Teil IX

Adressraum und Arbeitsspeicher

Überblick

Adressraum
- Physikalischer Adressraum
- Logischer Adressraum
- Virtueller Adressraum
- Zusammenfassung

Arbeitsspeicher
- Speicherzuteilung
- Platzierungsstrategie
- Ladestrategie
- Ersetzungsstrategie
- Zusammenfassung

Adressraumkonzepte

Betriebssystemsicht

physikalischer Adressraum definiert einen nicht-linear adressierbaren, von Lücken durchzogenen Bereich von E/A-Schnittstellen und Speicher ($*$RAM, $*$ROM), dessen Größe der Adressbreite der CPU entspricht
- 2^N Bytes, bei einer Adressbreite von N Bits

logischer Adressraum definiert einen linear adressierbaren Speicherbereich von 2^M Bytes bei einer Adressbreite von N Bits
- $M = N$ z.B. im Fall einer *Harvard-Architektur*
  - getrennter Programm-, Daten- und E/A-Adressraum
- $M < N$ sonst

virtueller Adressraum ein logischer Adressraum, der 2^K Bytes umfasst
- $K > N$ bei Speicherbankumschaltung, Überlagerungstechnik
- $K \leq N$ sonst

Adressraumorganisation

Adressenbelegung (engl. *address assignment*)

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Ungültige Adressen

Zugriff ⇝ Busfehler (engl. *bus error*)

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Reservierte Adressen

Zugriff ⇝ Schutzfehler (engl. *protection fault*)

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Freie Adressen

Arbeitsspeicher (engl. *working memory*)

Adressbereich	Größe (KB)	Verwendung
`00000000–0009ffff`	640	RAM (System)
`000a0000–000bffff`	128	Video RAM
`000c0000–000c7fff`	32	BIOS Video RAM
`000c8000–000dffff`	96	keine
`000e0000–000effff`	64	BIOS Video RAM (*shadow*)
`000f0000–000fffff`	64	BIOS RAM (*shadow*)
`00100000–090fffff`	147456	RAM (Erweiterung)
`09100000–fffdffff`	4045696	keine
`fffe0000–fffeffff`	64	SM-RAM (*system management*)
`ffff0000–ffffffff`	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

Segmentierung

Logische Unterteilung von Programmadressräumen

Ebene$_4$-Programme sind in (mind.) zwei Segmente logisch aufgeteilt:

- Text → Maschinenanweisungen, Programmkonstanten
- Daten → initialisierte Daten, globale Variablen, Halde

Ebene$_3$-Programme kennen (mind.) ein weiteres Segment:

- Stapel → lokale Variablen, Hilfsvariablen, aktuelle Parameter

Betriebssysteme verwalten diese Segmente im physikalischen Adressraum

- ggf. mit Hilfe einer MMU (engl. *memory management unit*)
  - Hardware, die nur logische in physikalische Adressen umsetzt
  - für die Verwaltung des Speichers ist das Betriebssystem verantwortlich
- die MMU legt eine Organisationsstruktur auf den phys. Adressraum
  - sie unterteilt ihn in Seiten fester oder Segmente variabler Länge

Ausprägungen von Programmadressräumen

Seitennummerierter oder segmentierter Adressraum

eindimensional in Seiten aufgeteilt (engl. *paged*)

- eine Programmadresse A_P bildet ein Tupel (p, o):

  $p = A_P \texttt{ div } 2^N$ ⇝ Seitennummer (engl. *page number*)
  $o = A_P \texttt{ mod } 2^N$ ⇝ Versatz (engl. *byte offset*)

  - mit 2^N gleich der Seitengröße (engl. *page size*) in Bytes
- Seite ↦ Seitenrahmen (auch: Kachel) des phys. Adressraums

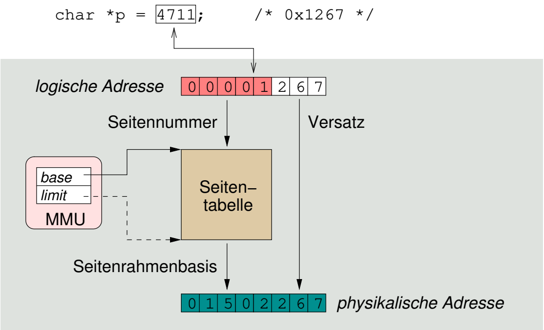
zweidimensional in Segmente aufgeteilt (engl. *segmented*)

- eine Programmadresse A_S bildet ein Paar (S, A)
  - mit der Adresse A relativ zu Segment(name/nummer) S
  - bei seitennummerierten Segmenten wird A als A_P interpretiert
- Segment ↦ Folge von Bytes/Seitenrahmen des phys. Adressraums

---

Adressumsetzung

Seitennummerierter Adressraum (engl. *paged address space*)



```
char *p = 4711;     /* 0x1267 */
```

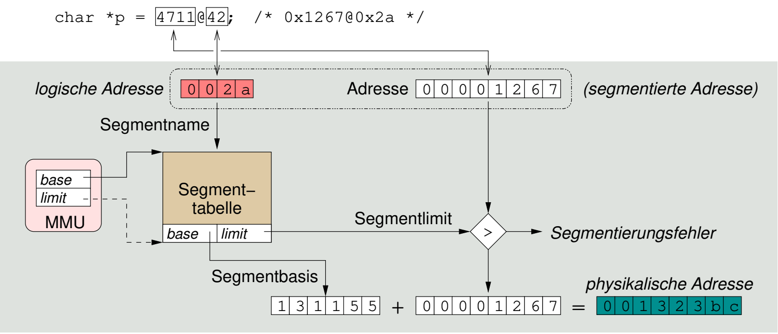
- die Seitennummer ist ein Index in eine Seitentabelle
  - pro Prozess
  - dimensioniert durch die MMU
- ein ungültiger Index führt zum *Trap*

- die indizierte Adressierung (der MMU) liefert einen Seitendeskriptor
  - enthält die Seitenrahmennummer, die die Seitennummer ersetzt
  - entspricht der Basisadresse des Seitenrahmens im phys. Adressraum
- setzen der Basis-/Längenregister der MMU ⇝ Adressraumwechsel

---

Adressumsetzung (Forts.)

Segmentierter Adressraum (engl. *segmented address space*)

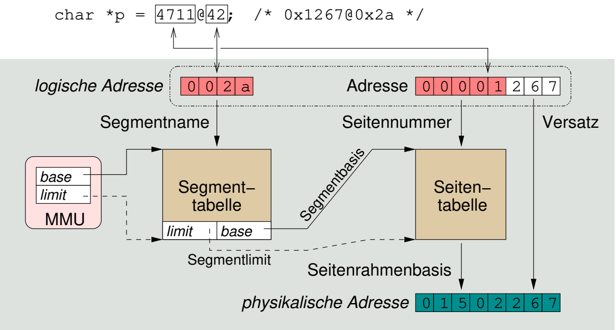


```
char *p = 4711@42;  /* 0x1267@0x2a */
```

- der Segmentname ist ein Index in die Segmenttabelle eines Prozesses
  - dimensioniert durch die MMU, ungültiger Index führt zum *Trap*
- die indizierte Adressierung (der MMU) ergibt den Segmentdeskriptor
  - enthält Basisadresse und Länge des Segments (engl. *base/limit*)
  - $address_{phys} = address > limit \; ? \; Trap : base + address$

Adressumsetzung (Forts.)

