182.711 Betriebssysteme

Schönen Dank geht hier bei an ***–symm3try—***für seine Zusammenfassung welche im VoWi hinterlegt ist. Meine Ausarbeitung bezieht sich im Allgemeinen auf seine Zusammenfassung in einer verkürzten Fassung.

Inhalt

1. Deadlock Bedingungen 2
2. Erklärung und Unterschiede Prozess vs. Thread 3
3. Inodes 5
4. Thrashing inkl. Lösungsansatz 6
5. Mutal Exclusion und Synchronisation 6
6. Bell and LaPadula's Model 6
7. 4 Arten der Datei Implementierung 7
8. Disk Scheduling (wozu, Verfahren erklären) 8
9. Security Grundprinzipien bzw. Ziele 9
10. PCB 11
11. ULT, KLT 12
12. Memory Management 13
13. Praxisbeispiele: 15
13.1. RMS Scheduling inkl. Schedulability Test 15
13.2. Round Robin Scheduling 15
13.3. Producer/Consumer mit Semaphoren 16
13.4. Semaphoren in Pseudocode implementieren, simples Beispiel zum synchronisieren 16
13.5. Reader/Writer mit Semaphoren 17

1. Deadlock Bedingungen

- Der Zugriff auf die Betriebsmittel ist exklusiv (**Mutal Exclusion**)
- Die Prozesse fordern Betriebsmittel an, behalten aber zugleich den Zugriff auf andere (**Hold and Wait**)
- Die Betriebsmittel werden ausschließlich durch die Prozesse freigegeben (Da Ressourcenzugriff eines Prozesses nicht unterbrochen werden kann. (**No Preemption**)
- Mindestens zwei Prozesse besitzen bezüglich der Betriebsmittel eine zirkuläre Abhängigkeit. (**Circular Wait**). D.h. eine geschlossene Kette von Prozessen, von denen jeder Prozess mindestens eine Ressource, die von einem anderen Prozess benötigt wird, hält.

Die Behandlung von Deadlocks kann auf **3 Arten** geschehen. Man spricht von Verhinderung, wenn mindestens einer der vier Bedingungen für eine Verklemmung nicht erfüllt wird.

1) **Deadlock Prevention**: Man verhindert das Eintreten einer der vier Deadlock-Bedingungen
   a. ***Indirect Deadlock Prevention***
      Bei der indirekten Deadlock Prevention versucht man eine der ersten drei Bedingungen (Mutal Exclusion, Hold & Wait, No Preemption) zu verhindern
   b. ***Direct Deadlock Prevention***
      Bei der direkten Deadlock Prevention verucht man die vierte Deadlockbedingung (Circular Wait) zu verhindern. Algorithmen die Circular Waits verhindern verbieten Interrupts in Kritischen Abschnitten und führen eine strikte lineare Ordnung für Ressourcenarten ein.

2) **Deadlock Avoidance**: Ressourcenreservierungen, die zu Deadlock führen könnten, werden nicht gewährt.
   a. ***Process Initiation Denial***
      Der Prozess wird nicht gestartet, wenn seine Anforderungen zu einem Deadlock führen könnten
   b. ***Process Alloctaion Denial***
      Eine Ressourcenanforderung eines Prozesses wird verwehrt, wenn dadurch ein Deadlock entstehen könnte.
      Stichwort: **Banker's Algorithmus**

3) **Deadlock Detection**: Ressourcenanforderungen werden immer gewährt und es wird durch periodisches Überprüfen sichergestellt ob ein Deadlock vorliegt. Einen Deadlock aus zu machen ist relativ einfach, da die Ressourcen die die Prozesse besetzt halten, dem Betriebssystem bekannt sind. Auch dieses Verfahren ist oft nicht automatisierbar. Eine weitere Möglichkeit ist das Ausführen eines Rollbacks auf einen bestimmten Prozess, der für den Deadlock verantwortlich sein könnte.
   Nicht alle Prozesse können problemlos zurückgesetzt werden. (ein Festplatten-Schreibvorgang etwa eignet sich meist besser als ein CD/DVD Brennvorgang)

2. Erklärung und Unterschiede Prozess vs. Thread

Prozess:

Ein Prozess ist ein „ausführbares Programm“ mit zughörigen Daten wie Variablen, Puffer, etc. Den aktuellen Zustand des Prozesses nennt man Kontext der die Daten zur Prozessverwaltung (Wartebedingung, Prioritäten, etc.) enthält

Der **Dispatcher** im Betriebssystem entscheidet welcher Prozess als nächstes bearbeitet werden soll.

Ein sogenannter **Trace** beschreibt das Verhalten eines Prozesses, bzw. die Abfolge der Instruktionen die für den Prozess ausgeführt werden. Die Traces können in Zeitscheiben in der eine Anzahl von Traceinstruktionen abgearbeitet werden aufgeteilt werden. Der Dispatcher entscheidet welcher Traceabschnitt als nächstes dran kommt.

