Technische Informatik 2
Fragenkatalog
WiSe 2018/2019
Universität Bremen

26. Februar 2021

Seit jeher gibt es auf den **Übungsblättern** des Kurses *Technische Informatik 2* der Universität Bremen Zusatzfragen, die der Vorbereitung auf das **Fachgespräch** dienen sollen. Nachfolgend finden Sie die von freiwilligen Studenten und Tutoren zusammengetragenen Antworten auf diese Fragen.

Jeder kann dazu beitragen, die Antworten zu erweitern, zu verbessern oder zu aktualisieren, indem er das Repository auf Github cloned, das Dokument überarbeitet und anschließend Merge-Requests einreicht. Das Repository befindet sich unter https://github.com/momesana/ti2-fragenkatalog/.

Inhaltsverzeichnis

1 Betriebssysteme - Grundlagen

1.1 Welche zwei Hauptaufgaben hat ein Betriebssytem?

- Abstraktion von Geräteeigenschaften
  - Geräteunabhängige Schnittstelle zu den Anwendungen
  - Geräteüberwachung und -steuerung
- Unterstützung des Mehrbenutzerbetriebs
  - Betriebsmittelverwaltung
  - Zuteilungsstrategien
  - Kostenabrechnung
  - Schutz

1.2 Was ist ein Prozess?

Ein Programm in Ausführung.

1.3 Was sind Dateien?

Dateien sind langlebige, über eindeutige Namen (Pfade) identifizierbare Datenobjekte. Prozesse können auf sie zugreifen und sie modifizieren (wenn die nötige Berechtigung gegeben ist). Für UNIX stellt der Inhalt lediglich eine Bytefolge dar (beliebige interne Struktur).

1.4 Welche Arten von Dateien gibt es?

- Normale Dateien (plain files)
- Verzeichnisse
- Symbolische Links
- Geräte-Dateien
  - Blockorientierte Gerätedateien
  - Zeichenorientierte Gerätedateien
- Sockets (3.5) und Named Pipes (3.5)

1.5 Wie ist ein UNIX-Dateisystem strukturiert? Wie können Dateien darin (eindeutig) aufgefunden werden?

Ein UNIX-Dateisystem ist eine hierachisch organisierte Verzeichniss-Struktur. Ganz oben steht die Wurzel (root). In Vezeichnissen können Dateien abgelegt werden. Vollständige Dateinamen (absoluter Pfad) sind (eindeutige) Pfadnamen von der Wurzel abwärts. Relative Pfade sind ebenfalls möglich.

1.6 Was ist ein *symbolic link* (symbolischer Link)?

Ein *symbolic link* ist ein Verweis auf eine Datei oder auf ein Verzeichnis (, die nicht notwendigerweise existieren müssen). Es ist also lediglich eine Referenz auf die Zieldatei bzw. das Zielverzeichnis (d.h. ein weiterer Pfad zu einer Datei). Ein Löschen oder Verschieben der eigentlichen Datei oder das Verweisen auf eine nicht existierende Datei führen üblicherweise dazu, dass die Referenz „ins Leere“ weist (dangling link).

1.7 Was ist ein *hard link*?

Ein Hardlink ist im Grunde genommen ein weiterer, regulärer Name der Datei (Zwei Namen für die selbe Datei). Wird also die Datei unter ihrem ursprünglichen Namen gelöscht, ist sie unter ihrem zweiten Namen immer noch vorhanden. Erst wenn alle "Namen"gelöscht sind ist die Datei nicht mehr zugreifbar. Intern wird dies durch Reference-Counting erreicht. Hardlinks können nicht auf Verzeichnisse gelegt werden (Vermeidung unkontrollierte Schleifen).

1.8 Ist das UNIX-Dateisystem wirklich ein Baum? Begründung.

Ein UNIX-Dateisystem entspricht eher einem gerichteten Graphen als einem Baum. Verzeichnisse und Dateien sind zwar hirarchisch organisiert, aber Möglichkeit von Hardlinks, d.h. das mehrfache Vorhandensein in der Dateistruktur einer physikalisch nur einfach vorhanden Datei stört das Bild eines Baumes. In einem Baum werden die Knoten benannt, unter UNIX werden Pfadnamen benutzt.

1.9 Welche Zugriffsrechte kann man auf eine UNIX-Datei haben? Welche Dateiattribute steuern dies, und wie?

Die Zugriffsrechte sind Lesen, Schreiben und Ausführen. Es können unterschiedliche Rechte für den Besitzer (user) einer Datei, die Gruppe (group) und den Rest der Welt (others) festgelegt werden. Zusätzlich gibt es noch die erweiterten Rechte Setuid, Setgid und das Sticky Bit. [1]

Detaillierte Auflistung der Zugriffsrechte:

	Erweiterte Rechte			User (Owner)			Group	Others
	SUID	SGID	Sticky	r	w	x	r w x	r w x
Plain file	Ausführen: Benutzer vs. Besitzer der Datei	Ausführen: Benutzer vs. Dateigruppe	(Bleibt im Swap Space kleben)	Lesen	Schreiben	Ausführen	(dito)	(dito)
Directory		Dateien darin erhalten Gruppe der Dir	Nur eigene Dateien in Dir löschbar (bsp.: /tmp)	Einträge lesen	Einträge (indirekt) ändern	Auf Dateien darin zugreifen	(dito)	(dito)

[1] http://de.wikipedia.org/wiki/Unix-Dateirechte#Sonderrechte.2Ferweiterte_Rechte

1.10 Welche Vorteile bietet es, auf Terminals in UNIX wie auf Dateien zuzugreifen? Was versteht man unter Ein-/Ausgabeumlenkung?

Da unter **UNIX** der Zugriff auf Geräte im allgemeinen (d.h. auch auf Terminals) als Zugriff auf eine Dateien abgebildet sind, bietet dies den Vorteil der einheitlichen Schnittstelle und Handhabung. Der Benutzer kann auf das Terminal zugreifen wie auf eine Datei auch und muss sich daher nicht mit den konkreten Geräteeigenschaften auseinandersetzen.
Es gibt Standardein- und Standardausgaben von Prozessen. Will der Benutzer andere Ein- bzw. Ausgabemöglichkeiten nutzen, kann er Ein- und Ausgaben umlenken. Beispielsweise können die Fehlermeldungen in eine Datei umgelenkt werden.

```
1  user $ echo -e "#\!/bin/bash\necho Hello world" > /tmp/bla # (1)
2  user $ bash < /tmp/bla # (2)
3  Helloworld
```

Listing 1: Ein-/Ausgabeumlenkung

1. Die Ausgabe des Befehls `echo` wird in eine Datei `/tmp/bla` geschrieben, und somit eine Ausgabeumlenkung getätigt.

2. Unter (2) liest die neu erzeugte shell von der soeben erstellten Datei `/tmp/bla` anstelle des Terminal, was einer Eingabeumlenkung entspricht.

1.11 Welche Aufgabe hat ein Kommando-Interpreter (z.B. in UNIX die Shell)?

Ein Kommando-Interpreter stellt dem Benutzer eine einfache (und zugleich mächtige) Schnittstelle zum Betriebssystem (Benutzungsschnittstelle) zur Verfügung. Damit lassen sich Aufträge an das Betriebssystem absetzen. Die Shell ist ebenfalls ein Prozess.

1.12 Was machst Du, wenn Dir die genaue Semantik eines Unix-Kommandos entfallen ist?

Zunächst die Hilfe (mit dem Kommandozeilenswitch –help), und wenn erforderliche die Man- oder Infopages zur Rate ziehen:

```
1  user $ mkdir --help
2  user $ man mkdir
3  user $ info mkdir
```

Listing 2: Hilfe zu einem Kommando am Beispiel von `mkdir`

1.13 `bla` sei ein ausführbares Programm. Was ist der Unterschied zwischen dem Aufruf

```
1  bla
```

und dem Aufruf

```
bla &
```

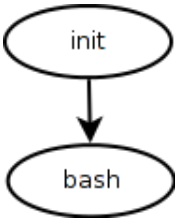
in der Shell? Welche Auswirkungen hat dies, wenn `bla` von **Standard Input** liest bzw. auf **Standard Output** schreibt?

`bla` wird im Vordergrund ausgeführt und beansprucht das Terminal (=Tastatur) für sich. Weitere Shell-Eingaben sind daher erst möglich, nachdem `bla` terminiert.

Hängt man einem Befehl ein & an, so wird dieser im Hintergrund ausgeführt (Hintergrundprozess). Er gibt damit die Kontrolle über das Terminal auf, so dass dieses für neue Prozesse verwendet werden kann. Dieser kann dann zwar auf `stdout` schreiben, wird aber sofort in den Zustand suspended versetzt , wenn er versucht von der Standardeingabe (*stdin*) zu lesen.

1.14 Prozessbäume: Gegeben sei der folgende Prozessbaum:

Abbildung 1: Prozessbaum



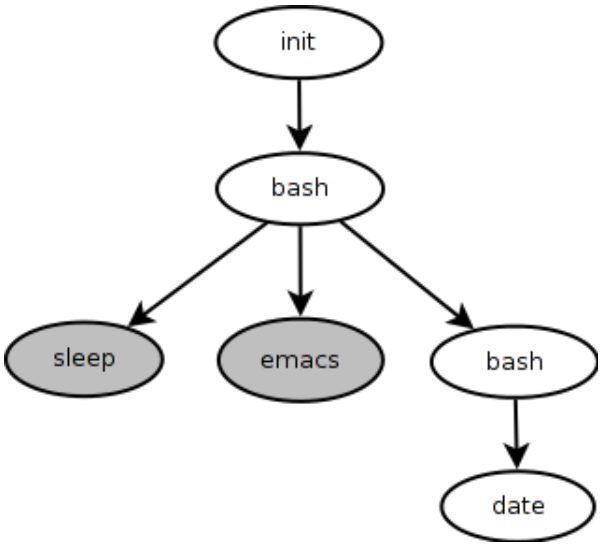
Wie ändert er sich nach Eingeben der folgenden Kommandofolge im Shell?

```
sleep 1000 &
emacs &
bash
date
```

Listing 3: Eingegebene Befehle

Der Prozessbaum sieht nach Eingabe der obigen Befehle folgendermaßen aus:

Abbildung 2: Prozessbaum nach den Eingaben



`sleep` und `emacs` laufen dabei als Hintergrundprozesse.

1.15 Nenne drei Beispiele für Informationen, die der Betriebsystemkern über einen Prozess wissen muss.

- Prozess ID
- Parent-Prozess ID
- Zustand des Prozesses

1.16 Was ist eine Pipe?

Eine Pipe ist die Umlenkung eines unidirektionalen sequentiellen Bitstroms von der Standardausgabe eines Prozesses auf die Standardeingabe eines Prozesses.

1.17 Wie macht man ein soeben editiertes Shell-File ausführbar?

Setzen des Ausführbar-Bits (executable bit) (x).

```
1    chmod +x shellfile.sh
```

Listing 4: Setzen des **executable bit** für ein Shellskript *shellfile.sh*

1.18 In welche Bereiche (Segmente) ist der virtuelle Adressraum eines Programmes in Ausführung unter UNIX unterteilt, und welche Eigenschaften kennzeichnen sie?

Der virtuelle Adressraum ist in Text, Data und Stack unterteilt. Im Text-Teil befindet sich der eigentliche Programmcode, im Data-Teil die Daten und auf dem Stack alle temporären Daten (z.B. Variablen).

1.19 Wozu wird der Stack verwendet?

Auf einem Stack werden die innerhalb von Funktionsaufrufen temporär anfallenden Daten eines Prozesses abgelegt. Das umfasst unter anderem die Rücksprung-Addresse, die an die Funktion übergebenen Argumente und lokale Variablen.

