<u>Prozess</u>: ein laufendes Programm, was nur im persistenten Speicher als Liste von Befehlen liegt und Betriebssystem überführt es in in den laufenden Zustand

-Gebrauch: - sollen viele Programme gleichzeitig laufen können und
- Anzahl der Prozesse soll unabhängig von Anzahl physischer Prozessoren sein
- Betriebssystem erzeugt die Illusion vieler existierender virtueller Prozessoren

-Darstellung: - modelliert verschiedene Zustände der Maschine /- Was laufendes Programm les en und verändern kann

- Welche Teile des Rechners wichtig für Ausführung sind (Register, Arbeitsspeicher, persistenter Speicher, etc.)

-ProzessZustände: - Running: Prozess läuft derzeit auf der CPU. Befehle werden ausgeführt.

Ready: Prozess kann ausgeführt werden, wird aber derzeit nicht ausgeführt.
Aus irgendeinem Grund hat das Betriebssystem den Prozess pausiert.

- Blocked: Prozess hat Operation durchgeführt, die ihn in blockierten Zustand gebracht hat bis bestimmtes Ereignis eintritt ->typisches Beispiel ist die Ein- und Ausgabe auf persistenten Speicher. - Scheduled: Überführung eines Prozesses von Ready nach Running

40

80

[B,C arrive] A | B C

- Descheduled: Überführung eines Prozesses von Running nach Ready

 Nachdem Ereignis eingetreten ist, das den Blocked-Zustand löst, geht der Prozess immer erst in den Ready-Zustand
imesharing: - Prozess wird eine Zeit lang ausgeführt und dann gestoppt, damit nächster Prozess Rechenzeit bekommt (usw)
 -> Rechenzeit auf physischem Prozessor wird auf Prozesse verteilt, welche nacheinander laufen für gewisse Zeit - erlaubt beliebig viele gleichzeitig laufende Prozesse (virtuell) --> erzeugt Eidnruck der Parallelität ( noch nicht da)

- <u>Nachteil</u>: je mehr Prozesse, desto langsamer <u>Vom Betriebsystem braucht man **Mechanismen**:</u>

Zur Implementierung von CPU-Virtualisieriung braucht es low-level Machinerie (wird durch Mechanismen erreicht) und smarte Strategien auf hohem Level. -> Mechanismen: beinhalten grundlegende Methoden und Protokolle, die notwendige

Funktionalität liefern

ContextSwitch: notwedig.Mechanismus für Timesharing -> siehe weiter unten (direktes Ausführen)

ProgrammCounter(PC): Befehlszähler (auch Instruction Pointer, IP) zeigt auf den nächsten auszuführenden Befehl

StackPointer: dient zur Organisation von Informationen des Stacks, zB.Funktionsparameter, lokale Variablen, Rückgabeadressen

Schnitstellen: - Create -> Prozess erstellen, z.B. nach Kommando in Konsole oder Klick auf Icon
- Destroy -> erzwungenes Beenden, z.B. wenn Prozess nicht selbständig endet
- Wait -> Prozess unterbrechen / anhalten, kann fortgesetzt werden

Misc. Control -> weitere Kontrollmechanismen, z.B. zeitlich befristet zurückstellen

- Status -> Rückgabe von Informationen, z.B. Laufzeit oder aktueller Zustand

Datenstrukturen: - Betriebssystem hat verschiedene Datenstrukturen, um relevante Informationen über ablaufende Prozesse zu speichern und In Prozessliste werden alle Prozesse und deren Zustände vorgehalten

## xv6 Prozess-Struktur aus OSTEP:

```
information xv6 tracks about each process including its register context and state
                                           // Start of process memory
char *mem;
                                              Size of process memory
Bottom of kernel stack for this process
Process state
Process ID
uint sz;
char *kstack;
 enum proc_state state;
int pid;
                                               Parent process
struct proc *parent:
void *chan;
int killed;
struct file *ofile[NOFILE];
struct inode *cwd;
                                               If !zero, sleeping on chan
If !zero, has been killed
Open files
                                               Current directory
struct context context:
                                               Switch here to run process
                                           // Trap frame for the current interrupt
struct trapframe *tf;
```

zesssteuerung -> Systemaufrufe als Schnittstelle zwischen Betriebssystem und Programmen

Befehle: - fork() -> neuer Prozess wird erstellt, welcher fast vollständige Kopie des aktuellen Prozesses darstellt

