Betriebssysteme Zusammenfassung

**Multitasking und Ressource Management:**

PThreads:

*Thread erstellen:* Thread die Run-Funktion die der Thread ausführt

pthread create(thread, attribute, runner, argumente);

Threadattribute Parameter für die Run-Funktion

*Threads joinen:* Return-Wert der Run-Funktion

pthread join(thread, ergebnis);

*Mutual Exclusion (Lock):*

Mutex erstellen:

pthread\_mutex\_t \*mutexp;

mutexp = (pthread\_mutex\_t \*)malloc(sizeof(pthread\_mutex\_t));

pthread\_mutex\_init(mutexp, NULL);

Sperre anfordern:

pthread\_mutex\_lock(mutexp) oder

pthread\_mutex\_trylock(mutexp) (Polling)

Sperre freigeben:

pthread\_mutex\_unlock(mutexp)

Coffman-Bedinungen

1. **Serielle Benutzung von Ressourcen** (*Mutual Exclusion*):

z.B. synchronized-Methoden/Locks

=> Parallelen (nicht exklusiven) Zugriff auf Ressourcen erlauben

1. **Inkrementelles Ressourcen-Erlangen** (*Hold and Wait*):

Prozess wartet auf Ressource(n) und gibt dabei Zugriff auf schon erlangte Ressource(n) nie frei

=> Prozess kann Ressourcen nur „ganz oder gar nicht“ (wenn alle auf einmal frei sind) bekommen

1. **Keine Preemtion** (*No Preemption*):

Einem Prozess können schon erlangte Ressource nicht abgenommen werden

=> Prozess werden (nach einer Wartezeit, …) Ressourcen entzogen und anderem Prozess zugeordnet

1. **Wartezyklus** (*Circular Wait*):

min. zwei Prozesse haben zirkuläre Abhängigkeit (warten auf Ressource die der jeweils andere hält)

=> Reihenfolge in der Prozesse die Betriebsmittel anfordern (dürfen) wird festgelegt

Bankieralgorithmus:

**G**esamtressourcen: **G**(4, 9, 5)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **H**at Ressourcen | **B**raucht zusätzliche Ressourcen | F = G - Hi |
| Prozess 1 | **H**1(1, 1, 2) | **B**1(3, 1, 0) | F(3, 8, 5) |
| Prozess 2 | **H**2(0, 0, 4) | **B**2(1, 3, 2) | F(3, 8, 1) |
| Prozess 3 | **H**3(1, 1, 0) | **B**3(0, 4, 0) | F(1, 7, 1) |

=> Ressourcen die **f**rei sind: F(1, 7, 1)

Prozess 3 kann beenden: F(1, 7, 1) - B3(0, 4, 0) = F(1, 3, 1) => Ressourcen von Prozess 3 werden freigegeben

=> Ressourcen die frei sind: F(2, 8, 1)

Prozess 1 kann nicht beenden: F(2, 8, 1) - B1(3, 1, 0) = F(-1, 7, 1)

Prozess 2 kann nicht beenden: F(2, 8, 1) - B2(1, 3, 2) = F(1, 5, -1)

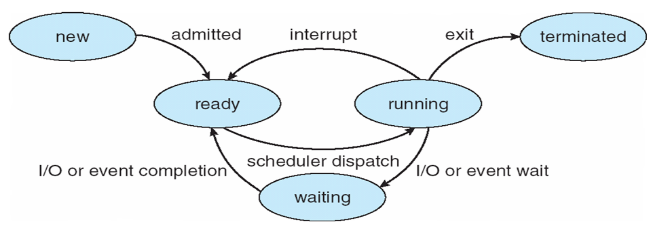
=> **Deadlock**

Spinlock:

Mechanismus zur Prozesssynchronisation: Lock auf Ressource wird angefordert -> ist diese belegt muss gewartet werden

