5.5 Virtueller Speicher

Wenn der reale Speicher sogar für einzelne Prozesse zu klein ist :

Virtueller Speicher *(virtual memory),*

- ist „beliebig" groß, nimmt alle Prozesse auf,
- ist in gleichgroße Teile – „Seiten" – aufgeteilt,
- die zwischen Arbeitsspeicher und Auslagerungsbereich umgelagert werden.

Rechtfertigung: *nicht alle Teile* eines Programms werden immer gleichzeitig benötigt.

5.5.1 Das Seitenverfahren

(paging)

Seite *(page)* =
Einheit der Speicherzuweisung und Umlagerung;
einheitliche Größe *s*, typischerweise zwischen 4 und 16 KB.

Genauer:

- *(virtuelle) Seite* eines Prozesses =
  virtueller Adressbereich *[k*s .. (k+1)*s – 1] (k=0,1,..)*
- *(reale)* Seite = einer der Teile des virtuellen Speichers

Rahmen *(frame)* =

Adressbereich *[k*s .. (k+1)*s – 1] (k=0,1,..)*
im Arbeitsspeicher, kann eine Seite aufnehmen;

entsprechend im Auslagerungsbereich.

Beachte:

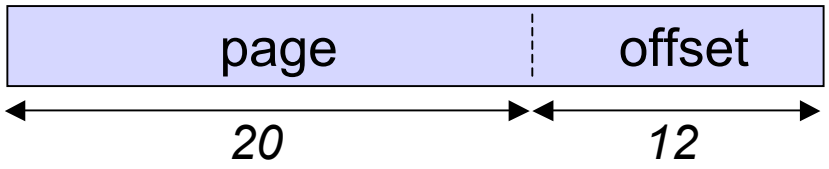
1. Durch einheitliche Seitengröße wesentlich vereinfachte Speicherverwaltung gegenüber der Segmentierung.
2. Zwar keine externe, aber **interne Fragmentierung**.

Adressierung z.B. bei 32-Bit-Adressen

(d.h. 4-GB-Adressraum)

und Seitengröße $s = 4096$:

virtuelle Adresse:



(vgl. 5.4.1←) page = *(virtuelle)* Seitennummer

offset = Distanzadresse

→ je Prozess maximal 2^{20} Seiten à 4096 Bytes

Adressumsetzung: MMU enthält ein **Kontextregister**, über dessen Inhalt die **Seitendeskriptoren** erreicht werden.

5.5.1.1 Deskriptorverwaltung

Seitendeskriptor *(page descriptor)*

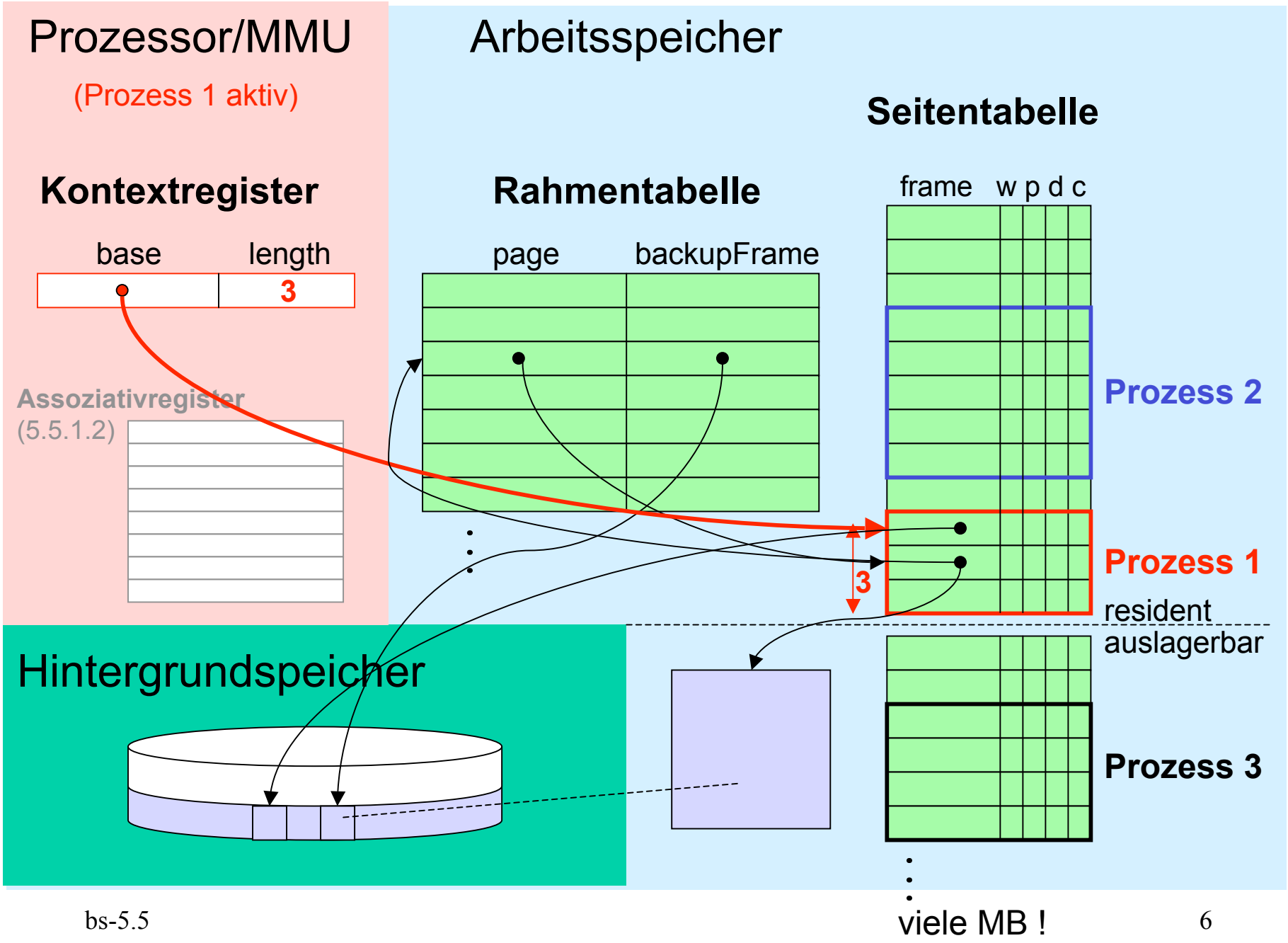
enthält Rahmennummer im Arbeitsspeicher - wenn eingelagert -
bzw. im Auslagerungsbereich - wenn ausgelagert -
ferner *d-Bit, c-Bit, p-Bit („present")*

