→ Intel 64 kennt verschiedene Privilege-Levels: PL0 bis PL3. OS läuft auf PL0 und Prozesse auf PL3. PL2 wenig verwendet, hauptsächlich für Virtualisierungen. OS gibt dem Prozess nur virtuelle Adressen. MMU übersetzt virtuelle Adressen in reale (logisch = virtuelle = lineare Adresse).

Gültiger Zugriff

Prozess will auf gültige Adresse zugreifen, MMU findet mapping in Mappingtabelle, MMU legt reale Adresse auf den Speicherbus, Prozessor liest/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

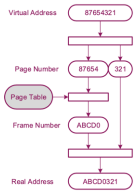
1. Nicht benötigte Inhalte auf sekundären Speicher (Festplatte) aus.
2. Prozess kann mehr Speicher verwenden als Hauptspeicher vorhanden.
3. Monoprogrammierung, Absolute Addressen, Prozess kann gesamten Adressraum nutzen, Prozess kennt keine anderen Prozesse

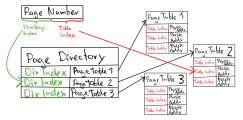
Seitenbasierter Virtueller Speicher

Weitere Arten: Segmentbasiert (legacy), Kombiniert (legacy, Vorteil zu klein)
Page → Ein **virtueller** Block Speicher
Frame → **Physischer** Block Speicher

- Page passt genau in 1 Frame
- verschiedene zusammenhängende Pages müssen nicht direkt nacheinander im Speicher liegen
- Mapping der Pages auf Frames ist Aufgabe des OS
- pro Prozess ein virtueller Addressraum und eine Page Table

Page Tables



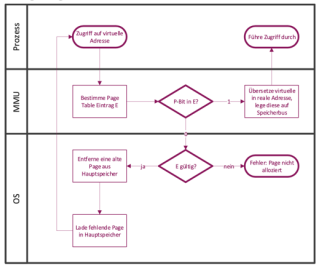


Wird vom OS konfiguriert
4 Implementierungsarten: **(Single-Level)** verbraucht viel Speicher. Grösse hängt von Grösse des virtuellen Addressraums ab. Für jede mögliche Page 1 Eintrag.
(Multi-Level) ist in den meisten Geräten verbaut.
(Inverted), (Hashtable) Transaction Lookaside Buffer: Cache der MMU.

Berechnungen (Single-Lvl) Hilfestellung vorheriges Bild	
Bits für Virtuelle Adresse =	Exponent der Adressraumgrösse → $16MB \rightarrow 2^{24} \rightarrow 24Bits$
Anzahl Pages =	Grösse virtueller Adressraum / Pagegrösse
Offset (relative Adresse innerhalb Page) =	Exponent der Pagegrösse
Bits Page Number =	Bits virtuelle Adresse - Offset
Bits Frame Number =	Exponent Anzahl Frames
Grösse Page Table pro Prozess (ohne Status Bits)	test

Multi Level Page Tables: Page Number wird zusätzlich aufgeteilt in Directory Index und Page Index (können auch mehr sein, je nach Anzahl Lvls). Page Number wird immer durch Anzahl Lvls geteilt (jedes Lvl einen Teil als "Key"). "Value"jedes Lvls ist Pointer zu tieferem Lvl oder Physische Adresse (ohne Offset). Auf x86 hat jeder Page-Table Eintrag 32 Bits.
Present-Bit (P-Bit): Ist Page im Hauptspeicher 1=Ja, 0=Nein
Accessed-Bit MMU: setzt Bit bei jedem Zugriff auf Page. OS löscht in z.B. Zeitintervallen
Dirty-Bit: MMU setzt Bit auf 1 bei jedem Schreibzugriff. OS löscht es, wenn Page zurückgeschrieben wurde.

Paging:



Dreschen

Problem für häufiges Pagen. Wenn Hauptspeicher zu klein und zu viele Prozesse vorhanden → CPU ist nur mit Paging beschäftigt. Dies kann durch mehr Hauptspeicher, Beschränkung der Anzahl Prozesse oder Paging Strategien vermindert werden.