Segmentierter seitennummerierter Adressraum (engl. *page-segmented address space*)



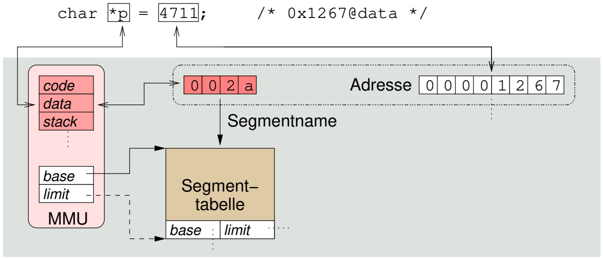
```
char *p = 4711@42;  /* 0x1267@0x2a */
```

- Reihenschaltung von zwei Adressumsetzungseinheiten der MMU:
  Segmenteinheit löst eine segmentierte Adresse auf
  - adressiert und begrenzt die Seitentabelle

  Seiteneinheit generiert die physikalische Adresse
- jeder Prozess hat (mind.) eine Segment- und eine Seitentabelle

Adressumsetzung (Forts.)

Segmentregister bzw. Segmentselektor (engl. *segment selector*)



- je nach Art des Speicherzugriffs selektiert die MMU implizit das passende Segment

Befehlsabruf (engl. *instruction fetch*) ↦ *code* ✓

Operandenabruf (engl. *operand fetch*) ⇝ Text, Daten oder Stapel ?

- Direktwerte ↦ *code*
- globale/lokale Daten ↦ *data*/*stack*
  - `char *foo (char bar) { return &bar; }`

☞ Programme können weiterhin 1-dimensionale log. Adressen verwenden

---

Seiten- bzw. Segmentdeskriptor

Abbildung steuernder Verbund

Adressumsetzung basiert auf Deskriptoren der MMU, die für jede Seite/Segment eines Prozesses Relokations- und Zugriffsdaten verwalten

- die Basisadresse des Seitenrahmens/Segments im phys. Adressraum
- die Zugriffsrechte des Prozesses
  - lesen (*read*), schreiben (*write*), ggf. ausführen (*execute*)

Segmente sind (im Gegensatz zu Seiten) von variabler, dynamischer Größe und benötigen daher zusätzliche Verwaltungsdaten ⇝ Segmentdeskriptor

- die Segmentlänge, um Segmentverletzungen abfangen zu können
  - Basis-/Längenregister (S. IV-33) ⊂ Segmentdeskritor
- die Expansionsrichtung: Halde „*bottom-up*“, Stapel „*top-down*“

Programmierung der Deskriptoren erfolgt zur Programmlade- und -laufzeit

- bei Erzeugung/Zerstörung schwer- und leichtgewichtiger Prozesse
- bei Anforderung/Freigabe von Arbeitsspeicher

Seiten- bzw. Segmenttabelle

Adressraum beschreibende Datenstruktur

Deskriptoren des Adressraums eines Prozesses sind in einer Tabelle im Arbeitsspeicher zusammengefasst

- die Arbeitsmenge (engl. *working set*) von Deskriptoren eines Prozesses wird im Zwischenspeicher (engl. *cache*) gehalten
  - TLB (engl. *translation lookaside buffer*) der MMU
- Adressraumwechsel als Folge eines Prozesswechsels bedeutet:
  1. zerstören der Arbeitsmenge (TLB „*flush*“; teuer, schwergewichtig)
  2. Tabellenwechsel (Zeiger umsetzen; billig, federgewichtig)

Basis-/Längenregister (engl. *base/limit register*)

- beschreibt eine Tabelle und damit exakt einen Prozessadressraum
- bei der Adressumsetzung wird eine Indexprüfung durchgeführt:
  - $descriptor = index \le limit\ ?\ \&base[index] : Trap$
  - wobei *index* die Seitennummer/den Segmentnamen repräsentiert

---

Adressraumabbildung einhergehend mit Ein-/Ausgabe

Integration von Vorder- und Hintergrundspeicher

Abstraktion von Größe und Örtlichkeit des verfügbaren Arbeitsspeichers

- vom Prozess nicht benötigte Programmteile können ausgelagert sein
  - sie liegen im Hintergrundspeicher, z.B. auf der Festplatte
- der Prozessadressraum könnte über ein Rechnernetz verteilt sein
  - Programmteile sind über die Arbeitsspeicher anderer Rechner verstreut

Zugriffe auf nicht eingelagerte Programmteile fängt der Prozessor ab: *Trap*

- sie werden stattdessen partiell interpretiert vom Betriebssystem
- der unterbrochene Prozess wird in einen E/A-Stoß gezwungen
  - er erwartet die erfolgreiche Einlagerung eines Programmteils
  - ggf. sind andere Programmteile aus dem Arbeitsspeicher zu verdrängen
- Wiederaufnahme des CPU-Stoßes führt zur Wiederholung des Zugriffs

Seiten- bzw. Segmentdeskriptor (Forts.)

Zusätzliche Attribute

Adressumsetzung unterliegt einer Steuerung, die „transparent“ für den zugreifenden Prozess die Einlagerung auslöst

- die Gegenwart eines Segments/einer Seite wird erfasst: *present bit*
  - 0 ↦ ausgelagert; *Trap*, partielle Interpretation, Einlagerung
    - die Basisadresse ist eine Adresse im Hintergrundspeicher
    - wird nach der Einlagerung vom Betriebssystem aus 1 gesetzt
  - 1 ↦ eingelagert; Befehl abrufen, Operanden lesen/schreiben
    - die Basisadresse ist eine Adresse im Vordergrundspeicher
    - wird nach der Auslagerung vom Betriebssystem aus 0 gesetzt
- das „Gegenwartsbit“ dient verschiedentlich noch anderen Zwecken:
  - um z.B. Zugriffe zu zählen oder ihnen einen Zeitstempel zu geben
  - Betriebssystemmaßnahmen zur Optimierung der Ein-/Auslagerung

---

Zugriffsfehler

Seitenfehler (engl. *page fault*) bzw. Segmentfehler (engl. *segment fault*)

present bit $= 0$ je nach Befehlssatz und Adressierungsarten der CPU kann der Behandlungsaufwand im Betriebssystem und der damit verbundene Leistungsverlust für die Programme beträchtlich sein

```
void hello () {
    printf("Hi!\n");
}

void (*moin)() = &hello;

main () {
    (*moin)();
}
```

```
main:
    pushl %ebp
    movl  %esp,%ebp
    pushl %eax
    pushl %eax
    andl  $-16,%esp
    call  *moin
    leave
    ret
```

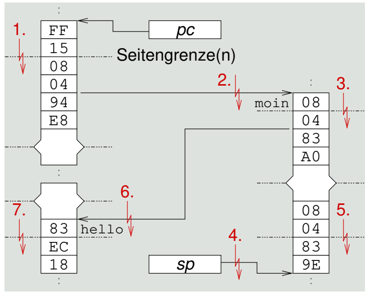
```
     ⋮
FF15080494E8
     ⋮
```

Zugriffsfehler (Forts.)

Beispiel des schlimmsten Falls eines Seitenfehlers. . .

`call *moin` (x86) Aufruf einer indirekt adressierten Prozedur

- der Operationskode (`FF 15`) wurde bereits gelesen



1. Operandenadresse holen (`08 04 94 E8`)
2. Funktionszeiger lesen (`08`)
3. Funktionszeiger weiterlesen (`04 83 A0`)
4. Rücksprungadresse stapeln (`08 04`)
5. Rücksprungadresse weiterstapeln (`83 9E`)
6. Operationskode holen (`83`)
7. Operanden holen (`EC 18`)

- Seitenfehler 6. und 7. sind eigentlich bereits der Ausführung des ersten Maschinenbefehls der aufgerufenen Prozedur zuzurechnen

---

Aufwandsabschätzung eines Seitenfehlers

Seitenfehler sind nicht-funktionale Programmeigenschaften

effektive Zugriffszeit (*effective access time*, *eat*) auf den Arbeitsspeicher

- hängt stark ab von der Seitenfehlerwahrscheinlichkeit (p) und verhält sich direkt proportional zur Seitenfehlerrate:

$$eat = (1 - p) \cdot pat + p \cdot pft, 0 \leq p \leq 1$$

- angenommen, folgende Systemparameter sind gegeben:
  - 50 ns Zugriffszeit auf den RAM (*physical access time*, *pat*)
  - 10 ms mittlere Zugriffszeit auf eine Festplatte (*page fault time*, *pft*)
  - 1 % Wahrscheinlichkeit eines Seitenfehlers ($p = 0,01$)
- dann ergibt sich:
  $eat = 0,99 \cdot 50\,\text{ns} + 0,01 \cdot 10\,\text{ms} = 49,5\,\text{ns} + 10^5\,\text{ns} \approx 0,1\,\text{ms}$

☞ Einzelzugriffe sind im Ausnahmefall um den Faktor 2000 langsamer

Aufwandsabschätzung eines Seitenfehlers (Forts.)

