BSVO Fragen

Zusammenfassung aller Fragen der mündlichen Prüfungen von 2022-11-10 bis 2013-09-03.

Für die Infos wurden hauptsächlich die Folien, BS Fragensammlung + Antworten.pdf und Zusammenfassung 2021W.pdf verwendet.

Neben jeder Überschrift steht wie oft dieses Thema gefragt wurde.

BSVO Fragen

- Mündliche Fragen
  - Prozesse und Threads
    - Was sind Prozess-Zustände und welche gibt es? (2)
    - Ursachen für ein Suspend? (2)
    - Was ist das Prozess Image? (8)
    - Was ist der PCB? (13)
    - Was ist ein Process Switch? (8)
    - Was ist der Microkernel? (8)
    - Was sind Threads? (9)
    - Prozesse vs. Threads (6)
    - User Level Threads vs. Kernel Level Threads (11)
  - Mutual Exclusion und Bedingungssynchronisation
    - Was sind die Anforderung für einen kritischen Abschnitt? (2)
    - Was ist ein Semaphore? (2)
  - Deadlock
    - Was sind Deadlocks? (2)
    - Was sind die Bedingungen für einen Deadlock? (14)
    - Deadlock Prevention (5)
    - Deadlock Avoidance (6)
  - Speicherverwaltung
    - Worum geht es beim Memory Management bei Prozessen? (1)
    - Was ist Fragmentierung? (9)
    - Was wird unter Relocation verstanden? (7)

- Was ist Paging? (11)
- Was ist Virtual Memory? (9)
- Was ist eine Inverted Page Table? (6)
- Was ist der Multi-Level Page Table? (1)
- Was ist der Translation Lookaside Buffer? (11)
- Was ist die Replacement Policy? Welche Strategien gibt es? (8)
- Was ist das Working-Set? (7)
- Welche Scheduling-Ebenen gibt es? (4)

Scheduling

- Round Robin und Virtual Round Robin - Scheduling (12)
- Shortest Process Next - Scheduling (1)
- Shortest Remaining Time - Scheduling (1)
- Highest Response Ratio Next - Scheduling (8)
- Feedback Scheduling (4)

Ein-/Ausgabe

- In welche Schichten ist die I/O organisiert?
- Was ist I/O-Buffering? (5)

File Management

- Wie ist ein Dateisystem realisiert? (5)
- Disk Scheduling (1)

Security

- Was sind die drei Security Objectives (2)
- Was ist eine Access Control List? (5)
- Was ist eine Capability List? (6)
- Was ist das Bell und LaPadula's Modell? (2)

Schriftliche Fragen

- Semaphoren - Wie kann man Semaphoren realisieren? (8)
- Semaphoren - Zwei Reader ein Writer (9)
- Semaphoren - Conditional Sync (2)
- Producer Consumer mit Ringbuffer (Fixe Größe K) (13)
- Reader Writer Problem (hier haben Writer Priorität) (13)
- Prozessfortschrittsdiagramm (3)
- Buddy System (4)
- Bankers Algorithm (7)
- Monitor (2)
- Deadlock Detection Algorithmus (1)

Round Robin Scheduling (4)

Mündliche Fragen

Prozesse und Threads

Was sind Prozess-Zustände und welche gibt es? (2)

Ein Prozess kann sich in mehreren Zuständen befinden, aber nicht gleichzeitig. Die wesentlichen Zustände sind:

- **New**: Prozess wurde erstellt
- **Running**: Prozess rennt und ist im besitzt der CPU
- **Ready**: Prozess ist lauffähig, wartet bis er CPU Zeit bekommt
- **Blocked**: Prozess wartet auf ein Ereignis (z.B. I/O, Timer)
- **Ready, Suspend**
- **Blocked, Suspend**
- **Exit**: Wird durch das Terminieren des Prozesses erreicht.

Prozesse kommen in den *Ready, Suspend* und *Blocked, Suspend* Zustand, wenn sie durch das Swapping von Hauptspeicher ausgelagert werden.

Ursachen für ein Suspend? (2)

- Swapping
- Problembehaftete Prozesse auslagern
- Interaktive Request: z.B. Debugging
- Timing: periodischer Prozess wird ausgelagert, bis er wieder drankommen sollte.
- Parent Request: Untersuchung, Modifikation, Koordination von Kindprozessen

Was ist das Prozess Image? (8)

Das Prozess Image ist eine Datenstruktur, die beschriebt, wie ein Prozess im Betriebssystem dargestellt wird. Es besteht aus:

- **User Program**
  Der Code für das Programm.
- **User Data**
  Daten, welche sich im Verlauf der Abarbeitung verändern können.
  User Stack, Daten, modifizierbare Programme
- **System Stack**
  Parameter und Calling Adressen von System Calls. Ist speziell geschützt, damit der User nicht mit dem System interferiert.
- **Process Control Block (PCB)**
  - Process Identification
  - Process State Information
  - Process Controll Information

Was ist der PCB? (13)

Der Process Control Block ist ein Teil des Process Image. Der PCB wird jedes Mal aktualisiert, wenn ein Prozess den Zustand ändert. Er beinhaltet folgende Teile:

- **Process Identification**
  Identifikation des Prozesses.
  Process number, User idenfier, Nummer des Parent/Caller Process
- **Process State Information**
  Registerinhalte, Stack Pointer, Kontroll- und Statusregister
- **Process Control Information**
  Beinhaltet Scheduling und Zustandsinformation sowie Querverweise auf andere Prozesse.
  State des Prozesses, Priorität, Ereignis auf die der Prozess wartet, Verweis auf Parent oder Child

Was ist ein Process Switch? (8)