Fork() in einem Thread: nur der AKTUELLE Thread wird geklont

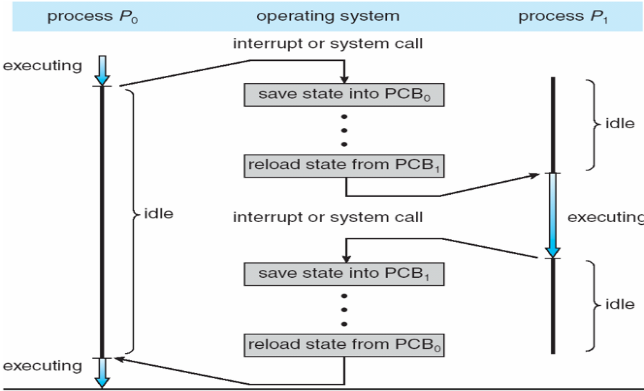
**Prozesse:**

Interrupts:

(Hardware-)Unterbrechung der laufenden Prozesse zur Behandlung einer Aufgabe

Kontextwechsel:

PCB (Process Control Block) = enthält Prozessvariablen



Process Scheduling Queues

**Job queue**: Alle Prozesse im System

**Ready queue:** Alle Prozesse die im Ready State sind und im Hauptspeicher liegen

**Device queues**: Prozesse die auf ein I/O-Gerät warten

Dämon:

Hintergrundprozess, der Ergebnis liefert

Multiprozess-Programmierung:

Erzeugen eines neuen Prozesses: pid = fork();

Process ID (PID): pid < 0 -> Fehler beim Fork

pid == 0 - > Kindprozess;   
 pid > 0 -> Elternprozess

Warten auf Kindprozesse: wait();

Terminieren eines Prozesses: exit();

Interprocess Communication (IPC):

Shared Memory (POSIX):

1. Prozess erzeugt Shared Memory Sement:

segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);

1. (anderer) Prozess der den Shared Memory nutzen will hängt sich an ihn (attach)

shared memory = (char \*) shmat(id, NULL, 0);

1. Der Prozess kann jetzt in den Shared Memory schreiben

sprintf(shared memory, "Writing to shared memory");

1. Wenn er fertig ist, kann sich der Prozess wieder ausklinken (detach)

shmdt(shared memory);

Nachrichtenübertragung:

* direkt: send(Id von Empfängerprozess, Nachricht)

recieve(Id von Senderprozess, Nachricht)

* indirekt: send(Id eines Postfachs, Nachricht)

recieve(Id eines Postfachs, Nachricht)

Pipes:

Pipe initialisieren: int toPlayers[2][2]

pipe(toPlayers[0]);

pipe(toPlayers[1]);

Write: //Elternprozess

close(toPlayers[0][0]); //Read-Pipe von Kinderproz. schließen

close(toPlayers[1][0]);

write(toPlayers[0][1], message, msglength); //auf Write-Pipe

write(toPlayers[1][1], message, msglength);

Read: //Kindprozess 1

close(toPlayers[1][1]); // Write-Pipe zu Elternproz. schließen

read(toPlayers[1][0], signal, 8); //Read auf Read-Pipe

//Kindprozess 2:

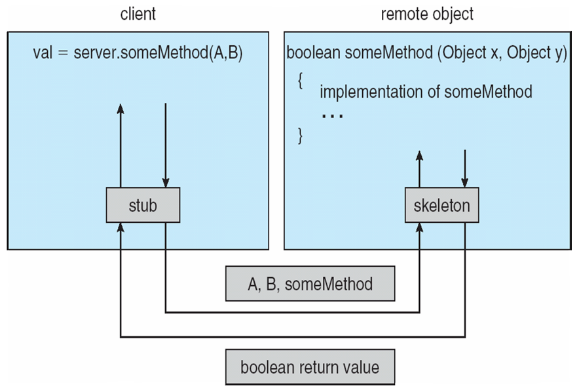
close(toPlayers[0][1]);

read(toPlayers[0][0], signal, 8);

Multithreading-Patterns:

|  |  |  |
| --- | --- | --- |
| Compute Farm | Workcrew | Pipeline |
|  |  |  |

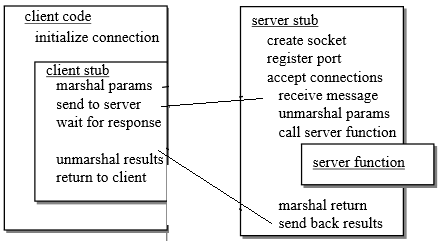
Remote Procedure Calls:

Aufruf abstrakter Methoden von Prozessen über das Netzwerk

**Stubs:** Sever/Clientseitige Proxyprozeduren für den Aufruf der tatsächlichen Prozedur

Client-Stub: findet den Server und marshallt die Parameter

Server-Stub: empfängt die Nachricht, entpackt die marshallten Parameter und führt die Prozedur aus



Sun-RPC:

Input:

* Remote Procedures definiert in einer spezifischen interface definition language

Output:

* die main-Funktion des Servers
* die Stubs des Client
* Header Dateien
* Funktionen zur Datenkonvertierung

(optional)

**Synchronisation:**

Producer-Consumer-Problem:

Es müssen Regeln für das Zugreifen auf gemeinsame Daten bestehen, wenn:

1. es sich um einen kritischen Abschnitt handelt:

Erzeugerprozess erzeugt/entfernt Element in Datenstruktur oder Verbraucherprozess entfernt Element -> es muss verhindert werden, dass anderer Erzeuger- oder Verbraucherprozess diesen Vorgang unterbricht. Sonst: Inkonsistentenz

1. zugegriffen wird, aber keine Daten vorhanden sind
2. die Datenstruktur bereits voll ist

Critical Section Problem:

1. Mutual Exclusion: Wenn Prozess Pi seinen kritischen Bereich ausführt, darf kein anderer Prozess seinen kritischen Bereich betreten
2. Progress: Wenn momentan kein Prozess den kritischen Bereich ausführt, und mehrere Prozesse den kritischen Bereich ausführen wollen, muss die Entscheidung welcher Prozess den kritischen Bereich ausführen darf in endlicher Zeit gefällt werden.
3. Bounded Waiting: Wenn ein Prozess Pi darauf wartet seinen kritischen Bereich ausführen zu dürfen muss die Menge, wie oft andere Prozesse ihre kritischen Bereiche ausführen dürfen bevor Pi seinen kritischen Bereich ausführen darf, beschränkt sein

|  |  |  |
| --- | --- | --- |
| Petterson-Algorithmus: | TestAndSet: | Swap: |
| do {  flag[i] = TRUE;  turn = j;  //busy waiting:  while (flag[j] && turn == j);  //critical section  flag[i] = FALSE;  //remainder section  } while (TRUE); | do {  //busy waiting:  while(TestAndSet(&lock));  // critical section  lock = FALSE;  // remainder section  } while(TRUE);  //ATOMAR:  bool TestAndSet(bool \*target)  {  bool rv = \*target;  \*target = TRUE;  return rv:  } | do {  key = TRUE;  //busy waiting:  while ( key == TRUE) {  Swap (&lock, &key );  }  //critical section  lock = FALSE;  //remainder section  } while (TRUE);  //ATOMAR:  void Swap (bool \*a, bool \*b)  {  boolean temp = \*a;  \*a = \*b;  \*b = temp;  } |

Semaphor:

Threadsichere Zählvariable: Enthält Zähler (Anzahl noch verfügbarer RessourcenI) und Methoden zum Reservieren sowie Freigeben von Ressourcen (=> Wenn alle Ressourcen belegt -> wait)

Polling:

fortwährendes Anfragen

Mutex (**Mut**ual **Ex**clusion):

Verfahren, das verhindert, dass nebenläufige Prozesse/Threads gemeinsam genutzte Datenstrukturen unkontrolliert ändern

Monitor:

Threadsichere Datenstruktur: Besitzt Prozeduren -> Aufruf einer „Start“-Prozedur durch einen Prozess (Prozeduren können nur von einem Prozess gleichzeitig aufgerufen werden)

-> noch Ressourcen vorhanden? Ja => Thread darf kritischen Bereich betreten

Nein => Thread wird in Monitorinterne Warteschlange gehangen (wait)

Ende kritischer Bereich -> „End“-Prozedur wird aufgerufen: Freigabe der Ressourcen des Prozesses -> wecke einen Prozess aus Warteschlange auf, der jetzt genug Ressourcen hat um Arbeiten zu können (signal)

**Scheduling:**

First Come, First Serve (FCFS):

