Disclaimer

Die Übungen die hier gezeigt werden stammen aus der Vorlesung *Betriebssysteme*! Für die Richtigkeit der Lösungen wird keine Gewähr gegeben. Anlagen sind im Ordner *Assets/-Betriebssysteme_uebung/* zu finden.

Übung 1

Aufgabe 1

Wie würden Sie heute, zu Beginn des Kurses, den Begriff "Betriebssystem"beschreiben? Sehen Sie Analogien zwischen den Aufgaben und Funktionen von Betriebssystemen und denen gesellschaftlicher oder wirtschaftlicher Einrichtungen?

Begriff "Betriebssysteme"beschreiben: Verknüpfung zwischen Hard- und Software.

- Ermöglicht Kommunikation
- Software: Browser, Office, Bibiliotheken,...
- Betriebssystem: Betriebssystemdienste, Resourcenmanagement, Schnittstellen, Bibiliotheken,...
- Hardware: CPU, GPU, E/A, Speicher, Netzwerk,...

eine Abstraktion von Hardware-Ressourcen, Sammlung von programmen, Vorteil: Komplexität verborgen vor dem Nutzer

Aufgabe 2

Wie sind diejenigen Betriebssysteme in Erscheinung getreten, mit denen Sie bisher schon gearbeitet haben? Welche Aufgaben haben sie gelöst? Welche Probleme haben sie gezeigt? verschiedene

Betriebssysteme:

- Universalsysteme (Windows, Linux, Mac, Android, iOS): Benutzerfreundlich, keine Spezialisierung (GUI gehört nicht dazu)
- eingebettete Systeme (Microkontroller): resourcensparend, belastbarkeit
- Echtzeitsysteme (motorsteuerung): deadlines!
- chipkarten/sicherheitssysteme: nicht wiederbeschreibbar, meist genau 1 Aufgabe

Aufgabe 3

In der Vorlesung haben Sie ein breites Spektrum an Einsatzgebieten für Betriebssysteme kennen gelernt, das verdeutlichen sollte, welche sehr unterschiedlichen Anforderungen heute an Betriebssysteme gestellt werden. Kennen Sie über diese Szenarien hinaus Beispiele für Einsatzgebiete, in denen folgende eine zentrale Rolle spielen? Was wären die jeweiligen Konsequenzen, wenn ein Betriebssystem dabei diese Eigenschaften nicht besitzen würde?

Echtzeitfähigkeit einhalten von Fristen ist Hauptziel, harte oder weiche EZS
Robustheit betrieb in widrigen Umgebungen (Umwelteinflüsse, Anwenderfehler)
Sicherheit Security: schutz gegen Angriffe von außen (Verschlüsselung etc); Safety: Schutz gegen Ängriffe von innen"(Speichersicherheit etc)
Korrektheit (gegenüber einer Spezifikation) Testen (keine vollständige Sicherheit), Verifikation (mathematischer Beweis)
Performanz Geschwindigkeit von Anwenderprogrammen, Effiziente Algorithmen für BS Dienste
Sparsamkeit BS Größe, energie- und ressourcenschonend
Skalierbarkeit Lastskalierbarkeit (Bsp Online shops), Ressourcenskalierbarkeit

Aufgabe 4

Wenn Sie auf einem der heute üblichen Computer Bürosoftware, Internetprogramme (Webbrowser, E-Mail-Clients etc.) und Audio/Video-Applikationen nutzen möchten und die Wahl eines Betriebssystems hätten: Welche der oben genannten und welche weiteren Eigenschaften würden Sie von diesem Betriebssystem erwarten? Anforderungen an Office-PC-BS (universalsystem): In-

teraktivität, Performance, Robustheit, korrektheit, sicherheit, sparsamkeit, bequemlichkeit, hohe abstraktionsebene