-> wird dann in child- und parent-Prozesse unterschieden
- wait() -> parent Prozess wartet auf erfolgreich beenden des child Prozesses

-> Prozess verschiebt die Ausführung seiner restlichen Befehle bis zur Rückgabe aus child

- exec() -> anderer Prozess (der keine inhatliche Kopie des Originalprozesses ist) wird an bestimmter Stelle ausgeführt

close() -> schließt file Deskriptor und die die standardmäßig offen sind open() -> öffnet Datei gemäß Parametern und nimmt ersten verfügbaren file descriptor

kill() -> sendet Prozessoren verschiedene Signale (Beenden, Pausieren, ...)

--> Signale stellen eine Infrastruktur dar, um durch externe Ereignisse Einfluss auf Prozesse nehmen zu können.

Direktes Ausführen: - Programm läuft direkt auf dem Prozessor

- dazu wird Prozessoreitrag in Prozessorliste erstellt, Speicher zugelassen und Programmcode geladen

--> dann wird an Stelle gesprungen, um Ausführung zu starten
- D.A. ist Ausgangspunkt um Kontrolle über CPU zu behalten, wärend Prozess läuft
- Programme laufen direkt auf CPU (ist schnell und effizient) + Ein-/Ausgaben bzw. Zugriff auf (mehr)Speicher

Problem: sind Operationen, die nicht uneingeschränkz benutzt werden dürfen. Jeder Prozess darf nich auf alles zugreifen.

Modi: -User Mode: stellt begrenzten Umfang an Operat. bereit + Ein-/Ausgaben führen zu Exeptions. BetrSyst beendet Prozesse -Kernel Mode: alle Operationen erlaubt (Bsp. Ein-/Ausgabe) -> Komponenten des Betriebssystem laufen in diesem Modus Fexible Rechtevergabe durch Verschlussmechanismus: Um Systemaufruf auszuführen, muss Programm speziellen Trap-Befehl Ausführen -> dieser bewirkt Übergang von User Mode in Kernel Mode -> System führt dort gewünschte Operation aus

(wenn erlaubt) und springt dann zurück zum Programm im User Mode

> Beim Wechsel muss durch Hardware sichergestellt werden, dass irgendwann in den Ursprungsztand zurückgespruwerden kann (Speicherinhalte, Registerzustände, etc.). — Solche Informationen landen auf dem Kernel Stack
Trap-Tabelle: -wird angelegt beim Booten des Systems + beinhalten Infos (als Code) für Hardware darüber, was bei versch.

besonderen Ereignissen passieren soll -> Zu diesen Ereignissen gehören neben Interrupts auch Systemaufrufe

Eingeschränkte direkte Ausführen (Trap-Table-Anlage): -hat 2 Phasen -> BootZeit und Laufzeit - Betriebssystem muss die Hardware über den Ort der Trap Handler informieren

Hardware merkt sich diese Informationen bis zum nächsten Neustart des Systems

Systemaufrufe werden üblicherweiße über eine eindeutige Nummer identifziert

- Programm-Code platziert benötigte Nummer im Register u. Betriebssystem prüft deren Gültigkeit (Sicherheitsmechanis Prozesswechsel: <u>Situation</u>: Programme laufen direkt auf CPU, dann läuft Betriebssystem per Definition gerade nicht <u>Problem</u>: gibt keine Möglichkeit eine Aktion auszuführen, wenn gerade keine Rechenzeit zur Verfügung steht

Ansätze: Kooperativ(Abwarten): - Betriebssystem vertraut darauf, dass laufende Prozesse zu wechseln (2 Ansätze)

Ansätze: Kooperativ(Abwarten): - Betriebssystem vertraut darauf, dass laufende Prozesse sich vernünftigt verhalten.