1.20 Welchem Zweck dienen Bibliotheken (Libraries)?

Bibliotheken stellen Funktionen zur Verfügung. Durch das Vorhandensein von Bibliotheken muss ein Programmierer nicht von Grund auf alle Funktionen neu schreiben.

Eine Programmbibliothek bezeichnet in der Programmierung eine Sammlung von Softwaremodulen, die Lösungswege für thematisch zusammengehörende Problemstellungen anbieten. Bibliotheken sind im Unterschied zu Programmen keine eigenständig lauffähigen Einheiten, sondern Hilfsmodule, die von Programmen angefordert werden. [2]

1.21 Welche Aufgabe erfüllt ein Linker?

Ein Linker oder Binder verbindet einzelne Programmmodule zu einem ausführbaren Programm. Im Zuge dessen werden die Symbole in den Objectfiles aufgelöst. Man unterscheidet zwischen statischem und dynamischem Linken.

Statisches Linken: Die Symbole schon während des Zusammenstellens der Programmdatei aufgelöst

Dynamisches Linken: Das Auflösen der Symbole erfolgt erst bei der tatsächlichen Ausführung des Programms.

1.22 Wozu wird beim Assemblieren eine Symboltabelle angelegt?

Zum Auflösen von Verweisen zu anderen Programmteilen.

[2] http://de.wikipedia.org/wiki/Programmbibliothek

1.23 Welchen Vorteil hat es, Bibliotheken mit *Position Independent Code* zu versehen?

Position-independent Code (PIC, engl. für positionsunabhängiger Code) ist Maschinencode, der ausgeführt werden kann, unabhängig davon, an welcher absoluten Adresse im Hauptspeicher er sich befindet. PIC wird üblicherweise für dynamische Bibliotheken verwendet, damit diese für jedes Programm an eine Speicherposition geladen werden können, wo sie sich nicht mit anderen Objekten dieses Programms überlappen. [3]

1.24 Durch welche "Qualitätsmerkmale“ sollten Betriebssysteme gekennzeichnet sein? Nenne Beispiele für konkurrierende Anforderungen.

Effizienz, Kosten, Bedienbarkeit, Zuverlässigkeit, Verfügbarkeit, Sicherheit, Wartbarkeit. Der Kostenpunkt kann mit jeder anderen Anforderung konkurrieren. Je besser ein Aspekt werden soll, desto mehr Zeit und Entwicklungsarbeit, sprich Geld muss in diesen Aspekt investiert werden. Am Beispiel Windows lässt sich wunderbar erkennen, wie Bedienbarkeit und Zuverlässigkeit im Widerspruch zueinander stehen können. Die Sicherheit eines OS's lässt sich oft auch nicht mit einer ausgereiften Bedienbarkeit vereinbaren, oder umgekehrt, siehe Win vs. Linux.

1.25 Worin unterscheidet sich der Kernel-Mode vom User-Mode (in UNIX)? Warum wird diese Unterscheidung getroffen?

Kernel und Useradressraum sind aus Sicherheitsgründen getrennt. Im User-Mode gibt es keinen direkten Zugriff auf die Hardware. Das hat auch seinen Sinn, da diese Zugriffe kontrolliert erfolgen sollen, damit das System sicher und zuverlässig bleibt.

1.26 Was passiert etwa bei einem System-Aufruf? (Reihenfolge der Arbeitsschritte.)

Es erfolgt ein Trap, darauf wird der Zustand des Systems gesichert, dann erfolgt der Sprung in den Kernel-Mode, in dem der eigentliche Hardware-Zugriff erfolgt. Daraufhin gibt es eine Rückmeldung und einen Rücksprung zum Prozess, wo dann auch der Kontext wieder hergestellt wird.

1.27 Was ist ein Interrupt? Nenne Beispiele für mögliche Interrupt-Quellen. Warum werden sie unterschiedlich priorisiert? Wie wird ein Interrupt in etwa behandelt?

Ein Interrupt ist eine externe Unterbrechung eines Prozesses. Eine Quelle kann z.B. die Hardware sein. Hardwareinterrupts haben generell eine höhere Priorität als Softwareinterrupts. Beim Clock-Tick ist die Quelle eben die Hardware, und der Prozess hat eine sehr hohe Priorität, da ansonsten die System-Uhr nicht mehr genau laufen würde, was sich fatal auf andere Programme auswirken kann, die sich nach dieser Uhr zu richten

[3] http://de.wikipedia.org/wiki/Position-Independent_Code

haben. Beispiele für einen Softwareinterrupt wären SIGKILL oder SIGSTOP. Der Interrupt entsteht und löst im Prozessor einen Interrupt Request (IRQ) aus. Daraufhin wird der aktuelle Befehlszyklus zu Ende ausgeführt und es erfolgt ein Sprung in die Interrupt Service Routine (ISR). Die ISR sichert die für sich benötigten Register, führt den entsprechenden Code aus und stellt daraufhin die gesicherten Register wieder her. Zum Schluss erfolgt ein Rücksprung zum zuvor ausgeführten Code.

1.28 Was ist ein Trap? Nenne Beispiele. Inwiefern unterscheiden sich Traps von Interrupts?

Ein Trap ist eine interne Unterbrechung eines Programms in Ausführung. Beim Trap wird in den Kernel-Mode gewechselt, da es sich um einen Systemaufruf oder ein Problem/undefiniertes Ereignis handelt, wofür das BS eingreifen muss. Die Behandlung erfolgt durch den Trap-Handler, der bei Bedarf dann auch Signale senden kann. Division durch 0 wäre ein solches Beispiel. Es wird ein Trap ausgelöst, der durch den Trap-Handler gejagt wird, welcher wiederum das Signal sigfpe lossendet (SignalFloating- PointError). Die Folge ist, wie bei vielen anderen Signalen auch, dass der Prozess terminert oder gekillt wird.

1.29 Was ist ein Signal? Nenne Beispiele für mögliche Signalquellen. Wie kann ein Prozess auf ein Signal reagieren?

Signale dienen der Meldung (von Ausnahmezuständen) an einen Prozess. Sowohl der Benutzer als auch Prozesse können Signale an Prozesse senden. Beispiele für Signale sind u.a. Stop (SIGINT), Kill (SIGKILL) und Terminate (SIGTERM). Der Traphandler ist oft eine Signalquelle.

1.30 Beschreibe kurz einige Zustände, in denen sich ein (UNIX-)Prozess befinden kann.

Einfaches Modell:

ready
: Der Prozess wartet auf Zuteilung der CPU (Zeitschlitz). Gibt es den Ready-Zustand, so befinden sich höchstens so viele Prozesse im Zustand running, wie CPUs vorhanden sind.

running
: Entweder genau der Prozess, der gerade bearbeitet wird, oder alle Prozesse, die momentan Rechenarbeit verrichten können.

sleep
: Der Prozess wurde auf eigenen Wunsch zurückgestellt. Er kann Signale entgegennehmen, wie z. B. Timer, oder Ergebnisse von Kindprozessen.

zombie
: Der Prozess wurde beendet und aus dem Arbeitsspeicher gelöscht, aber noch nicht aus der Prozessliste entfernt.

Mit erweitertem Modell:

dead
: Der Prozess wurde beendet, er belegt jedoch noch Speicherplatz.

trace
: Der Prozess wurde von außen angehalten, üblicherweise durch einen Debugger.

wait
: Der Prozess wartet auf ein Ereignis, üblicherweise eine Benutzereingabe.

uninterruptible sleep
: Der Prozess wartet auf ein Ereignis, üblicherweise Hardware. Tritt dieses Ereignis ein, ohne dass der anfragende Prozess es entgegennimmt, so kann das System instabil werden.

1.31 Nenne einige Randbedingungen, auf die man beim Entwurf eines Schedulers achten sollte. Wie sollten rechenintensive bzw. Ein-/Ausgabe-intensive Prozesse dabei behandelt werden?

Ein Scheduler soll schlank (effizient) und fair sein, d. h. der Rechenaufwand der Prozessverwaltung darf nicht so groß werden, dass die Effizienz darunter leidet, und jeder Prozess soll gerechterweise auch drankommen. Auch die Zeitscheiben sollen eine angemessene Grösse haben.

1.32 Wie könnte man mit Hilfe eines Round-Robin-Schedulers Prozess-Prioritäten "simulieren"?

Man könnte die Anzahl der Durchläufe für die Prozesse mit einer hohen Priorität erhöhen, so dass diese in einem Durchlauf mehrere Male dran kommen, oder man gibt den wichtigen Prozessen grössere Zeitscheiben, um ihnen mehr CPU-Zeit zukommen zu lassen.

1.33 Warum bestehen die *Sleep-* und die *Run-Queue* in UNIX nicht aus jeweils einer einzigen Warteschlange? Wie sind sie stattdessen organisiert?

Sleep-Queue und Run-Queue bestehen aus jeweils einem Array. Jede Priorität hat darin eine eigene Liste, das Handling zwischen den Queues ist abhängig von der Priorität. Diese Aufteilung besteht, da sonst immer alles durchsucht werden müsste.

1.34 Warum werden die Zustandsinformationen eines UNIX-Prozesses teilweise in der *Proc-Struktur* und teilweise in der *User-Struktur* abgelegt? Nenne jeweils drei charakteristische Beispiele für Angaben darin.

Die in der Proc-Struktur abgelegten Zustandsinformationen sind für alle Prozesse verfügbar, die in der User-Struktur abgelegten sind nur für den laufenden Prozess verfügbar. Beispiele für Angaben in der Proc-Struktur: Signale, die zur Verarbeitung anstehen Prozesszustände Scheduling Beispiele für Angaben in der User-Struktur: Zugriffsrechte Aktuelles Verzeichnis Geöffnete Dateien.

1.35 Skizziere kurz die Prozesserzeugung in UNIX. Welche Rolle spielen die Systemaufrufe fork() und exec() dabei? Wie unterscheiden sich Vater- und Kindprozess voneinander?

Der Prozess Nummer Eins (init) wird „von Hand" beim Systemstart erzeugt (standardmäßig `/sbin/init`, andere Pfade können dem Kernel im Boot-Loader als Parameter übergeben werden). Andere Prozesse werden bei Bedarf als Kindprozesse mit fork() (oft als Kind-Prozesse der Shell) erzeugt. So entsteht eine Hierachie von Prozessen. Der Kindprozess erhält eine neue Prozess-ID, ist aber zunächst eine Kopie des Vaters, d.h. der „gleiche" Adressraum mit gleicher Aufteilung (d.h., gleiche Segmente) von Text, Data und Stack. Normalerweise soll das Kind ein anderes Programm ausführen. Dieses wird mit exec() gestartet. Damit wird dann auch der Adressraum ausgetauscht.

1.36 Wie erfährt ein UNIX-Prozess, ob ein Kindprozess terminiert ist? Wozu gibt es in UNIX den Prozesszustand SZOMB („Zombie")?

Das Kind meldet bei der Termination das Signal SIGCHLD an den Vater, dieser behnadelt durch den Aufruf wait() den Kindprozess und der Kindprozess terminiert oder der Vater ignoriert das Signal. In diesem Fall wird das Kind zum Zombie. Der Zombie wird dann entweder später vom Vater behandelt oder (je nach UNIX-Variante) durch das Betriebssystem. Z.B. kann der INIT-Prozess die Proc-Struktur wieder freigeben.