Seitenfehler sind nicht wirklich transparent

mittlere Zugriffszeit (*mean access time*, *mat*) auf den Arbeitsspeicher

- hängt stark ab von der effektiven Seitenzugriffszeit und der Seitengröße (in Bytes pro Seite bzw. Seitenrahmen):

$$mat = (eat + (sizeof(page) - 1) \cdot pat)/pat$$

- angenommen, folgende Systemparameter sind gegeben:
  - Seitengröße von 4 096 Bytes (4 KB)
  - 50 ns Zugriffszeit (*pat*) auf ein Byte im RAM
  - effektive Zugriffszeit (*eat*) wie eben berechnet bzw. abgeschätzt
- dann ergibt sich:
  $mat = (eat + 4\,095 \cdot 50\,\text{ns})/50\,\text{ns} = 6\,095,99\,\text{ns} \approx 6\,\mu\text{s}$

☞ Folgezugriffe sind im Ausnahmefall im Ø um den Faktor 122 langsamer

---

Seitenüberlagerung „*Considered Harmful*“

Pro und Contra

Virtuelle Adressräume sind . . .

vorteilhaft wenn „übergroße“ bzw. gleichzeitig/nebenläufig viele Programme in Anbetracht zu knappen Arbeitsspeichers auszuführen sind

ernüchternd wenn der eben durch die Virtualisierung bedingte Mehraufwand zu berücksichtigen ist und sich für ein gegebenes Anwendungsszenario als problematisch bis unakzeptabel erweisen sollte

Seitenfehler sind . . .

- nicht wirklich transparent, wenn zeitliche Aspekte relevant sind
  - z.B. im Fall von Echtzeitverarbeitung oder Hochleistungsrechnen
- erst zur Laufzeit ggf. entstehende nicht-funktionale Eigenschaften

---

Adressräume

Ebenen der Abstraktion

physikalischer Adressraum enthält gültige und ungültige Adressen

ungültige Adressen Zugriff führt zum Busfehler

gültige Adressen Zugriff gelingt, ist jedoch zu bedenken. . .

- reservierte Adressbereiche sind ggf. zu schützen

logischer Adressraum enthält gültige Adressen

- Zugriffsrechte der Prozesse bestimmen, welche Adressen gültig sind
  - Zugriff auf reservierte Adressen führt ggf. zum Schutzfehler
- Prozesse sind in ihrem Programmadressraum abgeschottet, isoliert
  - Zugriff auf „fremde“ freie Adressen führt zum Schutzfehler

virtueller Adressraum enthält „flüchtige Adressen“

- die Bindung der Adressen zu den Speicherzellen ist nicht fest
- sie variiert phasenweise zwischen Vorder- und Hintergrundspeicher

---

Adressraumdeskriptoren

Seitennumerierte und segmentierte Adressräume

Abbildung steuernde Verbunde zur Beschreibung einzelner Adressraumteile

- speichern Attribute von Seiten oder Segmente
  - d.h., Relokations- und Zugriffsdaten, Zugriffsrechte
- bilden Seiten fester Größe auf gleichgroße Seitenrahmen ab
  - seitennummerierter Adressraum
- bilden Segmente variabler Größe auf Byte- oder Seitenfolgen ab
  - segmentierter und ggf. seitennummerierter Adressraum

Abbildungstabellen fassen Deskriptoren (eines Adressraums) zusammen

- die Tabellen werden vom Betriebssystem im Arbeitsspeicher verwaltet
  - der dafür erforderliche Speicherbedarf kann beträchtlich sein
- im TLB sind Arbeitsmengen von Deskriptoren zwischengespeichert
  - ein *Cache* der MMU, ohne dem Adressumsetzung ineffizient ist

Zugriffsfehler sind intransparente, nicht-funktionale Eigenschaften

Überblick

- Adressraum
  - Physikalischer Adressraum
  - Logischer Adressraum
  - Virtueller Adressraum
  - Zusammenfassung
- Arbeitsspeicher
  - Speicherzuteilung
  - Platzierungsstrategie
  - Ladestrategie
  - Ersetzungsstrategie
  - Zusammenfassung

Verwaltung des Arbeitsspeichers: Fragmente

Fest abgesteckte oder ausdehn-/zusammenziehbare Gebiete für jedes Programm

statisch allen Programmen inkl. dem Betriebssystem sind Arbeitsspeichergebiete maximaler, fester Größe zugewiesen

- innerhalb der Gebiete ist Speicher dynamisch zuteilbar
- Gefahr von Leistungsbegrenzung/-verluste, z.B.:
  - Brache eines Gebiets ist in anderen Gebieten nicht nutzbar
  - kleine E/A-Bandbreite mangels Puffer im Gebiet des BS
  - erhöhte Wartezeit von Prozessen wegen zu kleinen Puffern

dynamisch das Betriebssystem ermittelt freie Arbeitsspeicherfragmente angeforderter Größe und teilt diese den Programmen zu

- Zusammenspiel Laufzeit- und Betriebssystem (S. V-22)
  - d.h., von `malloc(3)` und z.B. `brk(2)`

☞ die Zuteilungseinheiten können in beiden Fällen gleich ausgelegt sein

---

Zuteilungseinheiten und Verschnitt

Vielfaches von Bytes oder Seitenrahmen

Aufbau und Struktur der jeweils zugeteilten Arbeitsspeicherfragmente unterscheidet sich je nach Adressraumausprägung (S. IX-9)

Seitennummerierung Fragment ↦ Vielfaches von Seitenrahmen

- ggf. wird mehr Speicher als benötigt zugeteilt
- interne Fragmentierung des Seitenrahmens

Segmentierung Fragment ↦ Vielfaches von Bytes (Segment)

- ggf. ist ein passendes Stück nicht verfügbar
- externe Fragmentierung des Arbeitsspeichers

Verschnitt (als Folge in/externer Fragementierung) zu optimieren, ist eine der zentralen Aufgaben der Speicherverwaltung

anfallender Rest bei der Speicherzuteilung allgemein

- „Abfall" im Falle interner Fragmentierung
- „Hohlräume" im Falle externer Fragmentierung

Politiken bei der Speicherzuteilung

Wohin, wann und welches Opfer...

Platzierungsstrategie (engl. *placement policy*) obligatorisch

- wohin ist ein Fragment abzulegen?
  - wo der Verschnitt am kleinsten/größten ist
  - egal, weil Verschnitt zweitrangig ist

Ladestrategie (engl. *fetch policy*) optional: dyn. Binden, VM

- wann ist ein Fragment zu laden?
  - auf Anforderung oder im Voraus

Ersetzungsstrategie (engl. *replacement policy*) optional: VM

- welches Fragment ist ggf. zu verdrängen?
  - das älteste, am seltensten genutzte
  - das am längsten ungenutzte

VM Abk. für engl. *virtual memory*, d.h. virtueller Speicher

Freispeicherorganisation

Verwaltung der „Hohlräume“ im Arbeitsspeicher

Bitkarte (engl. *bit map*) von Fragmenten fester Größe
  - eignet sich für seitennummerierte Adressräume
  - grobkörnige Vergabe auf Seitenrahmenbasis
  - alle freien Fragmente sind gleich gut ✓

verkettete Liste (engl. *free list*) von Fragmenten variabler Größe
  - ist typisch für segmentierte Adressräume
  - feinkörnige Vergabe auf Segmentbasis
  - nicht alle freien Fragmente sind gleich gut ?