Seitentabelle *(page table)*

enthält die Deskriptoren für alle Seiten des virtuellen
Speichers (0, wo nicht benutzt)

Rahmentabelle *(frame table)*

enthält für jeden Rahmen die (reale) Nummer der
dort eingelagerten Seite (0, wenn nicht benutzt)
sowie deren Rahmennummer im Auslagerungsbereich


Prozessor/MMU
(Prozess 1 aktiv)
Kontextregister
base
length
3
Assoziativregister
(5.5.1.2)
Arbeitsspeicher
Seitentabelle
Rahmentabelle
page
backupFrame
frame
w p d c
Prozess 2
Prozess 1
3
resident
auslagerbar
Prozess 3
viele MB !
Hintergrundspeicher

Achtung 1:

Wo sind die Deskriptorregister? (→5.5.1.2)

Achtung 2:

Gemeinsame Benutzung von Speicherbereichen durch verschiedene Prozesse ist hier zunächst *nicht möglich*. (*Mehrere* Deskriptoren auf *einen* Rahmen verweisen lassen? Problematisch ...) (→5.6)

Achtung 3:

Die Seitentabelle ist *riesig!* Mit den obigen Daten enthält sie bis zu 2^{20} Deskriptoren für *jeden Prozess*. Die Tabelle speicherresident zu halten ist unrealistisch. (→5.5.1.3)

5.5.1.2 Adressumsetzung mit Assoziativspeicher

Kontextregister *(context register)* in der MMU

übernimmt die Rolle von *Basis/Längenregister* –
jetzt nicht für einen Bereich im Arbeitsspeicher,
sondern für einen Bereich in der Seitentabelle –
und damit des virtuellen Speichers.

Aber: Für eine schnelle Adressumsetzung müssten
alle Seitendeskriptoren des laufenden Prozesses
in MMU-Registern sein ➜ Tausende von Registern ?

MMU enthält einen

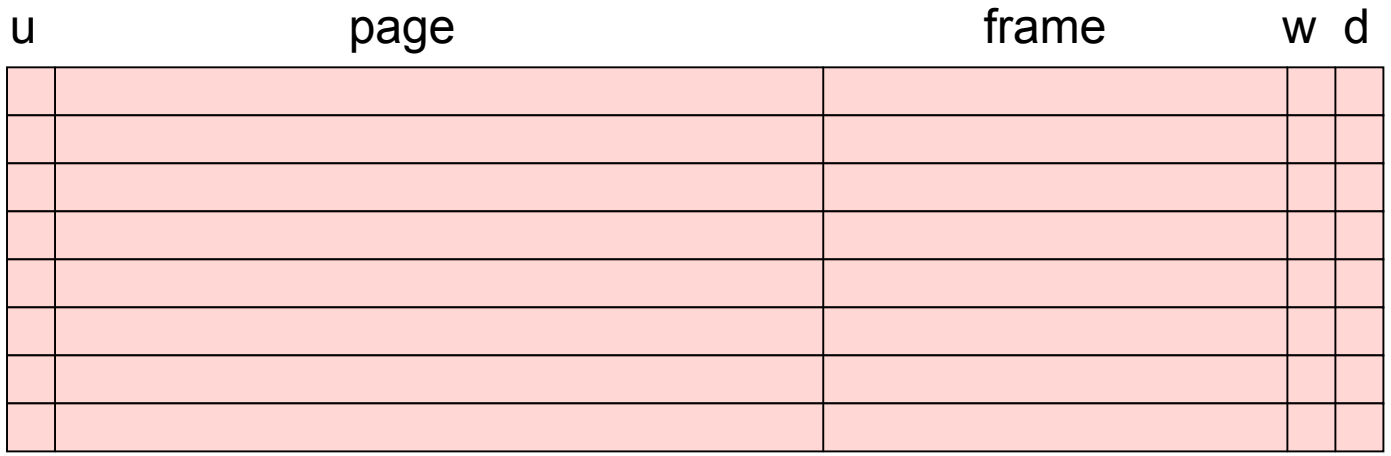
Assoziativspeicher *(associative memory, descriptor cache, TLB – translation look-aside buffer),* bestehend aus