Das Betriebssystem muss Datenstrukturen für die Prozesse zur Verfügung stellen. Es kontrolliert die Ausführung der Prozesse, überwacht die Zustände der Prozesse und allokiert den notwendigen Speicher.

Prozesszustände

Es gibt grundsätzlich 5 wesentliche Zustände in denen sich ein Prozess augenblicklich befinden kann:

- **New**: Der Prozess wurde vom Betriebssystem erzeugt und erhält eine eindeutige Process-ID. Im New-Zustand ist der Prozess jedoch **nicht ausführbar**.
- **Ready:** Prozess ist bereit zur Ausführung und wartet auf die CPU
- **Running:** Nun läuft der Prozess, d.h. er ist im Besitz der CPU und wird ausgeführt.
- **Blocked:** Wenn der Prozess auf ein Ereignis **wartet** (z.B. auf eine Eingabe auf der Tastatur), dann ist der Prozess im **Blocked State** und ist nicht laufbereit.
- **Exit:** Der Prozess ist terminiert und wird nicht mehr ausgeführt. Die Prozessinformationen werden gelöscht.

Das Prozessmanagement wird oft als **Queuing Modell** realisiert und vom **Dispatcher** bearbeitet.

Prozessswitch

Unter einem Prozess Switch versteht man das Umschalten des aktiven Prozesses. Dies ist immer dann möglich wenn das Betriebssystem im Besitz der CPU ist. (durch **Trap**/**Interrupt**)

Swapping

Wenn sich zu viele Prozesse im Hauptspeicher befinden, führt dies zur Verschlechterung der Performance. Deshalb werden gewisse Prozesse eine Zeit lang aus dem Arbeitsspeicher ausgelagert (Swapping). Durch das Swapping ist es notwendig **zwei zusätzliche Prozesszustände** einzuführen.

- **Ready suspend**: Ein Prozess mit dem Ready State wurde aus der Ready Queue entfernt und in den Sekundärspeicher ausgelagert.
- **Blocked suspend**: Ein Prozess mit dem Blocked State wurde aus der Wait Queue entfernt und in den Sekundärspeicher ausgelagert.

Threads:

Bisher bildete ein Prozess eine Einheit für die Ressourcenverwaltung und Dispatching. Die Motivation für die Einführung von Threads ist die Entkoppelung dieser zwei Dinge. Ein Prozess (ein „Task“) ist nur noch die Einheit der Ressourcenverwaltung und die Threads (leichtgewichtetet Prozesse) die Einheit für das Dispatching.

Dies bringt Vorteile:

- Erstens ist die **Threaderzeugung** viel **schneller** als ein **Prozess Switch**. Es muss nur mehr der „*Threadkontext*“ umgeschaltet werden. Die Prozessressourcen bleiben die gleichen.
- Die **Kommunikation** zwischen den **Threads** eines **Prozesses** ist **ohne** Einschaltung des **Kernels** möglich.

Die Threads haben also **alle Ressourcen gemeinsam**, deshalb muss die Konsistenz der Daten mittels **Synchronisationsmechanismen** gewährleistet werden, was in der Eigenverantwortung des Programmierers liegt.

Zustände des Threads:

- **Blocked**
- **Ready**
- **Running**

Zustände aller Arten von **Suspend** existieren hier nicht, da es beim Suspendieren darum geht Ressourcen freizugeben, was bei Threads keinen Sinn macht. Bei Terminierung eines Prozesses, terminieren auch all seine zugehörigen Threads.

Es gibt **ULT´s** (User-Level-Threads) und **KLT´s** (Kernel-Level-Threads)

3. Inodes

I-Nodes (Index Node)

Gehören im File-Management zum File System Layout unter den Strategien für die Datei Implementierung. Neben **Continous Allocation, Chained Allocation und Indexed Allocation**.

Im Allgemeinen gibt eine I-Node Informationen für die Datenverwaltung bekannt. In den meisten Unix-Betriebssystemen werden Dateien über ihre I-Nodes verwaltet. Jeder I-Node ist durch eine eindeutige Inode-Nummer gekennzeichnet und verweist auf genau eine Datei. Der I-Node enthält Attribute der betroffenen Datei, nicht jedoch Namen und Inhalt. Dateinamen verweisen auf den Inode, dieser verweist auf den Speicherplatz des Datei-Inhalts.

Wie ist die Datenstruktur der I-Nodes, wie werden sie auf dem System abgebildet?

Blocking

Beim File Blocking geht es um die Abbildung von Records auf Blöcke. Dafür gibt es verschiedene Methoden:

- ***Fixed Blocking:*** Jeder Record hat eine fixe Länge und eine bestimmte Anzahl von Records pro Block ist möglich. Dies führt zu einem Verschnitt.
- ***Variable Blocking Spanned:*** Die Records haben variable Länge und Records können auch auf zwei Blöcke verteilt sein. -> kein Verschnitt.
- ***Variable Blocking Unspanned:*** Auch hier haben die Records eine variable Länge, allerdings muss ein Records als Ganzes in einem Block gespeichert werden. -> Verschnitt.