1.37 Wie werden mehrere Prozesse in einer Einprozessormaschine verwaltet?

In einer Einprozessormaschine kann zu jedem Zeitpunkt nur ein Prozess in Ausführung sein. Damit alle Programme einmal an die Reihe kommen und ausgeführt werden, gibt es den Scheduler. Dieser „simuliert" auf Einprozessormaschinen Parallelität, indem jeder Prozess nur für einen begrenzten Zeitraum („Zeitfenster") ausgeführt wird.

Es gibt innerhalb des Kernels zwei Warteschlangen: die Run- und die Sleep-Queue. Letztere kommt zum Tragen, wenn ein Prozess sich schlafenlegt (zum Beispiel mit `sleep()`). In diesem Fall wäre es unsinnig, den Prozess weiter auszuführen. Stattdessen findet ein Kontextwechsel statt und ein anderer Prozess wird ausgeführt. Für letzteres existiert die Run-Queue. In dieser befinden sich alle fortzuführenden Prozesse.

Für die Verwaltung der Warteschlangen gibt es unterschiedliche Strategien: FIFO und Shortest Job Next. Bei ersterem werden die Prozesse nach und nach eingereiht, während

der erste immer bearbeitet wird. Bei Shortest Job Next sind alle Prozesse nach ihrer approximierten Bearbeitungszeit sortiert.

Besser sind jedoch Zeitscheiben-orientierte Verfahren. Prozesse werden hier für eine begrenzte Zahl an Sekunden ausgeführt (bei Unterbrechungen wie `sleep()` kürzer). Ein solches Verfahren ist Round Robin. Es gibt Variationen, bspw. unterschiedlich lange Zeitscheiben oder mehrfaches Einreihen in Run-Queue.

Ebenfalls gibt es die Möglichkeit des Schedulings nach Prioritäten, die sich aus dynamischen und statischen Faktoren berechnen ließen.

1.38 Welche Scheduling-Strategie ist empfehlenswert? Was wird bei Unix-Systemen genutzt?

In Unix findet eine Kombination der genannten Methoden statt: zunächst wird die CPU nach Ablauf der Zeitscheibe abgegeben. Bei Warten auf Ereignisse wird die CPU ebenfalls abgegeben. Es gibt eine Basispriorität (`nice()`), welche in der Proc-Struktur gespeichert wird. Diese berechnet sich teilweise aus dem CPU-Konto, welches die bereits zugesagten CPU-Zeiten führt. Die Zeit vom CPU-Konto wird automatisch reduziert, damit Prozesse die lange laufen, nicht benachteiligt werden. Wartende Prozesse befinden sich in der Sleep-Queue und werden aufgeweckt, wenn das Ereignis eintritt. In dem Fall werden sie wieder in die Run-Queue eingereiht. Die Sleep-Queue ist über den Waiting-Channel indiziert während die Run-Queue über die Priorität indiziert ist. Damit mehrere Prozesse auf den selben Waiting-Channel warten können bzw. die gleiche Prioritäten haben können, sind die Queues als Arrays von Listen (genauer: Hash-Tabellen) organisiert.

1.39 Welche Vor- und Nachteile hat der First-Fit- bzw. der Best-Fit-Algorithmus zur Speicherverwaltung? Wie funktioniert der Buddy-Algorithmus in etwa?

First-Fit-Algorithmus: Der erste Block, der groß genug ist, wird verwendet indem er in zwei Teile gespalten wird (gross genug für die Anforderung und Rest). Das führt zu einer Ansammlung von kleinen Blöcken am Anfang der Liste, es werden immer mehr Blöcke und sie werden im Mittel immer kleiner. Das führt zu im Mittel längerer Suche, kleine Anforderungen können schnell erfüllt werden.

Best-Fit-Algorithmus: Die Liste wird immer ganz durchsucht, der Block mit dem kleinsten Verschnitt wird ausgew ählt. Es entstehen extrem viele kleine Blöcke. Die Suche dauert sehr lange da immer die ganze Liste durchsucht wird. Es sind lange auch grosse Blöcke verfügbar.

Buddy-Algorithmus: Blöcke sind immer 2^k groß. Bei der Freigabe eines Blockes wird er, wann immer sein Nachbar (Buddy) frei ist, mit dem Nachbarn verschmolzen. Das vereinfacht das splitten und zusammenfügen der Blöcke. Allerding entsteht freier Speicher innerhalb der Blöcke. Optimierung: mehrere Listen mit jeweils gleich grossen Blöcken, das verringert die Suchdauer.

1.40 Wozu bieten Systeme eine Speicherhierachie an? Welche Beobachtung über den Speicherzugriff realer Programme liegt dem zugrunde? Welche verschiedenen Arten von Speicher werden typischerweise bereitgestellt?

Die Systeme bieten eine Speicherhierachie an, da die Geschwindigkeit sich proportional zu den Kosten verhält - der schnellste Speicher ist der teuerste. Reale Programme greifen oft nur auf eine sehr kleine Bereich zu (Working Set). Es werden meist Cache, Hauptspeicher und Massenspeicher bereitgestellt.

1.41 Warum ist es in der Regel nicht sinnvoll, den Adressraum eines Prozesses in einem Stück im Speicher abzulegen?

Ein Prozess muss nicht zwangsläufig komplett im Speicher verfügbar sein (s.o.:Working Set). Bei der Aufteilung in kleinere Einheiten ist auch die Nutzung kleinerer Freispeichereinheiten möglich, heisst insgesamt eine erheblich bessere Nutzung des Haupspeichers.

1.42 Was versteht man unter *Paging*, was unter *Segmentierung*? Wo tritt *interne Fragmentierung*, wo *externe Fragmentierung* auf? Was versteht man darunter?

Segmentierung: Speicher wird in verschieden große Stücke unterteilt, das ermöglicht die flexible Zuteilung kleiner Speicherbereiche und Shared Memory. Es sind Adressumsetzungstabellen pro Prozess notwendig. Segmente enthalten unterschiedlich viele Pages und sind maximal 210 Byte groß. Segmentierung hat das Problem der externen Fragmentierung.

Paging: flexible Zuteilung von gleich grossen Speichereinheiten. Der Hauptspeicher wird in Kacheln (Page-Frames) fester Größe unterteilt. Die Prozessadressräume wiederum liegen in Seiten (Pages) von gleicher Größe (meist 4 KiB). Auch dieses Verfahren ermöglicht Shared Memory, impliziert aber interne Fragmentierung (statt externer).

externe Fragmentierung: Wenn durch die angewendete Logik Speicher in unterschiedlich große Stücke eingeteilt wird und ganze Stücke freibleiben.

interne Fragmentierung: Wenn innerhalb von gleich großen Stücken der Speicher nicht restlos ausgenutzt wird und so Speicher ungenutzt bleibt.

1.43 Was ist Paging und warum macht man das? Wann tritt ein Page-Fault auf und was passiert dann?

Beim Paging wird der Hauptspeicher in Kacheln (Page-Frames) fester Größe unterteilt. Wird eine neue Page im Speicher benötigt und ist kein Page-Frame frei, kommt es zur Verdrängung. Wenn diese Page später angefordert wird, tritt ein Page-Fault auf, da sich diese Seite nicht im Working-Set des Hauptspeichers befindet.

Es gibt verschiedene Verdrängungsstrategien wie FIFO, LFU und LRU oder NRU.

1.44 Aus welchen Teilen besteht eine *virtuelle Adresse* zumeist? Wie ermittelt sich daraus die entsprechende Hauptspeicheradresse, d.h. wie läuft die Adressverwaltung in etwa ab?

Eine virtuelle Adresse besteht zumeist aus zwei Teilen: jPAGEjADDRj. Der erste Teil wird bei Gebrauch in der Page-Tabelle nachgeschaut und entsprechend ersetzt: jADDRjADDRj. Dieser Vorgang wird üblichwerweise durch Hardwareunterstützung realisiert: die Memory-Managment-Unit (MMU).

1.45 Wie können mehrere Prozesse mit Hilfe virtueller Adressierung auf dieselben Programmstücke (oder auch Datenbereiche) zugreifen?

Shared Memory: In den jeweiligen (unterschiedlichen) Page-Tabellen sind dieselben Seiten eingetragen. (Doppelter bzw. dreifacher Eintrag ein und desselben Inhalts in die 'Inhaltsverzeichnisse').

1.46 Wie arbeiten die folgenden Algorithmen zur Verdrängung von Seiten aus dem Hauptspeicher in etwa?

LRU (Least-Recently-Used): LRU entfernt die Seite im Hauptspeicher, auf die am längsten nicht mehr zugriffen wurde. Das Lokalitätsprinzip wird in der Regel gut erfasst. LRU ist eine gute Aproximation an den optimalen Algorithmus, aber das erfassen aller Zugriffszeiten auf die Seiten ist notwendig, d.h. bei jedem Zugriff müssen weitere Speicherzugriffe erfolgen. LRU ist zu aufwendig ohne Spezialhardware.

FIFO (First-in-First-Out): FIFO entfernt die älteste Seite und ist einfach zu realisieren: als verkettete Liste der Page Frames nach Belegungsalter. Aber: bestimmt das Working Set in der Regel nicht gut.

Second-Chance-FIFO: Funktioniert (im Prinzip) wie FIFO: als verkettete Liste nach Belegungsalter. Wenn eine Seite (nach FIFO) zum Löschen an der Reihe ist wird überprüft ob sie seit ihrer letzten überprüfung referenziert wurde. Wenn ja: sie wird wieder (als jüngste Seite) in die Liste eingehängt. Wenn nein: löschen der Seite. In diesem Algorithmus wird unterschiedliche Benutzungshäufigkeit einkalkulliert, aber er ist sehr aufwendig, da auch eine FIFO-Liste geführt werden muss (s.o.)

NRU (Not-Recently-Used): NRU trifft eine zufällige Auswahl aus den kürzlich nicht referenzierten Seiten (z.B. mit zyklischer Suche nach Page Frame-Nummern). Der gewählte Modus des Zurücksetzens des Referenzbits entscheidet über die Güte". Mögliche Optimierung: Unterscheidung zwischen Nur-Lese-Zugriffen und Schreib-Zugriffen, wird in Dirty- Bit (D) = Modifikationsbit (M) angegeben. Da bei kürzlich beschriebenen Seiten der veränderte Seiteninhalt erst gerettet werden muss, sollte wegen des Aufwandes eher eine Seite mit Nur-Lese-Zugriffen zum überschreiben gwählt werden.

Bei Lastspitzen summiert sich die Verdrängung von beschriebenen Seiten, daher wird NRU oft nicht verwendet.

Aging: Beim Aging altern die Seiten duch shiften eines Schieberegisters das für jede Seite angelegt wird. Das führt zu einer guten Annäherung an LRU, aber auch zu

unbertretbar hohem Aufwand. Bei Vereinfachung des Algorithmus (z.B. von Schieberegister mit 8 Bit auf 2 Bit) gehen die LRU-ähnlichen Vorteile verloren und Aging wird zu komplexen Variante von Second-Chance-FIFO.

1.47 In welche dieser Kategorien kann man den Clock-Hand-Algorithmus einordnen?

LRU (ohne Dirty-Bit).

1.48 Warum ist ein perfekter Algorithmus zur Verdrängung von Pages aus dem Hauptspeicher nicht realisierbar?

Ein optimaler Algorithmus ist für ein 'normales' Betriebssystem nicht realisierbar, da die Prozesse durch unterschiedliche Working Sets eine unterschiedlich große Anzahl von Seiten unterschiedlich lange benötigen. Im Gegensatz dazu ist bei einem Betriebssystem mit wenigen, speziellen Aufgaben (Embeded Systems) ein optimaler bzw. annähernd optimaler Algorithmus möglich.

