Spick für Betriebsysteme 1
von Janosch Bühler, Seite 1 von 2

1 Logik

Nulläre Funktion: Funktion mit null Parametern
Bsp.: $f() = 1$ => eine Möglichkeit mit null Argumenten
Unäre Funktion: Funktion mit einem Parameter
Bsp.: $f(x) = -> x$ => Zwei mögliche Argumente: x=0 oder x=1
Binäre Funktion: Funktion mit zwei Parametern
Bsp.: $f(x,y) = x \wedge y$ => 4 mögl. Argumente (0,0),(0,1),(1,0),(1,1)
n-äre Funktion: Funktion mit n Parametern (n-stellig)
Bsp: $f(x_0,...,x_{n-1}) = x_0 \wedge x_1 \wedge ... \wedge x_{n-1} \Rightarrow 2^n$ mögliche Kombinationen

1.1 Disjunktion & Konjunktion

Neutrales Element:
$(x \vee 0) \equiv (x)$ $(x \wedge 1) \equiv (x)$
Komplement:
$(x \vee \neg x) \equiv (1)$ $(x \wedge \neg x) \equiv (0)$
Kommutativität:
$(x \wedge y) \equiv (y \wedge x)$ $(x \vee y) \equiv (y \vee x)$
Assoziativität:
$(x \wedge (y \wedge z)) \equiv ((x \wedge y) \wedge z)$ $(x \vee (y \vee z)) \equiv ((x \vee y) \vee z)$
Distributivität:
$(x \wedge (y \vee z)) \equiv ((x \wedge y) \vee (x \wedge z))$ $(x \vee (y \wedge z)) \equiv ((x \vee y) \wedge (x \vee z))$
Idempotenz:
$(x \wedge x) \equiv x$ $(x \vee x) \equiv x$
Doppelnegation: $\neg(\neg x) \equiv x$
de Morgans Regeln:
$\neg(x \wedge y) \equiv ((\neg x) \vee (\neg y))$ $\neg(x \vee y) \equiv ((\neg x) \wedge (\neg y))$

1.2 Exklusive Disjunktion (XOR)

Abbildung der Addition zweier Bits und die Konjunktion den Übertrag.

1.3 16 Binäre Funktionen

x	y	0	xy	$x\bar{y}$	x	$\bar{x}y$	y	$x \oplus y$	$x \vee y$	$\overline{x \vee y}$	$x \odot y$	$\bar{y}$	$x \vee \bar{y}$	$\bar{x}$	$\bar{x} \vee y$	$\overline{xy}$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

2 nulläre: $0, 1$ 4 unäre: $x, y, \neg x, \neg y$

2 Assembler

Programm, welches textuelle Befehle in Maschinencode übersetzt.
Die Sprache dazu heisst Assembly. Die Konventionen zu Assembly sind jeweils abhängig vom Hersteller.

Befehlssatz: Menge aller Maschinencodes, die ein Prozessor kennt.

Diese Maschinencodes können unterschiedlich lang sein:

- 1 Byte (8 Bit): byte, DB, RESB –> hat 2 Hexadez. Stellen
- 2 Byte: word, DW, RESW
- 4 Byte: dword, DD, RESD
- 8 Byte: qword, DQ, RESQ

2.1 Register

AL 8 Bit Register
AX: 16 Bit Register mit je zwei 8 Bit Registern (AH (oben), AL (unten))
EAX: 32 Bit Register, erweitert AX links mit 16 Bit
RAX: 64 Bit Register, erweitert EAX links mit 32 Bit

2.2 Allzweckregister

RAX: Accumulator, für einige Rechenoperationen das einzige Register
RCX: Counter für Schleifen und Stringoperationen
RDX: Pointer für I/O-Operationen
RBX: Datenpointer
RSI / RDI: Quell- und Zielindizes für Stringoperationen
RSP: Stackpointer, Adresse des allozierten Stacks
RBP: Basepointer, Adresse innerhalb des Stacks, Basis des Rahmens der Funktion
R8 – R15: Zusätzliche Register

2.3 Endians

Little-Endian: BE | BA | FE | CA –> innerhalb des Bytes bleibt es gleich, aber das LSB (Least Significant Bit) ist am «Ende»
Big-Endian: CA | FE | BA | BE –> unsere "normale" Art. MSB (Most Significant Bit) steht am «Ende».