Directory Implementierung

Um Dateien zu finden, wird zuerst das Root Directory lokalisiert. Danach wird der Pfadname interpretiert. In Unix-Systemen werden im Superblock die schon erwähnten I-Nodes gespeichert. Der erste dieser I-Nodes verweist auf das Root Directory.

4. Thrashing inkl. Lösungsansatz

Durch häufige Page Faults verbringt der Prozessor sehr viel Zeit mit dem Laden von Sekundärspeicher. Das geht auf Kosten der Effektivität. Dies passiert meistens wenn das Residentset (Teile des Prozesses, die sich im Hauptspeicher befinden) zu klein ist. *(Warum ist es zu klein? -> Es werden zu viele Prozesse im Hauptspeicher gehalten oder bearbeitet schedeult)*

Lösungsansatz: Translation Lookaside Buffer

Die Pagetables selber sind auch im Speicher. Eine Adressübersetzung beim Paging hat also zwei Zugriffe auf den physikalischen Speicher. Einmal in der PageTabelle und einmal wegen der physikalischen Adresse. Um das zu verhindern wird ein sogenannter Translation Lookaside Buffer (TLB) (=Cache für Einträge der Seitentabelle) verwendet. Aus dem TLB bekommt man sofort die Framenummer. Wenn wir noch keinen Eintrag im TLB finden, so muss man wohl einen Zugriff auf die Pagetable machen. Es wird aber hier dann auch gleichzeitig der Eintrag in der TLB geschrieben, damit es beim nächsten Mal schneller geht.

Working Set

Diese Strategie beschreibt die variable Allokation von Frames für einen Prozess aufgrund der Lokalitätsannahme. Das Working Set W(D,t) beschreibt die Menge der Seiten des Prozesses, die in den letzten D virtuellen Zeiteinheiten referenziert wurden. Das Working Set wächst stark beim Start eines Prozesses an und stabilisiert sich während der weiteren Prozessausführung. Das Working Set wird beobachtet und die Seiten die sich in diesem Working Set befinden werden periodisch gelöscht.

5. Mutal Exclusion und Synchronisation

Lösungen zu Kritischen Abschnitt-Problem müssen folgende Kriterien haben:

- **Mutal Exclusion:** Nur ein Prozess pro Zeit im KA
- **Prozess**: Falls mehrere Prozesse gleichzeitig in KA eintreten wollen, wird die Reihenfolge in endlicher Zeit festgelegt, wobei nur die eintretenden Prozesse an der Entscheidung beteiligt sind.
- **Bounded Waiting:** Kein Prozess darf beliebig lange auf den Eintritt in KA warten müssen (kein starvation)
- Keine Annahmen über relative oder absolute Ausführungsgeschwindigkeiten
- Wenn ein Prozess im unkritischen Abschnitt stoppt, beeinfluss das nicht den Ablauf der anderen Prozesse

6. Bell and LaPadula's Model

Objekte haben Sicherheitsstufen und Benutzer auch. Benutzer dürfen keine Objekte höherer Stufe lesen oder schreiben. Das Bell LaPadula Modell hat folgende Security Axiome, welche sicherstellen, dass Information höherer Stufen nicht in niedrigere Stufen gelangen:

No read Up: Ich darf nichts lesen, was höhere Sicherheitsstufe hat

No write Down: Ich darf nur auf höherer Sicherheitsstufe schreiben. Damit wird verhindert dass Informationen von höherer Stufe an jemanden in einer niedrigeren Stufe gelangen.

Subject S kann Objekt O1 lesen und O2 schreiben wenn: SC(O1) <= SC(S) <= SC(O2)

7. 4 Arten der Datei Implementierung

I-Nodes (Index Node)

Gehören im File-Management zum File System Layout unter den Strategien für die Datei Implementierung. Neben **Continous Allocation, Chained Allocation und Indexed Allocation**.

Im Allgemeinen gibt eine I-Node Informationen für die Datenverwaltung bekannt. In den meisten Unix-Betriebssystemen werden Dateien über ihre I-Nodes verwaltet. Jeder I-Node ist durch eine eindeutige Inode-Nummer gekennzeichnet und verweist auf genau eine Datei. Der I-Node enthält Attribute der betroffenen Datei, nicht jedoch Namen und Inhalt. Dateinamen verweisen auf den Inode, dieser verweist auf den Speicherplatz des Datei-Inhalts.

Continous Allocation

Dabei handelt es sich um das klassische Memory Allcocation Model, wobei eine Datei eine andere angrenzende Menge von Blöcken belegt.

- ***Vorteil***: gute Performance
- ***Nachteil***: Platzprobleme beim Vergrößern einer Datei und die externe Fragmentierung

Chained Allocation

Hierbei belegt die Datei eine Menge von einzelner Blöcke die über Zeiger miteinander verkettet sind.