1.49 Was passiert, wenn die Umlaufzeit des Zeigers beim Clock-Hand-Algorithmus zu groß bzw. zu klein gewählt wird? Wie kann ein zweiter Zeiger den Algorithmus verbessern?

Wenn die Umlaufzeit zu groß gwählt wird ist irgendwann kein freier Speicher mehr verfügbar (die Freispeichereserve leer). Wird die Umlaufzeit zu klein gewählt, werden die Seiten vor ihrer nächsten Benutzung aus dem Speicher entfernt und müssen dann erneut geladen werden, die Freispeicherreserve ist grösser als sinnvoll.

Ein zweiter Zeiger verbessert den Clock-Hand-Algorithmus insbesondere bei großen Mengen von Speicher. Zum Beispiel kann der erste Zeiger das Referenzbit gegebenenfalls zurücksetzen und der Zweite die Seiten entfernen falls sie in der Zwischenzeit nicht erneut referenziert wurden.

1.50 Was ist Swapping? Warum wenden auch Paging-Systeme häufig dieses Verfahren an beziehungsweise unter welcher Bedingung?

Swapping wird angewandt, wenn die Working Sets der aktiven Prozesse nicht vollständig in den Speicher passen. Dadurch entsteht eine hohe Page Frame-Rate, das so genannte Seiten-Flattern. Da das Lesen von der Festplatte recht lange dauert, sinkt die Geschwindigkeit, in der die Instruktionen abgearbeitet werden können, rapide.

Durch Swapping werden nicht nur Teile von Prozessen (Paging), sondern ganze Prozesse auf die Platte ausgelagert. Längere Zeit inaktive Prozesse werden aus dem Hauptspeicher entfernt, um dort Platz verfügbar zu machen, dadurch wird natürlich ein Austausch der Prozesse im Hauptspeicher nach gewisser Zeit erforderlich.

1.51 Wie kann man die Vorteile von Paging und Segmentierung kombinieren?

Bei einem einzigen Adressraum kann dieser beim Paging nicht logisch aufgeteilt werden. Der gesamte Adressraum wird in gleich große Kacheln eingeteilt. Deswegen wird heute der Speicher häufig in mehrere Segmente unterteilt, die jeweils eigene Pagetabellen enthalten. Die Größe der Segmente ist nicht statisch, sondern kann jede Gösse mit n multipliziert mit Page Frame Größe umfassen.

Dadurch entstehen dreiteilige Adressen: Region mit eigener Pagetabelle, Page in der Region, Adresse in Page. Regionen folgen dabei der logischen Größe des Adressraumes (Text, Data, Stack), Pages sind in feste Größen unterteilt.

1.52 Wozu bzw. wo wird bei der Speicherverwaltung häufig ein *Assoziativspeicher* eingesetzt?

Assoziativspeicher wird im allgemeinen als Spezialhardware innerhalb der MMU (Memory-Managment-Unit) realisiert. D.h. Assoziativspeicher == Hardware-Cache. Er wird häufig zur Adressumsetzung genutzt. Für die gesamte Pagetabelle wäre dies sehr teuer, deswegen wird beim Prozesswechsel neu geladen, es sind (getreu dem Lokalitätsprinzip) immer nur einige Seiten in Gebrauch. Daher enthält der Cache jeweils nur diese Seiten.

Ablauf: Adresszugriff, Page Table Entry im Cache? Wenn ja: Adressumsetzung, wenn nein: Nachladen aus der Pagetabelle. Es ist der parallele Zugriff auf alle im Cache befindlichen Einträge möglich. Die gesamte Pagetabelle im Hauptspeicher vorzuhalten, wäre zu langsam, da zusätzlicher Speicherzugriff nötig wäre.

1.53 Beschreibe kurz die Zugriffsoperationen open(), close(), lseek(), read() und write() auf einem UNIX-Filesystem. Welche Rolle spielt der File-Deskriptor dabei?

`open()`
Öffnet eine Datei für die weitere Arbeit. open() werden die Parameter `path` (Pfadname der Datei), `ags` (erlaubte Folgeoperationen wie lesen, schreiben,...) und `mode` (Zugriffsrechte bei einer NEU angelegten Datei) übergeben. Der File Descriptor wird zurückgegeben.

`close()`
Dieser Systemaufruf gibt eine Datei wieder frei.

`create()`
Es wird mit den Parametern path und mode eine neue Datei erzeugt. Der File Descriptor wird zurückgegeben.

`lseek()`
wird mit den Parametern File Descriptor, offset und whence aufgerufen. Die aktuelle Position im File Descriptor wird um offset Bytes gemäß whence verschoben (d.h. 0 = vom Anfang, 1 = vom Ende, 2 = von der aktuellen Position aus). Zusammengefasst: lseek setzt den Zeiger, der innerhalb einer Datei die Position anzeigt.

`read()`
mit den Parametern File Descriptor (fd), buf und len liest von len Bytes ab der aktuellen Position des fd in den Puffer buf, dabei wird die aktuelle Position um die gelesenen Bytes weitergeschoben. read() liefert die Anzahl der tatsächlich gelesen Bytes.

`write()`
mit den Parametern fd, buf und len funktioniert analog zu read() und liefert die Anzahl der geschriebenen Bytes zurück.

1.54 Wie sieht die Struktur des UNIX-V7-Dateisystems auf der Platte in etwa aus? Warum erfolgt die Verwaltung der Freispeicherliste über *Indirekt*-Blöcke?

Das Dateisystem V7:

- Der Dateiinhalt ist Block-orientiert mit speziellem Block-Index organisiert.
- Der Boot-Block ist Block 0 des Root-Dateisystems und wird beim booten geladen.
- Der Superblock enthält die Verwaltungsinformationen des Dateisystems: Größe, Verwaltung der freien Inodes, Verwaltung der freien Blöcke (mit verketteter Liste von Blöcken mit freien Blocknummern)
- Der Superblock verweist auf Inodes und freie Blöcke
- Inodes dienen zur Ablage der Verwaltungsinformationen der Dateien. Jede Datei hat einen Inode mit:
  - eindeutigem Bezeichner (heisst: Inodenummer, diese gibt die Position an)
  - Besitzer (nid), Gruppe (gid)
  - Zeitpunkt der letzten änderung, des letzten Zugriffs, ...
  - Anzahl der Hard Links (mehrere Namen für eine Datei verweisen auf denselben Inode)
  - Anzahl von Bytes
  - Dateityp
  - Verweise auf Datenblöcke
  - Zugriffsrechte: Kleine Dateien bis zu zehn Blöcken werden direkt im Inode-Block gespeichert.

- Organisation der Indirekten Blöcke:
  - Dateien größer als 10 Blöcke
  - Jeder ind. Block ist in Inodes eingetragen
  - Sehr große Dateien mit maximal vier Zugriffen/Dateiblock realisierbar
  - Blockaufteilung für Sicherheit:
    - Superblock mehrfach in Kopie vorhanden
    - Mechanisch kaputte Blöcke werden aussortiert
  - Das Gegenteil wäre Organisation in verketteter Liste, ist hier ein Stück kaputt, ist die Liste auch kaputt.

1.55 Welche Aufgaben enthält ein Inode? Welche Angaben enthält eine Verzeichnis-Datei (Euch besser bekannt als *Directory*)?

Aufgaben von Inodes: siehe oben. Eine Verzeichnis-Datei (Directory) ist eine Datei und hat entsprechend einen eigenen Inode und Datenblöcke. Verzeichnisse sind eine Folge von Einträgen, die jeweils den Dateinamen (auch Dateinamen von Unterverzeichnissen) und die jeweilige Inodenummer enthalten.

1.56 Welche Aufgaben hat der Buffer Cache in UNIX?

Der UNIX-Buffer Cache ist ein Zwischenspeicher (Puffer) für gelesene oder geschriebene Plattenblöcke im Kernadressraum. Er puffert Ein- und Ausgabe vom User-Adressraum entkoppelt. Der Buffer Cache ermöglicht Mehrfachzugriff ohne weitere Plattenzugriffe. Die Daten werden von Platte zu Cache in Blöcken transportiert, von Cache zu CPU in Bytes.

1.57 Was geschieht durch einen mount()-Systemaufruf in etwa?

Unter UNIX kann eine physikalische Harddisk in mehrere logische Partitionen eingeteilt sein. Jede dieser Patitionen enthält ihr eigenes Dateisystem, ein UNIX-System kann also aus mehreren solcher Dateisysteme zusammengesetzt sein. Mit dem Befehl mount wird ein weiteres Dateisystem in einen solchen Dateibaum (Wurzel: Root) eingehängt, d.h. verfügbar gemacht. Beim mounten werden logische Geräte (z.B. USB-Stick, CDROM, Partition, ...) und der Mount Point übergeben.

1.58 Welche Vorteile bietet es, Dateien mit dem UNIX-Systemaufruf `mmap()` in den virtuellen Adressraum eines Prozesses abzubilden?

I/O Performance steigt durch:

- Lazy Loading
- Keine Systemaufrufe notwendig

- Das Betriebssytem arbeitet meistens direkt auf dem Buffercache, so dass keine Kopie im Userspace Addressraum angelegt werden muss.

1.59 Wie ist eine Platte intern organisiert? Wie wirkt sich dies auf den Informationszugriff aus? Wie geht das UNIX Fast File System damit um?

Eine physikalische Festplatte besteht aus sechs bis zehn internen Platten (Scheiben). Auf jede dieser internen Platten greift jeweils von oben und von unten ein Lese-Schreibkopf zu. Jede Oberfläche ist in Spuren unterteilt, jede Spur in Sektoren. übereinanderliegende Spuren auf den verschiedenen Oberflächen bilden Zylinder. Da für jede Armbewegung relativ viel Zeit benötigt wird, sollten Armbewegungen möglichst vermieden werden. D.h. zusammengehörende Daten sollten in hintereinanderliegenden Sektoren in derselben Spur untergebracht werden, damit das Lesen und Schreiben in Vorgang ohne weitere Armbewegung vorgenommen werden kann. Ist die Datenmenge größer, sollten die Spuren innerhalb des gleichen Zylinders verwendet werden.

UNIX Fast File System: Das System arbeitet auf vier bzw. acht Byte großen Datenblöcken. Die Platte ist in Zylindergruppen unterteilt. Jede Zylindergruppe hat einen eigenen Inode- und Datenbereich, Dateien und ihr Inode werden möglichst in derselben Zylindergruppe gespeichert. Ein Verzeichnis und die darin enthaltenen Dateien werden wiederum in derselben Zylindergruppe gespeichert, dies gilt aber nicht für im Verzeichnis befindliche Unterverzeichnisse.

1.60 Welche Vorteile bietet eine vereinheitlichte Betriebssystemschnittstelle zum Zugriff auf Geräte? Wie sieht die in UNIX in etwa aus?

In UNIX sind Geräte als Dateien dargestellt. Dies bietet für Anwender und Programmierer den Vorteil, dass sie einheitlich zu behandeln sind.

Laut Wikipedia: Everything is a file (engl. ‚Alles ist eine Datei') beschreibt eine der definierenden Eigenschaften von Unix und seinen Abkömmlingen, dass eine große Bandbreite an Ein-/Ausgabe-Ressourcen wie Dokumente, Verzeichnisse, Festplatten, Modems, Tastaturen, Drucker und sogar Interprozess- und Netzwerkverbindungen als einfache Byteströme via Dateisystem verfügbar sind.