Assoziativregistern *(associative registers),*

welche nur die *aktuell benötigten* Deskriptoren enthalten (typische Anzahl: 8 – 64);

diese werden von der MMU automatisch nachgeladen,
wenn auf eine Seite „erstmalig" zugegriffen wird.

(„assoziativ", weil nicht über Registernummer identifiziert,
sondern über die enthaltene Nummer der virtuellen Seite !)

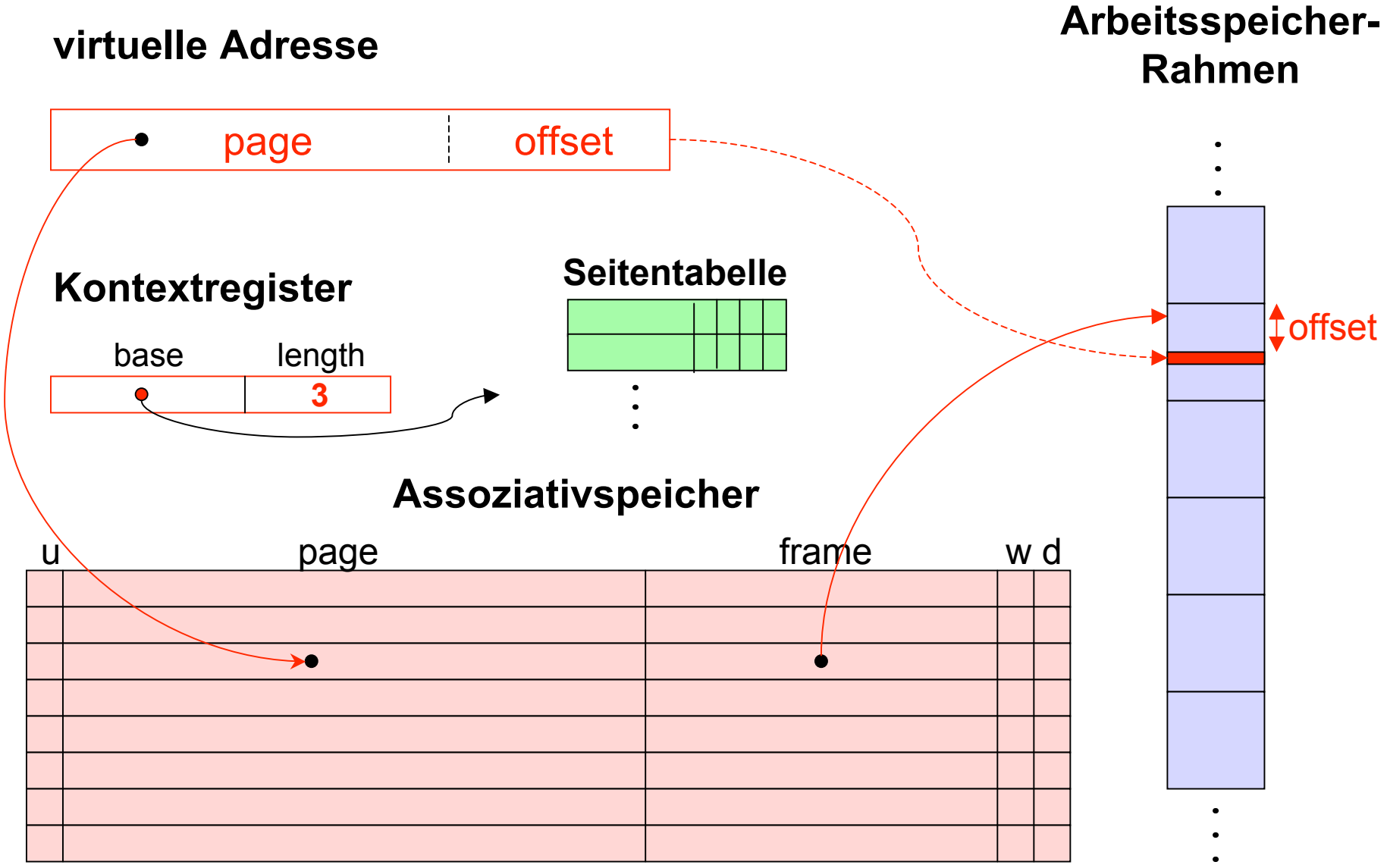


d = dirty bit

w = writable bit – für Zugriffsschutz (s.u.)

u = used bit: wird bei Verdrängung eines Deskriptors durch einen neu benötigten Deskriptor in allen Registern gelöscht und bei der Benutzung eines Deskriptors in dessen Register gesetzt.