-wird angenommen, dass lange laufende Prozesse in regelmäßigen Abstände die Kontrolle zurückgeben

-inPraxis: viele Prozesse haben Systemaufrufe (I/O,...) - 2 gibt yield()-Aufruf, um Kontrolle freiwillig zurückzugeben

-Programme geben Kontrolle zurück, wenn sie etwas illegales tun (Trap zum Betriebssystem)

Imperativ(Kontrollübernahme): -Betriebssystem braucht Funktionen,um Prozesse unterbrechen zu können.
-typisches Bip ist zeitbasierter Unterbrechungsmechanismus, der Prozesse nach gewisser Zeit anhält

Problem: Programm bleibt durch Fehler in in Unendlicher Schleife hängen -> Lsg: Kontrollmechansismen (Bsp.: Zeit)

-> Zeit: - Zeitgeber wird so programmiert, dass alle paar Millisekunden ein Interrupt ausgelöst wird

Prozesse werden angehalten und der Interrupt Handler des Betriebssystem läuft
 spezieller Code für Hardware zur Boot-Zeit konfiguriert

-> Betriebssystem erhält Kontrolle nach gewisser Zeit zurück, wenn Timer gestartet wurde

--> aber es gibt previligierte Operationen die Timer deaktivieren können

Anmerk: -Hardware hat bei Interrupt Verantwortung dafür, Zustand so zu speichern, dass Prozess später fortgesetzt werden kann -Wenn Scheduler entscheidet, den Prozess zu wechseln, wird ein <u>Context Switch</u> durchgeführt 
Context-Switch: Betriebssystem sichert zunächst Registereinträge des aktuell laufenden Prozesses in Datenstruktur (Kernel Stack

->dann lädt es die entsprechenden Einträge für nächsten Prozess aus dieser Datenstruktur

-> ann iaor es die entsprecienden Eintrage Tur nachsten Prozess aus dieser Datenstruktur
-> so stellt Betriebssystem sicher, dass nach Rückgabe aus der Trap, der neue Prozess ausgeführt wird

SwitchSchritte: 1-Inhalte des allgemeinen Registers sichem 2-Program Counter (Befehlszähler) sichern

3-Zeiger für Kernel Stack sichern 4-spezielle Switch()-Methode aufrufen

Register: User Register: werden von Hardware mittels Kernel Stack gesichert -> Zeitpunkt Sichern: Timer Interrupt (Timer stopt) Kernel Register: werden von Software in Struktur für Prozesse gesichert -> Zeitpunkt: Betriebsys. entscheidet Wechsel

Strategien der Ablaufplanung (Scheduling Policies) -> Zeitmanagment von Prozessen

Strategien der Ablaufplanung (Scheduling Policies) - Zeitmanagment von Prozessen
Annahmen und Vereinfachung: müssen den Aufwand von Prozessen quantifizieren können
Annahmen über Prozessee (Jobs): 1. Jeder Job läuft für genau die gleiche Zeitspanne. 2. Alle Jobs kommen zur gleichen Zeit an.
3. Wenn einmal gestartet, dann läuft Job bis zum Ende
4. Jeder Job nutzt nur Prozessorzeit (kein I/O). 5. Laufzeit jedes Jobs ist bekannt.
Gütermaß für Sheduling: Turnaround Time: T\_turnaround = T\_completion - T\_arrival (TurnaroundT ist Metrik für Performance)
- Bemerk: -derzeitige Annahme: Alle Jobs kommen zur gleichen Zeit an. -> T\_arrival = 0
- Andere Metriken bemessen dagegen Fairness (Bsp. Jain's Fairness Index)
- Performance und Fairness sind entgegengesetzte Ziele

FIFO (First in, First Out): -einfa Prinzip: Abarbeitung in Rehienfolge der Ankunft -> einfach zu implementieren (als Warteschlange)
Bsp1: Jobs A, B, C; Tarrival = 0 (nahezu); 10s je Job -> alle 3 Jobs sind gleich lang (10s) (Bild nicht dabei)
-> Durchschnitliche TurnaroundTime: (/(10 + 20 + 30) - 0) / 3 = 20.

Bsp2 -> siehe oben rechte seite

Konvoi-Effekt: kurze/schnelle Jobs müssen auf einen langen/langsamen Prozess warten

Beispiel 2: Jobs A, B, C; T\_arrival = 0 (nahezu); Jobs unterschiedlich lang (ohne Annahme 1) --> Ø Turnaround Time: ((100 + 110 + 120) - 0)/3 = 110 .