Aufgabe 5

Warum könnte es problematisch sein, ein und dasselbe Betriebssystem auf Großrechnern (Mainframes) und gleichzeitig auf eingebetteten Systemen einzusetzen? gleiches Betriebssystem für

Großrechner und eingebetteten Systemen? Portierbarkeitsprobleme

- Aufgaben sehr unterschiedlich; Batch-Verarbeitung, Interaktiv
- Ressourcen Verwaltung/Größen
- Platformspezifische Einheiten (Befehlssätze, RISC vs CISC)
- Funktionsumfang

Aufgabe 6

Welche Vorteile hat es aus Anwendersicht, Betriebssysteme als Virtualisierung von Maschinen zu verstehen? Vorteile für Anwender die Betriebssysteme als Virtualisierung zu sehen

- Bequemlichkeit
- Beherschbarkeit + Isolation

Übung 2

Repetitorium

- **Welcher Zusammenhang besteht zwischen den Konzepten Nebenläufigkeit und Parallelität? Wann können Aktivitäten auf einem System nur pseudoparallel ausgeführt werden?** Nebenläufigkeit ist die Voraussetzung für Parallelität. Nebenläufigkeit beschreibt wiederum den Umstand, dass zwischen zwei Aktivitäten keine kausalen Abhängigkeiten bestehen. Parallel, wenn zwei oder mehr Aktivitäten zeitgleich/ zeitlich überlappend ausgeführt werden können. Das wiederum bedeutet, dass man Aufgaben, bei welchen die Teillösungen immer aufeinander aufbauen nicht parallelisieren kann. Allerdings ist die gleichzeitige Berechnung von unabhängigen Lösungen kein Problem und somit kann sich auch ein enormer Geschwindigkeitszuwachs bieten. Aktivitäten können nur dann echt parallel ausgeführt werden, wenn auch mehrere unabhängige Prozessorcores dafür zur Verfügung stehen, das bedeutet, sobald nicht mehrere Prozessoren verfügbar sind, kann die Parallelität nur durch zeitliches Switching simuliert werden. Pseudoparallel weiter bei Abhängigkeit (z.b. Synchronisation)
- **Wozu dienen Prozessmodelle? Aus welchen Bausteinen setzt sich ein Prozessmodell zusammen? Welche finden sich typischerweise in Prozessdeskriptoren wieder?** Da wir einem Computer nicht einfach mehrere Aufgaben ohne Kontext und ähnliches vorwerfen können und das Rechnen darauf beginnen können, müssen wir etwas neues Einführen. Hier wählen wir Prozesse. Prozesse sind eine Betriebssystemabstraktion zur vollständigen Beschreibung einer sequentiell ablaufenden Aktivität. Im weiteren Verlauf ist es nun so, dass parallele Aufgaben durch parallele Prozesse beschrieben/repräsentiert werden.
  Ein Prozessmodell definiert nun Prozesse und die konkreten Prozesseigenschaften: zum Beispiel: Semantik der Operationen auf Prozessen, also die Auswirkungen von Erzeugen, Beenden, Anhalten, Fortsetzen, sowie auch die nichtfunktionalen Eigenschaften von Prozessen, wie beispielsweise die Echtzeiteigenschaften, Safetyeigenschaften, Securityeigenschaften.
  Ein Prozessdeskriptor oder PCB „Process Control Block“ beinhaltet Informationen zur Identifikation, Größe, Position, Zugriffsberechtigungen und Verwendung eines Segments und erlaubt somit erst das parallele Rechnen.
  Aufbau des Prozessmodells/ Process Control Block: Identifikationsbereicht + Schedulinginformation + Programmkontext (Instruction Pointer + Stackpointer + PSR) + Ereignismanagement + Accounting + Rechte + Ressourcen
- **Aus welchem Grund wurde das Thread-Konzept entwickelt? Welche zwei eigentlich unabhängigen Konzepte sind im Modell des ursprünglichen (Schwergewichts-)Prozesses vereint?** Das Thread-Modell wurde für die Parallelisierung nebenläufiger Aktivitäten erschaffen, da sich zeigte, dass ein neuer Prozess pro nebenläufiger Aufgabe eher unpraktisch ist. Die Hauptprobleme oder die wichtigsten negativen Aspekte zeigten sich bei dem hohen Managementaufwand, den hohen Kosten für die Isolation und die hohen Kosten für die Kommunikation der Prozesse untereinander.
  Im ursprünglichen schwergewichtigen Prozessen war das Konzept des Ressourcenmanagements und das Management der Parallelität vereint.
- **Wozu dienen Prozess- bzw. Threadzustände? Welche elementaren Zustände finden sich in jedem Prozessmodell?** Thread und Prozesszustände erlauben es mehrere Threads (pesudo)parallel auszuführen, bzw. erlauben das Scheduling, indem man per Zustand bestimmt, was ein Thread gerade macht. Hier gibt es als unterschiedliche Zustände „frisch“, „aktiv“, „blockiert“, „beendet“, „bereit“.