Beim nächsten Verdrängen wird dort verdrängt,
wo *u* nicht gesetzt ist (zuvor *d* in Seitentabelle retten!).
(Rechtfertigung: Extrapolation von *"not recently used"*)


virtuelle Adresse
page
offset
Arbeitsspeicher-
Rahmen
Kontextregister
base
length
3
Seitentabelle
offset
Assoziativspeicher
u
page
frame
w d

```
hardDesc = A<page> ;

if hardDesc == 0 then                (descriptor not loaded, access page table)
    if page >= length then address fault end;          (beyond last page)
    softDesc = pageTable[base+page];
    if softDesc.frame == 0 then address fault end;     (page non-existent)
    if softDesc.p == 0 then page fault end;            (page non-resident)
    x = findPreemptable();           (where u-bit is not set)
    pageTable[base+x] = pageTable[base+x] or A<x>.d;   (save dirty bit)
    A<x> = (1, page, softDesc.frame, softDesc.w, 0);   (+ clear other u-bits)
    hardDesc = A<page> end;

if write and not hardDesc.w then access fault end;     (no write access)
location = hardDesc.frame * pageSize + offset;
A<page>.d = A<page>.d or write ;  (set dirty bit)
A<page>.u = 1 .
```

Beachte: Der Assoziativspeicher enthält stets eine Teilmenge der Deskriptoren der *eingelagerten* Seiten des aktiven Prozesses.

Prozessumschaltung:

- *dirty bits* aus dem Assoziativspeicher retten
- Löschen des Assoziativspeichers
- Umsetzen des Kontextregisters

5.5.1.3 Auslagerung von Seitentabellen

Die Seitentabelle ist *riesig!*

Mit den obigen Daten enthält sie bis zu 2^{20} *Deskriptoren* für *jeden Prozess* - und das gilt bereits für 32-Bit-Adressen!

Die Tabelle speicherresident zu halten ist unrealistisch.

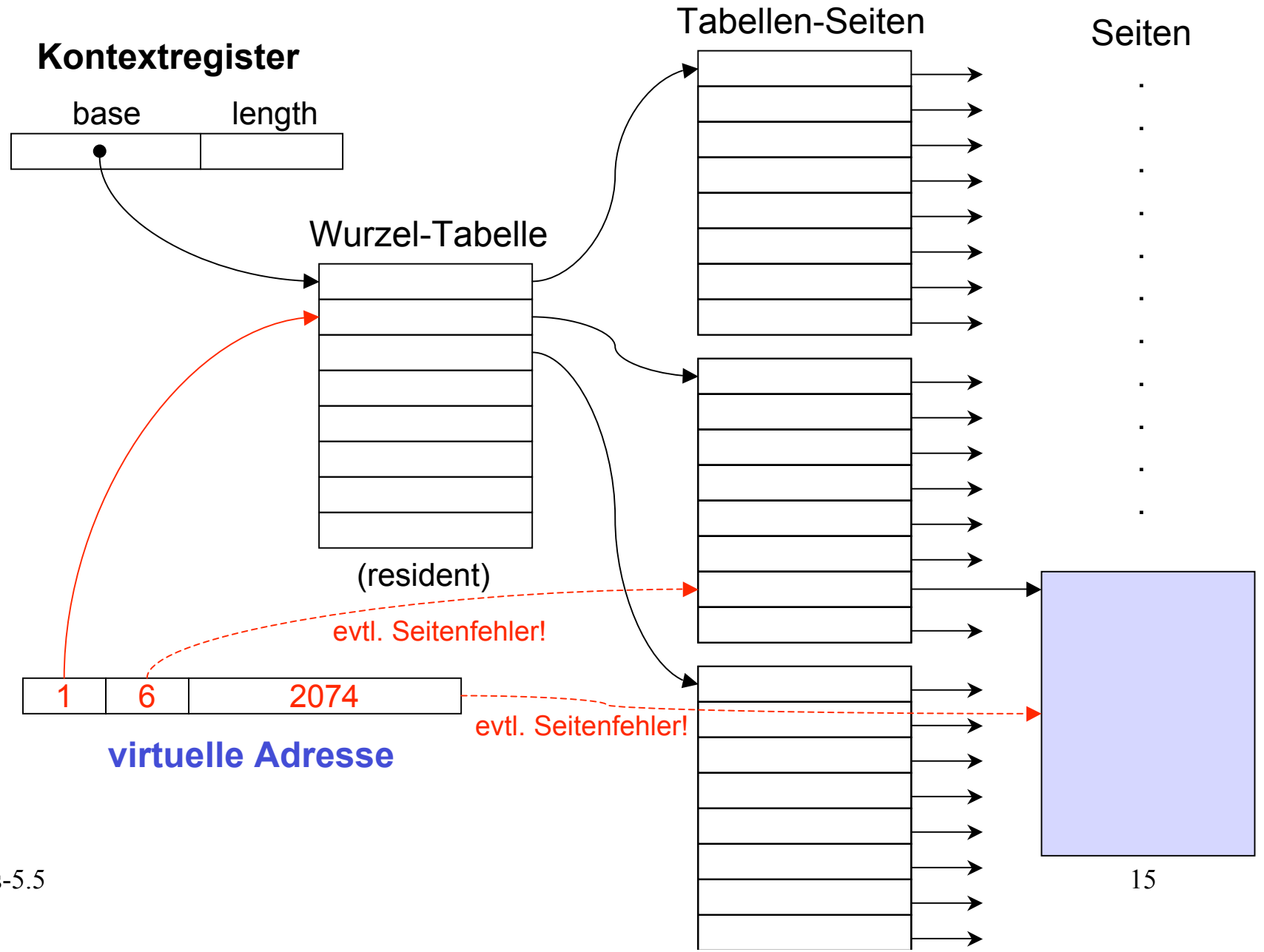
Alternativen:

❶ Hierarchisch gestufte Tabellen

❷ Invertierte Tabellen

❸ Prozess ein/auslagern bedeutet dessen Deskriptoren ein/auslagern (→5.5.4)

❶ Hierarchisch gestufte Tabellen



❷ Invertierte Tabellen (zusätzlich zu ❶):

Deskriptoren der eingelagerten Seiten werden in einer Hash-Tabelle (vom Umfang der Rahmentabelle) gehalten und dort über den Schlüssel *(procno, pageno)* gefunden.

Falls Deskriptor nicht im TLB, in der Hash-Tabelle suchen.

Falls Deskriptor nicht in der Hash-Tabelle, *Seitenfehler*.

(All dies macht die MMU-*Hardware!)*

→ Betriebssystem übernimmt wie bei ❶

5.5.2 Das Lokalitätsprinzip

? **Strategie** für die Umlagerung von Seiten ?

Ausgangspunkt:

nur diejenigen Seiten eines Prozesses einlagern,
die der Prozess tatsächlich benötigt !

Beobachtungen:

1. Prozess greift *nicht permanent auf alle* Bereiche seines Adressraums zu.

2. Es genügt, wenn die *aktuell benötigten Teile* eines Prozesses sich im Arbeitsspeicher befinden.

3. Dies erlaubt auch die Bearbeitung von Programmen, die insgesamt *größer als der Arbeitspeicher* sind.

Programme zeigen „Lokalitätsverhalten“:

sie greifen während längerer Zeiträume („**Phasen**“)

jeweils nur auf eine *bestimmte Teilmenge ihrer Seiten* zu.

aktuelle **Lokalität** *(locality)*

Konsequenz: nur diejenigen Seiten umlagern,
die zur Lokalität gehören,
den Rest ausgelagert lassen !

5.5.2.1 Request Paging

bedeutet, dass der Prozess dem Betriebssystem über Systemaufrufe mitteilt, welche Seiten jeweils zu seiner Lokalität gehören:

`include``(addr)`

erweitert die Lokalität um die Seite, die `addr` enthält
(→ Einlagerung *„on request"*, auch *prepaging)*

`exclude``(addr)`

verringert die Lokalität um die Seite, die `addr` enthält

Rechtfertigung: Das Programm weiß am besten über sein eigenes Verhalten Bescheid !

→ Modifizierte Prozess-Umlagerung *(locality swapping)*:

- anstelle des gesamten Prozesses wird jeweils seine Lokalität umgelagert;
- aus der Lokalität entfernte Seiten werden zwar ausgelagert, aber nicht wieder eingelagert;
- Erweiterung der Lokalität führt zu vorzeitiger Auslagerung.

Vorteil: effizient, *keine Seitenfehler!*

Nachteil: schwer praktikabel: erfordert Kenntnis sowohl des Verhaltens als auch des Layouts des Programms.
(Wo sind die Seitengrenzen?)

5.5.2.2 Demand Paging

geht davon aus, dass man über die Lokalität nichts weiß:

Seitenfehler *(page fault)* steuern das Paging:

Hardware: Unterbrechung (Alarm)

page fault handler der Speicherverwaltung:

- veranlasst das Einlagern der Seite („*on demand*"),
- setzt nach erfolgtem Einlagern den Prozess mit dem unterbrochenen Befehl fort.

! Wiederaufsetzen des Prozesses nach Seitenfehler ist eventuell *problematisch:*

unterbrochener Befehl kann *partiell* ausgeführt sein

(z.b. bei Autoinkrement-Adressierung oder komplexen Befehlen)

Lösungsmöglichkeiten:

❶ *Keine.* (Notorisches Beispiel: Motorola 68000 war für *Demand Paging* ungeeignet.)

❷ *Hilfsregister* der MMU, aus denen der *page fault handler* die benötigten Informationen entnehmen kann.

❸ *Prozessor* annulliert automatisch die partiellen Effekte.

Fragen zum *Demand Paging:*

? Wenn für das Einlagern keine freien Rahmen verfügbar sind, welche Seite soll dann *verdrängt* werden ?