2.4 Labels

Am Anfang jeder Zeile kann ein sogenanntes Label stehen, welches nicht in Bytecode übersetzt und ausgegeben wird. Intern assoziiert der Assembler Speicher für das Label.

2.5 Syntax

_myfunction: Definiert _myfunction als Pointer auf die Stelle im generierten Byte-Stream
mov x, y: Kopiert den Wert vom Register rbx in rax
Bsp: mov cl, [rax] –> es werden 8 Bit bewegt.
mov x, [y]: Kopiert das Byte nach x, das an der Hauptspeicheradresse liegt, die in y steht.
inc rax: Erhöht Wert in rax um 1
dec rax: Verringert Wert in rax um 1
cmp x, y: Vergleicht x mit y und setzt Z(ero)-Flag, wenn beide gleich sind.
sub x, y: y wird von x abgezogen und Differenz in x geschrieben.
jnz _myfunction: Springt zu _myfunction wenn Z-Flag nicht gesetzt.
ret: Return, Rücksprung zum Aufrufer und vorher Rücksprungadresse vom Stack holen und in Befehlszähler schreiben.

2.6 Calling Convention

Calling Convention sind Vereinbarungen zwischen dem Caller und der aufgerufener Funktion(Callee): Wo Argumente, Wo Rückgabewerte, welche Register bearbeitet, etc.

2.7 Lokal & globale Variablen

Lokale Variable: Liegt auf dem Stack
Globale Variable: Fixe Adresse im Speicher(Label)

3 Grössen

2^{10}	$1.024 \cdot 10^3$	K	Kilo	Ki	Kibi
2^{20}	$1.049 \cdot 10^6$	M	Mega	Mi	Mebi
2^{30}	$1.074 \cdot 10^9$	G	Giga	Gi	Gibi
2^{40}	$1.100 \cdot 10^{12}$	T	Tera	Ti	Tebi
2^{50}	$1.126 \cdot 10^{15}$	P	Peta	Pi	Pebi
2^{60}	$1.153 \cdot 10^{18}$	E	Exa	Ei	Exbi

2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	1'0000
2^5	32	10'0000
2^6	64	100'0000
2^7	128	1000'0000
2^8	256	1'0000'0000
2^9	512	10'0000'0000

Bsp benötigte Bits:
$0 - 32T - 1 \rightarrow 2^5 * 2^{40} = 45bit$
$0 - 32T \rightarrow 46bit$
$0 - 1012_d \rightarrow 40bit$ (weil $2^{39} = 0.550 * 10^{12}$ nicht reicht)

4 Binär

4.1 Begriffe

Bit: Stelle einer Binärzahl
Bit = 1: Set Bit
Bit = 0: Gelöschtes Bit (cleared bit)
LSB: Least Significant Bit, niederwertigstes Bit, Bit 0
MSB: Most Significant Bit, höchstwertiges Bit, Bit n-1
Nibble: Binärzahl mit vier Bit.
Byte/Oktett: Binärzahl mit acht Bit.
Carry Bit: Übertragsbit, wird bei einem Übertrag gesetzt.

4.2 Formeln

Grösste Darstellbare Zahl(unsigned): 2n-1
Anzahl Zahlen: 2n
Bereich(unsigned): 0 bis 2n-1
Grösste Darstellbare Zahl(signed): 2n-1-1
Kleinste Negative Zahl (signed): -2n-1
Bereich(signed): -2n-1 bis 2n-1-1
MSB = 0: Dient als positives Vorzeichen
MSB = 1: Dient als negatives Vorzeichen

4.3 Operationen

Muliplikation
Wie schriftliche Multiplikation im Dezimalsystem.
2er Potenz über Binar zu Hex
$2_d^{12} \rightarrow 0001\,0000\,0000\,0000 \rightarrow 1000_h$
Invertieren
0001 0000 0000 0000 → 1110 1111 1111 1111
Zweierkomplement
Durch MSB als Vorzeichen können Binärzahlen negativ dargestellt werden.
Zweierkomplement Binär:Invertieren und +1
0001 0000 0000 0000 → 1110 1111 1111 1111 → 1111 0000 0000 0000
Zweierkomplement in Dezimal: MSB abziehen den Rest addieren
$1101_b \rightarrow -8 * 1 + 4 * 1 + 2 * 0 + 1 * 1 = -3$
Links- & Rechtsshift
Rechtsshift: Wenn negativ dann 1 nachschieben, sonst 0
Bsp negativ: 1010 → 1101
Bsp positiv: 0110 → 0011
Linksshift: Es wird immer eine 0 von rechts nachgeschoben.
Bsp: 0110 → 1100