- ***Vorteil:*** keine externe Fragmentierung
- ***Nachteil:*** Bei Random Access ist dieses Verfahren sehr langsam und die Nutzdaten pro block sind < 2^n da ja der Zeiger auf den Nachfolgeblock selbst auch Platz braucht.

Indexed Allocation

Um den Nachteil der reduzierten Nutzdaten von Chained Allocation entgegenzuwirken wird bei dieser Strategie der Pointer auf den Folgeblock nicht in den Blöcken gespeichert, sondern in einer Tabelle im Speicher. (**F**ile **A**llocation **T**able, **FAT**)

- ***Vorteil:*** direkter und sequentieller Zugriff gut möglich. Die Blöcke können ganz für die Nutzdaten verwendet werden.
- ***Nachteil:*** Platzbedarf für FAT im Arbeitsspeicher

8. Disk Scheduling (wozu, Verfahren erklären)

Beim Disk I/O – Scheduling versucht man die Plattenzugriffe möglichst zu beschleunigen.

Begriffe:

Seek Time: Unter der Seek Time versteht man die benötigte Zeit um den Disk-Arm zur gewünschten Spur zu bewegen (Mittelwert).

Rational Delay: Bezeichnet die Zeitverzögerung, bis der Anfang des gesuchten Sektors gefunden wurde. Dies entspricht im Mittel eine halbe Umdrehungszeit der Disk.

Transfer Time: Dabei handelt es sich um die benötigte Zeit zum Übertragen der Daten.

Aus diesen Werten kann man die benötigte Zeit für einen Datenzugriff, die Average Access Time errechnen.

Die Requests können überall auf der Disk verteilt liegen. Um die Seek Time bei einem Disk-I/O Request zu minimieren werden verschiedene Strategien verwendet.

Priority: Der Lesekopf wird zu denjenigen Requests bewegt dessen Priorität am nächsthöchsten ist.

First-In, First-Out (FIFO): Es wird der Request genommen der in zeitlicher Reihenfolge als nächstes dran kommt.

Last-In, First-Out (LIFO): Der Lesekopf nimmt die jüngsten zuerst.

Shortest Service Time First (SSTF): Es werden jene Requests genommen welche am nähesten sind.

SCAN (Elevator Alg.): Beim Elevator Algorithmus bewegt sich der Lesekopf immer in eine Richtung. D.h. Der Lesekopf bewegt sich zunächst nach Innen und arbeitet die Requests ab. Ist er ganz Innen angekommen bewegt er sich wieder ganz nach außen. ***Nachteil*** dieser Strategie ist, dass Requests in der Diskmitte bevorzugt werden und Request an den Rändern länger warten müssen.

C-SCAN: Um den Nachteil des Elevator Algorithmus zu beheben wird beim C-SCAN Algorithmus immer nur in eine Richtung gegangen. D.h. Der Lesekopf bewegt sich zunächst von innen nach außen und arbeitet Requests ab. Dann wird der Lesekopf an den Innen Rand zurückgesetzt (ohne Requests zu bearbeiten) und der Vorgang wiederholt sich.

N-step-SCAN: Bei diesem Algorithmus wird die Request Queue in Subqueues der Länge N aufgeteilt. Diese N Requests werden dann mit dem Elevator Algorithmus abgearbeitet.

FSCAN: Der FSCAN Algorithmus arbeitet mit zwei Subqueues. Während dem scannen, befinden sich alle Requests in der ersten Queue und neuangekommene Requests werden in die zweite Queue gereiht. Die neuen requests werden erst abgearbeitet wenn die alten abgearbeitet wurden.

Disc Cache

Beim Disk Cache handelt es sich um eine gewisse Menge RAM um den Plattenzugriff zu beschleunigen. Der Cache kann entweder in der Festplatte eingebettet sein oder sich im Hauptspeicher befinden. Hard Disk Caches sind effektiver aber auch teurer und deshalb auch kleiner. Fast alle modernen Disk Drives haben so einen internen Cache. Ein riskanter Nachteil von Disk Caching ist, wenn das Computersystem, aus welchem Grund auch immer, abstürzen sollte. Dann sind die Daten im Cache verloren. Deshalb werden Daten im Disk Cache regelmäßig zurückgespeichert, sodass im Falle eines Absturzes nicht alle Daten weg sind.

9. Security Grundprinzipien bzw. Ziele

Unter dem Begriff Security versteht man Strategien, Vorkehrungen und Tools, um die Vertraulichkeit, Integrität und Verfügbarkeit von Informationen in einer Organisation zu gewährleisten.