? Wie kann das *locality swapping* approximiert werden ?

? Oder sind bessere Lösungen möglich ?

5.5.3 Seitentausch-Strategien

(page replacement policies/algorithms)

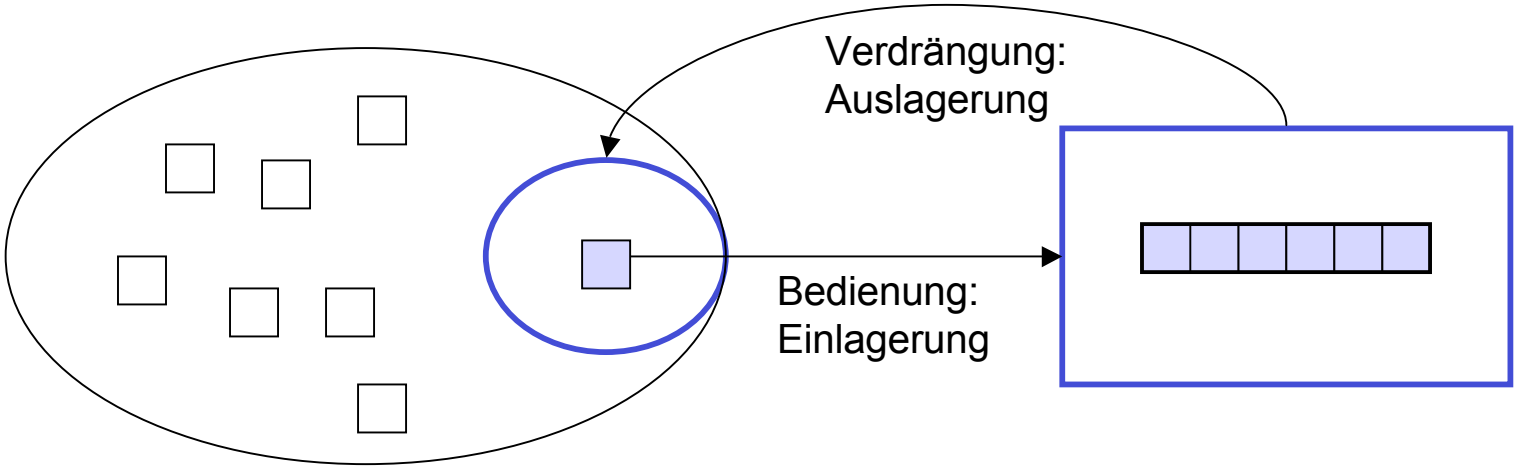
beantworten die Frage, welche Seite *verdrängt* werden sollte, wenn freier Rahmen für einzulagernde Seite benötigt wird.

Optimale Strategie *(B°)*, nicht praktikabel, nur für Vergleichszwecke:

diejenige Seite wird verdrängt, deren nächste Benutzung am weitesten in der Zukunft liegt.

5.5.3.1 ... ohne Berücksichtigung des Programmverhaltens

Zur Erinnerung - Ablaufsteuerung (3.3←), übertragen auf Seiten:



"Warteschlange": einzulagernde *Seite*

"Bedienstationen": *Rahmen*

Beachte: im Gegensatz zum Warteschlangenmodell geht es hier nicht darum, wer als nächster *bedient* wird, sondern wer als nächster *verdrängt* wird, und zwar möglichst schon *vor dem Auftreten des nächsten Seitenfehlers*.

Fairness bedeutet hier: keine der eingelagerten Seiten wird bezüglich ihrer *Einlagerungsdauer* deutlich bevorzugt.

RANDOM Per *Zufallsauswahl* wird ein Rahmen bestimmt, dessen Seite verdrängt wird.
[Einfach ! Schwach fair.]

RR *(round robin)* Reihum wird aus dem jeweils *nächsten* Rahmen eine Seite verdrängt.
[Einfach ! Fair.]

FIFO *Am längsten eingelagerte* Seite wird verdrängt.
[Rahmen-Schlange führen! Sehr fair.]

! Bei allen diesen Verfahren kann ausgerechnet eine solche Seite ausgelagert werden, die gerade intensiv benutzt wird !

5.5.3.2 ... mit Berücksichtigung des Programmverhaltens

LRU *(least recently used,* „am wenigsten jüngst" benutzt)
Verdrängt wird diejenige Seite,
deren *Benutzung am längsten zurückliegt.*
Begründung: Extrapolation des vergangenen Programm-verhaltens, „wahrscheinlich nicht mehr in Lokalität".
„**Abgestandene Seite**" *(stale page)*

[Unrealistisch ohne extrem aufwendige Hardware, die automatisch eine entsprechende Rahmen-Schlange führt.]

NRU *(not recently used,* „jüngst nicht benutzt")

Heuristische Approximation von *LRU:*

Hardware bietet neben *dirty bit* ***d***
zusätzlich *referenced bit* ***r***
(auch *„accessed bit"*)

Periodisch werden alle r-Bits gelöscht.