- **Warum benötigt jeder Thread einen eigenen Stack?** Da jeder Thread seine eigene Ausführungssequenz / seinen eigenen Code haben kann, muss er einen eigenen Stack (LIFO Speicher) haben, auf den er den Inhalt seines Programmzählers schieben/einfügen kann (wenn z.B. Funktionsaufrufe und Rückgaben stattfinden).
- **Worin besteht der Unterschied zwischen Kernel- und User-Level-Threads? Welche Vor- und Nachteile besitzt die jeweilige Form? Wo befinden sich die PCB- und TCB-Datenstrukturen?** Kernel-Level-Threads werden direkt durch das OS verwaltet und das Thread Management geschieht direkt durch den Kernel. Dadurch, dass sowohl die Kontextinformation als auch die Prozessthreads alle durch den Kernel gemenaged werden, kann man sagen, dass Kernel-level-threads typischerweise eher langsamer als User-Level-Threads sind und ihre Performanz durch Parallelität erreichen. (Multithreadingbetriebssystem)

  - Vorteile: Effiziente Nutzung von Multicore-Architekturen. Mehrere Threads des selben Prozess können auf verschiedenen Prozessoren geschedult werden. Ein blockierender Systemaufruf in einem Thread blockiert nicht auch gleichzeitig alle anderen Threads des gleichen Prozess.
  - Nachteile: Da der Kernel sowohl Threads als auch Prozesse verwalten und planen muss, benötigt er einen vollständigen Thread-Kontrollblock (TCB) für jeden Thread, um Informationen über Threads zu erhalten. Dies führt zu einem erheblichen Overhead und erhöht die Komplexität des Kernels. Die Threads auf Kernel-Ebene sind langsam und ineffizient. Beispielsweise sind die Thread-Operationen hundertmal langsamer als die Threads auf Benutzerebene.
  - User Level Threads werden durch Nutzer implementiert und der Kernel selbst hat kein Wissen über die Existenz dieser Threads und behandelt diese, als wären sie Single-Thread Prozesse. Userlevelthreads sind kleiner und schneller als kernel level threads. Sie werden durch einen Programmzähler (PC), Stapel, Register und einen kleinen Prozess-Steuerblock dargestellt. Außerdem gibt es keine Kernel-Beteiligung an der Synchronisation für Threads auf Benutzerebene.
  - ULT: Thread-Implementierung in Anwendung (OS kennt keine Threads)
    Vorteile:
    Threads auf Benutzerebene sind einfacher und schneller zu erstellen als Threads auf Kernel-Ebene. Sie lassen sich auch leichter verwalten. Threads auf Benutzerebene können auf jedem Betriebssystem ausgeführt werden. In Threads auf Benutzerebene sind keine Kernelmodus-Privilegien zum Threadwechsel erforderlich. Thread-Management ohne Systemaufrufe anwendungsindividuelle Thread-Schedulingstrategien möglich (für Spezialanwendungen sinnvoll)
    Nachteile:
    Multithread-Anwendungen in Threads auf Benutzerebene können Multiprocessing nicht zu ihrem Vorteil nutzen. Der gesamte Prozess wird blockiert, wenn ein Thread auf Benutzerebene blockierende Operationen durchführt.
    Beim Kernel-level-Thread befinden sich sowohl der PCB als auch TCB im Kernelspace. Beim User-level-Thread befinden sich PCB im Kernelspace und der TCB im Userlevel.