A B C ->Bessere Strategie: SJF(Shortest Job First): Abarbeitung in Reihenfolge der Länge -> implemt als Vorrangwarteschladen -- geändertes Bsp2: geänderte Reihenfolge (B,C,A): Berechnung: --> Ø Turnaround Time: ((100 + 20 + 120) - 0)/3 = 50 . > nach Ahnnahme 2 ist Strategie beweisbar optimal

-BpS2: T\_arrival(A) = 0, T\_arrival(B) = T\_arrival(C) = 10 (ohne Annahme 2); Längen wie Bsp2 → Reihenfolge(A,B,C)

100 120 --> Ø Turnaround Time: ((100 - 0) + (110 - 10) + (120 - 10)) / 3 = 103,33. --> Annahme 3 wird verworfen in dem Fall Geändertes Bsp3 durch STCF: selbe geg Daten --> Ø Turnaround Time: ((120 - 0) + (20 - 10) + (30 - 10)) / 3 = 50.

->hier muss Job nicht erst beenden, also: A arrived bei 0, dann B,C, dann A weiter -> Optimal, wenn Jobs nicht zur gleichen Zeit ankommen -STCF(Shortest-Time-to-Completion First): -auch: Preemptive Shortest Job First (PSJF) ->hat Werkzeuge(TimerInterrupt, ContextSwitch), um Prozesse zu pausieren/wechseln ->bei Ankunft neuer Jobs: Entscheidung für Job, der am schnellst beendet werden kann 80 100 120

> В С

Shortest lob First

20 25 Round Robin Zeitscheibe: 1s

20

 $\varnothing T_{\text{response}}$ :  $\frac{0+5+10}{2} = 5$ 

 $\varnothing T_{\text{response}}$ :  $\frac{0+1+2}{3} = \underline{1}$ 

Ansprechverhalten: -bei stapelverarbeitenden Systemen war STCF valide Strategie.

Bei Maschinen mit paralleler Benutzung nicht valide.

-> man Braucht Response Time: T\_response = T\_firstrun - T\_rrival

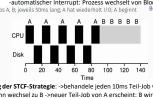
>T\_firstrun bezeichn Zeitpunkt, wann Job erste mal zum Ausführen vorgesehen ist
-Annahme: Job erzeugt sofort ohne Verzögerung eine Rückmeldung (Best Case) Round Robin(RR): -Jobs laufen für eine gewisse Zeitspanne, statt immer bis

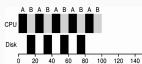
zum Ende (Time Slicing, Zeitscheibe) erzwungener Wechsel des Prozesses am Ende der Zeitspanne

-Prinzip wiederholt, bis alle Prozesse beendet ->Rundlaufverfahren genannt wichtig: Zeitscheibe bildet ein Vielfaches der Zeit des Timer Interrupts ab

- Länge der Zeitscheiben ist entscheidend für die Response Time (je kürzer desto besser) -> if aber zu kurz, dann anfaller Kosten für KontextSwitch zu groß -> man muss optimale Länge festlegen

Ein-/Ausgabe: -bei I/O: erste Entscheid, da Prozess Rechenzeit nicht benötigt /-zweite Entscheidung, wenn I/O beendet wurde -automatischer Interrupt: Prozess wechselt von Blocked nach Readv beispiel 5b: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt Beispiel 5b: Jobs A, B; jeweils 50ms lang: A hat wiederholt I/O; A beginnt Beispiel 5a; Jobs A, B; jeweils 50ms lang; A hat wiederholt I/O; A beginn





t**zung der STCF-Strategie**: ->behandele jeden 10ms Teil-Job von A als eigenen Job ->Start mit ersten Teil-Job von A (weil kürzer) bis I/O, dann wechsel zu B ->neuer Teil-Job von A erscheint: B wird unterbroc (usw) ->effizice (überlappende) Nutzung d Ressourcen 4. Annahme IO: Man kann Prozesse ausführen, während der andere Prozess Blocked ist

5. Annahme: Wie SJF oder STCF ohne Wissen über Laufzeit? (u. TA-Time; Response-T optimieren) -> MLFQ

MLFQ(Multi-Level Feedback Queue): - bekannteste Strategie für Ablaufplanung -> kann verbessert werden (Laufzeitverteilung):

Festlegung: MLFQ hat eine festgelegte Anzahl an Queues mit vers. Prioritäten

->Zu jederzeit sind Jobs, die fertig sind, einer der Queues zugeordnet

Regeln: 1-Job mit höherer Priorität hat Vorrags III, einer Ger Queues zügechner Berner Round-Robin Prinzip | 3-Neuer Job höchste Priorität im System (oberste Queue) | 4a-Befindet sich Job für ganzen Zeitabschnitt im Zustand Running, dann Priorität reduziert (eine Warteschlange nach unten) | 4b-Gibt Job den exklusiven Zugriff auf Prozessor vor Ablauf des Zeitabschnitts

ruturück, dann bleibt Priorität unverändert | 5-Nach festgelegtem Zeltraum bekommen alle Jobs höchste Priorität