Unter einem Process Switch versteht man das Umschalten des aktiven Prozesses der CPU. Falls das BS im Besitz der CPU ist, kann dies ausgelöst werden durch:

- **Supervisor Call**: expliziter Aufruf durch das Programm
- **Trap**: bedingt durch aktuelle Instruktionen

- **Interrupt**: Ursache liegt außerhalb des Prozesses, Kontrolle geht an Interrupt-Handler

Ein Process-Switch läuft folgendermaßen ab:

1. Umschalten des aktiven Prozesses (Speicherung des PCB, Register, Stack um fortsetzen zu können)
2. Alten Prozess in die Blocked-Queue setzen oder terminieren
3. Neuen Prozess aus der Ready-Queue holen
4. Neuen Prozess der CPU zuteilen

Was ist der Microkernel? (8)

Die Microkernel-Architektur ist eine Betriebssystem-Architektur. Hier gibt es nur einige Basisservices im Kernel (Process Switching, Basic Memory Management, Interrupts und Hardware Access (I/O), Nachrichtenaustausch). Die nicht zentralen BS Service sind als Server Prozesse realisiert. Es gibt eine Nachrichtenkommunikation zwischen Betriebssystem und Prozessen. Der Kernel soll so klein wie möglich sein, der Rest sollen Prozesse sein.

Vorteile, die durch diese Architektur zustande kommen:

- Einheitliche Interfaces
- Flexibilität und Erweiterbarkeit
- Portabilität
- leichter zum Analysieren und Debuggen

Was sind Threads? (9)

Threads haben das Ziel, die Ressourcenverwaltung von der Ausführung zu trennen. Threads sind Teile eines Prozesses, die für sich selbst ausgeführt werden können. Der Prozess beinhaltet die Daten, der Thread ist eine Ausführungsstrang des Prozesses. Jeder Thread hat seinen eigenen User Stack und Kernel Stack. Die Vorteile von Threads sind:

- Threads können schneller erzeugt und terminiert werden.
- Umschalten zwischen den Threads ist schneller als ein Process Switch.
- Kommunikation ohne Einschaltung des Kernels.

Prozesse vs. Threads (6)

Ein Prozess ist ein Programm in Ausführung, das Ressourcen wie Speicher und CPU-Zeit beansprucht. Jeder Prozess hat seine eigene Speicheradresse, Ressourcen und Laufzeitumgebung.

Ein Thread ist ein Teil eines Prozesses, der eine Ausführungssequenz für den Prozess bereitstellt. Threads teilen sich den Speicher und die Ressourcen des Prozesses, in dem sie laufen, und können parallel ausgeführt werden, um die Effizienz zu erhöhen. Ein Prozess kann mehrere Threads haben, die alle auf gemeinsame Ressourcen zugreifen können.

In Kurzform: Ein Prozess ist ein Programm in Ausführung mit eigener Speicheradresse und Ressourcen. Ein Thread ist eine Ausführungssequenz innerhalb eines Prozesses, die den Prozess teilt und parallel laufen kann.

User Level Threads vs. Kernel Level Threads (11)

User Level Threads:

- Sind für den Kernel unsichtbar.
- Management durch eine Library.
- Thread Switching im User-Mode: kein Mode Switch notwendig.
- Applikationsspezifisches Scheduling

Kernel Level Threads:

- Sind natürlich für den Kernel sichtbar.
- Management durch den Kernel mittels Kernel Thread API (ist keine Library)
- Threadspezifisches Scheduling
- Thread Switching innerhalb eines Prozesses benötigt zwei Mode Switches.

Ein hybrider Ansatz ist möglich, mit ULTs innerhalb einer KLT Architektur.

Mutual Exclusion und Bedingungssynchronisation

Was sind die Anforderung für einen kritischen Abschnitt? (2)

- **Mutual Exclusion**
- **Progress**
  Wenn sich kein Prozess im k.A. befindet und Prozesse in den k.A. eintreten möchten, darf die Entscheidung über den nächsten Prozess nicht unendlich lange verzögert werden (kein Lifelock, „After You“)
- **Bounded Waiting**
  Nachdem ein Prozess einen Request für den k.A. abgegeben hat, ist die Anzahl der Eintritte in den k.A. durch andere Prozesse begrenzt. Verhinderung einer Starvation.

Was ist ein Semaphore? (2)

Ein Semaphor ist eine Technik, um Synchronisation ohne Busy Waiting zu ermöglichen. Semaphor S ist eine Int-Variable, auf die nach der Initialisierung nur noch mit den atomaren Funktionen `wait(S)` und `signal(S)` zugegriffen werden kann. Diese Funktionen inc. bzw decr. den Wert der Variable.

Natürlich besteht ein Semaphor auch noch aus einer Queue, wo sich die Prozesse anstellen, welche auf die Anweisungen nach dem `wait` hinwollen.

Deadlock

Was sind Deadlocks? (2)

Permanentes Blockieren einer Menge von Prozessen, die um Ressourcen konkurrieren oder miteinander kommunizieren. Es ist ein zyklischer Ressourcenkonflikt zwischen zwei oder mehreren Prozessen.

Was sind die Bedingungen für einen Deadlock? (14)

- **Mutual Exclusion**
  Ein Prozess hat exklusiven Zugriff auf eine Ressource.
- **Hold and Wait**
  Ein Prozess behält Anspruch auf seine Ressourcen, während er auf andere Ressourcen wartet.

- **No Preemtion**
  Zugewiesene Ressourcen werden den Prozessen nicht weggenommen.
- **Circular Wait**
  Wenn alle diese Bedingungen eintreten, kommt es zu einem Deadlock. Prozess1 hat Ressource2 und braucht Ressource1, Prozess2 hat Ressource1 und braucht Ressource2. Beide warten aufeinander und es passiert ein Deadlock.