Freispeicher erscheint als „Hohlräume“ (auch “Löcher“ genannt) im RAM, die mit Programmtext/-daten auffüllbar sind

- als Bitkarte/verk. Liste implementierte Löcherliste (engl. *hole list*)

Freispeicher(bit)karte

Erfassung freien Speichers fester Größe

Fragmenten des Arbeitsspeichers ist (mind.) ein Zustand zugeordnet, der durch einen Bitwert repräsentiert wird: $0 \mapsto$ belegt, $1 \mapsto$ frei

- Suche, Belegung und Freigabe $\leadsto$ Operationen zur Bitverarbeitung

Anforderungen von K Bytes zu erfüllen bedeutet, M Zuteilungseinheiten in der Bitkarte zu suchen, deren Zustand „frei“ anzeigt:

$$M = \frac{K + sizeof(unit) - 1}{sizeof(unit)}$$

- mit *unit* definiert als $char[N]$, d.h. allgemein ein Bytefeld darstellend
  - $N = 1$ im Falle segmentierter Adressräume
  - $N = sizeof(page)$ im Falle seitennummerierter Adressräume

☞ Abfall $M * sizeof(unit) - K$ ist nur möglich für $sizeof(unit) > 1$

Freispeicher(bit)karte (Forts.)

Umfang hängt ab von der Größe der Zuteilungseinheiten

Erfassung freier Fragmente beansprucht mehr oder weniger viel Speicher:

- z.B. ein System mit 1 GB Hauptspeicher und 4 KB Seitengröße
- die dazu passende Bitkarte hat eine Größe von 32 KB:

$$\begin{aligned} sizeof(bit\ map) &= 1\,\text{GB} \div 4\,\text{KB} \div 8\,\text{Bits} \\ &= 2^{30} \div 2^{12} \div 2^{3} = 2^{15}\,\text{Bytes} \end{aligned}$$

- der Bitkartenumfang variiert mit der Seiten(rahmen)größe
  - gleich gr. Speicher $\leadsto$ ungleich gr. Gemeinkosten (engl. *overhead*)
- die MMU unterstützt ggf. eine Abstimmung (engl. *tuning*)
  - durch einstell- bzw. programmierbare Seiten(rahmen)größen

☞ je feinkörniger die Speicherzuteilung, desto größer die Bitkarte

---

Freispeicherliste

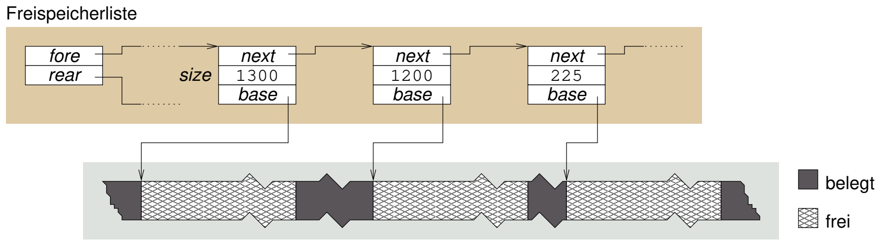
Erfassung freien Speichers variabler Größe

Löcherliste (engl. *hole list*) führt Buch über freie Speicherbereiche

- ein Listenelement hält Anfangsadresse und Länge eines Bereichs
  - d.h., es erfasst genau ein freies Fragment
- für die Liste ergeben sich zwei grundsätzliche Repräsentationsformen:
  1. Liste und Löcher sind voneinander getrennt
     - die Listenelemente sind Löcherdeskriptoren, sie belegen Betriebsmittel
     - Löcher haben eine beliebige Größe N, $N > 0$
  2. Liste und Löcher sind miteinander vereint
     - die Listenelemente sind die Löcher, sie belegen keine Betriebsmittel
     - Löcher haben eine Mindestgröße N, $N \geq sizeof(list\ element)$
- strategische Überlegungen bestimmen die Art der Listenverwaltung

Freispeicherliste (Forts.)

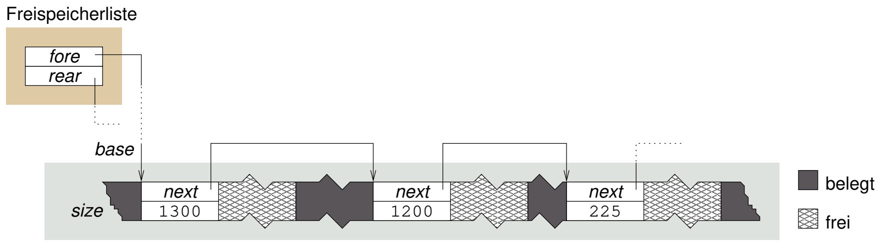
Listenelemente als Löcherdeskriptoren



- die Liste und der Listenkopf liegen im Betriebssystemadressraum
  - *fore*, *rear* und *next* sind logische/virtuelle Adressen
  - *size* ist die Größe des Lochs, Vielfaches von *sizeof*(*unit*) (S. IX-30)
  - *base* ist die physikalische Adresse des freien Fragments
- Listenmanipulationen laufen innerhalb eines log./virt. Adressraums ab

Freispeicherliste (Forts.)

Listenelemente als Löcher



- nur der Listenkopf liegt im Betriebssystemadressraum
  - *fore*, *rear*, *next* und *base* sind physikalische Adressen
  - *size* ist die Größe des Lochs, Vielfaches von *sizeof*(*unit*) (S. IX-30)
- Listenmanipulationen müssen ggf. Adressraumgrenzen überschreiten
  - die Listenoperationen laufen im Betriebssystemadressraum ab
  - der Betriebssystemadressraum ist ein logischer/virtueller Adressraum
  - die Liste ist im physikalischen Adressraum $\rightsquigarrow$ Adressraumumschaltung

Zuteilungsverfahren

Löcher der Größe nach sortieren

best-fit verwaltet Löcher nach aufsteigenden Größen
- Ziel ist es, den Verschnitt zu minimieren
  - d.h., das kleinste passende Loch zu suchen
- erzeugt kl. Löcher am Anfang, erhält gr. Löcher am Ende
  - hinterlässt eher kleine Löcher, bei steigendem Suchaufwand

worst-fit verwaltet Löcher nach absteigenden Größen
- Ziel ist es, den Suchaufwand zu minimieren
  - ist das erste Loch zu klein, sind es alle anderen auch
- zerstört gr. Löcher am Anfang, erzeugt kl. Löcher am Ende
  - hinterlässt eher große Löcher, bei konstantem Suchaufwand

Ist die angeforderte Größe kleiner als das gefundene Loch, fällt Verschnitt an, der als verbleibendes Loch in die Liste neu einsortiert werden muss:
- d.h., die Freispeicherliste ist ggf. zweimal zu durchlaufen

Zuteilungsverfahren (Forts.)