Aufgabe 1: Prozesserzeugung in Linux-Systemen

In Betriebssystemen der Unix/Linux-Familie werden neue Prozesse durch den fork-Systemaufruf erzeugt. Dabei entsteht sukzessive eine Abstammungshierarchie, in der ein Prozess, der ein

fork() (erfolgreich) ausführt, zum Elternprozess ("parent") des von ihm erzeugten Kind-Prozesses ("child") wird. Die bei Unix/Linux-Systemen benutzte Technik funktioniert wie folgt: Durch fork wird eine nahezu exakte Kopie des Elternprozesses zum Zeitpunkt des fork()-Aufrufs erzeugt, bei der der neue Kindprozess eine Vielzahl der Eigenschaften des Elternprozesses erbt. Falls der Kindprozess ein anderes als das vom Elternprozess vererbte Programm ausführen soll, kann das Kind unmittelbar nach fork einen Systemaufruf der exec[ute]-Familie verwenden, der das durch den aufrufenden Prozess ausgeführte Programm austauscht. a) Informieren Sie sich über $fork$

und exec∗ und finden Sie Antworten auf die folgenden Fragen. Wie unterscheiden sich Eltern- und Kindprozess unmittelbar nach dem fork()-Aufruf?

Unterscheidung: getrennte Speicherbereiche, unterschiedliche PIDs, Programmierung gleich Direkt nach dem fork Aufruf unterscheiden sich Vater und Kindprozess durch:

- Das Kind hat seine eigene und einzigartige PID, also ProzessID
- Die ProzessID des Vaterprozesses des Kindes ist die selbe wie die ProzessID des Vaters.
- Das Kind erbt keine Speichersperren/Memorylocks der Eltern
- Prozessressourcenauslastung und CPU-Zeitzähler werden im Kind auf 0 gesetzt
- Der Satz ausstehender Signale ist ursprünglich leer
- Das Kind erbt keine Semaphoranpassungen des Elternteils
- Das Kind erbt keine prozessbezogenen Datensatzsperren von seinem Elterneteil
- Das Kind erbt keine Zeitgeber von seinem Elterneteil
- Das Kind erbt keine austehenden E/A Operationen oder Kontexte

Weiterhin gibt es einige Linux-Spezifische Prozessattribute welche sich verändern, hierzu zählen insbesondere:

- Das Kind erbt keine Verzeichnisänderungsbenachrichtigungen von seinem Elternteil
- Speicherzuordnungen, die mit dem *madvise*(2) MADV_DONTFORK-Flag markiert wurden, werden nicht über einen $Fork()$ vererbt.

Woran können sich Eltern- und Kindprozess unmittelbar nach einem fork()-Aufruf selbst erkennen ("Wer bin ich?")? Finden Sie mindestens 3 Möglichkeiten.

Selbsterkkennung: *getpid*(), *getppid*(), *system*()-calls, Rückgabewert von $fork()$

Welche unterschiedlichen Werte kann der Funktionsaufruf fork() zurückgeben, und was bedeuten sie?

Bei Erfolg wird im Elternprozess der PID des Kindprozesses ¿0 zurückgegeben, im Kindprozess wird 0 zurückgegeben. Bei einem Fehlschlag wird -1 im Elternprozess zurückgegeben, es wird kein Kindprozess erstellt und errno wird entsprechend gesetzt.