Ladestrategien fetching policies

Demand Paging

Laden nur dann, wenn benötigt. Erfordert jedes Mal ein Interrupt. Minimaler Aufwand, Maximale Ladezeit

Preparing

Wird eine Page geladen, werden zusätzlich andere (welche vermutlich auch gebraucht werden, durch statische Analyse oder Tracken eines Prozesses ermittelt) frühzeitig geladen. Fast nicht verwendet in Praxis.

Kombination

Wird eine Page geladen, werden Nachbaren auch geladen. Weniger Page Faults und Blocktransfer. Möglicherweise nicht benötigte Pages mitgeladen.

Entladestrategien cleaning policies

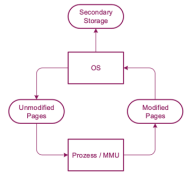
Demand Cleaning

Page zurückschreiben, wenn Frame wiederverwendet werden soll. Minimaler Aufwand, erhöhte Wartezeit.

Precleaning

Vorausschauend schreiben. Modifizierte Pages werden frühzeitig in Sekundärspeicher geschrieben. Reduzierte Wartezeit, mehr Aufwand, evtl unnötiges Schreiben in Sekundärspeicher

Page Buffering



Zwei Listen:
List 1 → Unmodified Pages **U**
List 2 → Modified Pages **M**
OS verschiebt periodisch Pages mit Dirty bit von **U** nach **M**. OS schreibt Pages von **M** in sekundären Speicher und verschiebt sie von **M** nach **U**

Verdrängungsstrategien page replacement policies

Wenn Speicher knapp wird, gibt es verschiedene Strategien, um Pages aus dem Hauptspeicher zu entfernen.

Stategie Optimum

Ersetze Seite, die am spätesten in der Zukunft gebraucht werden wird. (in der Theorie, Praxis nicht möglich) Beste Lösung für das Problem.

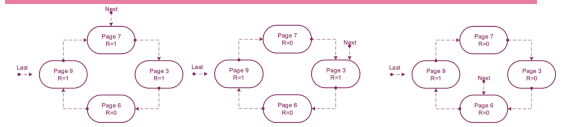
Strategie FiFo First-In-First-Out Linked-List Implementation

Ersetzen der ältesten Seite. Grosser Hauptspeicher führt zu mehr Pagefaults, Alte, häufig benutzte Pages werrden entfernt und gleich wieder geladen, Benötigt keine Statusbits

Second Chance

Wie FIFO, zusätzlich wird geprüft, ob A-Bit gesetzt ist. Wenn ja wird es an Anfang der Linkedlist gesetzt. Wenn Nein wird gelöscht → Entf. der ältesten ungenutzten Page

Clock



Effiziente Implementierung von Second Chance. Linkedlist wird zum Kreis (clock). Keine Verschiebung der Elemente sondern Arbeit mit "Next-Pointer"

Least Recently Used

Ersetzt am längsten unbenutzte Page anhand von Zeitstempel, die MMU bei Zugriff notiert. Grosser HW Aufwand, viele MMUs nicht in der Lage dazu

Not Frequently Used

Benötigt zusätzliche Counter Tabelle. Bei Zugriff Counter erhöhen. Ersetzen der Page mit niedrigstem Counter. Alte Pages können lange bleiben (wenn früher oft gebraucht)

Working Set

Pro Page-Table-Eintrag ein Zeitstempel, der immer mit dem Fault-Interrupt gescannt wird (anstelle Timer-Interrupt).

1. Wenn $A = 1$:
   Setze $t = now$ und $A = 0$
2. Wenn $A = 0$:
   Wenn $now - t < T$ Page im Working Set und bleibt
   Wenn $now - t >= T$ Page nicht im Working Set
   und wird entfernt
3. Wenn keine Page gefunden wird: Entferne älteste Page (bevorzuge saubere Page)

Working Set mit Clock

Speichert Zeitstempel, Iteriert über Clock im Fault-Interrupt
Vermeidet Aufwand alle PTEs durchsuche, Praxistauglich