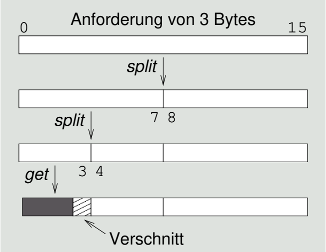
Löcher der Größe (2er-Potenz) nach sortieren

buddy (Kamerad, Kumpel) verwaltet Löcher nach aufsteigenden Größen
- das kleinste passende Loch $buddy_i$ der Größe 2^i suchen
  - i ist Index in eine Tabelle von Adressen auf Löcher der Größe 2^i
- $buddy_i$ entsteht durch (sukzessive) Splittung von $buddy_{j, j>i}$:
  - $2^n = 2 \times 2^{n-1}$
  - zwei gleichgroße Blöcke, die *Buddy* des jeweils anderen sind
- der ggf. anfallende Verschnitt kann beträchtlich sein
  - schlimmstenfalls $2^i - 1$ bei $2^i + 1$ angeforderten Einheiten
- ein Kompromiss zwischen *best-fit* und *worst-fit*
  - vergleichsweise geringer Such- und Aufsplittungsaufwand
  - passt gut zum Laufzeitsystem, das in 2er-Potenzen anfordert

Jeder Rest ist als Summe freier *Buddies* darstellbar, wie auch jede Dezimalzahl als Summe von 2er-Potenzen.

Zuteilungsverfahren (Forts.)

Sukzessive Aufsplittung bei *Buddy*



1. Block 2^4 teilen: $3 < 2^4/2$
2. Block 2^3 teilen: $3 < 2^3/2$
3. Block 2^2 vergeben: $3 \geq 2^2/2$
    - Verschnitt von $2^2 - 3 = 1$ Byte

Ob der Verschnitt als interne Fragmentierung zu verbuchen ist, hängt von der MMU ab.

Verschmelzung bei Speicherfreigabe wird zum „Kinderspiel“...

- zwei freie Blöcke können verschmolzen werden, wenn sie *Buddies* sind
    - die Adressen von *buddies* unterscheiden sich nur in einer Bitposition
- zwei Blöcke der Größe 2^i sind genau dann *Buddies*, wenn sich ihre Adressen in Bitposition i unterscheiden

---

Zuteilungsverfahren (Forts.)

Löcher der Adresse nach sortieren

first-fit verwaltet Löcher nach aufsteigenden Adressen

- Ziel ist es, den Verwaltungsaufwand zu minimieren
    - invariante Adressen sind das Sortierkriterium
    - die Liste ist bei anfallendem Rest nicht umzusortieren
- erzeugt kl. Löcher vorne, erhält gr. Löcher am Ende
    - hinterlässt eher kl. Löcher, bei steigendem Suchaufwand

next-fit reihum (engl. *round-robin*) Variante von *first-fit*

- Ziel ist es, den Suchaufwand zu minimieren
    - Suche beginnt immer beim zuletzt zugeteiltem Loch
- nähert sich einer Verteilung von „gleichgroßen Löchern“
    - als Folge nimmt der Suchaufwand ab

Ist die angeforderte Größe kleiner als das gefundene Loch, fällt Verschnitt an, der jedoch nicht als Restloch in die Liste einsortiert werden muss:

- d.h., die Freispeicherliste ist nur einmal zu durchlaufen

Verschmelzung

Vereinigung eines Lochs mit angrenzenden Löchern

Verschmelzung von Löchern produziert ein großes Loch, die Maßnahme...

- beschleunigt die Speicherzuteilung, verringert externe Fragmentierung
- erfolgt bei Speicherfreigabe oder scheiternder Speichervergabe

Löchervereinigung sieht sich mit vier Situationen konfrontiert, je nach dem, welche relative Lage ein Loch im Arbeitsspeicher hat:

1. zw. zwei zugeteilten Bereichen	▶ keine Vereinigung möglich
2. direkt nach einem Loch	▶ Vereinigung mit Vorgänger
3. direkt vor einem Loch	▶ Vereinigung mit Nachfolger
4. zwischen zwei Löchern	▶ Kombination von 2. und 3.

☞ der Aufwand variiert z.T. sehr stark mit dem Zuteilungsverfahren

---

Verschmelzung vs. Zuteilungsverfahren

Aufwand ist klein bei *buddy*, mittel bei *first/next-fit*, groß bei *best/worst-fit*

buddy anhand eines Bits der Adresse des zu verschmelzenden Lochs lässt sich leicht feststellen, ob sein *Buddy* bereits als Loch in der Tabelle verzeichnet ist

first/next-fit beim Durchlaufen der Freispeicherliste (bei Freigabe) wird jeder Eintrag daraufhin überprüft, ob...

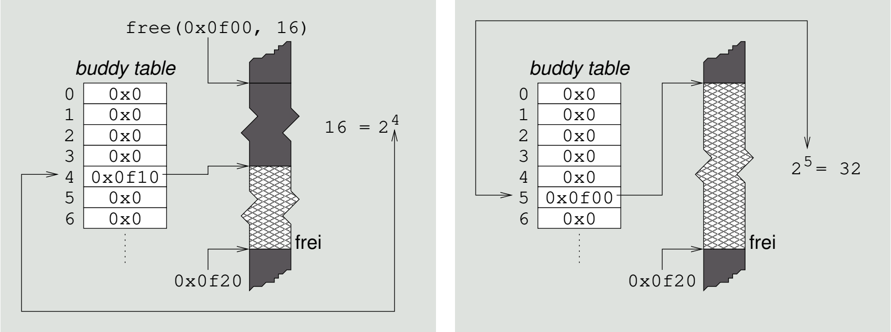
- Adresse plus Größe eines Eintrags gleich der Adresse des zu verschmelzenden Lochs ist $\leadsto$ 2.
- Adresse plus Größe eines zu verschmelzenden Lochs der Adresse eines Eintrags entspricht $\leadsto$ 3.

best/worst-fit ähnlich wie bei *first/next-fit*, jedoch kann im Gegensatz dazu nicht davon ausgegangen werden, dass bei einem angrenzenden Loch das Vorgänger- bzw. Nachfolgerelement in der Liste das ggf. andere angrenzende Loch sein muss

- es muss weitergesucht werden...

Verschmelzung — *buddy*

Zwei Blöcke 2^i sind *Buddies*, wenn sich ihre Adressen in Bitposition *i* unterscheiden

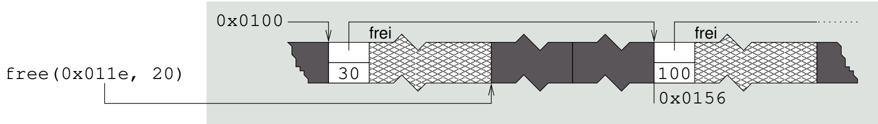


$$\begin{aligned} 0f00_{16} &= 0000\,1111\,0000\,0000_2 \\ 0f10_{16} &= 0000\,1111\,0001\,0000_2 \\ 16_{10} &= 0000\,0000\,0001\,0000_2 \end{aligned}$$

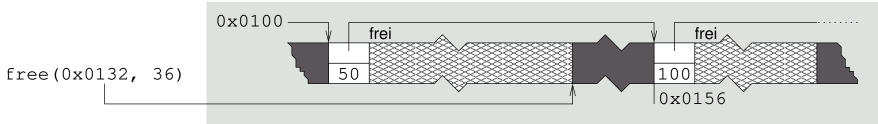
$$\begin{aligned} 0f00_{16} &= 0000\,1111\,0000\,0000_2 \\ 0f20_{16} &= 0000\,1111\,0010\,0000_2 \\ 32_{10} &= 0000\,0000\,0010\,0000_2 \end{aligned}$$

Verschmelzung — *first/next-fit*

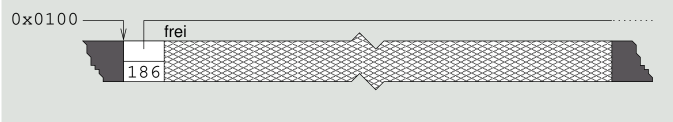
Freizugebender Block ist Nachfolger und/oder Vorgänger