Zielsetzung zum Thema Sicherheit (Anforderungen):

- **Confidentiality (Secrecy):** Regelung der Geheimhaltung von Informationen (Zugriffsschutz)
- **Integrity:** Sicherstellen, dass Daten dem Sollzustand entsprechen. Daten dürfen nicht verändert werden.
- **Availability:** Verfügbarkeit der Daten. Daten und Services die angegeben werden sollen auch tatsächlich verfügbar sein.
- **Authenticity:** Korrektheit der Identität. Maßnahmen die Versuchen die Korrektheit der Identität der Person die auf den Daten arbeiten sicherzustellen.

Man unterscheidet grundsätzlich zwischen zwei Typen von Bedrohungen:

- **Passive Threats:** Unter passiven Bedrohungen versteht man das Abhören von Information oder Beobachten des Datenflusses von nicht autorisierten Personen. Dies für zur ***Confidentiality Violation***.
- **Active Threats:** Aktive Bedrohungen sind manipulative Angriffe auf das System um an Informationen zu kommen. Dies führt zur ***Beeinträchtigung*** von ***Integrität*** und ***Service***.

Es gibt grob 3 Klassen von Bedrohungen

- **Denial of Service:** Als Denial of Service (DoS) wird die Folge einer Überlastung von Infrastruktursystemen bezeichnet.
- **Exposure/Interception:** Nicht autorisierter lesezugriff auf Daten.
- **Modification/Fabrication:** Manipulation von Daten durch nicht-autorisierte Personen/Prozesse. Verletzung der Datenintegrität.

Confidentiality wird durch Exposure /Interception bedroht.
Integrity wird durch Modification / Fabrication bedroht.
Availability wird durch Denial of Service bedroht.

Malware: eine Menge von Software die Computersysteme bedrohen
Virus: Ein Virus ist ein in einem Programm versteckter Code, der sich selbst in andere Programme kopiert und dort destruktives Verhalten bis hin zu Denial of Service (DoS) oder sogar Distributet Denial of Service (DDoS) Attacken durchzuführen.
Worm: Unter einem Worm versteht man ein Programm, das sich selbst repliziert und Kopien seines Codes über ein Netz an andere Computer sendet.
Trojan Horse: Das ist ein Programm mit gewünschter Funktionalität, das jedoch versteckten Code mit bösartiger Funktionalität enthält.
Logic Bomb: Eine Logic Bomb ist ein Programmstück, das sich selbst beim Auftritt eines bestimmten Ereignisses aktiviert.
Trapdoor: Ein Trapdoor ist ein geheimer Einstiegspunkt in ein System bzw. die Umgehung der Zugriffskontrolle.
Standard Attacken: Anfordern und Auslesen von Speicher, fälschen der Login-Routine, gezieltes Abbrechen von Prozessen.
Design Prinzipien für Security

- ***Open Source:*** Eine Community steht dahinter. Es besteht hier natürlich das Risiko: Wer findet den Fehler zuerst?
- ***Security by Obsurity:*** wobei alle Informationen versteckt werden.

Passwörter: Die Wahl eines guten Passworts ist überlebensnotwendig um nicht einer Attacke zum Opfer zu fallen.

Reference Monitor: Beim Reference Monitor handelt es sich um einen Schutzmechanismus sodass nicht jeder Benutzer Vollzugriff hat, sondern nur gewisse Berechtigungen. Der Reference Monitor überwacht alle Operationen/Zugriffe im System und analysiert das Systemverhalten (mitprotokolieren).

Protection Domains: Eine Protection Domain beschreibt eine Menge von Paaren von Objekten und zugehörigen Rechten. Die Domain kann sich ändern. Zum Beispiel wenn ein Prozess vom Usermode in den Kernelmode wechselt. Die Zugriffsrechte für jede Domain wird manchmal in der sogenannten Access Matrix gespeichert. Die Rechte sind direkt an die Objekte gekoppelt (Access Control List). Eine andere Möglichkeit ist das Einsetzen von Capability Lists. Dabei gibt es für jeden Prozess eine Liste mit Objekten und den erlaubten Zugriffsoperationen (read, write, execute). Hier können die Rechte auch an andere Objekte vererbt werden.

10. PCB

Prozess Control Block

Dieser wird vom Betriebssystem zur Verwaltung der Prozesse herangezogen und enthält eine eindeutige Prozessnummer (PID), welcher der Prozess bei der Erzeugung zugewiesen bekommt. Weiters enthält der PCB User Identifier für die Zugriffsrechte und die Nummer des Prozesses der den Prozess generiert hat (Parent Process Identifier). Jeder Prozess ist Kind Prozess eines anderen.

Das PCB gibt einerseits Auskunft über den Status des Prozesses und andererseits Informationen über die Kontrolle des Prozesses. So werden Informationen für das Scheduling des Prozesses (Zustand, Priorität, ...), Querverweise auf andere Prozesse (Parent, Child, ...), Informationen für Interprozesskommunikation, Rechte, Verweise auf Segment- und Seitentabellen und Informationen über verwendete Ressourcen bereitgestellt. Die PCB´s der Prozesse sind miteinander verlinkt.