Priotitätsverhalten: NLFQ variert Prioritäten ständig nach beob. Verhalten | Jobs, die oft auf Eingaben warten, bekommen hö

Priorität Jobs, die lange und kontinuierlich CPU ausdasten, bekommen niedrigere Priorität

Probleme von 4 auf 5: - Zu viele interaktiven Jobs können verfügbare CPU-Zeit allein aufbrauchen, andere Jobs vernachlässigt.

- User können Programme so schreiben, dass sie künstlich die Priorität hoch halten und Scheduler ausspielen Oser Kollnen Frogramm kann über Zich kinweg sein Verhalten werändern (nach langer Berechnung folgt interaktive Phase)
 Prozesse können nicht verhungern, da Round Robin fair verteilt
 Jobs, die zu Interaktivität werbelen, werden wie vorgesehen behandelt
 Manipulation des Systems: -Problem des Ausspielens entsteht durch 4a und 4b

-Priorität wird beibehalten, wenn Job den Zugriff auf Prozessor zurückgibt Kontoführung für Jobs: -Scheduler speichert Information, wieviel Zeit ein Job je Prioritätslevel genutzt hat - jeder Job bekommt ein Anteil der Zeit je Level zugewiesen

neue Regel4: Sobald Job zugewiesenen Zeitanteil für ein Level aufgebraucht hat, wird Priorität reduziert (eine Warteschlange nach unten) -> geschieht unabhängig davon, wie oft Job bisher Zugriff auf den Prozessor selbständig zurückgegeben hat Weitere Probleme: -unterschiedlich lange Zeitabschnitte je Prioritätslevel (je höher desto kürzer) /-Paretre über Tabellen für Admins veränderbar (Solaris) /-mathematische Funktionen für Anpassung der Prioritäten (FreeBSD) /-höchste Prioritätslevels nur für Betriebssystem (OS); Benutzerempfehlungen (nice Kommando);.

Bsp4: Jobs A, B; versucht Scheduler auszuspielen; ohne Schutzmechanismus (Regel4): - ohne Schutz bekommt Job A kaum Rechenzeit, während B existiert -> durch Regel 4 wandert Job B in der Priorität nach unten, um Fairness zu gewährleisten GerechteVerteilung: jedem Job soll garantierter Anteil der Prozessor-Zeit zugestanden werden (FairShare, Proportional-Share Schedu Lottery Scheduler (Bsp): - Tickets repräsentieren den Anteil der Ressource, den ein Job oder Benutzer erhalten kann

- je mehr Tickets ein Job hat, desto größer ist die Wahrscheinlichkeit der Zuweisung
- regelmäßige Lotto-Ziehungen bestimmen nächsten laufenden Job

Mechanismen: Verschiedene Währungen: - erlauben dem Benutzer zugewiesene Tickets nach eigenen Jobs zu verteilen - System rechnet die Währungen eines jeden Benutzers in den richtigen globalen Wert um

Transfer von Tickets: - Prozess kann seine Tickets temporär an andere Prozesse übergeben (Client-Seiver-Systemen)
Ticket-Inflation: - Sinnvoll in nicht-kompetitiven Systemen: Wenn Prozess mehr Prozessor-Zeit benötigt, dann erhöh seine Ticketanzahl entsprechend selbständig. (keine Kommunikation mit adneren Prozessen)

Mehrere Prozessoren (Mehrkernsystem): durch Mehrkernprozessoren weiterhin starker Anstieg in Rechenkapazitäten
-> Parallelität der Programme und Scheduling muss angepasst werden
Geäderte Architektur: - in moderenen Prozessoren, haben Einkemprozessoren Cache-Hierachie, in der genutzte Daten als Kopie
zwischengelagert werden (schnellerer Zugriff darauf als auf Areitsspeicher)

Cache-Speicherung: -Zeitliche Lokalität: die selben Daten werden wiederholt genutzt (z.B. in Schleife)

-Ortliche Lokalität: benachbarte Daten werden wahrscheinlich auch genutzt (z.B. aus Array)
-->Prozessoren können auch geteilten Arbeitsspeicher haben , dann gibt es Cache-Kohärenz und Cache-Affinität