- das alte 30 Byte große Loch kann um 20 Bytes vergößert werden



1. das alte 50 Byte große Loch kann um 36 Bytes vergößert werden
2. das neue 86 Byte große Loch kann um 100 Bytes vergößert werden



Fragmentierung

Abfall eines zugeteilten Bereichs oder Hohlräume im Arbeitsspeicher

(lat.) Bruchstückbildung; Zerstückelung des Speichers in immer kleinere, verstreut vorliegende Bruchstücke

intern bei seitennummerierten Adressräumen ⇝ Verschwendung
- Speicher wird in Einheiten gleicher, fester Größe vergeben
  - eine angeforderte Größe muss kein Seitenvielfaches sein
  - am Seiten(rahmen)ende kann ein Bruchstück entstehen
- der „lokale Verschnitt“ ist nutzbar, dürfte aber nicht sein

extern bei segmentierten Adressräumen ⇝ Verlust
- Speicher wird in Einheiten variabler Größe vergeben
  - eine linear zusammenhängende Bytefolge passender Länge
  - anhaltender Betrieb produziert viele kleine Bruchstücke
- der „globale Verschnitt“ ist ggf. nicht mehr zuteilbar
  - Kompaktifizierung des Arbeitsspeichers schafft ggf. Abhilfe

Kompaktifzierung

Auflösung externer Fragmentierung durch Vereinigung des globalen Verschnitts

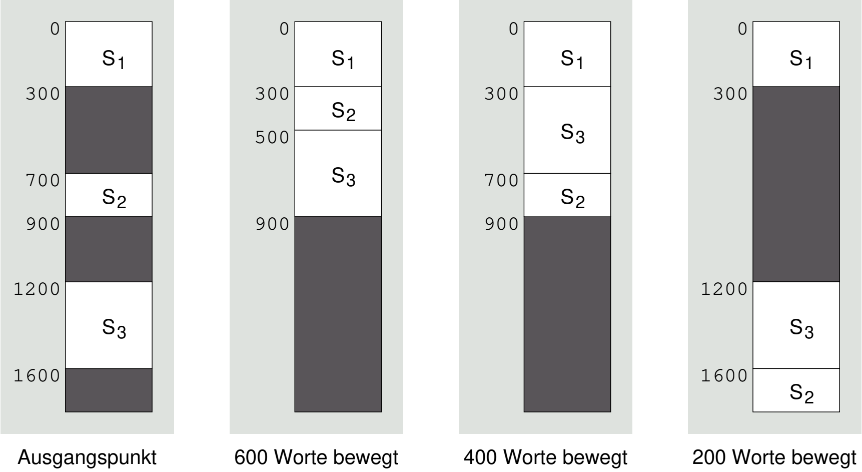
Segmente von (Bytes oder Seitenrahmen) werden so verschoben, dass am Ende ein einziges großes Loch vorhanden ist
- alle in der Freispeicherliste erfassten Löcher werden sukzessive verschmolzen, so dass schließlich nur noch ein Loch übrigbleibt
- durch Umlagerung (engl. *swapping*) kompletter Segmente bzw. Adressräume wird der Kopiervorgang „erleichtert“

Relokation der verschobenen Segmente/Seiten(rahmen) ist erforderlich
- das Betriebssystem implementiert logische/virtuelle Adressräume oder
- der Übersetzer generiert positionsunabhängigen Programmtext

☞ je nach Fragmentierungsgrad ein komplexes Optimierungsproblem. . .

Kompaktfizierung (Forts.)

Loslegen und Aufwand riskieren oder vorher nachdenken und Aufwand einsparen. . .



Ausgangspunkt | 600 Worte bewegt | 400 Worte bewegt | 200 Worte bewegt

---

Einlagerung von Seiten/Segmenten

Spontanität oder vorauseilender Gehorsam

Einzelanforderung „*on demand*“ → *present bit* (S. IX-17)
- Seiten-/Segmentzugriff führt zum *Trap*
  - engl. *page/segment fault*
- Ergebnis der Ausnahmebehandlung ist die Einlagerung der angeforderten Einheit

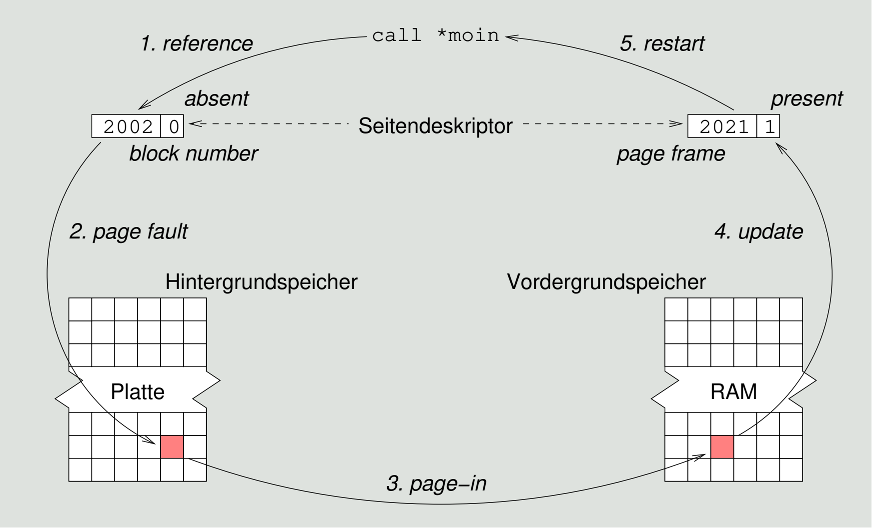
Vorausladen „*anticipatory*“
- Heuristiken liefern Hinweise über Zugriffsmuster
  - Programmlokalität, Arbeitsmenge (*working set*)
- alternativ auch als Vorabruf (engl. *prefetch*) im Zuge einer Einzelanforderung
  - z.B. zur Vermeidung von Folgefehlern (S. IX-19)

☞ ggf. fällt die Verdrängung (Ersetzung) von Seiten/Segmenten an

---

Einzelanforderung

On-demand paging — durch ein in/externes „Rufgerät“ (engl. *pager*) des Betriebssystems



Vorausladen im Zuge einer Einzelanforderung

Vorbeugung ggf. nachfolgender Seitenfehler

`call *moin` (S. IX-19)

1. den gescheiterten Befehl dekodieren, Adressierungsart feststellen
2. da der Operand die Adresse einer Zeigervariablen (`moin`) ist, den Adresswert auf Überschreitung einer Seitengrenze prüfen
3. da der Befehl die Rücksprungadresse stapeln wird, die gleiche Überprüfung mit dem Stapelzeiger durchführen
4. in der Seitentabelle die entsprechenden Deskriptoren lokalisieren und prüfen, ob die Seiten anwesend sind
   - jede abwesende Seite (*present bit* $= 0$) ist einzulagern
5. da jetzt die Zeigervariable (`moin`) vorliegt, sie dereferenzieren und ihren Wert auf Überschreitung einer Seitengrenze prüfen
   - hierzu wie bei 4. verfahren
6. den unterbrochenen Prozess den Befehl wiederholen lassen

☞ Teilemulation eines Maschinenbefehls durch das Betriebssystem

Verdrängung eingelagerter Fragmente

Platz schaffen zur Einlagerung anderer Fragmente (d.h., Seiten oder Segmente)

Konsequenz zur Durchsetzung der Ladestregie bei Arbeitsspeichermangel

- wenn eine Überbelegung des Arbeitsspeichers vorliegt
  - der Speicherbedarf aller Prozesse ist größer als der verfügbare RAM
- aber auch im Falle (extensiver) externer Fragmentierung

OPT (optimales Verfahren) Verdrängung der Seite, die am längsten nicht mehr verwendet werden wird — unrealistisch

- erfordert Wissen über die im weiteren Verlauf der Ausführung von Prozessen generierten Speicheradressen, was bedeutet:
  - das Laufzeitverhalten von Prozessen ist im Voraus bekannt
  - Eingabewerte sind vorherbestimmt
  - asynchrone Programmunterbrechungen sind vorhersagbar
- bestenfalls ist eine gute Approximation möglich/umsetzbar

☞ Seitenfehlerwahrscheinlichkeit senken und Seitenfehlerrate verringern

---

Ersetzungsverfahren

Praxistaugliche Herangehensweisen

Approximation des optimalen Verfahren greift auf Wissen aus der Vergangenheit/Gegenwart zurück, d.h., ersetzt wird ...