Deadlock Prevention (5)

Ein Betriebssystem Design welches Deadlocks von vorn hinein ausschießt. Hier gibt es die indirekte und die direkte Deadlock Prevention.

- **Indirekte**
  Durch Verhindern der Bedingungen 1-3 wird Deadlock abgewiesen.
  - Mutal Exclusion kann schlecht verhindert werden, da diese meinst Zielsetzung ist.
  - Hold and Wait: Ein Prozess muss gleich zu Anfang alle seine Ressourcen anfordern und blockiert, bis er diese erhält.
  - No-Premtion: (1) Falls Prozess eine Ressource nicht kriegt muss er seine Ressource freigeben und es später versuchen.
    (2) Falls die Ressource von einem Prozess angefordert, wird muss diese freigegeben werden.
    Der Zustand der Ressource speichern, bevor andere Prozess diese bekommt, damit die Ressource recovered werden kann.
- **Direkte**
  Verhinderung des Circular Wait durch ein Protokoll.
  Allen Ressourcenklassen wird eine linare Ordnung O zugewiesen. Prozess muss zu Anfang alle seine Ressourcen aus der Klasse R_i anfordern. Falls er neue Ressourcen aus der Klasse R_k haben will, muss gelten $O(R_k) > O(R_i)$.
  Nachteil: Ineffizienz durch unnötiges Zurückweisen aufgrund der Ordnung.

Deadlock Avoidance (6)

Selektive Verteilen von Ressourcen, um einen Deadlock zu verhindern. Die Avoidance führt zu einer höheren Paralellität als die Prevetion. Der Ressourcenbedarf muss aber bekannt sein. Hier gibt es zwei verschiedene Ansätze:

- **Process Initiation Denial**

Falls die womöglich angeforderten Ressourcen des Prozesses zu einem Deadlock führen könnte, wird dieser erst gar nicht gestartet. Es wird überprüft, ob die Ressourcenanforderung eines Prozesses in Kombination mit den bereits laufenden Prozessen zu einem Deadlock führen kann, wenn ja → Prozess wird gar nicht erst gestartet

$$R_i \geq C_{(n+1)_i} + \sum_{k=1}^{n} C_{k_i} \tag{1}$$

- **Ressource Allocation Denial**
  Falls die neu angeforderten Ressourcen eines Prozesses zu einem *unsafe state* führen würden, werden diese nicht bewilligt. Mittels des Banker's Algorithms überprüfen.

Speicherverwaltung

Worum geht es beim Memory Management bei Prozessen? (1)

- **Partitioning**: Speicheraufteilung auf Prozesse
- **Relocation**: Positionierung von Code und Daten im Speicher, Virtual-Memory Management
- **Protection**: Speicherschutz
- **Sharing**: gemeisamer Zugriff auf Speicher
- **Performance**: effektive logische und physische Organisation

Was ist Fragmentierung? (9)

Fragmentierung kann auf zwei Weisen auftreten:

- **Interne Fragmentierung**: Die Objekte, welche in der Partition sind, füllen diese nicht ganz aus und verschwenden so Platz. (z.B. bei Fixed Partitioning)
- **Externe Fragmentierung**: Zerstückelung des Speicherbereichs außerhalb der Partitionen. Es bleibt nicht nutzbarer Speicher zwischen den Partitionen über. (z.B. Dynamic Partitioning)

Den Vorgang um die Partitionen wieder näher zueinander zubringen und diesen Speicher nutzbar zu machen, nennt man **Compaction**.

Bei Pagination kann interne Fragmentierung, aber nicht externe Fragmentierung auftreten.

Was wird unter Relocation verstanden? (7)

Unter Relocation wird verstanden, dass der wirklichen belegte Speicherbereich, wo sich ein Prozess im Hauptspeicher befindet, nicht fixiert ist. Der Speicherbereich kann sich durch z.B. durch Swapping, Compaction ändern. Deswegen müssen Referenzen auf den physischen Speicher veränderbar sein, und es muss zwischen physischen und logischen Adressen unterschieden werden.

Es kann unter folgenden Adressarten unterschieden werden:

- **physische Adresse**: Absolute, tatsächlicher Position im Hauptspeicher
- **logische Adresse**: Referenz auf einer Position im Speicher. Wie ein Name für einen Speicherort.
- **relative Adresse**: Position relativ zu einem bekannten Ort im Programm (meistens eine logische Adresse)

Was ist Paging? (11)

Der Hauptspeicher wird in Frames (Blöcke gleicher Größe) unterteilt. Prozesse werden in Pages derselben Größe wie die Frames unterteilt. Es gibt einen Page-Table pro Prozess, der die aktuellen Frame Nummern für jede Page enthält. Des weiteren gibt es eine Free Page List, welche auf freie Frames zeigt.

Ein **Page Fault** ist die Ausnahmebehandlung, wenn eine Page im VM referenziert wird, aber sich nicht im Hauptspeicher befindet. Dann muss der entsprechende Speicherbereich aus dem Sekundärspeicher geladen werden. Dies nennt man **Demand Paging**.

Falls durch häufige Page Faults der Prozessor sehr viel Zeit mit dem Nachladen von Sekundärspeicher verbringt, spricht man vom **Thrashing**. Dies führt zu einem drastischen Einbruch der Effektivität. Grund dafür ist, dass das Resident Set des Prozesses zu klein ist.

Die **Adressübersetzung** funktioniert indem aus dem Page-Table die Framenummer herausgelesen wird und mit dem Page-Nummer-Teil der logischen Adresse ersetzt wird, um auf die physische Adresse zu kommen.