Prozesse: P1 = 24 Burst Times, P2 = 3 Burst Times, P3 = 3 Burst Times

-> Wartezeiten: P1 = 0, P2 = 24, P3 = 27 🡪 Average Waiting Time: (0 + 24 + 27) / 3 = 17

=> Convoy-Effekt: kurze Prozesse reihen sich hinter langem Prozess ein -> lange Wartezeiten

Shortest Job First (SJF):

Je nach Länge des nächsten CPU-Bursts: Kürzester wird zu Erst bearbeitet -> schwer vorauszusagen

=> Starvation: ein Prozess wird dauerhaft vernachlässigt

-> besonders gefährlich bei geringer Priorität -> Ageing: Priorität eines Prozesses nimmt über Zeit zu

Berechnung des Next CPU Burst:



Round Robin:

Jeder Prozess bekommt kleinen Teil CPU Zeit -> danach ans Ende der Queue

=> Maximale Wartezeit: (Anzahl Prozesse - 1) \* festgelegte Zeiteinheit

Multilevel Queue:

2 Queues mit unterschiedlichen Scheduling-Verfahren:

* Foreground: Round Robin
* Background: FCFS

-> Scheduling zwischen den Queues:

1. Fixed Priority: Erst alle Prozesse auf der Foreground-Queue, dann Background-Queue
2. Time Slice: Beide Queues bekommen einen bestimmten Prozentsatz an CPU Leistung (z.B. 80% Foreground / 20% Background) die sie auf ihre Prozesse verteilen können

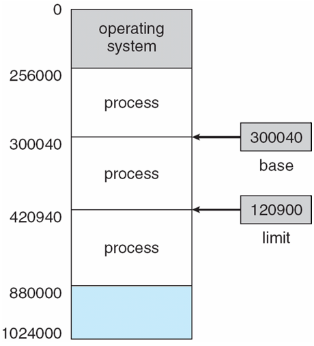
Multilevel Feedback Queue:

-> Prozesse können zwischen den Queues wandern

wird definiert durch:

* Anzahl der Queues
* Anzahl der (verschiedenen) Scheduling-Algorithmen
* Methode, die entscheidet, welcher Prozess geupgradet wird
* Methode, die entscheidet, welcher Prozess gedowngradet wird
* Methode, die entscheidet, in welcher Queue Prozess startet/eingeordnet wird

**Memory Management:**

Main Memory: wird unterteilt in 2 Teile:

1) Low memory: Das Betriebssystem selbst

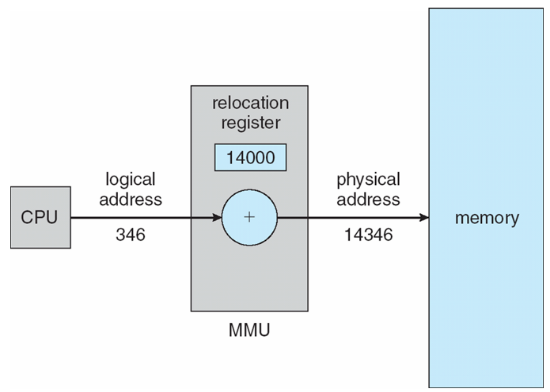
2) High memory: Benutzerprozesse

Logischer Adressraum eines Prozess = Limit Pointer - Base Pointer

Logische (virtuelle) Adresse: Wird von CPU generiert

Physikalische Adresse: Adresse die von der Speichereinheit gesehen wird

=> Werden gemappt von Memory Management Unit (MMU)



Dynamic Loading: Routine wird erst geladen wenn sie benötigt

Swapping:

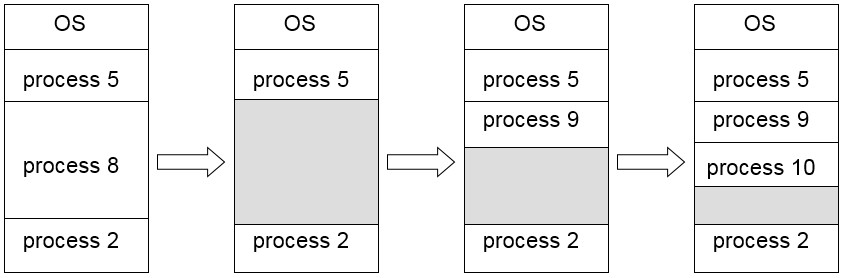
temporäres tauschen eines Prozesses in einen Backing Store -> späteres zurücktauschen um ihn weiter auszuführen