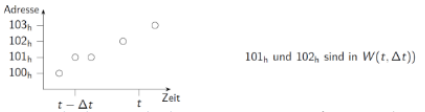
5 Hexadezimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

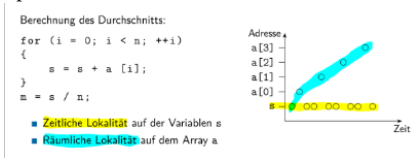
Bsp: $2A_h -> 42_d$, $D0_h -> 208_d$

6 Lokalitätsprinzip

Arbeitsbereich eines Programmes: Die Speicherstellen welche in einem Zeitintervall $(t - \Delta t, t()$ referenziert werden.



Räumliche Lokalität: Wenn auf eine bestimmte Adresse im Hauptspeicher zugegriffen wird ist die Wahrscheinlichkeit hoch, dass die nachfolgenden Zugriffe auf eine Adresse in der Nachbarschaft erfolgt. Im Speicher wird dies genutzt in dem man immer Datenblöcke verschiebt.
Zeitliche Lokalität: Wird auf eine bestimmte Adresse im Hauptspeicher zugegriffen, so ist die Wahrscheinlichkeit hoch, dass in naher Zukunft wieder darauf zugegriffen wird. Im Speichersystem will man also die zuletzt zugegriffenen Daten auf der schnellsten Stufe der Speicherhierarchie halten.
–> Wenn man den Arbeitsbereich kennt kann man auf den Zukünftigen schliessen.
–> Hätte man dieses Prinzip nicht, dann würde man immer nur die langsamste Speicherstufe verwenden.



7 Speicherallokation

First-Fit: Erste passende Lücke am Anfang
Next Fit: Erste passende Lücke nach zuletzt reserviertem Bereich
Best-Fit: Durchsucht alle Lücken, wählt kleinste passende aus
Worst-Fit: Durchsucht alle Lücken, nimmt grösste Lücke

8 C

8.1 Pointer

Ein Pointer ist eine Variable, dessen Wert (im Normalfall) die Adresse einer anderen Variablen ist. Adresse, welche auf ein Objekt zeigt.
Definition: void* oder int* oder auch int**

```
#include <stdio.h>
int main () {
    int  var = 20;   /* actual variable declaration */
    int  *ip;        /* pointer variable declaration */

    ip = &var;  /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var  );

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );

    return 0;
}
```

8.2 Conditionals

Ausdrücke, welche interpretiert werden, dass 0 falsch ist und jeder andere Wert wahr.

8.3 Referenz- und Dereferenzoperator

Der Operator «&» erzeugt die Adresse eines Ausdrucks.
Pointer (*) werden jeweils Adressen (&) zugewiesen, sprich der eigentliche Wert von T* ist &a.
«*» und «&» heben sich gegenseitig auf» –> *&a = a = &*a
Achtung bei &*a muss a Pointer sein, weil die Adresse verlangt wird.