Was ist Virtual Memory? (9)

Dynamische Adressübersetzung. Speicher, der gerade nicht benötigt wird, ist im Sekundärspeicher ausgelagert. Durch VM erscheint es immer noch so das diese Teile im Hauptspeicher liegen. Falls Platz im Hauptspeicher benötigt wird, können Pages aus dem Hauptspeicher in den Sekundärspeicher ausgelagert werden. Diese befinden sich dann in der Virtual Memory. Falls der Prozess diese Pages wieder benötigt, kommt es zu einem Page Fault, da sie sich nicht mehr im Hauptspeicher befinden. Diese werden dann aus dem Hauptspeicher nachgeladen (Demand Paging).
Der virtuelle Adressraum erscheint also größer als der eigentliche physische Speicher. Nicht alle logischen Adressen müssen im physischen Speicher abgebildet werden.

Das **Resident-Set** bezeichnet die Teile eines Prozesses die sich wirklich im Hauptspeicher befinden.

Was ist eine Inverted Page Table? (6)

Statt für jeden einzelnen Prozess einen Page Table zu haben, wird ein Inverted Page Table für den ganzen Hauptspeicher gemacht. Hier gibt es eine Zuordnung, welche einen Eintrag pro Frame darstellt, welche sagt in welchem Frame, welche Seite von welchem Prozess dort zu finden ist.
Nachteil: die assoziative Suche im IPT

Was ist der Multi-Level Page Table? (1)

Bei großen Prozessen beenötigt der Page Table viel Speicherplatz, der sogar über mehrere Seiten gehen kann. Die Page Number wird noch einmal in zwei Teile unterteilt.
Der Outer Page Table ist eine Indexierung aller Seiten im Arbeitsspeicher und speichert Verweise auf die Pages im Page Table. Die höherwertige Page Number p1 wird verwendet, um im Outer Page Table die entsprechende Page im Page Table zu finden. Innerhalb des Page Tables kann dann mit der niedrigeren Page Number p2 auf die richtige Page zugegriffen werden. Der Outer Page Table muss immer im Arbeitsspeicher bleiben.

Was ist der Translation Lookaside Buffer? (11)

Der Translation Lookaside Buffer wird im Kontext des Pagings als High-Speed-Buffer für Page-Table-Einträge verwendet, die kürzlich aufgerufen wurden. Das Hauptziel ist die doppelte Speicherzugriffszeit zu reduzieren. Ein normaler Zugriff auf einen Page-Table-Eintrag erfordert zusätzlich das Laden der Daten aus dem Hauptspeicher oder der Virtual Memory.

Wenn eine Adresse gesucht wird, wird zuerst der Cache überprüft. Wenn die Adresse im Cache gefunden wird, kann sie verwendet werden. Wenn die Adresse nicht im Cache gefunden wird, wird im Page Table nach dem passenden Eintrag gesucht. Wenn der Eintrag gefunden wird und das Present Bit gesetzt ist, kann die Adresse im Cache gespeichert und verwendet werden. Wenn das Present Bit nicht gesetzt ist, kommt es zu einem Page Fault und die Page muss in den Arbeitsspeicher geladen werden, was zu einer I/O Operation und wahrscheinlich zu einem Process Switch führt. Wenn der Prozess zurück zur CPU kommt, sollte die gesuchte Page im Arbeitsspeicher vorhanden sein und es sollte kein Page Fault mehr auftreten.

Nachteil: assoziative Suche

Was ist die Replacement Policy? Welche Strategien gibt es? (8)

Die Replacement Policy entscheidet, welche Pages im Hauptspeicher ersetzt werden sollen. Hier gibt es zwei Ansätze:

- **lokal**: Austauschstrategie wird auf einzelne Pages des Prozesses angewandt.
- **global**: Austauschstrategie wird auf alle Pages des Hauptspeichers angewandt.

Die Replacement Strategien sind:

- **Optimal Policy**
  Führt zu den wenigsten Page-Faults, ist aber nur eine theoretische Strategie. Ersetzt die Seite, deren Referenzierung am weitesten in der Zukunft liegt.
- **Least Recently Used Policy**
  Ersetzt die Page deren letzte Refenzierung am weitesten in der Vergangenheit liegt. Basiert auf den Lokalitätsprinzip. Anzahl der Page-Faults kaum höher als bei OPT. Nachteil ist aber, dass die Implementierung sehr aufwendig ist.

- **FIFO Policy**
  Die älteste Page wird ersetzt. Ist sehr einfach zu implementieren. Nachteil ist aber, dass eine Seite, die gerade oft verwendet wird, trotzdem die älteste sein könnte.
- **Clock Policy** (4)

  Hat kaum mehr Page-Faults als LRU. Ringpuffer aus den Frames, welche infrage kommen, sowie einem Positionszeiger. Der Puffer ist mit den Use Bits befüllt. Der Zeiger wandert nun von Bit zu Bit weiter. Falls er auf eine Page kommt, wo das Use Bit mit 1 gesetzt ist, stellt er dieses auf 0 um und wandert weiter. Falls das Use Bit 0 aufweist, wird dieser Frame ersetzt und das Use Bit auf 1 gesetzt und der Zeiger geht um eine Stelle weiter.

Was ist das Working-Set? (7)

Die variable Allokation von Frames für einen Prozess, basierend auf Lokalitätsannahme. Das Working-Set sind alle aktiven Pages eines Prozesses. Anhand einer Funktion $W(D, t)$ wird schrittweise immer wieder die Aktivität von Frames überprüft und inaktive Pages werden dementsprechend aus dem Working-Set gelöscht.