FIFO (*first-in, first-out*) das zuerst eingelagerte Fragment

- Fragmente entsprechend des Einlagerungszeitpunkts verketten

LFU (*least frequently used*) das am seltenste genutzte Fragment

- jeden Zugriff auf eingelagerte Fragmente zählen
- Alternative: MFU (*most frequently used*)

LRU (*least recently used*) das kürzlich am wenigsten genutzte Fragment

- Zeitstempel, Stapeltechniken oder Referenzlisten einsetzen
- bzw. weniger aufwändig durch Referenzbits approximieren

☞ die zu ersetzenden/verdrängenden Fragmente sind vorzugsweise Seiten

---

Zählverfahren

Auswahlkriterium ist die Häufigkeit von Seitenreferenzen

Zähler basierte Ansätze führen Buch darüber, wie häufig eine Seite innerhalb einer bestimmten Zeitspanne referenziert worden ist:

- im Seitendeskriptor ist dazu ein Referenzzähler enthalten
- der Zähler wird mit jeder Referenz zu der Seite inkrementiert
- aufwändige Implementierung, bei mäßiger Approximation von OPT

LFU ersetzt die Seite mit dem kleinsten Zählerwert

- Annahme: aktive Seiten haben große Zählerwerte und weniger aktive bzw. inaktive Seiten haben kleine Zählerwerte
- große Zählerwerte können dann aber auch jene Seiten haben, die z.B. nur in der Initialisierungsphase aktiv gewesen sind

MFU ersetzt die Seite mit dem größten Zählerwert

- Annahme: kürzlich aktive Seiten haben eher kleine Zählerwerte

---

Zeitverfahren

Auswahlkriterium ist die Zeitspanne zurückliegenden Seitenreferenzen

LRU$_{time}$ verwendet einen Zähler („logische Uhr“) in der CPU, der bei jedem Speicherzugriff erhöht und in den zugehörigen Seitendeskriptor geschrieben wird

- verdrängt die Seite mit dem kleinsten Zeitstempelwert

LRU$_{stack}$ nutzt einen Stapel eingelagerter Seiten, aus dem bei jedem Seitenzugriff die betreffende Seite herausgezogen und wieder oben drauf gelegt wird

- verdrängt die Seite am Stapelboden

LRU$_{ref}$ führt Buch über alle zurückliegenden Seitenreferenzen

- verdrängt die Seite mit dem größten Rückwärtsabstand

- entspricht OPT, wenn allerdings die Vergangenheit betrachtet wird
  - gute Approximation von OPT, bei sehr aufwändiger Implementierung

Approximation von LRU

Alterung von Seiten (engl. *page aging*) verfolgen

Referenzbit (engl. *reference bit*) im Seitendeskriptor zeigt an, ob auf die zugehörige Seite zugegriffen wurde:

0 ↦ kein Zugriff
1 ↦ Zugriff bzw. Einlagerung

- das Alter eingelagerter Seiten wird periodisch (Zeitgeber) bestimmt:
  - für jede eingelagerte Seite wird ein N-Bit Zähler (*age counter*) geführt
  - der Zähler ist als Schieberegister (engl. *shift register*) implementiert
  - nach Aufnahme eines Referenzbits in den Zähler, wird es gelöscht
- „kürzlich am wenigsten genutzte" Seiten haben den Zählerwert 0
  - d.h., sie wurden seit N Perioden nicht mehr referenziert (→ NT)

Approximation von LRU (Forts.)

Schieberegistertechnik zur Bestimmung des Lebensalters einer Seite

page aging (Forts.) mit Ablauf eines Zeitquantums (Tick), werden die Referenzbits eingelagerter Seiten des laufenden Prozesses in die Schieberegister seiner Seitendeskriptoren übernommen

- z.B. ein 8-Bit „*age counter*": $age = (age >> 1) \mid (ref << 7)$

Referenzbit	Alter (*age*, initial 0)
1	10000000
1	11000000
0	01100000
1	10110000
⋮	⋮

> Den Inhalt des 8-Bit Schieberegisters (*age*) als ganze Zahl interpretiert liefert ein Maß für die Aktivität einer Seite: mit abnehmendem Betrag (sinkender Aktivität) steigt die Ersetzungspriorität.

☞ Aufwand steigt mit der Adressraumgröße des unterbrochenen Prozesses

---

Approximation von LRU (Forts.)

Referenzierte Seiten sind vermeintlich aktive Seiten

second chance (auch: *clock policy*)

- arbeitet im Grunde nach FIFO, berücksichtigt jedoch zusätzlich noch die Referenzbits der jeweils in Betracht zu ziehenden Seiten
- periodisch (Zeitgeber, Tick) werden die Seitendeskriptoren des (unterbrochenen) laufenden Prozesses untersucht

Referenzbit	Aktion	Bedeutung
1	Referenzbit auf 0 setzen	Seite erhält zweite Chance
0	—	Seite ist Ersetzungskandidat

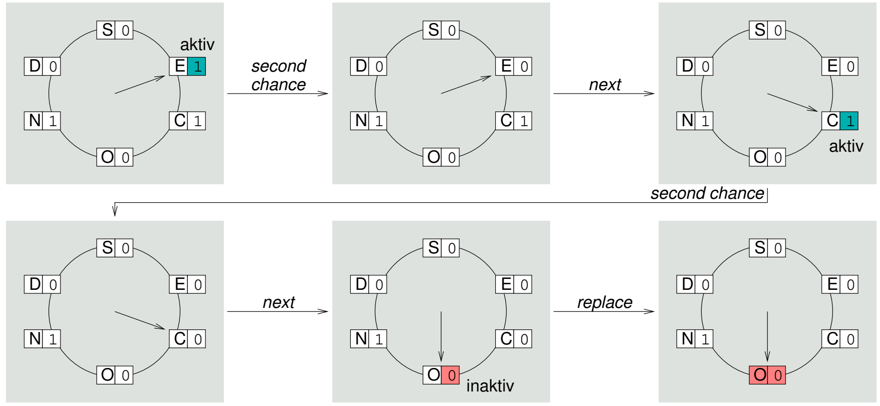
- im schlimmsten Fall erfolgt ein Rundumschlag über alle Seiten, falls die Referenzbits aller betrachteten Seiten (auf 1) gesetzt waren
  - die Strategie „entartet" dann zu FIFO

☞ unterscheidet nicht zwischen lesende und schreibende Seitenzugriffe

---

Approximation von LRU (Forts.)

Clock Policy



Approximation von LRU (Forts.)