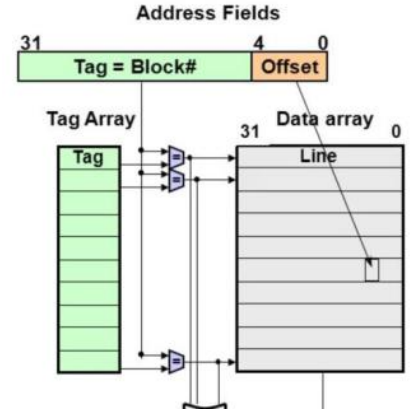
9 Cache

Der Cache ist ein kleiner aber sehr schneller (Zwischen)Speicher. Er ist viel schneller als der Hauptspeicher. Es können Daten und Tags gespeichert werden.
Cache Hit: Gesuchte Adresse im Cache vorhanden
Cache Miss: Gesuchte Adresse nicht im Cache
Berechnung der mittleren Zugriffszeit: T_C = Zugriffszeit auf Cache, T_M = Zugriffszeit auf Hauptspeicher, p_C = Chance auf Cache Hit
$E(T) = p_C * T_C + (1 - p_C) * T_M$
l Länge einer Cachezeile
s Grösse des Hauptspeichers
w_n Anzahl Wege des Caches auf Stufe n (Anzahl Cacheeinträge)
s_n Grösse des Caches auf Stufe n
s_n' Grösse eines Wegs des Caches auf Stufe n
z_n Anzahl Zeilen des Caches auf Stufe n
z_n' Anzahl Zeilen pro Weg des Caches auf Stufe n
t_n Anzahl Bits pro Tag im Cache auf Stufe n
T_n Anzahl Bits für alle Tags im Cache auf Stufe n (Overhead)
Offset: Bei n-Byte Zeilenlänge –> $2^x = n$ –> x Bit Offset bzw. l in 2er-Potenz und Potenz = Anzahl Byte Offset
$s_n' = l * z_n$, $z_n = s_n / l$
$z_n' = z_n / w_n$,
in FAC $z_n' = z_n$
$t_n = l$ – Bits(2er-Potenz) von z_n' – Offset,
in FAC $t_n = l$ – Offset

$T_n = t_n * z_n$
Speicherstelle des Hauptspeichers auf gleichen Cacheeintrag: $s/s_n{}' = s/(s_n/w_n) = s * w_n/s_n$

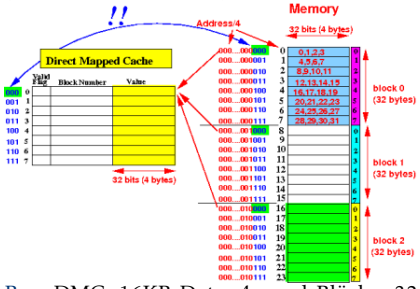
9.1 Fully Associative Cache(FAC)

Adressen aufgeteilt in Tags und Offset. Zuerst sucht man die Cachezeile mit dem richtigen Tag, danach sucht man in diesem das richtige Offset. Beste Cache-Leistung, aber teuer da aufwendige HW. Besitzt viele Vergleichsbausteine.

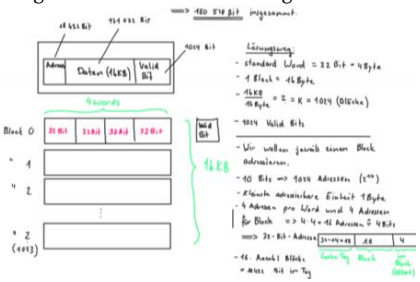


9.2 Direct Mapped Cache

Aus dem Main Memory kommen mehrere Einträge in eine Cachezeile. Es ist fixiert in welche Zeile ein Eintrag im Cache hingehört. (blau: Cache Index Bits)

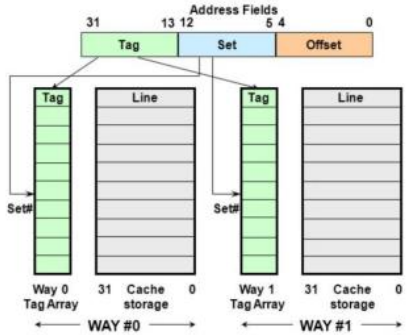


Bsp. DMC, 16KB Data, 4-word Blöcke, 32 Bit-Adressen.
Frage: nach Anz. Bits für den ganzen Cache



9.3 n-Way Set associative Cache

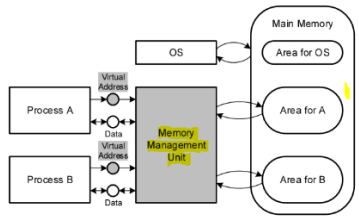
Ist ein Kompromiss zwischen Fully Associative Cache und Direct Mapped Cache. Er ist weniger komplex als FAC, hat weniger Kollisionen als DMC ist aber genauso schnell wie DMC. Es gibt für jede Cachezeile n Möglichkeiten(pro Way eine), da n DMCs parallel verwendet werden. Ein Way ist genau eine Cachezeile gross.