Welche Scheduling-Ebenen gibt es? (4)

- **Long-term Scheduling**
  Welche Prozesse sollen kreiert werden? Bestimmt den Grad der Paralellität.
- **Mid-term Scheduling**
  Kümmert sich um das Memory-Management sowie Virtual Memory.
- **Short-Term Scheduling**
  Welcher Prozess bekommt jetzt CPU-Zeit.
  Aktivierung des Dispatchers, wenn Prozessumschaltung angebracht sein kann:
  - System Calls und Traps
  - I/O-Interrupts, Signale
  - Clock-Interrupt

Scheduling

Round Robin und Virtual Round Robin – Scheduling (12)

Round Robin ist eine Scheduling Technik, welche die gleiche Selection Funktion hat wie FCFS. Der Decision Mode ist aber preemtive. Es können auch kurze Prozesse fair behandelt werden, ohne die Service Time zu kennen. Die Prozesse bekommen fixe Zeitscheiben, in denen sie arbeiten können, danach wird zum nächsten Prozess in der Queue gewechselt. Dies lässt im Gegensatz zu FCFS auch kurze Prozesse gut zum Zug kommen. Es gibt aber einen Nachteil, da I/O-Intensive Prozesse benachteiligt werden. Diese müssen sich nach jeder I/O-Interaktion, auf die sie warten müssen, wieder in die Queue stellen.

Virtual Round Robin löst dieses Problem, indem es eine weitere Queue, die *Auxilary Queue*, hinzufügt, wo sich Prozesse, nachdem sie ihre I/O-Sachen erledigt haben, wieder queuen können. Diese kommen dann mit einer höheren Priorität zur CPU.

Shortest Process Next – Scheduling (1)

Hier wird für die Selection Funktion die Prozesse präferiert, welche die kürzeste erwarteten CPU-Burst haben. Der Decision Mode ist non-preemptive. Es entstehen bessere Response Times als bei FCFS. Nachteile sind aber das, die Service Time geschätzt werden muss und das lange Prozesse in die Starvation kommen können.

Shortest Remaining Time – Scheduling (1)

Die Selection Function ist die gleiche wie bei SRN. Der Decision Mode ist aber preemtive. Es kommt zu nicht so vielen Interrupts wie Round Robin. Die Service Times müssen aber mitprotokolliert werden, und Starvation ist immer noch möglich.

Highest Response Ratio Next – Scheduling (8)

Die HRRN rechnet sich einen Wert aus, welcher für die Selection Function verwendet wird. Der Decision Mode ist non-preemtive. Dieser wird sich folgendermaßen ausgerechnet.

$$RR = \frac{(w+s)}{s} \tag{2}$$

w ist die Wartezeit die ein Prozess bis jetzt schon gehabt hat. s ist die geschätzte Service Time.

So werden kürzere Prozesse fair behandelt und es erfolgt keine Starvation. Es muss aber die Service Time geschätzt werden.

Feedback Scheduling (4)

Im Feedback Scheduling basiert die Selection Function auf der bisherigen Ausführungszeit. Je größer die Ausführungszeit ist, desto niedriger wird die Priorität des Prozesses, es erhöht sich aber die Länge der Zeitscheibe, welche ein Prozess bekommt. Eigene Queue für jede Prioritätsstufe. Ein neuer Prozess hat eine hohe Priorität, dafür aber eine kurze Zeitscheibe. Starvation langer Prozesse kann aber passieren, weswegen die Priorität nach einer Zeit angehoben werden muss.

Ein-/Ausgabe

In welche Schichten ist die I/O organisiert?

- **Logical I/O**
  stellt das Interface mit den Funktionen dar
- **Device I/O**
  Operationen werden in Gerätekommandos übersetzt
- **Scheduling & Control**
  Verarbeitung/Verwaltung von Gerätekommandos, Interaktion mit Hardware

Was ist I/O-Buffering? (5)

Ist das Zwischenspeichern von Daten bei einem I/O-Transfer. Das bietet mehrere Vorteile: es reduziert den Aufwand, ermöglicht eine Entkopplung von Prozess und Betriebssystem sowie eine Entkopplung von I/O und Swapping. Es kann vorübergehende Lastspitzen abfangen und kann die Geschwindigkeit des I/O Vorgangs erhöhen, solange der Puffer nicht voll ist. Es gibt aber auch Nachteile, wie die Notwendigkeit, Speicher zu reservieren und die Komplexität des Puffermanagements. Die Größe des Puffers begrenzt den Geschwindigkeitsgewinn und die langfristige Leistung wird durch die Geräteleistung limitiert.

File Management

Wie ist ein Dateisystem realisiert? (5)

Hier gibt es vier verschiedene Möglichkeiten, wie ein Dateisystem realisiert werden kann:

- **Contingous Allocation** (1)
  Dateien werden in einem durchgehenden, einzigen Teil auf die Platte geschrieben, welcher nicht unterbrochen wird.
  Pros: schnelle Lesezeiten
  Cons: Fragmentierung, wenn Dateien vergrößert werden müssen
- **Chained Allocation**
  Dateien werden auf Blöcken im HS gespeichert. Können irgendwo sein. Sind mittels Zeiger verbunden. Zeiger zeigt auf den nächsten Block.
  Pros: keine Fragmentierung
  Cons: langame Readspeeds, keine Lokalität, Pointer selbst benötigt Platz im Block
- **Indexed Allocation** (1)
  Wie Chained Allocation, nur dass die Zeiger in einer eigenen Tabelle sind (FAT - File Allocation Table). Dieser befindet sich im Hauptspeicher.
  Pros: keine Fragmentierung, sowohl sequenzielle und direkte Zugriffe gut unterstützt, Block ganz für Nutzdaten zur Verfügung
  Cons: FAT nimmt viel Platz im HS ein.
- **I-Nodes** (5)
  Datenstruktur, bei dem für jede Datei die Fileattribute (z.B. Flags, Security-Richtlinien) und Referenzen der Blöcke in den I-Nodes stehen. Wird für Unix verwendet.