Execution Modes

Durch Hardware Unterstützung werden die Datenstrukturen (wie z.B. das PCB) geschützt und können im sogenannten **Privileged Mode** oder **User Mode** ausgeführt werden. Jedes Mal wenn das Betriebssystem die Kontrolle übernehmen soll, muss ein Mode Switch durchgeführt werden. Erst dann ist es erlaubt diese Ressourcen zu modifizieren.

Bsp. Process Switch -> laufendem Prozess wird die Kontrolle entzogen -> Mode Switch (Privileged Mode) -> Update des PCB -> Process Switch (do things) -> Mode Switch (User Mode)

11. ULT, KLT

User-Level Threads (ULT)

Diese Art von Threads sind Implementierungen auf Benutzerebene und für den Kernel unsichtbar. Das Thread Management geschieht mittels Thread Libraries (Applikationslibaries im Userspace, Code für die Erzeugung und Terminierung von Threads, Thread Scheduling, Sichern/Herstellen von Thread, Kontexten,...). Das Betriebssystem hat keine Ahnung dass es diese Threads überhaupt gibt. Deshalb ist der Zugriff auf mehrere Prozessoren nicht garantiert.

Ein großer **Vorteil** von ULT ist, dass das Einschalten des Kernels bei einem Threadswitching entfällt (**kein Mode Switch**). Weiters kann das Scheduling sehr applikationsspezifisch erfolgen (Wähle den besten Algorithmus für die Situation). ULT´s können auf verschiedene Betriebssysteme laufen. Man benötigt nur die geeignete Thread-Library. **Nachteil**: Der Kernel kann nur Prozesse zu Prozessoren verteilen. Zwei Threads im gleichen Prozess können also nicht simultan auf zwei Prozessoren ausgeführt werden.

Kernel-Level Threads (KLT)

Bei dieser Art von Threads wird das Thread Management vom Kernel übernommen. Es gibt eine Kernel Thread API. Kernel-Level Threads garantieren den Zugriff auf mehrere Prozessoren, dafür ist die Performance dieser etwas geringer als bei User-Level Threads, wegen dem System Overload. **Vorteil**: Der Kernel kann mehrere Threads des gleichen Prozess simultan schedulen. **Nachteil**: Thread-Switching auch innerhalb des gleichen Prozesses involviert den Kernel, was sich auf die Performance niederschlägt.

12. Memory Management

Anforderung an die Speicherverwaltung:

- Partitionierung (Speicheraufteilung auf Prozesse
- Relocation (Positionierung von Code und Daten im Speicher, Virtual-Memory Management)
- Protection (Speicherschutz)
- Shareing (gemeinsamer Zugriff auf Speicher)
- Performance (effektive log./physikal. Organistation)

3 Arten der Partionierung:

- ***Fixed partitioning*** (fix definierte Partitionsgröße)
    - Interne Fragmentierung (Verschleiß)
- ***Dynamic partioning*** (Partition sind genau so groß wie die Anforderungen es verlangen)
    - Externe Fragmentierung (teile werden unbenutzbar, da zu klein)
- ***Buddy System:*** aufteilen in Blöcke mit 2er Potenzen (z.B. 2^8 = 256 MB)

Fragmentierung

Unter Fragmentierung (Fragment = Bruchstück) versteht man die verstreute Speicherung von logisch zusammengehörigen Datenblöcken des Dateisystems auf einem Datenträger, was als Spezialfall der allgemeinen Speicherfragmentierung betrachtet werden kann. Bei Speichermedien mit relativ hohen Zugriffszeiten wie beispielsweise Festplatten kann Fragmentierung zu einer spürbaren Verlangsamung der Lese- und Schreibvorgänge führen, da diese ihre volle Geschwindigkeit technisch bedingt nur bei sequentiellem Zugriff erreichen.

Clock Policy

Dabei handelt es sich um einen “Ringbuffer” mit Frames die für den Austausch in Frage kommen. Nach einem Page Fault wird der Pointer auf den folgenden Frame im Ringbuffer gesetzt und ein Use Bit des Frames wird auf 1 gesetzt wenn die Seite geladen oder referenziert wird. Bei Page Fault wird die erste Seite ab der Pointerposition mit Use Bit = 0 gesucht. Dabei werden Use Bits mit dem Wert 1 wieder auf 0 gesetzt.

FIFO Policy

Bei vollem Seitenkontingent wird die älteste Seite ersetzt (FIFO)

- **Nachteil**: Es können hier Seiten die oft verwendet werden ausgelagert werden weil sie an letzter Stelle sind.
- **Vorteil**: einfache Implementierung durch Pointer

LRU Policy

Bei der LRU (Last Recently Used) Policy werden Seiten ersetzt die am längsten nicht benutzt worden sind. Etwas mehr Page Faults als OPT Policy sind hier zu erwarten. Die Implementierung ist hier jedoch sehr aufwendig.