+ hohe Ebenen (schnell) + tiefere Ebenen (kostengünstig) zusammen

- Zeitbedarf durch Tauschen

Dynamische Speicherzuweisung (Dynamic Memory Allocation):

OS merkt sich: schon zugewiesenen Speicher, „Löcher“ mit freiem Speicher



Neuer Prozess fordert Speicher an:

* **First-fit**: Dem ersten freien Loch im Speicher zuweisen das groß genug ist
* **Best-fit**: Dem kleinsten Loch im Speicher, das immer noch groß genug ist zuweisen => ganze Liste muss durchsucht werden (außer Liste ist nach Größe sortiert), aber kleinster übrigbleibender Speicher
* **Worst-fit**: Dem größten Loch im Speicher zuweisen => Auch ganze Liste durchsuchen, aber der freie Speicher der übrig bleibt ist so groß wie möglich

Fragmentierung:

**externe Fragmentierung:** zugewiesener Speicher wird zu 100% von Prozess verwendet, dafür sind zugewiesene Speicherblöcke im Hauptspeicher nicht zusammenhängend

**interne Fragmentierung:** Speicherblöcke im Hauptspeicher hängen zusammen, aber einzelnen Prozessen wird ggf. mehr Speicher zugewiesen als sie benötigen

Paging:

Adressraum, der einem Prozess zugeordnet wird, muss nicht zusammenhängend sein:

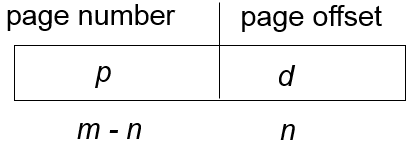
=> physikalischer Hauptspeicher wird in Blöcke mit fester Größe aufgeteilt = **Frames**

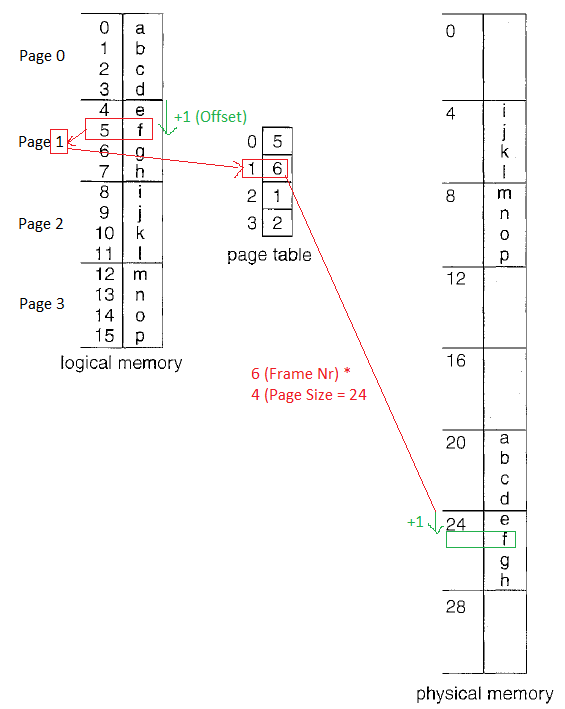
logischer Speicher wird in die gleichen Blöcke aufgeteilt = **Pages**

1. Frames merken, die schon zugewiesen wurden
2. Prozess mit n Frame großer Speicheranfrage

-> bekommt die nächsten n freien Frames zugeordnet

=> interne Fragmentierung (Frames: feste Größe => zugewiesener Speicher meistens > benötigter Speicher

Logische Adresse:



Theoretisch:

Page Tabelle = Register nah an der CPU

Praktisch:

Page Tabelle mit z.B. 1 Millionen Einträgen => Umsetzung als Register nicht praktikabel

**deshalb:** Page Tabelle selbst wird im Hauptspeicher gehalten und **page table base register** zeigt auf Speicheradresse der Page Tabelle