Auswahl zur Verdrängung gemäß dieser Rangfolge:

(Vergleiche u-Bit
in den Assoziativregistern, S. 10 !)

r	d
0	0
0	1
1	0
1	1

Kleine Übungsfrage: wie kann ein fehlendes r-Bit vom Betriebssystem „simuliert" werden ?

clock oder *second chance*

approximiert ebenfalls *LRU*.

Die Seiten werden in FIFO-Reihenfolge inspiziert:

r = 0 : Opfer gefunden

r = 1: lösche r (d.h. wenn die Seite aktiv war, erhält sie eine *weitere Chance*)

Einfach und leidlich gut; weit verbreitet.

5.5.4 Vermeidung von Seitenflattern

Seitenaustausch –

global: *alle* eingelagerten Seiten sind potentielle Opfer, d.h. Prozess kann anderem Prozess „einen Rahmen stehlen"

prozesslokal: Prozess verdrängt eine *eigene* Seite zugunsten einer anderen eigenen Seite, wenn die dem Prozess zugestandene Lokalitätsgröße überschritten wird.

Globaler Seitenaustausch:

alle Prozesse sind i.d.R. mit eingelagerten Seiten im Arbeitsspeicher vertreten;

➜ wenn die Gesamtheit der Lokalitäten aller Prozesse den Arbeitsspeicher sprengt, dann droht

Seitenflattern *(thrashing):*

Prozesse stehlen sich gegenseitig die Rahmen:

- hektisches Ein/Auslagern
- häufig warten *alle* Prozesse auf fehlende Seiten!

➜ Globaler Seitenaustausch ist *keine gute Idee*
(und widerspricht im übrigen dem Umlagern ganzer Prozesse)

Prozesslokaler Seitenaustausch:

Seitenflattern ist auch hier nicht ausgeschlossen
- wenn dem Prozess zu wenige Rahmen zugestanden werden.

Daher:

❶ Lokalität wird jeweils approximiert durch die

Arbeitsmenge *(working set),*

das sind diejenigen Seiten, die „in der letzten Zeit" vom Prozess benutzt wurden.

❷ Dem Prozess werden jeweils so viele Rahmen zugestanden wie er für seine Arbeitsmenge benötigt.

❸ Beginnendes Seitenflattern weist auf eine Vergrößerung der Arbeitsmenge hin, so dass dem Prozess mehr Rahmen zugestanden werden.

→ Empfehlenswerte Strategie: **Working Set Swapping**

1. Regelmäßige Prozessauslagerung (außer gemeinsame Seiten).
2. Dabei *Working-Set* merken - oder nur seine *Größe*.
3. Wiedereinlagerung setzt voraus, dass entsprechend viel Arbeitsspeicher verfügbar ist (*wo* ist egal!).
4. Arbeitsspeicher entsprechend *reservieren* und *Working Set* einlagern - ***prepaging*** - oder nur aktuelle Code-Seite (!).
5. Im letzteren Fall Seiten einlagern ***on demand***.
6. Beim nächsten Auslagern evtl. kleineren *Working Set* feststellen. Oder bei (lokalem) Seitenflattern Prozess auslagern und *Working-Set*-Größe erhöhen.

5.5.5 Der Speicherverwalter

Speicherverwalter *(memory manager)*

= im einfachsten Fall *ein Systemprozess*,
der für die Umlagerung der Seiten *(page swapping)*
zuständig ist (analog zum Umlagerer/*Swapper*)

reagiert auf

- *Seitenfehler* (Alarm)
- *Beendigung* einer Seitenumlagerung (Eingriff)
- *Uhr* (Eingriff) zwecks evtl. *Working-Set*-Umlagerung

Achtung:

Seiten, in denen eine *Ein/Ausgabe*-Operation läuft,
dürfen nicht ausgelagert werden!

Entweder Ein/Ausgabe grundsätzlich nur über *Pufferbereiche*
im speicherresidenten Teil des Betriebssystems

oder Seite temporär *einfrieren (freezing, pinning, locking)*
und nach beendeter Ein/Ausgabe wieder *auftauen*
(thawing).

Einfrieren/Auftauen ist auch beliebte Technik
für Echtzeit-Anwendungen u.ä. !

+ viele technische Details,

z.B. Seiten benötigen *keine* feste Heimatadresse im Hintergrundspeicher: eine Seite kann an eine beliebige freie Stelle mit geringer Latenzzeit (!) ausgelagert werden.

Beim Einlagern eines *Working Set* sorgt der Gerätetreiber dafür, dass die Seiten in optimaler Reihenfolge - gemäß den anfallenden Latenzzeiten - eingelagert werden.

5.5.5.1 Beispiel Unix System V

betreibt *Working Set Swapping* mit globalem Seitenaustausch:

❶ Wenn freie Rahmen knapp werden *(Niedrigwassermarke)*, *abgestandene* Seiten auslagern,
bis freie Rahmen reichlich vorhanden *(Hochwassermarke)*.

❷ Dies wird *global* praktiziert, ohne Berücksichtigung der verschiedenen Prozesse und gemeinsamer Seitenbenutzung.