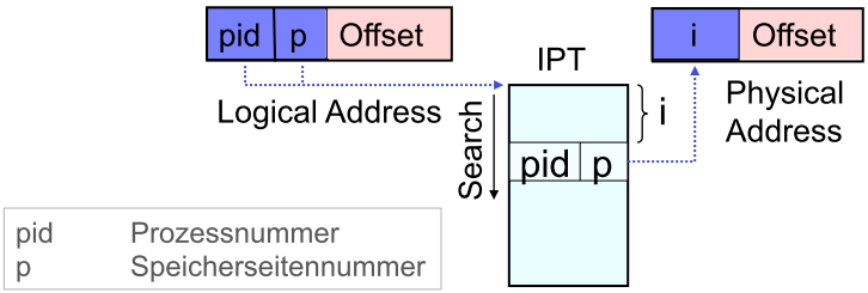
OPT Policy

Bei der OPT Policy werden Seiten ersetzt deren nächste Referenz am weitesten in der Zukunft liegen. Diese Strategie erzeugt sehr wenige Page Faults.

Inverted Page Table (IPT)

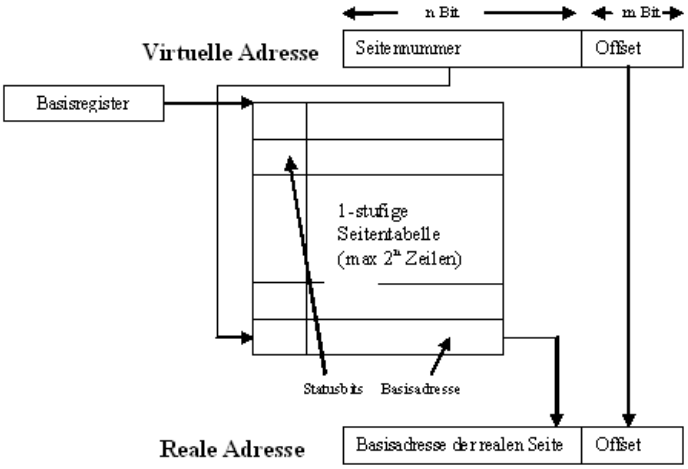
- Eine IPT für das ganze System4

- Ein IPT-Eintrag pro physikalischem Frame
  - Bei Zugriff auf Speicher: assoziative Suche in der IPT bzw. Hashing um Suche in der IPT zu beschleunigen
  - Page Fault, wenn gesuchter Eintrag nicht in IPT



Paging

Die Segmentierung wird nicht automatisch verwaltet. Deshalb braucht man das sogenannte „Paging“. Als Paging oder zu Deutsch Kachelverwaltung bezeichnet man die Methode der Arbeitsspeicher-Verwaltung per Seitenadressierung durch Betriebssysteme. Dabei wird der Hauptspeicher in (kleine) Seitenrahmen (**Frames**) gleicher Größe unterteilt. Prozesse werden in Seiten (**Pages**) derselben Größe unterteilt. Die Pages passen genau in die Frames. In einer **Page Table** pro Prozess wird festgehalten welche Framenummer jede Page hat. Außerdem gibt es noch eine **Free Frame List** welche auf freie Frames im Speicher verweist.



Relocation

Bei Systemen mit virtuellem Memory-Management muss es möglich sein, dass Programme in unterschiedlichen Positionen im Speicher abgelegt sein können. Bei der Relocation handelt es sich um eine Unterstützung um Instruktionen und Daten an verschiedenen Stellen im Speicher zu halten.

13. Praxisbeispiele:

13.1. RMS Scheduling inkl. Schedulability Test

Dieses Verfahren wird auch häufig in Echtzeitsystemen eingesetzt. Die Prioritäten der Tasks werden statisch nach der Periodendauer eines Jobs festgelegt. Je kürzer die Periodendauer eines Jobs, desto höher ist seine Priorität. (Periode = Deadline) Die Tasks müssen periodisch sein, die Ausführungszeit muss bekannt sein und die Tasks sollten unabhängig sein (schwierig zu erfüllen).

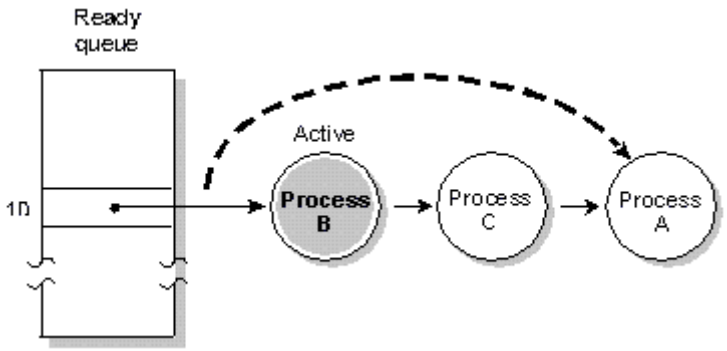
- **Schedulability Test**
  Bei einem Scheduability Test wird überprüft ob alle Deadlines eingehalten werden können.
  Hinreichende Bedingung für RMS:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (\sqrt[n]{2} - 1)$$

  Ci... Ausführungszeiten
  Ti... Periodenlängen
  n... Anzahl der Jobs.
  Mit zunehmender Anzahl von Jobs (n --> unendlich) nähert sich die Schranke dem Wert ~0.69
  Notwendig Bedingung: Summe (Ci/TI) <= 1 … das die Summe der Ausführungszeiten dividiert durch die Periode muss kleiner, gleich 1 sein. Andernfalls ist das Set nicht schedulbar.