1.61 Was ist ein *Gerätetreiber*, was ein *Geräte-Controller*? Welche Aufgaben haben sie?

Gerätetreiber:

- ist Code innerhalb des Betriebssystems zur Geräteverwaltung
- es existiert jeweils ein Treiber pro Gerätetyp, die Typen werden durch die Major Number unterschieden
- als Parameter wird die Minor Number benötigt, um die konkrete Hardware zu identifizieren / anzusprechen. Controller:

- ist Hardware, die sich zwischen CPU und Gerät befindet
- enthält unter anderem einen Puffer für die Zwischenlagerung von Aufträgen an das Gerät
- zur Aktivierung des Controllers werden die Aufträge in Controllereigenen Registern abgelegt Ein Treiber ist also zum Kernel gehörende Software, ein Controller ist die zu einem Gerät gehörende, vom Treiber gesteuerte, Hardware.

1.62 Warum erfolgt der Zugriff auf Geräte häufig über Warteschlangen? Wozu besitzten diese in der Regel eine *High Water Mark* bzw. eine *Low Water Mark*?

Der Zugriff über Warteschlangen erfolgt um eine Vermischung von Aufträgen und das Verlorengehen von Aufträgen zu verhindern. Desweiteren um die komplette Abarbeitung eines Auftrages zu gewährleisten und die Reihenfolge der Auftragsabarbeitung zu organisieren. Die Warteschlangen besitzen in der Regel eine High Water Mark, um anzuzeigen, dass die Warteschlange voll ist, d.h. um zu verhindern das entweder ein Auftrag in der Schlange überschrieben wird und ein Auftrag verlorengeht. Die Low Water Mark zeigt dagegen an, dass weitere Aufträge "nachgefüllt" werden können.

1.63 Worin unterscheidet sich Direct Memory Access (DMA) von Programmed I/O?

Beim Programmed I/O ist der Ablauf wie eben beschrieben. Da der Treiber ein Teil des Betriebssystems ist, bleibt letztendlich die Kontrolle des Vorgangs bei der CPU.

Beim Direct Memory Access (DMA) liegt nach dem Anstoss eines Auftrages der Zugriff auf den Hauptspeicher und die daraus vorzunehmenden Kopiervorgänge in der "Verantwortung" des Controllers. Der Controller greift nach der Abarbeitung eines Auftrages ohne Einbeziehung der CPU auf den Hauptspeicher zu. Zu diesem Zweck muss der Auftrag selbst die entsprechenden Speicherbereiche angeben.

1.64 Warum werden Terminal-Treiber in UNIX parametrisiert? Nenne typische Parameter.

Geräte können unter Umständen in verschiedenen Modi laufen. Um den gewünschten Modus zu erzeugen, benötigt ein Treiber die Angabe desselben mit Hilfe eines Parameters $\Rightarrow$ Parametrisierung. Ein typischer Parameter wäre bei einem Monitor z.B. die Farbtiefe. Bei Terminals werden grundsätzlich zwei Modi unterschieden: der Raw-Modus (Canonical Mode), in ihm werden einem Terminal Tasteneingaben unverändert und zeichenweise an des Pozess weitergereicht. Im Cooked-Modus (Noncanonical Mode)wird die Tasteneingabe zeilenweise an den Prozess weitergegeben. In diesem Modus ist es möglich, Tastenkombinationen abzufangen (also erst einmal nicht an den Prozess weiterzugeben) und ihnen besondere Funktionen zuzuweisen. Z.B. Zeileneditierfunktionen

wie: BS, DEL, Strg-w,... zu verarbeiten und dann in der neuen Form an den Prozess weiterzugeben. Weitere Beispiele hierfür sind die Flusskontrolle mit Strg-s (stoppt Ausgabe) oder Strg-q (weiter) oder Signale wie Strg-c (stoppt den Prozess mit SIGINT).

2 Nebenläufigkeit

2.1 Erkläre den Begriff „Nebenläufigkeit". Welchen Zweck haben Petri-Netze in diesem Kontext?

engl. concurrency

Nebenläufigkeit entsteht, wenn mehrere Ereignisse in keiner kausalen Beziehung zueinander stehen, sich also nicht beeinflussen. Aktionen können nur dann nebenläufig ausgeführt werden (sie sind parallelisierbar), wenn keine das Resultat der anderen benötigt.

Eine Modellierungssprache, die diese Abhängigkeiten wiedergibt, sind Petri-Netze.

2.2 Skizziere kurz einige Probleme des nebenläufigen Zugriffs auf Betriebsmittel.

Zwei Prozesse dürfen nicht gleichzeitig auf ein Betriebsmittel zugreifen (kritischer Abschnitt). Es kann hier zu Verklemmungen (Deadlocks), dem After-you-after-you Problem und dem Verhungern kommen.

Bei nebenläufigem Zugriff kann es zu inkonsistenten Daten kommen, wenn der gegenseitige Ausschluss nicht gewährleistet ist.

2.3 Grenze die Begriffe *Nebenläufigkeit*, *Quasi-Parallelität* und *Parallelität* voneinander ab. Was verstehen wir unter Nichtdeterminismus?

Nebenläufigkeit
: ist das Abarbeiten von mehreren Prozessen mit einer CPU.

Quasi-Parallelität
: ist z.B. paralleles Arbeiten auf einem Einprozessorsystem. In der CPU wird nichts parallel bearbeitet, für uns scheint es am Monitor aber so.

Paralellität
: ist das Abarbeiten von Prozessen mit einer Mehrprozessormaschine.

Nichtdeterminismus
: ist ein Konzept, in dem Algorithmen oder Maschinen bei gleicher Eingabe mehrere Möglichkeiten für den Übergang in den nachfolgenden Zustand haben.

2.4 Welche Nebenläufigkeitseigenschaften bzw. -probleme werden durch die drei folgenden „klassischen" Szenarien ausgedrückt:

1. Kritischer Abschnitt (Critical Section)
2. Leser/Schreiber (Reader/Writer)
3. Erzeuger/Verbraucher (Producer/Consumer)
4. Speisende Philosophen (Dining Philosophers)?

Kritischer Abschnitt
Zwei nebenläufige Vorgänge greifen auf das selbe Betriebsmittel zu (z.B. zwei Threads auf eine gemeinsame Variable). Das Ergebnis hängt von der tatsächlichen Ausführungsreihenfolge ab und es entsteht (ungewollter) nicht-Determinismus.
Lösung: Gegenseitiger Ausschluss oder Mehrseitige Synchronisation.

Leser/Schreiber
Datenbestand, auf den verschiedene Prozesse lesend bzw. schreibend zugreifen. Z.B. Datenbank (Datensätze mit mehreren Komponenten). Paralleles Lesen ist ok. Aber nicht parallel lesen und schreiben.
Abhilfe: z.B. Verwendung eines Locking-Verfahrens

Erzeuger/Verbraucher
Gemeinsames Kommunikationsobjekt (Warteschlange: Speicherbereich, Datei, Pipe,...). Erst schreiben, dann lesen - Verbraucher muss auf den Erzeuger warten. Viele mögliche Probleme (Count-Zähler führt nicht korrekt Buch: kritischer Abschnitt; Elemente in scheinbar leerer Warteschlange, Platz in scheinbar voller Warteschlange; Lost-Wakeup, Verklemmung).

Speisende Philosophen
Fünf Philosophen und nur fünf Stäbchen zum Reisessen. Mehrseitige Synchronisation nötig und Gefahr von Verklemmungen, Philosophen verhungern.
Mögliche Lösung durch Vermeidung (Bankiers Algorithmus): Es dürfen nur vier Philosophen an den Tisch.

2.5 Was ist ein Thread („Faden")? Skizziere ein sinnvolles Anwendungsbeispiel für die Verwendung mehrerer Threads innerhalb eines Prozesses.

Threads können parallel durchlaufen werden, auch wenn sie zu einem gleichen Prozess gehören. Sie ermöglichen also Parallelisierung/Nebenläufigkeit. Bringen aber damit auch bekannte Gefahren mit sich.

Ein Prozess könnte eine Operation auf mehrere Datensätze ausführen, das ist sehr gut parallelisierbar.

Oder ein Prozess hat eine Grafische Benutzeroberfläche (GUI) und führt Berechnungen durch. Dann soll die GUI immer beidenbar sein, auch wenn grade eine aufwändige Berechnung läuft.

2.6 Was besagt Prämemption?

Präemption ist die zeitweise Unterbrechung der Bearbeitung einzelner Prozesse zugunsten anderer.

2.7 Grenze den Thread-Begriff gegen den UNIX-Prozess ab (Adressraum, Zustandsinformationen etc.). Was haben Light-Weight-Prozesse (LWPs) damit zu tun?

Ein Prozess hat einen eigenen Adressraum und ist somit stark abgeschottet gegenüber anderen Prozessen. Prozesse können aus anderen Prozessen erzeugt werden und arbeiten trotzdem weiter, wenn der Vater-Prozess terminiert ist. Prozesse sind eigenständig. Threads sind mehrere Kontrollfäden in einer Hülle, nämlich dem Prozess, aus dem sie aufgerufen wurden. Threads haben zwar einen eigenen Ausführungszustand (PC, Stack) dafür aber eine gemeinsame Umgebung (Adressraum, offenen Dateien, gemeinsame Datenstrukturen). Hier kann es zu kritischen Abschnitten kommen. Eine Synchronisation ist erforderlich. Einem Thread muss Arbeit (eine Prozedur) zugewiesen werden. Ein Thread ist beendet, wenn die Umgebung, also der Prozess, terminiert. Ein Thread ist also von der Umgebung abhängig. Ein Problem ist, dass der Betriebssystemkern nichts vom Erzeugen der Threads erfährt, es ist also keine „echte Nebenläufigkeit" gegeben. Light-Weight-Prozesse dagegen sind dem Bestriebsystemkern bekannt. Das stellt die Grundlage für das Scheduling dar.

2.8 Die Routinen `pthread_create()`, `pthread_join()`, `pthread_exit()` realisieren die Erzeugung und Termination von Threads in der UNIX-Multithreading-Umgebung. Vergleiche ihre Funktionalität mit den Systemaufrufen zur Erzeugung und Termination von Prozessen (`wait()`, `fork()` und `exit()`). Warum arbeitet `pthread_create()` deutlich anders als `fork()`?

pthread_create()
: Thread wird Prozedur zugewiesen, hat eigenen Ausführungszustand (PC,Stack)

fork()
: erstellt eine Kopie des Adressraumes. Kind kann weiterarbeiten nachdem der Vater bereits terminierte.

pthread_join()
: sammelt Thread ein wenn er beendet ist. Bei Prozessende werden eventuell nicht-abgeschlossene Threads beendet.

wait()
: wartet auf die Terminierung eines Prozesses. Welcher Prozess kann über die Prozess-ID abgefragt werden.

pthread_exit()
: zum vorzeitigen Beenden von Threads

exit()
: zum vorzeitigen Beenden von Prozessen

`pthread_create()` erzeugt in einer gemeinsamen Umgebung einen eigenen Ausführungszustand. Dazu muss beim Erzeugen unter anderem auch eine Stack-Info erstellt werden. Mit Threads ist es möglich, von einem Programmfaden in mehrere überzugehen.

`fork()` ist dagegen so ausgelegt, dass ein eigenständiger Prozess erzeugt wird. Dieser wird auch vom Scheduler berücksichtigt, da er dem Betriebsystemkern bekannt ist.