10 Virtueller Speicher

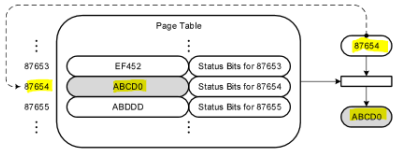
Prozesse bekommen vom OS Speicher zugeteilt und das OS schaut, dass sich die Prozesse nicht gegenseitig stören.
Lösung: Die Prozesse kennen nur virtuelle Adressen. Das MMU übersetzt virtuelle Adresse in physische Adresse. Das OS konfiguriert einen MMU pro Prozess.
MMU: Memory Management Unit übersetzt virtuelle in physische Adresse.
Page: Virtueller Adressraum besteht aus Pages, eine Page hat jedoch keinen physischen Speicher. Sie benötigt einen Speicherort (Hauptspeicher/Sekundärspeicher).
Frames: Hauptspeicher wird in Frames aufgeteilt, in ein Frame passt genau eine Page. Ein Frame = eine Page.
Virtueller Adressraum/Pagetable: Pro Prozess ein virtueller Adressraum.
- Pro Prozess gibt es eine Pagetable (Mapping Tabelle)
- OS verwaltet, welche Pages wann wo liegen müssen. MMU kennt nur den Hauptspeicher dh. MMU kann nur sagen ob Page, resp. zu Page gehörendes Frame im Hauptspeicher ist.

Status-Bit Used (P-Bit): Zeigt, ob Page used =1 oder unused = 0 ist.
Interprozesskommunikation (IPC): **Shared Memory**: Prozesse teilen sich den Speicher -> kein Schutz. **Message Passing**: OS kopiert Daten, sicher aber auch Overhead.
Paging: Verwaltung von pageorientiertem Speicher (Laden von Pages etc.)



10.1 Single-Level Page Table

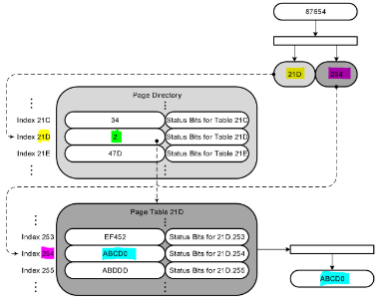
Für jede mögliche Page einen Eintrag. Die Grösse hängt vom virtuellen Adressraum ab. Lookup sehr schnell aber kann sehr schnell sehr gross werden.



10.2 Two-Level Page Table

Page Number wird in Directory Index & Page Table Index aufgeteilt
1. Page Table in Directory finden
2. Table Index in Page Table finden

Viele Page Tables, Page Directory zeigt auf Page Tables



10.3 Speicherfreigabe

Explizit: Programmierer bestimmt, wann Speicher freigegeben wird. Nur im OS ist explizite Speicherverwaltung möglich. →Mögliche Speicherlacks, falls nicht verwendeter Speicher nicht mehr freigegeben wird. Implizit: Speicher wird automatisch freigegeben, wenn er nicht benötigt wird. Dies geschieht in der App (JVM, Python, ...)

10.4 Befehle

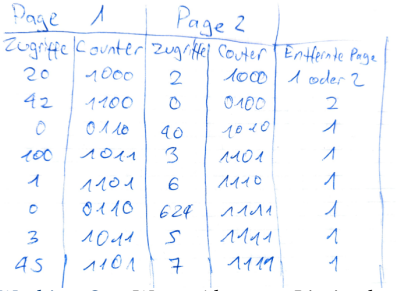
Malloc(s): Alloziiert Speicherblock mit Grösse S.
Free(*p): Gibt einen Speicherblock frei, der an Adresse p beginnt.
Malloc und free gehören wie Klammerpaare zusammen, damit es keine Speicherlacks gibt. Interne Fragmentierung: Heap reserviert einen grösseren Speicherblock, als angefragt wurde. Der überschüssige Speicher wird nicht verwendet.
Externe Fragmentierung: Das Programm reserviert immer wieder Speicher und gibt ihn unregelmässig wieder frei. Über Längere Zeit entstehen kleine Löcher, die aber, trotz in der Summe genügend gross wären, nicht in der Lage sind grösseren Speicher zu reservieren.

10.5 Suchalgorithmen

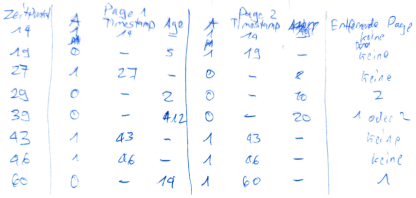
First fit: Erste passende Lücke
Next fit: Erste passende Lücke nach zuletzt verwendetem Bereich
Best fit: Durchsucht alle Lücken nach der passendsten.
Worst fit: Durchsucht alle Lücken nach der Grössten
Quick fit: Erstes Element mit kleinster passenden Grösse
Nachteil: Nachbarn schwer zu finden.