Vorteil: kostengünstig, da RAM vgl günstig

Nachteil: 2 Zugriffe auf Hauptspeicher

(1x Page Tabelle + 1x an physikalische Adresse)

=> langsam

Kompromiss: **translation look-aside buffer** (spezieller, schneller und kleiner Hardwarecache) wird als Lookup-Tabelle verwendet

Wenn Frame-Adresse nicht in TLB -> Zugriff auf Page-Tabelle

Schutz vor unberechtigten Speicherzugriffen:

Page-Tabelle bekommt ein valid-Flag:

v = „valid“ -> Zugehörige Page gehört zum logischen Adressraum des Prozesses

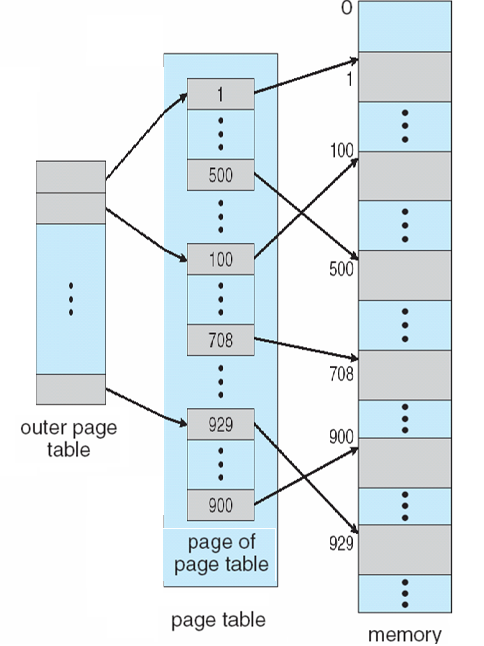
i = „invalid“ -> Zugehörige Page gehört NICHT zum logischen Adressraum des Prozesses

Effective Access Time:

(α = Hit ratio; ε = Kosten für assoziativen Lookup)

EAT = (1 + ε) α + (2 + ε)(1 – α)

= 2 + ε – α

Struktur der Page-Tabelle:

|  |  |  |
| --- | --- | --- |
| Hierarchisches Paging:  Aufteilen des Adressraums in mehrere Page-Tabellen | Hashed Page-Tabelle  Page Nr wird gehashed -> mehrere (PageNr, FrameNr)-Tupel an selber Stelle in Hashtable -> Richtiges suchen | Invertierte Page-Tabelle  Ein Eintrag pro echter Page im Speicher  Eintrag = Infos über Prozess + virtuelle Adresse der echten Page  => nur noch 1 PT für ALLE Prozesse |

Segmentierung:

Prozesse (z.B. Programm) besteht aus verschiedenen Segmenten -> unterschiedlicher Speicherverbrauch

=> Segmentation-Architektur: Logische Adresse = <Segment-Nr, Offset>

Segment-Tabelle: Mappt zweidimensionale physikalische Adressen

base (physikalische Starkadresse) + limit (Länge Segments)

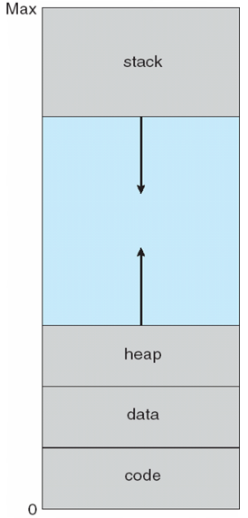
**Virtual Memory Management:**

Abgrenzung des logischen Benutzerspeichers vom physikalischen Speicher

* nicht ganzes Programm muss in Speicher geladen werden
* bestimmte Segmente können zwischen Prozessen geteilt werden