2.9 Was versteht man unter *einseitiger* bzw. *mehrseitiger Synchronisation*? Gib jeweils ein Anwendungsbeispiel an.

Einseitige Synchronisation Producer-Consumer-Problem mit unendlicher Pufferkapazität; benötigt eine Semaphore

```
Sem s(0);

Producer:
erzeugen();
s.V();

Consumer:
s.P();
verbrauchen();
```

Mehrseitige Synchronisation Producer-Consumer-Problem mit Pufferkapazität; benötigt zwei Semaphoren

```
Sem s1(0);
Sem s2(3);

Producer:
s2.P();
erzeugen();
s1.V();

Consumer:
s1.P();
verbrauchen();
s2.V();
```

2.10 Was ist der Unterbrechungsauschluss?

Beim Unterbrechungsausschluss werden alle Unterbrechungen auf Systemebene für den kritischen Abschnitt augeschaltet:

```
disable_interrupts(); (entspricht lock())
... kritischer Abschnitt ...
enable_interrupts(); (entspricht unlock())
```

Damit sind keine Unterbrechungen möglich, folglich auch kein „Prozesswechsel". Ebenso kann sich nur ein Prozess gleichzeitig im kritischen Abschnitt befinden.

Problem hierbei ist, dass Interrupts zeitkritisch sind und somit die Gefahr besteht, dass relevante Interrupts nie verarbeitet werden. Ebenso funktioniert der Unterbrechungsauschluss nur bei Einprozessorsystemen. Im User-Mode ist der Unterbrechungsauschluss gar nicht möglich.

In der Regel wird zuviel gesperrt; viele Aktivitäten wollen kritischen Abschnitt gar nicht betreten.

2.11 Was ist ein kritischer Abschnitt? Wie kann man den gegenseitigen Ausschluss gewährleisten? Warum ist ein Unterbrechungsausschluss dabei nicht immer das geeignete Mittel?

Ein kritischer Abschnitt ist ein Teil des Programmcodes, welcher geschützt werden muss, wenn zwei Threads auf die selben Ressourcen zugreifen.

Ein kritischer Abschnitt ist ein Programmteil, in dem von mehreren Programmabäufen auf eine gemeinsame Datenstruktur oder auf ein gemeinsames Betriebsmittel zugegriffen wird. Kritische Abschnitte müssen daher geschützt werden. Sonst würde es eventuell zu falschen Ergebnissen kommen. jeder kritische Abschnitt benötigt daher einen Schliessmechanismus, der durch eine „Schlossvariable" realisiert wird. Bei Betreten eines kritischen Abschnitts wird dann diese Variable gesetzt (oft Boolean TRUE).

Andere Programmabläufe müssen nun den Status der Schlossvariablen nachfragen, um zu erfahren, ob der kritische Abschnitt frei ist oder nicht. Möglichkeiten, den gegenseitigen Ausschluss zu gewährleisten: - eigene Absicht anmelden, dann überprüfen ob anderer auch will. Jeder Programmablauf benötigt eine eigene Schlossvariable. Sobald einmal die Absicht einzutreten erklärt wurde, ist der kritische 10 Abschnitt gesperrt. Der gegenseitige Ausschluss ist somit gewährleistet. Problem: Wenn zwei Programmabläufe nebenläufig die Absicht anmelden, können beide Abläufe in einer Endlosschleife hängenbleiben, weil sie darauf warten, dass der andere den kritischen Abschnitt freigibt. - Unterbrechungen ausschalten. Andere Programmabläufe können nun nicht mehr einen anderen unterbrechen, wenn sich dieser im kritischen Abschnitt befindet. - andere Prozesse blockieren>: hier werden andere programmabläufe einfach blockiert, egal ob sie nun auf kritische Abschnitte zugreifen können oder nicht. - wenn trotdem andere prozesse während eines kritischen Abschnitts zugelassen werden sollen, dann müssen wie oben mechanismen gefunden werden, um den kritischen Abschnitt zu schützen. Gegenseitiger Ausschluss durch Unterbrechungsausschltung funktioniert nur bei Einprozessorsystemen. Wenn ein Prozess sich in einem kritisdchen Abschnitt befindet, und nicht selbst wieder herauskommt (z.B. Endlosschleife), wäre es nicht sinnvoll, hier die Unterbrechung auszuschalten.

2.12 Nach welchen Kriterien wird Korrektheit bzw. Güte von *Locking-Algorithmen* bewertet? Wie geht man dabei vor?

Der Zugriff auf Schlossvariablen darf nicht selbst in einem kritischen Abschnitt liegen. Locking-Algorithmen sollten verklemmungsfrei arbeiten.

2.13 Warum sollte man die Bewertung von Locking-Algorithmen auf der Grundlage von *unteilbaren Operationen* durchführen?

Nur unteilbare Operationen sind nicht unterbrechbar. Unteilbare Operationenen sind Maschinen-Instruktionen, also die unterste Ebene der Programm- Ausführung. Wenn man ein Programm auf Maschinen-Code runterbricht, kann man sehr gut erkennen, dass nach jedem Befehl eine Unterbrechung, z.B. ein Interupt zum Prozesswechsel, erfolgen kann (z.B. durch Ablauf der Zeitscheibe). Die kritischen Abschnitte lassen sich daher genau analysieren und Anfang und Ende festlegen.

2.14 Auf welche verschiedenen Arten kann man *Verklemmungen* angehen? Wie arbeitet der *Bankiersalgorithmus*?

Ein Deadlock liegt an, wenn z.B. zwei oder mehrere Prozesse auf ein Betriebsmittel warten, das nur von anderen freigegeben werden kann, es aber nicht mehr zu der Freigabe kommen wird. Möglichkeiten:

- Deadlock ignorieren, wenn er nur selten auftreten kann, da eine Lösung unter Umständen zu teuer ist.
- Deadlock durch Untersuchung des Betriebsmittelgraphen entdecken und Deadlock durch Zwangsentfernen eines Prozesses beheben.
- Deadlock verhindern durch das erkennen der Randbedingung, die zum Deadlock führt:
  - Exklusivität (Spooling)
  - alles auf einmal anfordern (aber: Verschwendung von Ressourcen!)
  - Zwangsentzug
  - Zyklus verhindern (Betriebsmittel nach aufsteigender Nummer anfordern und belegen), dies führt zu einer Einschränkung der Prozesse.
- Deadlocks vermeiden: Bankiersalgorithmus.

Bankiersalgorithmus

Die Idee ist, dass die maximale Anforderung von Betriebsmitteln im voraus bekannt ist. Belegung wird nur dann gewährt, wenn das System dadurch nicht in einen Zustand gelangt, der zu einem Deadlock führen kann. Diesem Verfahren liegt das Modell eines Bankiers zugrunde, der mit Kunden einen Kreditrahmen aushandelt, der nach und nach eingelöst werden kann, aber die Grenzen der Leistungsfähigkeit nicht sprengt.

2.15 Wie kann man eine einseitige Synchronisation mit Hilfe von wait() und signal() vornehmen? Wie kann man diese Primitiven in etwa auf lock() und unlock() abbilden?

TODO

2.16 Grenze die Begriffe *aktives* und *blockierendes* Warten voneinander ab.

Beim aktiven Warten wird ständig die Schlüsselvariable eines kritischen Abschnitts überprüf, ob sieser nun frei geworden ist oder nicht. Durch das ständige Abfragen wird unter Unständen unnötig CPU-Zeit in Anspruch genommen. Beim blockierenden Warten legt sich der betreffende Pozess schlafen, falls der kritische Abschnitt nicht frei ist. Wird der kritische Abschnitt von dem anderen Prozess verlassen, so führt dieser ein wakeup() aus, um die schlafenden Prozesse zu wecken.

2.17 Was ist ein Spinlock?

Ein Spinlock ist aktives Warten im Programm selber.

2.18 Was ist aktives Warten?

Aktives Warten ist das Verharren in einer Schleife, bis der kritische Abschnitt wieder frei ist: Spinlocks

Nachteil: Verbrauchen unnötig Prozessorkapazität durch permanente Abfragen (bis Zeitscheibe aufgebraucht)

```
Gemeinsam von A und B genutzte Variable: lock
Interpretation des Werts:                0 gesperrt, ungleich 0 offen
Initialisierung:                         lock = 0

Prozess A                    Prozess B
   ...                          ...
   solange (lock == 0) {        ...
      ;                         lock = 1;
   }                            ...
   Aktion a                     ...
```

2.19 Was ist blockierendes Warten?

Beim blockierenden Warten legt sich der Prozess schlafen, wenn der kritische Abschnitt gerade nicht frei ist (sleep()).

Bei Verlassen des kritischen Abschnitts werden darauf wartende Prozesse aufgeweckt (wakeup()).

Interrupthandler sind zeitkritisch und haben keinen eigenen Prozesskontext, können sich also nicht schlafenlegen und können zur mehrseitigen Synchronisation kein blockierendes Warten verwenden.

2.20 In einer UNIX-Multiprozessorumgebung können mehrere Prozesse nebenläufig `sleep()` aufrufen. Warum ist dies ein kritischer Abschnitt? Warum kann man ihn nicht einfach davor schützen, dass man den Aufruf von `sleep()` von einem *Spinlock* umgibt? Was wird man stattdessen tun?

Die gemeinsame Datenstruktur ist hier die Sleep-Queue. Wenn sich ein schlafender Prozess von einem Spinlock umgibt, dann kann er unter Umständen nicht wieder aufgeweckt werden, da er die Sleep-Queue nicht freigegeben hat.

Als Lösung wird eine Variante von sleep() eingeführt: sleepl(). sleepl() gibt die Sleep-Queue wieder frei (gibt den Spinlock wieder ab), bevor die CPU abgegeben wird (vorm Schlafenlegen).

Tim: Was ist mit der Prozesstabelle? Laut Tutorium ist diese ebenfalls relevant.

2.21 Was sind Semaphore?

P() / V()
P(): blockieren [wait, acquire oder down]; counter–
V(): entblockieren [signal, release, post oder up]; counter++
auch; „counting semaphore"

Zähler >= 0 bei Eintritt in kritischen Bereich inkrementiert, dekrementiert bei Austritt. Bei Zähler = 0 warten.

P()/V() ist nicht an Blockstrukturen gebunden.
P()/V() ist nicht immer regelmäßig geschachtelt.

2.22 Welche zusätzlichen Eigenschaften zeichnen Semaphoren gegenüber blockierenden Locks aus?

- Semaphore ist FIFO-Queue, ist also fair.
- Semaphore: Anzahl der Zugriffe können begrenzt werden (<-> blockierendes Warten alle).
- Blockierendes Warten ist unabhängig.
- Synchronisation der Betriebsmittelverwaltung.

Es gibt zwei zusätzliche Erweiterungen bei einer Semaphore:

- Im Wartefall wird der Pozess in eine FIFO-Queue eingereiht, das heißt, die Prozesse werden nach Ankunftsreihenfolge beim kritischen Abschnitt abgearbeitet.
- Semaphore-Counting, das heisst die Semaphore enthält einen Counter, durch den die Semaphore erst blockiert, wenn dich n Prozesse im kritischen Abschnitt befinden. Eine Semaphore wird mit n vorinitialisiert.

2.23 Was sind Monitore?

```
public synchronized void einzahlen(int betrag) {
.. ändere Konto ..
}
```

Schutz erfolgt nur gegenüber anderen Methoden der Klasse, die ebenfalls synchronized sind, und kritischen Abschnitten, die sich über das Objekt synchronisieren.