Rückgabewert von $fork()$:

- PID des Kindes von Parent
- 0 im Kindprozess
- -1 Fehler (errno gesetzt)

b) Demonstrieren Sie mit dem einfachen C-Programm p1 (in der Anlage), dass nach der Ausführung von fork() tatsächlich zwei Prozesse existieren. $cc - op1p1.c$ // Kompiliert das Pro-

gramm Dann in selbem Directory ./p1, dies startet das Programm

c) Führen Sie Programm p2 aus. Dieses enthält einen execl-Systemaufruf, mit dem ein Programm p4 ausgeführt werden soll. Das Programm p4 muss dabei ein (mit dem C-Compiler) übersetztes, ausführbares Programm im gleichen Verzeichnis sein. Sie können dazu das vorgegebene Programm p4 verwenden, das lediglich einen Ausdruck erzeugt.

d) Wie viele Prozesse werden durch das Programm p3 erzeugt? Warum? Was passiert, wenn execl() nicht erfolgreich ausgeführt werden kann, weil z. B. das Programm p4 nicht gefunden wird? Führen Sie zur Kontrolle p3 aus, während das Programm p4 einmal existiert und ein weiteres Mal, während dieses nicht existiert. Es werden insgesamt 4 Prozesse durch die Ausführung von

p3 erzeugt.

Erklärung: Es werden, falls es sich um einen Kindprozess handelt, fork() also = 0 ist, dessen PID und die des Elternknotens ausgegeben. Weiterhin wird, falls es sich um einen Kindknoten handelt, das Kindprozessimage durch execl durch das Prozessimage p4 ersetzt.

Sollte es sich allerdings nicht um einen Kindknoten handeln, dann wird die PID des Elternprozess ausgegeben.

Dann wird unabhängig von der bisherigen Auswahl ein Fork erstellt, „PID terminating“ ausgegeben und wait(NULL) aufgerufen. wait(NULL) blockiert den Elternprozess, bis eines seiner Kinder beendet ist. Wenn der Kindprozess beendet wird, bevor der Elternprozess wait(NULL) erreicht, wird der Kindprozess zu einem Zombie-Prozess, solange bis der Elternprozess auf ihn wartet und ihn aus dem Speicher freigibt.

Im Fall, dass p4 nicht vorhanden ist, wird zuerst der Elternprozess ausgegeben. Dann wird ein Fork erstellt und Terminating ausgegeben. Der Elternprozess verfällt in den Wartemodus. Daraufhin wird p3 wieder aufgerufen, da es sich nun um einen Fork handelt, wird Ausgegeben, dass ein Kind erstellt wurde und es wird versucht dieses Kindimage durch das p4 image zu ersetzen. Dies funktioniert allerdings nicht, also wird von execl -1 zurückgegeben.

Aufgabe 2: Prozessdeskriptoren und Prozesszustände

Beschäftigen Sie sich mit dem Shell-Kommando ps. Es dient dazu, bestimmte Informationen aus den Prozessdeskriptoren ausgewählter Prozesse auszugeben. Die ausgewählte Prozessmenge und der Umfang der wiedergegebenen Informationen kann dabei durch entsprechende Parameter beeinflusst werden. So bedeutet z. B. $ps-el$, dass eine Liste mit vielen Parametern (l, "long") für alle Prozesse (e) erzeugt wird. Aus Gründen der Übersichtlichkeit kann auch $ps-al$ verwendet werden. a) Welche Prozesszustände sind in Linux-Betriebssystemen definiert, und wie erkennt

man diese an den Ausgaben von ps? Man erkennt die Zustände an dem STAT Wert.