=> 1) Demand Paging

2) Demand Segmentierung



Demand Paging

Adresse nur in Speicher laden wenn sie gebraucht wird

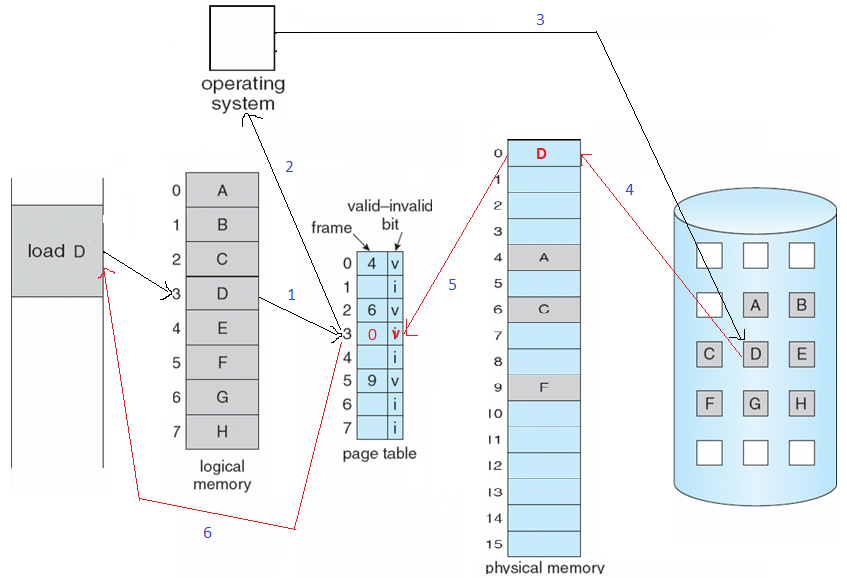
* weniger I/Os
* weniger Speicherverbrauch
* schnellere Antwortzeiten
* mehr Benutzer

=> Page wird gebraucht -> Refenzieren

1. Referenz invalide -> Abbruch
2. nicht im Speicher -> in den Speicher laden

Page Fault:

Programm greift auf Page zu, die sich im Moment nicht im Hauptspeicher befindet



Reaktion:

1. logische Adresse referenziert leeren Eintrag in der Page-Table
2. Trap wird vom OS geworfen
3. Page in einen Frame swapen
4. Page-Tabelle zurücksetzen
5. Valid-Flag auf v setzen
6. Anweisung die zum Page Fault geführt hat neustarten

Copy-on-Write:

Eltern- und Kindprozesse teilen sich Pages bis ein Prozess Daten ändert -> Kopie der Page

Page Replacement:

Wenn kein freier Frame vorhanden ist: Page in Speicher muss ersetzt werden

Ersetzungsstrategien:

Beispiel-String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

**First In, First Out (FIFO):** Die Page, die als erstes in den Speicher geladen wurde, wird als erstes ersetzt

-> Belady’s Anomaly: mehr Frames können zu mehr Page Faults führen

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 1 | 1 | 4 | 4 | 4 | 5 |  |  | 5 | 5 |  |
| 1 |  | 2 | 2 | 2 | 1 | 1 | 1 |  |  | 3 | 3 |  |
| 2 |  |  | 3 | 3 | 3 | 2 | 2 |  |  | 2 | 4 |  |

**Least Recently Used (LRU):**  Die Page, die am längsten nicht mehr benutzt wurde, wird ersetzt

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 1 | 1 | 1 |  |  | 1 |  |  | 1 | 1 | 5 |
| 1 |  | 2 | 2 | 2 |  |  | 2 |  |  | 2 | 2 | 2 |
| 2 |  |  | 3 | 3 |  |  | 5 |  |  | 5 | 4 | 4 |
| 4 |  |  |  | 4 |  |  | 4 |  |  | 3 | 3 | 3 |

Implementierung mit Timestamp:

Counter der bei jedem Page Fault erhöht wird -> angehängt an Page-Tabelle

->Victim-Suche: einmal durch komplette Liste gehen und Page mit kleinstem Timestamp finden

Implementierung mit “Stack”:

Doppelt verkettete Liste: Page wird in leeren Frame geladen -> ganz vorne einordnen

schon geladene Page wird aufgerufen -> Aus Liste entfernen und vorne einordnen

Page Fault -> Page ganz hinten in Liste entfernen und neue Page vorne einordnen

=> jedes Update ein wenig teurer, aber keine Suche durch ganze mehr nötig

Referenz-Bit:

Jede Seite bekommt ein zugehöriges Bit (Anfangs = 0) -> Page referenziert: bit = 1

Ersetze die Page, deren Bit = 0 (falls so eine Page existiert)

Second Chance:

Referenz-Bit (Anfangs = 1) + Timestamp -> Wenn Page ersetzt werden müsste -> Referenz-Bit = 0 und Page bleibt im Speicher

Zählende Algorithmen:

Jeder Eintrag in der Page Tabelle bekommt eine Zählvariable -> wird bei Zugriff auf Page erhöht

**Least Frequently Used (LFU):** Ersetzt Page die am seltensten verwendet wurde

**Most Frequently Used (MFU):** Ersetzt Page die am häufigsten verwendet wurde

Thrashing:

ein Prozess nicht genug Pages -> hohe Page Fault-Rate -> CPU nicht ausgelastet -> CPU startet neue Prozesse

-> Prozess muss ständig Pages austauschen = Thrashing

Zuweisung von Kernelspeicher:

wird anders behandelt als Anfragen von Userprozessen und kommt meist aus dem free memory pool

**Buddy-System:** 2n große Segmente => baumartige Struktur, fortlaufend reservierter Speicher, fragmentiert,

freigegeben Segmente können ggf. verschmolzen werden

**Slab Allocator:** Kernel benötigte Datenstrukturen auf Vorrat (in Caches) -> Ein Slab bekommt mehrere Caches

=> keine Fragmentierung, schnelle Beantwortung von Speicheranfragen

Memory Mapped Files:

Disk I/O wie Speicherzugriff behandeln -> pagegroßer Fileblock wird in Speicher geladen -> weiter I/Os auf Page

=> bessere Performance, kann von mehreren Prozessen gemapped werden

**Dateimanagement:**

Dateistruktur:

Zeilen, fixe Länge (fscanf -> Speicherverschwendung!), variable Länge (CSV -> linearer Aufwand)

=> „flache“ Datei: eine lange Zeile

formatiertes Dokument, \*.obj (relocatable load file)

=> Komplexe Datei

Links in UNIX:

**Hard Link:** Ein Link direkt auf den Index Node einer Datei im Dateisystem -> Index node wird erst gelöscht, wenn alle Hard Links gelöscht wurden => nur im eigenen Dateisystem gültig

**Symbolischer/Soft Link:** Ein Link auf einen anderen Namen im Dateisystem => Über Dateisysteme hinweg gültig

**Bash-Befehle:**

cat: Zusammenfügen

cat text1.txt text2.txt > gesamt.txt

cat text1.txt => Ausgabe von text1.txt auf Konsole

cp: Kopieren

cp datei.txt /Dokument/...

ln: Link erstellen (-s = Softlink)

ln von nach

ls: Dateien in aktuellem Verzeichnis listen(-u = versteckte Dateien anzeigen; -l = langform; -R = Rekursiv)

Angezeigt wird: Dateityp/Rechte, Anzahl Hardlinks, Eigentümer, Dateigröße, Änderungszeit, Name

mkdir: Verzeichnis erstellen

mv: Verschieben einer Datei (im eigenen Verzeichnis = Umbenennen)

mv Datei.txt /Dokument/…

rm: Entfernen

rmdir: Löschen eines leeren Verzeichnisses

sudo: Rootrechte für einen Befehl

touch: Erstellen einer leeren Datei/Änderungszeitpunkt einer existierenden Datei ändern

Pipe: weiterleiten der Ausgabe eines Befehls

ps ax | sort -nr

cat text.txt > ges.txt (Überschreiben)

cat text.txt >> ges.txt (Anhängen)

find: Finden einer Datei im aktuellen Verzeichnis

find /Dokument/ -name “\*test.txt“

grep: Durchsuchen einer Datei

grep -e “[Hh]allo“ /Dokument/Datei.txt

mount: Einbinden eines Datenträgers ins Dateisystem

sed: Ersetzen: sed -n ‘ s/Test/TEST‘ < text.txt

Anhängen: sed -n ‘ i/…‘ < text.txt

Ändern: sed -n ‘ c/…‘ < text.txt

awk: Aufbau: BEGIN {Action}

pattern{action}, …

END{Action}

Aufruf: -awk ‘ Programm‘ Datei

Beispiel:

ls | awk ‘Begin{print „Liste.txt Files:“} /\.txt$/{print} End{print „End“}‘