2.24 Was sind Mutexes?

kurz für: mutual exclusion

Erlauben den gegenseitigen Ausschluss. Sie sind als mehrseitige Synchronisation (allerdings Zugriff auf die selbe Semaphor-Variable) realisiert.

```
IrgendeineKlasse myObj = new IrgendeineKlasse();
synchronized (myObj) {
.. kritischer Abschnitt ..
}
```

äquivalent zu:

```
Mutex myObj;
myObj.lock();
.. kritischer Abschnitt ..
myObj.unlock();
```

Mutexes lassen sich auch über Semaphore umsetzen.
Signale/Ereignisse: myObj.wait()/myObj.notify()

2.25 Wie wird eine einseitige bzw. mehrseitige Synchronisation durch Semaphore ausgedrückt?

- einseitige: Erzeuger-Verbraucher-Prinzip
  Die Semaphore muss mit 0 vorinitialisiert werden. Wenn in einem Prozess B etwas ausgeführt werden soll, was abhängig von einem anderen Prozess A ist (zum Beispiel ein Ereignis), dann muss vorher P() ausgeführt werden. In A muss, wenn das betreffende Ereignis stattgefunden hat, V() ausgeführt werden. Wurde das V() nicht ausgeführt, und versucht B jetzt P() auszuführen, so wird blockiert und gewartet bis V() kommt.

  P1: .. Daten produzieren .. (kritischer Abschnitt); S.V (Producer-Thread)
  P2: S.P; .. Daten verarbeiten .. (kritischer Abschnitt) (Verbraucher-Thread)

  S = Sempaphor

- mehrseitige Synchronisation: Funktioniert wie einseitige, jedoch mit einer oder mehren Semaphoren. Bei einer Semaphore S greifen alle Threads auf S zu:

  P1: S.P; kritischer Abschnitt 1; S.V
  P2: S.P; kritischer Abschnitt 2; S.V

  Für ein Beispiel mit mehreren Semaphoren, siehe 2.9.

2.26 Was sind die speisenden Philosophen?

Die Philosophen sitzen am Tisch und denken über philosophische Probleme nach. Wenn einer hungrig wird, greift er zuerst die Gabel links von seinem Teller, dann die auf der rechten Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.

Solange nur einzelne Philosophen hungrig sind, funktioniert dieses Verfahren wunderbar. Es kann aber passieren, dass sich alle fünf Philosophen gleichzeitig entschließen, zu essen. Sie ergreifen also alle gleichzeitig ihre linke Gabel und nehmen damit dem jeweils links von ihnen sitzenden Kollegen seine rechte Gabel weg. Nun warten alle fünf darauf, dass die rechte Gabel wieder auftaucht. Das passiert aber nicht, da keiner der fünf seine linke Gabel zurücklegt. Die Philosophen verhungern.

2.27 Wie können Semaphore zur Lösung des Problems der speisenden Philosophen eingesetzt werden? In welches Problem wird eine allzu einfache „Implementierung" laufen?

Fünf Philosophen sitzen an einem Tisch mit fünf Tellern Reis. Zwischen jedem der Teller befindet sich ein Stäbchen, also insgesamt sind auch fünf Stäbchen vorhanden. Das Problem ist, dass jeder Philosoph zwei Stäbchen zum Essen benötigt, es können also nicht alle Philosophen gleichzeitig essen. Nur für zwei gleichzeitig speisende Philosophen sind ausreichend Stäbchen vorhanden. Jedes Stäbchen ist ein kritischer Abschnitt. Es kommt aber zu Verklemmungen, wenn jeder Philosoph z.B. sein linkes Stäbchen nimmt, kann keiner der Philosophen essen. Lösung: Einsatz von Semaphoren für Stäbchen und Philosophen, das heißt immer Paare schützen.

2.28 Warum bietet eine einfache Semaphor-Implementierung mit den UNIX-eigenen `sleep()/wakeup()`-Routinen keine vollständige Semaphore-Semantik? Welche zusätzlichen Massnahmen müsste man ergreifen?

Bevor sich ein Prozess schlafenlegt, wäre es sinnvoll den kritischen Abschnitt wieder freizugeben, da dieser sonst unter Umständen nicht wieder aufgeweckt werden kann. Als zusätzliche Massnahme muss der Abschnitt also wieder freigegeben werden.

2.29 Welche Probleme gibt es mit „fairen Semaphoren"? Was sind „Konvois", was sind „donnernde Herden"?

Duch das Einreihen der Prozesse in eine FIFO-Queue kommt es zu einer festen Reihenfolge bei der Abarbeitung. Es kann hier zu Konvois kommen. Weiterhin kann es zu unnötig vielen Prozesswechseln kommen, wenn die Prozesse aus der FIFO-Queue entnommen werden, prüfen ob sie den kritischen Abschnitt betreten können und wieder eingereiht werden.

Zu donnernden Herden kann es kommen, wenn man das Semaphore-Counting benutzt. Alle Prozesse, die beim Eintreten in den kritischen Abschnitt schlafen gelegt wurden, warten praktisch auf eine Veränderung des Counters. Wenn nun ein anderer Prozess den kritischen Abschnitt frei gibt, indem er ein V() ausführt, werden alle wartenden Prozesse geweckt und donnern los. Es kommt aber nur einer in den kritischen Abschnitt hinein.

2.30 Was ist ein Monitor? Unter welchen Bedingungen wird ein Monitor betreten bzw. wieder verlassen?

Ein Monitor ist ein ADT (Abstrakter Datentyp), also mit Daten und Operationen auf diesen Daten. Die Zugriffsoperationen sind implizit gegen nebenläufigen Zugriff geschützt. Das heisst, der Programmierer muss sich keinen Gedanken mehr um den Schutz von kritischen Abschnitten innerhalb des Programmes zu machen, da der Nenbenläufigkeitsschutz im Monitor bereits implementiert ist.

Vorteil dieser Lösung ist, dass eventuelles Vergessen von Schutzanweisungen nicht mehr möglich ist. Ein Monitor wird betreten, wenn von einer Prozedur des Programms eine entsprechende Monitorfunktion aufgerufen wird. Verlassen wird er, wenn er warten muss (wait()) oder wenn er fertig ist.

2.31 Aus welchen Komponenten besteht ein Petrinetz (mit Marken)? Was kann man damit beschreiben?

Ein Petrinetz ist ein gerichteter Graph, der aus Zuständen (Stellen) und Zustandübergängen (Transistionen) besteht. Mit Marken (Tokens) ist der aktuelle Ausführungszustand beschreibbar. Mit einem Petrinetz kann man grafisch die Synchronisationszusammenhänge darstellen.

2.32 Wie kann man durch ein Petrinetz typische Synchronisationsvorschriften ausdrücken:

1. Sequenz
2. Beschränkte Nebenläufigkeit
3. Unabhängigkeit?

a) Sequenz b) beschränkte Nebenläufigkeit c) Unabhängigkeit? Bei einer Sequenz a, b gibt es die Transistionen a und b, die jeweils eine Ein- und Ausgangsstelle haben. Die Ausgangsstelle von a ist allerdings die Eingangsstelle von b. Die beschränkte nebenläufigkeit kann man darstellen, indem man Token verwendet, um die Anzahl der möglichen Prozesse/Threads anzuzeigen. Bei Unabhängigkeit können zwei Prozesse beliebig oft und in beliebiger Reihenfolge gestartet werden und somit unabhängig nebenläufig laufen.

2.33 Was kennzeichnet lebendige bzw. todesgefährdete Petrinetze?

Bei lebendigen Petrinetzen gibt es immer Transistionen, egal welche Ausführungsreihenfolge stattgefunden hat.. Bei todesgefährdeten Petrinetzen kann es nach einer Transistion zu einem Zustand kommen, von dem aus keine Transistionen mehr ausgeführt werden kann.

2.34 Was wird durch einen Pfadausdruck beschrieben? Welche Operatoren werden dazu vorgesehen? Was ist ihre Semantik?

Mit einem Pfadausdruck kann man Angaben über mögliche Nebenläufigkeit von Zugriffsoperationen eines ADTs machen. Operator : : (Beschränkte) Nebenläufigkeit Operator + : Alternative „Entweder ... oder ...” Operator ; : Sequenz „Erst ..., dann ...” Operator | : Unabhängikeit von Prozessen

2.35 Welche Vorteile bieten Pfadausdrücke zur Steuerung von Nebenläufigkeit im Vergleich zu Semaphoren bzw. Monitore?

Vorteil ist die korrekte Definition dessen, was erlaubt ist oder was nicht erlaubt ist. Unterschied

2.36 Was ist der Unterschied zwischen offenen und geschlossenen Pfadausdrücken?

Geschlossenen Pfadausdrücke: alle erlaubten Ausführungsreihenfolgen von Operationen eines synchronisierten Datentyps (ADT mit expliziter Synchronisationsvorschrift) sind zu spezifizieren. Das heisst alles davon abweichende ist verboten. Offene Pfadausdrücke: alle Einschränkungen von Ausführungsreihenfolgen sind im Pfadausdruck zu spezifizieren. Das heisst alles andere ist erlaubt.

2.37 Welches Problem entsteht bei Nebenläufigkeit, wenn bspw. zwei Threads die Variable i inkrementieren möchten und was verursacht dieses? Wie heißt dieses Problem genau? Wie kann man dies Semaphoren lösen?

Es handelt sich um das Leser-/Schreiber-Problem. Das Problem führt dazu, dass der zweite Thread u.U. mit dem alten Wert weiterarbeitet. Lesende Zugriffe stören sich nicht gegenseitig und können parallel erfolgen. Jedoch ist nebenläufiges Schreiben problematisch.

Auf Assembler-Ebene ist ersichtlich, dass Inkrementieren mehr als eine Instruktion erfordert. Deswegen ist das Definieren des kritischen Blocks notwendig.

Mit Semaphoren lassen sich beide Threads synchronisieren. Es gibt zwei Möglichkeiten: Gegenseitiger Ausschluss (Mutual Exclusion) und mehrseitige Synchronisation.

Gegenseitiger Ausschluss:

```
Sema1(1);

Thread 1:
Sema1.P()
i++
Sema1.V()

Thread 2:
Sema1.V()
i++
Sema1.P()
```

2.38 Wie würde man das klassische Reader-/Writer-Problem als Pfadausdruck formulieren?

Beim Reader-/Writer-Problem sollen beliebig viele Leser, aber kein Schreiber zugelassen werden, oder ein Schreiber ohne Leser. Damit werden inkonsistente Ergebnisse verhindert. Als Pfadausdruck: leser+writer

2.39 Was versteht man unter *synchronem* bzw. *asynchronem Nachrichtenaustausch*? Inwiefern sind diese beiden Kommunikationsformen aufeinander abbildbar?

Bei synchronem Datenaustausch warten Sender und Empfänger aufeinander. Das heist, send() und recieve() wirken blockierend aufeinander. Der Empfänger muss bereit sein, die Nachricht aufzunehmen. Bei asynchronem Datenaustausch muss der Empfänger nicht unbedingt empfangsbereit sein, damit der Sender senden kann. Die Nachricht muss dazu aber in einem Puffer innerhalb des Kommunikationskanals festgehalten werden. Wenn dieser Puffer zu klein ist, muss der Sender blockiert werden, da es sonst zu einem Pufferüberlauf (Bufferover ow) kommt. Synchrone Kommunikation kann durch asychrone Operationen vorgenommen werden, d.h. der Sender wartet nach dem senden auf die Bestätigung des Empfängers. Asynchrone Kommunikation kann durch sychrone Operationen vorgenommen werden. Dazu muss ein Pufferprozess eingeführt werden.