❸ .....

❸ Gleichzeitiges *Demand Paging*.

❹ Wenn abgestandene Seiten knapp werden (Seitenflattern droht!), *ganze Prozesse auslagern*, bis freie Rahmen reichlich vorhanden (Hochwassermarke).

❺ Wenn ausgelagerte Prozesse vorhanden, aktive Code-Seite des ältesten einlagern.

Realisierung durch Zusammenwirken von 3 Prozessen:

- Benutzerprozess steuert durch Seitenfehler das *Demand Paging* (Behandlungsroutine blockiert, bis die benötigte Seite eingelagert ist!).

- Systemprozess *Page Stealer* versucht sicherzustellen, dass dafür immer freie Rahmen verfügbar sind; gelingt das nicht, wird der *Swapper* benachrichtigt:

- Systemprozess *Swapper* besorgt das *Working Set Swapping.*

5.5.5.2 Beispiel Windows NT

Speicherverwaltung ist aufgeteilt auf

sechs *System Threads* mit wachsenden Prioritäten:

0 *Zero Page Thread*
hält einen Vorrat an freien Rahmen (gelöscht, *zeroed*)

16 *Balance Set Manager* (im Sekundenabstand aktiviert)
überwacht die *Working Sets*, aktiviert gegebenenfalls
den *Swapper*, erhöht die Priorität verhungernder Prozesse

17 *Modified Page Writer*
kopiert auszulagernde *dirty*-Seiten in den Hintergrundspeicher,
trägt die freien Rahmen in die Freispeicherliste ein

23 *Swapper*
besorgt das *Working Set Swapping*, wirkt aktiviert vom
Balance Set Manager

[ferner für *memory-mapped files* (→6):

17 *Mapped Page Writer* (analog zu *Modified Page Writer*)

18 *Dereference Segment Thread*]

5.5.6 Wahl der Seitengröße

? Gibt es eine optimale Seitengröße ?

Kriterium	spricht für kleine	große Seiten
Speicher sparen:		
Speicherverschnitt	✗	
Working-Set-Größe	✘	
Seitentabellen-Größe		✗
Zeit sparen:		
Umlagerungszeit/Byte		✘
BS-Aktivität		✘

Historische Tendenz:

von kleinen zu größeren Seiten 1 → 16 KB
(billigerer Arbeitsspeicher!)

Wenn allerdings Speicher knapp ist:

Balance zwischen *Speicherverschnitt* und
Speicheraufwand für die *Seitentabelle* ?

Prozessgröße	*p , z.B. 2^{20} Bytes (1 MB)*
Seitengröße	*s*
Deskriptorgröße	*d , z.B. 2 Bytes*

→

interner Verschnitt: $s/2$ (im Mittel)

Seiten je Prozess: p/s

Seitentabelle je Prozess: $d\ p/s$

→

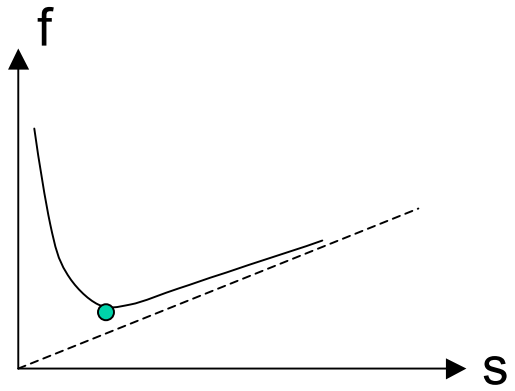
zu minimierender Speicherverbrauch:

$f(s) = s/2 + d\ p/s$

$f'(s) = 1/2 - d\ p/s^2$

$\mathbf{s_{opt} = \sqrt{(2\ d\ p)}}$

z.B. $2 * 2^{10}$ B = 2 KB



5.5.7 Verzögerte Speicherzuweisung

(lazy memory allocation)

Adressraum eines neuen Prozesses oder Erweiterungsteil eines Adressraums ist in der Regel noch nicht überall mit neuen Daten (bzw. Code) belegt.

➜ Deskriptoren noch nicht benutzter Seiten brauchen noch *nicht auf reale Rahmen* zu verweisen.

Erster Zugriffsversuch führt zu *Seitenfehler* oder *Zugriffsfehler,* und daraufhin wird der Rahmen eingerichtet, d.h. realer Speicher belegt.

(Eventuell kann BS sogar „Lesezugriff" erkennen und dies als echten Programmfehler behandeln!)

***copy-on-write*:**

Beim Klonen eines Prozesses (Unix `fork`) muss nicht sofort eine vollständige Kopie des Prozesses erzeugt werden:

Nur die Deskriptoren werden kopiert,
und in allen Deskriptoren wird das *w-Bit* gelöscht.

Lesezugriff von Erzeuger oder Kind: Seite wird gemeinsam benutzt.

1. Schreibzugriff: Zugriffsfehler veranlasst das BS, eine Kopie der Seite herzustellen, den Deskriptor darauf verweisen zu lassen und in beiden Deskriptoren das *w-Bit* zu setzen.