10.6 Paging

Dirty Bit: Page im Hauptspeicher ist anders als im sekundären Speicher.
Accessed Bit: Page wurde kürzlich von Prozess verwendet → wird von MMU gesetzt.
Page Fault: Wird gesetzt, wenn die referenzierte Page nicht im Hauptspeicher ist.
Bsp



Working Set: Wenn Alter >= Limit, dann wird Page entfernt, sonst bleibt sie.
Bsp. Limit des Alters zusätzlich 10

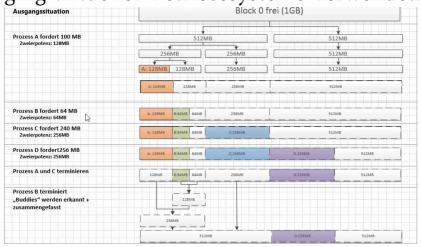


11 Verdrängungsstrategie

FIFO: Entferne jeweils älteste Page. Problem: alte, aber häufig benutzte Pages werden gleich wieder geladen.
Second Chance (Extension FIFO): Pages erhalten ein A(ccess)-Bit und es wird jeweils die älteste, nicht verwendete Page entfernt. Verliert eine Page ein "Leben"rutscht sie ganz an den Anfang. Bekommt eine Page ein "Leben"bleibt es an der Stelle (links alt, rechts neu).
Optimal: Ersetze die Page, die in Zukunft am spätesten verwendet wird (nicht umsetzbar, da man nicht weiss welche diese sind)
LRU (Least recently used): Ersetzt, die am längsten unbenutzte Page.Bei jedem Zugriff wird ein Timestamp gesetzt, Page mit kleinstem T wird ersetzt. (Nahe am Optimum)
NFU (not frequently used): Benötigt Counter Table pro Page Table eine n-Counter am selben Index. OS zählt Intervalle, in denen es Zugriffe gab.
Problem: Auch alte Pages können lange bestehen bleiben, wenn sie zu Beginn sehr oft verwendet wurden.
NFU mit Aging: Die Counter werden neu auch nach Zeit gewichtet.

11.1 Buddy-System

Das Buddy-System teilt Prozessen Speicher zu. Der Speicher wird in 2 k Bereiche aufgeteilt und zu Beginn gibt es einen Block, der möglichst den gesamten Speicher abdeckt. Falls Speicher nicht als Zweierpotenz ausgedrückt werden kann, dann kann er auch als mehrere Blöcke unterschiedlicher Grössen unterteilt werden. Wenn es keinen Block in der Grösse gibt, dann wird die nächstgrössere 2-er-Potenz genommen. Terminieren Prozesse, dann erkennt das System die Buddies wieder und fügt die Partitionen zusammen. Wird fürs Paging aktueller Betriebssysteme verwendet.



11.2 Berechnungen

Bei 16MB virtueller Adressraum, 4KB Pages, 64KB Hauptspeicher, Single Level:
Anzahl Bits für virtuelle Adresse $2^{24} \rightarrow 24Bit$
Grösste virtuelle Adresse $FFFFFF_h = 2^{24} - 1$
Kleinste virtuelle Adresse: $000000h$
Anzahl Bits für reale Adresse $2^{16} \rightarrow 16Bit$
Grösste physische Adresse: $FFFF_h = 2^{16} - 1$
Kleinste physische Adresse: 0000_h
Anzahl Pages: Virtueller Adressraum, Grösse einer Page. Hier $=2^{12}$ Pages
Anzahl Frames: Hauptspeicher, Grösse gleich wie Pagegrösse. Hier = $16Page$ Frames
Offset: Bei $4KB$ Page $= 2^{12} = 12Bit$ Offset
Grösse Framenummer: Ist der Teil der realen Adresse ohne den Offset. Hier $16Bit$ ohne Offset ($12Bit$) = $4Bit$ für die Framenummer. Oder: Page Frames als 2-er-Potenz und der Exponent als Bit. Grösse einer Pagetable (ohne Statusbits): Anzahl Pages * Framenummer.

Bestimmen einer Adresse: $3AB4_h \rightarrow$ Pagenummer = $3 \rightarrow$ Offset = $AB4 \rightarrow$ reale Adresse hier $5AB4_h$ In diesem Bsp. liegen Pages 2 bis 5 in den Frames 4 bis 7. Bsp. Auf Frame 5 werden Adressen von 3000_h bis $3FFF_h$ abgebildet.