  Verzeichnisse werden wie ganz normale I-Nodes gespeichert, nur das ein Flag gesetzt ist, welches Anzeigt das dieses I-Node ein Directory ist.
  Pros: Nur die I-Nodes welche gerade benötigt werden, müssen in den Speicher geladen werden. Blöcke ganz für Nutzdaten.
  Cons: I-Nodes haben nur eine gewisse Anzahl an Referenzen, weshalb bei großen Files Querverlinkungen auf andere I-Nodes sein könnten.

Disk Scheduling (1)

Beim Disk Scheduling geht es darum, in welcher Abfolge der Lesekopf die Prozessanfragen auf der Disk abarbeiten soll. Es gibt acht mögliche Taktiken für das Disk-Scheduling:

- **Priority**: Der Request mit der höchsten Priorität wird als nächster gescheduled.
- **FIFO**: Erster Request, der kommt, wird gescheduled.
- **LIFO**: Der letzte gerade angekommene Request wird gescheduled.
- **Shortest Service Time First**: Der Request wird gescheduled, der von der aktuellen Position des Lesekopfs am nächsten dran ist.
- **SCAN**: Auf dem Weg zum nächsten Request werden alle anderen Request am Weg auch abgearbeitet.
- **CSCAN**: Wie SCAN nur mit "Leerfahrt" bei der Rückfahrt von Request.
- **n-SCAN**: Request werden in Pakete der Größe n zusammengepackt, und arbeitet diese nacheinander ab.
- **FSCAN**: Hat zwei Gruppen von Request, die aktive Gruppe, welche gerade abgearbeitet wird. Request, die zu diesem Zeitpunkt reinkommen, werden zu einer anderen Gruppe gegeben. Nachdem die aktive Gruppe abgearbeitet ist, wird gewechselt.

Security

Was sind die drei Security Objectives (2)

- **C**onfidentiality: Wer kann die Daten sehen?
- **I**ntegrity: Daten entsprechen dem Sollzustand. Wer darf daten ändern?
- **A**vailability: Verfügbarkeit der Daten
- Autenticity: Korrektheit der Identität und berechtigter Zugriff
- Accountability: Nachvollziehbarkeit, Protokollieren von Zugriffen

Was ist eine Access Control List? (5)

Eine Access Control List ist eine Möglichkeit, Rechte für Domains zu speichern. Die Zugriffsrechte für die Domains sind direkt am Objekt gespeichert, also eine Spaltenzerlegung der Access Control Matrix. Dies ermöglicht leichtes Ändern der Zugriffsrechte von Objekten. Diese Access Control Lists werden bei der Ressourcenverwaltung durch User/Groups geschaffen.

Was ist eine Capability List? (6)

Mit dieser Strategie möchte man auch die leeren Felder einer Access Matrix umgehen. Für jeden Prozess gibt es eine List mit erlaubten Zugriffsoperationen auf die Objekte. Die Zugriffsrechte sind als Tickets realisiert und können weitergegeben/vererbt werden.

Was ist das Bell und LaPadula's Modell? (2)

Dies ist ein Modell für den Informationsfluss. Es wird eine Hierarchie von *Security Classifications (SC)* für Subjekte und Objekte aufgestellt. Weiters gibt es folgende **Security Axiome**:

- "no read up": Es darf nicht aus einer höheren SC gelesen werden. Es muss gelten $SC(S) \geq SC(O)$.
- "no write down": Es darf nicht in eine niedrigere SC geschrieben/geleakt werden. Es muss gelten $SC(S) \leq SC(O)$
- "read and write": Für read und write müssen beide die gleiche SC haben. Es muss gelten $SC(S) = SC(O)$

Schriftliche Fragen

Semaphoren – Wie kann man Semaphoren realisieren? (8)

```
init(S, value) {
  S.value = value;
  S.queue = new empty queue;
}
wait(S) {
  S.value = S.value-1;
  if (S.value < 0) {
    //add process to S.queue and block the process
  }
}
signal(S) {
  S.value = S.value+1;
```

```
    if(S.value <= 0) {
      //remove next process from S.queue and add it to the ready queue
    }
  }
```

Semaphoren – Zwei Reader ein Writer (9)

Die Reader müssen Gleichzeitig lesen können. Jeder Reader darf einen Wert nur genau einmal lesen. Danach muss wieder geschrieben werden.

```
  //init
  init(A, 2); init(B1, 0); init(B2, 0);

  //writer
  loop
    generate data;
    wait(A);
    wait(A);
    add data to shMem;
    signal(B1);
    signal(B2);
  end loop;

  // reader1
  loop
    wait(B1);
    x = get data from shMem;
    signal(A);
    use x;
  end loop;

  // reader2
  loop
    wait(B2);
    x = get data from shMem;
    signal(A);
```

```
    use x;
  end loop;
```

Semaphoren - Conditional Sync (2)

Drei Prozesse mit Operation a(), b(), c() sollen nacheinander aufgerufen werden.

```
  //init
  init(A, 1); init(B, 0); init(C, 0);

  // Process A
  loop;
    wait(A);
    a();
    signal(B);
  end loop;

  // Process B
  loop;
    wait(B);
    b();
    signal(C);
  end loop;

  // Process C
  loop;
    wait(C);
    C();
    signal(A);
  end loop;
```