13.2. Round Robin Scheduling

Hierbei wird **einem Prozess** die CPU für eine **kurze Zeiteinheit** zugeteilt. Danach wird der Prozess wieder hinten in die Warteliste eingereiht. Der nächste Prozess wird nach dem FIFO-Prinzip ausgewählt. I/O-intensive Prozesse werden von dieser Strategie benachteiligt weil sie die Zeitscheiben nicht voll ausschöpfen können. Deshalb wird häufig eine Hilfsqueue für die Prozesse (auxiliary queue) verwendet wo I/O Prozesse eingereiht werden und welche eine höhere Priorität als die herkömmliche Ready Queue besitzt (**Virtual Round Robin**)



13.3. Producer/Consumer mit Semaphoren

Das Producer-Consumer Problem ist eine klassische, abstrakt formulierte Problemstellung der Prozesssynchronisation, welche eine Regelung der Zugriffsreihenfolge auf eine Datenstruktur durch elementerzeugende (schreibende) und elementverbrauchende (lesende) Prozesse bzw. Threads thematisiert.

```
// Aufnahmekapazität der Datenstruktur
const N=4;
// Deklaration der Semaphore
semaphor mutex;
semaphor sem_read;
semaphor sem_write;
// Initialisierung der Semaphore
init (mutex, 1);      // genau einem Prozess wird die Modifikation der Datenstruktur ermöglicht
init (sem_read, 0);   // kein Element befindet sich in der Datenstruktur
init (sem_write, N);  // es sind N freie Ablageplätze für Elemente vorhanden

// Erzeugerprozess
while (1) {
  P (sem_write);     // Zeige Ablageaktion an
  P (mutex);         // Schütze die Datenstruktur vor Zugriffen anderer Prozesse
  write (element);   // Schreibzugriff auf die Datenstruktur
  V (mutex);         // Hebe den Datenstrukturschutz auf
  V (sem_read);      // Informiere einen evtl. blockierten Verbraucherprozess über die Ablage eines Elements
}

// Verbraucherprozess
while (1) {
  P (sem_read);      // Zeige Entnahmeaktion an
  P (mutex);         // Schütze die Datenstruktur vor Zugriffen anderer Prozesse
  read (element);    // Lesezugriff auf die Datenstruktur
  V (mutex);         // Hebe den Datenstrukturschutz auf
  V (sem_write);     // Informiere einen evtl. blockierten Erzeugerprozess über die Entnahme eines Elements
}
```

13.4. Semaphoren in Pseudocode implementieren, simples Beispiel zum synchronisieren

```
Gemeinsam von A und B genutzter Semaphor: mutex
Initialisierung: init (mutex, 1)  /* Es gibt 1 Ressourcen-Exemplar "Recht auf Betreten eines kritischen Abschnitts" (k.A.) */
```

```
Prozess A          Prozess B
   ...             ...
   P (mutex)       P (mutex) /* versuche "Betreten-Recht" zu reservieren (blockierend) */
   ka_A            ka_B
   V (mutex)       V (mutex) /* freigeben des "Betreten-Rechts" */
   ...             ...
```

13.5. Reader/Writer mit Semaphoren

Reader-Writer Problem

Reader:

```
loop
    P (z);
    P (rsem);
    P (x);
    rc := rc + 1;
    if rc = 1 then P (wsem);
    V (x);
    V (rsem);
    V (z);
    read;
    P (x);
    rc := rc - 1;
    if rc = 0 then V (wsem);
    V (x);
end loop
```

Writer:

```
loop
    P (y);
    wc := wc + 1;
    if wc = 1 then P (rsem);
    V (y);
    P (wsem);
    write;
    V (wsem);
    P (y);
    wc := wc - 1;
    if wc = 0 then V (rsem);
    V (y);
end loop
```

<u>Schreiber haben Priorität</u>

Reader-Writer Problem

```
init (x, 1); init (y, 1); init (z, 1); init (wsem, 1); init (rsem, 1);
rc := 0; wc := 0;
```

Reader:

```
loop
    P (x);
    rc := rc + 1;
    if rc = 1 then P (wsem);
    V (x);
    read;
    P (x);
    rc := rc - 1;
    if rc = 0 then V (wsem);
    V (x);
end loop
```

Writer:

```
loop
    P (wsem);
    write;
    V (wsem);
end loop
```

<u>Leser haben Priorität</u>