Cache-Kohärenz: -mehrere Caches der selben Ressource (Wert, Datei, Gerät) -> Abhilfe: Speicherzugriffe beobachten (Bus Snooping)

Cache-Affinität: -Laufende Prozesse erzeugen Daten und Zustände, die gecached werden -> nach Unterbrechungen sollte der Prozess auf gleichen Prozessor weiterlaufen -> auf anderen Prozessor müssen Zustände geladen werden

Single Queue Multiprocessor Scheduling (SQMS): geplanten Prozesse werden in einer Wartschlange organisiert
->Probleme: -Skalierbarkeit (Zugriffsperre auf Queue, damit mehere Prozesse laufen, Bremmst System, je mehr Prozesse) -Cache-Affinität (Prozessor wählt next Job in Queue, so wandern Prozesse tändig über alle Prozessoren) ->Lsg ->

Multi-Queue Multiprocessor Scheduling (MQMS): mehrere Queues -> Jede kann bestimmtes Scheduling-Verfahren umsetzen -> Neuankommende Jobs werden auf Basis einer Heuristik genau einer Warteschlange zugeordnet.

Beispiel 1: Jobs A bis D; Zwei Prozessoren mit eigenen Queues -- gegeben: (Q0 -> A -> B und Q1 -> C -> D)

CPU 0 A A C C A A C C C

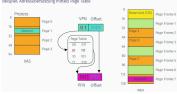


-Cache-Affinität sichergestellt: Jobs verbleiben auf dem selben Prozessor

<u>Beispiel 2</u>: wie zuvor; Job C erfolgreich beendet --> Nun bekommt A doppelt so viel Prozessorzeit, wie die beiden anderen Prozesse B und D. Wenn dann auch Prozess A fertig ist, dann hat CPU 0 gar nichts mehr zu tun! –> Load Imbalance
Alles im Gleichgewicht: Um Effekt ungleicher Lastverteilung zu verringern, müssen wir Jobs doch zw Prozessoren wechseln I

-> mgl: Jobs müssen kontinuierlich zwischen Prozessoren wandern, um diese voll auszulasten -> Lsg --> Work Stealing: Warteschlange mit wenig Auslastung schaut bei anderen Warteschlangen nach, wie voll diese sind
-> Sind andere Warteschlangen mehr ausgelastet, dann werden von dort Jobs gestohlen
ABER: wenn zu oft -> Overhead wird zu groß und Skalierbarkeit leidet; wenn zu selten -> Load Imbalance tritt ein

Wann und unter welchen Umständen Zustandswechsel: -beim Starten eines Prozesses wird in den Zustand "Ready" versetzt.
->Entscheidet das Betriebssystem, dass der Prozess ausgeführt werden soll, wird der Prozess in den Status "Running" versetzt und wird abgearbeitet |-zwischen "Ready" und "Running" befindet sich der Prozess im Zustand "Scheduled" -> wenn Prozess eine Operation ausführt, die ihn in blockierten Zustand versetzt (z.B. IO-Operationen), wird Prozess in Zustand "Blocked" versetzt -> ist IO-Operation fertig, wird Prozess erst wieder in den Zustand "Ready" versetzt und dann in den Zustand "Running", wenn Betriebssystem entscheidet, dass Prozess ausgeführt wird entscheidet Betriebssystem, dass anderer Prozess ausgeführt werden soll, wird ein laufender Prozess wieder in Zustand "Ready" versetzt ->Dabei hat er Zustand "Descheduled"



1.Berechnen der Anzahl der Seiten im virtuellen Adressraum: Geg: -gesamter VAS = 2 GiB; - Pagesize = 4 KiB -Anzahl der Seiten = Gesamter VAS / Pagesize

= (2 GiB) / (4 KiB) = (2 \* 1024 \* 1024 KiB) / (4 KiB) = (2 \* 1024 \* 1024 \* 1024) / (4 \* 1024) = 1024 \* 256) = 524288 Seiten

2.Berechnen der Größe der Page-Tabell Geg: -Jeder Eintrag in der Page-Tabelle ist 4 Byte groß.

-Größe Page-Tabelle = Anzahl Seiten \* Größe eines Eintrags = 524288 Seiten \* 4 Byte/Eintrag = 2097152 Byte = **2 MiB**