Producer Consumer mit Ringbuffer (Fixe Größe K) (13)

```
  //init
  init(S, 1); init(N, 0); init(E, K);
  in = out = 0;
```

```
// funcs
append(w) {
  b[in] = w;
  in = (in+1) % K;
}
take() {
  w = b[out];
  out = (out+1) % K;
  return w;
}

//producer
loop;
  x = generate data;
  P(E);
  P(S);
  append(x);
  V(S);
  V(N);
end loop;

//consumer
loop;
  P(N);
  P(S);
  x = take();
  V(S);
  V(E);
  use x;
end loop;
```

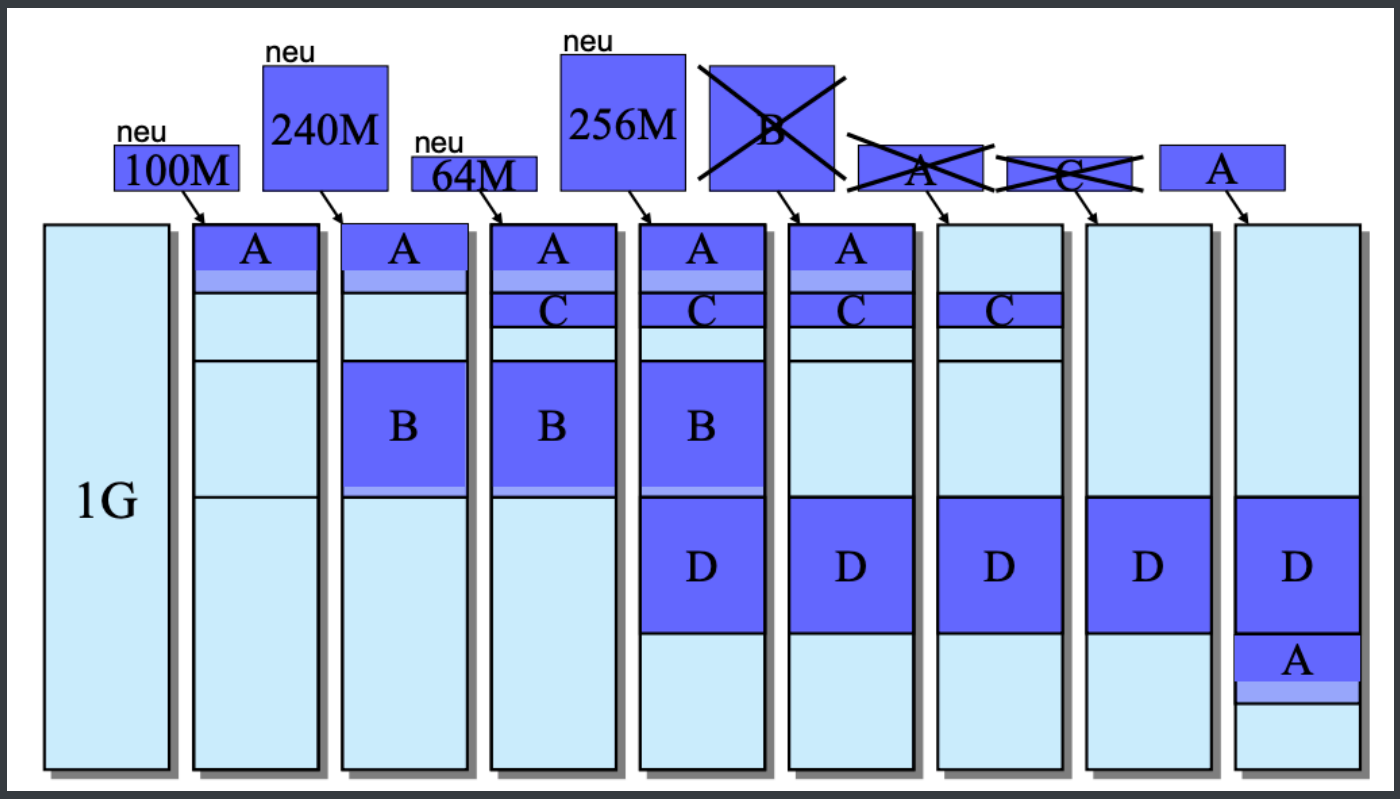
Reader Writer Problem (hier haben Writer Priorität) (13)