Schreibzugriffe stärker gewichten als Lesezugriffe

enhanced second chance prüft zusätzlich, ob eine Seite schreibend oder nur lesend referenziert wurde

- Grundlage dafür ist ein Modifikationsbit (*modify/dirty bit*)
  - ist als weiteres Attribut in jedem Seitendeskriptor enthalten
  - wird bei Schreibzugriffen auf 1 gesetzt, bleibt sonst unverändert
- zusammen mit dem Referenzbit ergeben sich vier Paarungen (R, D):

	Bedeutung	Entscheidung
(0, 0)	ungenutzt	beste Wahl
(0, 1)	beschrieben	keine schlechte Wahl
(1, 0)	kürzlich gelesen	keine gute Wahl
(1, 1)	kürzlich beschrieben	schlechteste Wahl

- kann für jeden Prozess(adressraum) zwei Umläufe erwirken
  - gibt referenzierten Seiten damit eine dritte Chance (→ MacOS)

Freiseitenpuffer

Reserve freier (d.h. ungebundener) Seitenrahmen garantieren

Alternative zur Seitenersetzung, die den Vorabruf von Seiten nutzt

- Steuerung der Seitenüberlagerung über Schwellwerte („*water mark*“):

low	Seitenrahmen als frei markieren, Seiten ggf. auslagern
high	Seiten ggf. einlagern, Vorausladen

- die Auswahl greift z.B. auf Referenz-/Modifikationsbits zurück

Freiseiten gelangen in einen Zwischenspeicher (engl. *cache*)

- die Zuordnung von Seiten zu Seitenrahmen bleibt jedoch erhalten
- vor ihrer Ersetzung doch noch benutzte Seiten werden „zurückgeholt“
- sog. „*reclaiming*“ von Seiten durch Prozesse (→ Solaris, Linux)

☞ vergleichsweise effizient: die Ersetzungszeit entspricht der Ladezeit

Seitenanforderung

Verteilung von Seitenrahmen auf Prozessadressräume

Prozessen ist mindestens die kritische Masse von Seitenrahmen zur Verfügung zu stellen, um in ihrer Ausführung voranschreiten zu können

- Rechnerausstattung/-architektur geben „harte“ Begrenzungen vor:

Obergrenze	die Größe des Arbeitsspeichers
Untergrenze	definiert durch den komplexesten Maschinenbefehl

  - schlimmster Fall möglicher Seitenfehler (S. IX-19)

- innerhalb dieser Grenzen, ist die Zuordnung . . .

gleichverteilt	in Abhängigkeit von der Prozessanzahl und/oder
größenabhängig	bedingt durch den (statischen) Programmumfang

- „weiche“ Begrenzungen ergeben sich z.B. durch die gegenwärtige Systemlast und den gewünschten Grad an Mehrprogrammbetrieb

Einzugsbereiche

Wirkungskreis der Seitenüberlagerung

lokal nur Seitenrahmen des von der Seitenersetzung betroffenen Prozessadressraums nutzen (→ NT)

- die Seitenfehlerrate ist von einem Prozess selbst kontrollierbar
  - Prozesse verdrängen niemals Seiten anderer Adressräume
  - fördert ein deterministisches Laufzeitverhalten von Prozessen
- statische Zuordnung von Seitenrahmen zum Prozessadressraum

global alle verfügbaren Seitenrahmen heranziehen; dynamische Zuordnung

- Verdrängung/Ersetzung von Seitenrahmen ist unvorhersehbar und auch nicht bzw. nur schwer reproduzierbar
  - Seitenfehler erhalten *Interrupt*-Eigenschaften
- der zeitliche Ablauf einer Programmausführung ist abhängig von den in anderen Adressräumen vorgehenden Aktivitäten

- Kombination beider Ansätze ist möglich: Prozess-/Adressraumklassen
  - z.B. nur Echtzeitprozesse der lokalen Seitenersetzung unterziehen

Seitenflattern

(engl. *thrashing*)

„Dresche beziehen“ — wenn durch Seitenüberlagerung verursachte E/A die gesamten Systemaktivitäten bestimmt

- eben erst ausgelagerte Seiten werden sofort wieder eingelagert
  - Prozesse verbringen mehr Zeit beim „*paging*“ als beim Rechnen
  - das Problem ist immer in Relation zu der Zeit zu setzen, die Prozesse mit sinnvoller Arbeit verbringen
- ein mögliches Phänomen der globalen Seitenersetzung
  - Prozesse bewegen sich zu nahe am Seiten(rahmen)minimum
  - zu hoher Grad an Mehrprogrammbetrieb
  - ungünstige Ersetzungsstrategie
- verschwindet ggf. so plötzlich von allein, wie es aufgetreten ist . . .

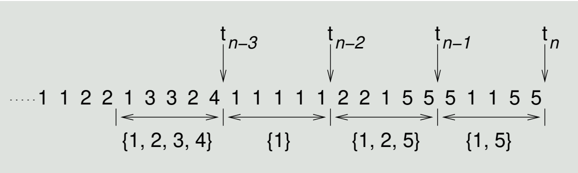
☞ Umlagerung (*Swapping*) von Adressräumen bzw. Arbeitsmengen

Seitenflattern vermeiden

Heuristik über zukünftig erwartete Seitenzugriffe erstellen

Arbeitsmenge (engl. *working set*) die Menge der Seiten, die ein Prozess lokal/das System global in naher Zukunft aktiv in Benutzung haben wird

- Berechnung der Arbeitsmenge ist nur näherungsweise möglich:
  - Ausgangspunkt ist die Seitenreferenzfolge der jüngeren Vergangenheit
  - regelmäßig wird ein Fenster (*working set window*) darauf geöffnet



- zu kleine Fenster haben wenig aktive Seiten
- zu große Fenster zeigen Überlappungen

- die Fensterbreite gibt eine „feste Anzahl von Maschinenbefehlen“
  - sie wird approximiert durch periodische Unterbrechungen
  - die Befehlsanzahl ergibt sich in etwa aus der Periodenlänge

Seitenflattern vermeiden (Forts.)

Approximation der Arbeitsmenge

Grundlage sind Referenzbit, Seitenalter und periodische Unterbrechungen

- bei jedem Tick werden die Referenzbits eingelagerter Seiten geprüft:

  1 ⇝ Referenzbit und Alter auf 0 setzen
  0 ⇝ Alter der Seite um 1 erhöhen

  - Alterswerte von Arbeitsmengenseiten sind kleiner als die Fensterbreite
- nur Seiten des laufenden Prozesses altern $\models$ lokale Arbeitsmenge
  - Problem: gemeinsam genutzte Seiten (z.B. *shared libraries*)
- Seiten aller „aktiven Adressräume“ altern $\models$ globale Arbeitsmenge
  - Problem: vergleichsweise (sehr) hoher Systemaufwand

Umlagerung bzw. Vorausladen kompletter Arbeitsmengen praktizieren:

- Mindestmenge von Seitenrahmen laufbereiter Prozesse vorhalten
- Prozesse mit unvollständigen Arbeitsmengen stoppen/suspendieren

Speicherverwaltung

Buchführung über freie/belegte Arbeitsspeicherfragmente

- Programme erhalten Arbeitsspeicher statisch/dynamisch zugeteilt
  - Zuteilungseinheiten sind Segmente oder Seitenrahmen
- Zuteilung von Arbeitsspeicher ist Aufgabe der Platzierungsstrategie
  - die Erfassung freier Fragmente hängt u.a. ab vom Adressraummodell
    - Seitenrahmen ⇝ Bitkarte, Segmente ⇝ Löcherliste
  - Zuteilungsverfahren: *best*/*worst*-fit, *buddy*, *first*/*next*-fit
  - Verschmelzung/Kompaktifizierung wirkt ext. Fragmentierung entgegen
- die Ladestrategie sorgt für die Einlagerung von Seiten (Segmenten)
  - auf Anforderung oder im Voraus
- eingelagerte Seiten unterliegen der Ersetzungsstrategie
  - Ersetzungsverfahren: OPT, FIFO, LFU, MFU, LRU (*clock*)
    - alternativer Ansatz ist der Freiseitenpuffer
  - Verdrängung arbeitet adressraumlokal oder systemglobal
  - Arbeitsmengen auseinanderreißen kann zum Seitenflattern führen