2.40 Wie kann man die Sychronisationseigenschaften von synchronem bzw. asynchronem Nachrichtenaustausch mit Semaphoren modellieren?

```
Sema1(0);
Sema2(0);
```

```
Synchron:
A       B
send(); s1.P();
s1.V(); receive();
s2.P(); s2.V();

Asynchron:
A       B
send(); s1.P();
s1.V(); receive();
```

3 Netzwerke und Interprozess-Kommunikation

3.1 Wozu verwendet man *Kanäle* bzw. *Ports*? Was ist das?

Kommunikation zwischen zwei Prozessen setzt die Benennung der Partner voraus. Besser wäre es, hier den Kanal zu benennen, über den sie kommunizieren. Die Partner bleiben dadurch „anonym". Ports geben an, wohin gesendet werden soll. Sie werden verwendet, wenn es nur einen Sender oder Empfänger auf einem Kanal gibt. Durch Ports ist eine dynamische Zuordnung zu Prozessen möglich.

3.2 Was ist ein *guarded command*? Warum kann die Verwendung eines solchen Konzepts gerade im Zusammenhang mit Nachrichtenaustausch interessant sein?

Guarded commands tauchen im Protokoll „CSP" (Communicating Sequential Processes) auf, welches das Beschreiben von Algorithmen mit synchronem Nachrichtenaustausch ermöglicht.

Durch guarded commands ist es möglich, auf verschiedene Sender als Empfänger entsprechend zu reagieren. Mit Hilfe von guarded commands können Unterscheidungen getroffen werden (zum Beispiel if...else if....). Wenn mehrere Wächter zutreffen, ist die Auswahl nichtdeterministisch.

3.3 Wie arbeitet der *Korridor*-Algorithmus zum Überwachen mehrerer Eingabequellen (`select()`) in etwa?

Der Korridor-Algorithmus arbeitet ähnlich wie ein Bürobote, der auf seinem Rundgang durch einen Korridor mit vielen Büros geht. Der Bote hat einen „Warteraum", in dem er im ruhenden Zustand sitzt. An jedem Büro ist eine Lampe o.ä. angebracht, die abzuarbeitende Aufträge anzeigt.

Wenn der Bote von seinem Warteraum aus sieht, dass ein Auftrag abzuarbeiten ist, geht er an den Türen aller Büros vorbei und sammelt die zu diesem Zeitpunkt anstehenden Aufträge ein. Dies tut er in festglegter Reihenfolge. Der Bürobote sollte sich, bevor er sich wieder in den Warteraum begibt, noch einmal umsehen, ob in der Zwischenzeit nicht neue Aufträge zu Beginn seiner Runde entstanden sind.

3.4 Worin unterscheiden sich die Eigenschaften der folgenden UNIX-Mechanismen zur Interprozesskommunikation:

- Pipes,
- Named Pipes,
- Sockets?

Wie lassen sich die Kommunikationseigenschaften von Sockets mit normaler Briefpost und Telefongesprächen vergleichen?

3.5

Pipe: Unidirektionaler sequentieller Bytestrom von stdout eines Prozesses zu stdin eines anderen Prozesses. Um eine Pipe verwenden zu können, müssen die beiden Prozesse verwandt sein.

Named Pipe (FIFO): Das Pipe-Objekt benötigt einen eindeutigen Namen (Kanal-Bezeichner). Die beiden miteinander kommunizierenden Prozesse müssen nicht verwandt sein, auch eine Named Pipe ist ein unidirektionaler sequentieller Bytestrom.

Sockets: Es gibt zwei Arten von Sockets: Stream-Sockets und Datagram-Sockets. Stream-Socket: Bidirektionaler Pipe, Bytestrom, sequentielle übertragung, zuverlässig und verbindungsorientiert (Vergleich: Telefongespräch) Datagram-Socket: Paketübergabe, Nachrichten, beliebige Reihenfolge, u.U. unzuverlässig (Verluste sind möglich), verbindungslos (Vergleich: Brief-/Paketzustellung)

3.6 Wie lassen sich die Kommunikationseigenschaften von Sockets in die generische Systemaufrufschnittstelle zum Zugriff auf Dateien einordnen?

Man erzeugt sie, bekommt den Deskriptor zurück und kann auf ihnen operieren.

3.7 Warum kommt dem Adressierungsproblem in der Interkommunikation eine so große Bedeutung zu? In welche zwei Teile zerfällt eine Adressangabe typischerweise?

Ohne eindeutige Adressierung ist Kommunikation unmöglich. Nur die eindeutige und funktionierende Adressierung ermöglicht den Austausch von Nachrichten bzw. Inhalten. Eine Adresse zerfällt typischerweise in die Benennung des Rechners und die Benennung des Ports. Protokolle enthalten z.B. Verabredungen zur Informationsrepräsentation, zur Fehlersicherung, zur Flusskontrolle, zur Adressierung und zur Netzkoppelung. Dies ist üblicherweise in einer ganzen Protokollhierachie festgelegt, TCP/IP, OSI-Schichten.

3.8 Was ist ein (Kommunikations-)Protokoll?

Protokoll: Verabredung über die Art und Weise des Kommunikationsvorgangs.

3.9 Skizziere kurz einige typische Kommunikationsprobleme und je einen Lösungsvorschlag im Rahmen eines entsprechenden Protokolls.

Probleme und Lösungsansätze: - Kabel, d.h. übertragungsmedium kapput: via Routing eines anderen Weg suchen. - Bitkipper: via Fehlererkennung, z.B. Prüfsummenberechnung eine überprüfung durchführen und den Vorgang gegebenenfalls wiederholen. - überlastung des Empfängers, d.h. die Nachricht kann nicht angenommen werden: via Flusskontrolle überprüfen (Staumeldung). - Empfänger liegt in einem anderen System,

d.h. er arbeitet mit einem anderen Protokoll: die Daten müssen entsprechen umgewandelt werden.

3.10 Skizziere einige Eigenschaften typischer Netztopologien.

- Vollständige Vermaschung: Ist eine aufwändige, d.h. teure Möglichkeit. Zum einen wird besonders viel Verkabelung benötigt, zum anderen müssen alle Adressen aller beteiligten Computer auf jedem einzelnen Netzwerkrechner gespeichert werden.

- Ring: Ein Ringsystem bietet den Vorteil, dass die Nachrichten einfach in eine Richtung losgeschickt werden und die beteiligten Computer nur überprüfen, ob die Nachricht für sie ist. Wenn nicht, wird das Datenpaket weitergeleitet. Es hat den Nachteil besonders großer Störanfälligkeit. Wenn ein einziger Netzwerkrechner oder ein Kabel ausfällt, ist die Kommunikation „dahinterliegender" Rechner gestört.

- Bus: Bei einem Bussystem ist die größte Störanfälligkeit der Bus selbst. Fällt diese Leitung aus, sind u.U. alle Rechner vom Netz abgeschnitten. Ausserdem muss bei dieser Netzvariante besonderes Augenmerk auf die Reihenfolge der Nachrichtenübertragung gelegt werden.

- Stern: Bei einem Stern geschieht die Speicherung der Adressen für die Weiterleitung nur auf dem Zentralrechner. Besonderer Nachteil eines Sterns ist die Möglichkeit, dass der zentrale Rechner ausfällt, in diesem Fall ist keinerlei Kommunikation mehr möglich. Allerdings stört der Ausfall eines einzelnen Computers oder einer übertragungsleitung die Funktionsfähigkeit des restlichen Netzes in keiner Weise.

- allgemeines Netz: Ein allgemeines Netz ist durch die Möglichkeit bei Störungen andere Wege zu suchen, in seiner Funktionsfähigkeit eine sehr Fehler tolerante Netzwerkvariante.

3.11 Welche besondere Bedeutung kommt dem Protokoll IP zu?

Das Internet Protokoll sorgt für die Integration von Anwendungen und Medien. Es ermöglicht eine Netzübergreifende Adressierung und ist Paket-orientiert. Durch die mit IP bestehenden Möglichkeiten ist es zum wichtigsten und am weitesten verbreiteten Protokoll geworden.

3.12 Was ist ein *Remote Procedure Call* (RPC), und welche Parameter wird man typischerweise dabei übergeben? Was haben RPCs mit dem *Client/Server*-Modell zu tun?

TODO

4 Sicherheit

4.1 Nenne einige absichtliche und unabsichtliche Angriffe auf Hardware, Software und/oder Daten. Wie können solche Angriffe klassifiziert werden?

absichtliche Angiffe: - Hardware: Zerstörung, herbeiführen von Störungen, Diebstahl - Software: Zerstörung, Fälschung, Kopie, Viren, Würmer unabsichtliche Angiffe: - Hardware: Speisen, Getränke, Wasserschäden, Mäuse - Software: Bugs, Bedienfehler Klassifizierung: - Abfangen: Anzapfen, Manipulation - Modifikation: Daten verfälschen, Programme verändern - Unterbrechung: Zerstörung, Löschen von Daten

4.2 Welche grundsätzlichen *Sicherheitsziele* kann man unterscheiden? Was ist eine *Sicherheitspolitik*?

Eine Sicherheitspolitik soll das reibungslose Erfüllen der vom System zu erledigenden Aufgaben sicherstellen. Dabei befinden sich die einzelnen Faktoren in einem Spannungsfeld, hohe Kosten stehen z.B. immer wieder einer besonders hohen Sicherheit gegenüber. Entsprechend muss im Einzelfall die Wahrscheinlichkeit einer Betriebsstörung und ihre möglichen Folgen gegen den Sicherheitsaufwand abgewogen werden. Als grundsätzliche Sicherheitsziele können Gemeinhaltung, Unversehrtheit und Verfügbarkeit unterschieden werden.

4.3 Auf welche verschiedenen Arten kann sich ein Benutzer *authentifizieren*?

- Wissen: z.B. Passwort
- Besitz: z. B. Schlüssel, Chipkarte
- Persönliche Merkmale: z.B. Fingerabdruck, Iris, Sprache

4.4 Welche Komponenten enthält eine Zugriffskontrollmatrix? Wie ordnen sich die Dateizugriffsrechte in UNIX in dieses Schema ein?

Eine dreidimensionale *Zugriffskontrollmatrix* enthält für jeden User, jede Datei Lese- und Schreibrechte. Die drei Dimensionen sind also User-Identity, Rechte und Datei. Die Dateizugriffsrechte in UNIX ordnen dem Besitzer einer Datei, der zugehörigen Gruppe und dem "Rest der Welt"Lese-, Schreib- und Ausführungsrechte zu.

4.5 Charakterisiere *symmetrische* und *asymmetrische* Verschlüsselungsverfahren. Wie können sie zur Realisierung einer Vertraulichkeit eingesetzt werden? Warum werden häufig Mischformen eingesetzt?

Bei symetrischer Verschlüsselung existieren zwei gleiche Schlüssel. Das hat den Nachteil, das bei bekannt werden des Schlüssels die Vertraulichkeit nicht mehr gegeben ist. Bei asymetrischer Verschlüsselung existieren insgesamt vier Schlüssel: zwei private und zwei

öffentliche Schlüssel. Die öffentlichen Schlüssel können allgemein bekannt sein. Die Verschlüsselung selbst passiert mit dem privaten Schlüssel des Senders und dem öffentlichen Schlüssel des Empfängers. Die Entschlüsselung einer Nachricht erfolgt mit dem privaten Schlüssel des Empfängers und dem öffentlichen Schlüssel des Senders.

Asymetrische Verschlüsselung ist erheblicher rechenintensiver als Symetrische. Allerdings ist die symetrische Verschlüsselung unsicherer als die Asymetrische. Deshalb werden oft Mischformen eingesetzt, z.B. werden die symetrischen Schlüssel mit Hilfe asymetrischer Verschlüsselung ausgetauscht um die Schlüsselübergabe so sicher als möglich zu machen, die Kommunikation selbst findet dann aber mit Hilfe der symetrischen Verschlüsselung statt um eine angemessene Geschwindigkeit (z.B. Videokonferenzen) zu gewährleisten.