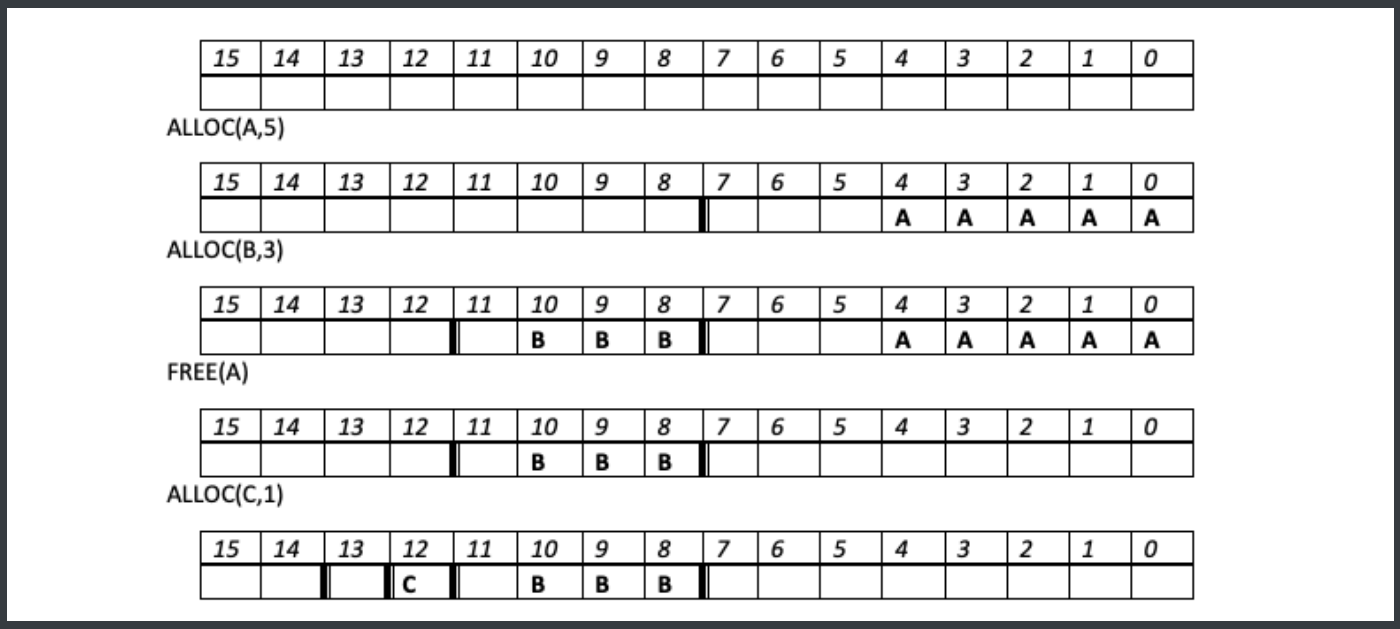
```
//init
init(x,1); init(y,1); init(z,1); init(wsem,1); init(rsem,1);
rc = wc = 0;

//reader
loop;
  P(z);
  P(rsem)
  P(x);
  rc = rc+1;
  if(rc == 1) P(wsem);
  V(x);
  V(rsem);
  V(z);

  read();

  P(x);
  rc = rc-1;
  if(rc == 0) V(wsem);
  V(x);
end loop;

//writer
loop;
  P(rsem);
  P(wsem);
  write();
  V(wsem);
  V(rsem);
end loop;
```

Prozessfortschrittsdiagramm (3)

Buddy System (4)

Der Speicherplatz wird in 2er-Potenzen aufgeteilt, bis der kleinstmögliche Bereich für den Prozess erreicht wurde. Anschließend kommen weitere Prozesse in den passenden des Speichers, wobei alle Bereiche aufgrund der Zerteilung die Größe einer 2er-Potenz haben.





Bankers Algorithm (7)

Im Bankers Algorithm geht es um die Ressource Allocation Denial zu entscheiden.

Bankers Algorithm

$$R = (8\ 3\ 6) \qquad V = (1\ 1\ 2) \qquad Q_2 = (1\ 0\ 1)$$

$$C = \begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 0 \\ 5 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad N = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 0 & 2 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix}$$

$$W = (0\ 1\ 1) \quad C = \begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 0 \\ 6 & 1 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad N = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 0 & 1 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix} \leftarrow$$

$$W = (6\ 2\ 3) \quad C = \begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad N = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 0 & 0 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix} \leftarrow$$

$$W = (7\ 2\ 3) \quad C = \begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad N = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix} \leftarrow$$

$$W = (8\ 3\ 4) \quad C = \begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{pmatrix} \quad N = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 4 & 2 & 0 \end{pmatrix} \leftarrow$$

$$W = (8\ 3\ 6) \quad C = \begin{pmatrix} 3 & 2 & 2 \\ 6 & 1 & 3 \\ 3 & 1 & 4 \\ 4 & 2 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad N = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$\Longrightarrow$ safe state

Monitor (2)

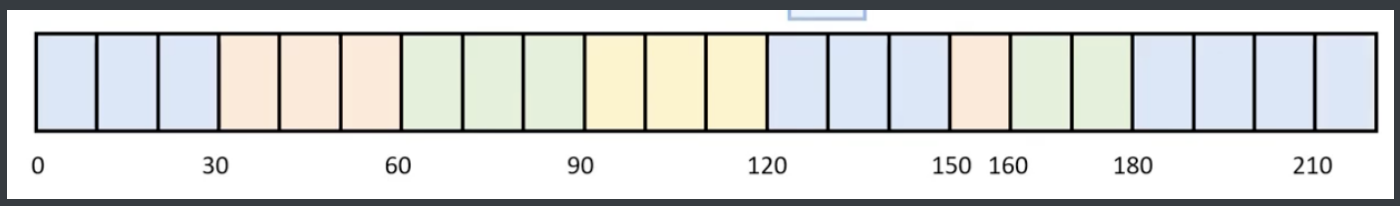
Benötigte Variablen erstellen, richtig initialisieren und eine Funktion "besuchen()" schreiben (maximal M Personen durften gleichzeitig besuchen)

```
atomic int x = M;
condition c = x > 0;

besuchen() {
   x--;
   cwait(c);
   // do stuff
   x++;
   csignal(c)
}
```

Deadlock Detection Algorithmus (1)

$$Q = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad V = (0\ \ 0\ \ 0\ \ 0\ \ 1)$$

markiere P_4 (belegt keine Ressourcen)
$W := (0\ 0\ 0\ 0\ 1)$
Q_{3i} (Anforderung von P_3) <= W_i , daher markiere P3 und setze $W := W + (0\ 0\ 0\ 1\ 0) \rightarrow (0\ 0\ 0\ 1\ 1)$
Algorithmus terminiert, liefert Deadlock von P_1 und P_2

Round Robin Scheduling (4)

Quantum: 30ms