Mit ps lassen sich Daten über die Prozesse in der Prozeßtabelle anzeigen. Die Prozeßtabelle

wird mit einer Titelzeile ausgegeben. Die Spalten haben folgende Bedeutung:

PID	
PPID	
UID	
USER	
PRI	
NI	
SIZE	
TSIZE	
DSIZE	
TRS	
SWAP	
D	
LIB	
RSS	
SHARE	
STAT	Der Status des Prozesse
WCHAN	
TIME	
%CPU	
%MEM	
COMMAND	
TTY	

b) Starten Sie das Programm p5, in dem der durch fork() erzeugte Kindprozess in einer Endlosschleife läuft (Zweck?). Welche Prozesszustände haben Eltern- und Kindprozess? Beobachten Sie, welche Komponenten der Prozessdeskriptoren sich in Abhängigkeit von der Zeit ändern und interpretieren Sie dies. Das einzige was sich an Programm p5 zeigt, ist dass sich die Rechenzeit

erhöht, sonst werden keine Prozesse mehr geforkt. Der Parentprozess ist in Zustand S+, d.h. er läuft zwar im Vordergrund, ist allerdings im unterbrechbaren Schlaf und wartet darauf, dass ein Event fertiggestellt wird. Der Kindprozess ist im Zustand R+, was bedeutet, dass er gerade ausgeführt wird und im Vordergrund läuft.

Im Lauf der Zeit verändert sich vor allem die Laufzeit des Programms, also die CPU Zeit

c) Wenn ein Prozess auf ein sogenanntes Ereignis warten muss, beispielsweise eine Eingabe, dann ändert sich dessen Zustand. Starten Sie Programm p6, welches mittels getchar() auf eine Eingabe vom Standardeingabegerät (ohne weitere Maßnahmen ist dies die Tastatur) wartet, und untersuchen Sie die Zustandsinformationen des wartenden Prozesses. Der Prozess p6 selbst ist

im S+ State, was bedeutet, dass er im aufweckbaren Zustand ist und auf die Eingabe wartet. Die CPU Zeit ändert sich nicht, was bedeutet, dass keine Zeit verbraucht wird.

d) Mit dem Systemaufruf sleep() kann sich ein Prozess selbst ßchlafen legen". Starten Sie p7 und untersuchen Sie, welchen Zustand der hierdurch erzeugte Prozess nach dem Aufruf von sleep() einnimmt. Bei aufruf von sleep() wird der Prozess sofort alle CPU Ressourcen freigeben,

also fällt die CPU Auslastung für den Prozess auf 0, weiterhin wird der Zustand für die Zeit von sleep(time) auf S+ gesetzt, nach Ablauf der Zeit wird es sich wieder auf R+ (also runnable)

wechseln. CPU Zeit wird währenddessen natürlich auch nicht verbraucht.

Aufgabe 3: Dateiformate ausführbarer Programme

Kompilierte Programme liegen immer in einem genau definierten Format vor. Ziel dieser Aufgabe ist es, Erkenntnisse darüber zu gewinnen, wie ein Betriebssystem aus einem kompilierten Programm einen Prozess erzeugt. Ein in Linux-Betriebssystemen verbreitetes Binärformat ist ELF (Executable and Link Format), welches Gegenstand dieser Aufgabe ist. a) Im Mittelpunkt

Ihrer Recherchen über ELF sollte stehen, welche Informationen das Betriebssystem zur Erzeugung eines Prozesses benötigt und wo und wie diese in ELF zu finden sind. Berücksichtigen Sie ebenfalls die Metainformationen, die sich im ELF-Header befinden. Finden Sie Antworten auf die folgenden Fragen.

- ***Wie findet man (bzw. das Betriebssystem) die erste auszuführende Instruktion innerhalb des Text-Segments?*** *Erste Instruktion e_entry gibt die virtuelle Adresse an, an welcher der Prozess zuerst beginnt.*
- ***Auf welche Weise bekommen bereits im Quellprogramm (z. B. C-Programm) initialisierte Variablen ihre Anfangswerte vor dem Start der Ausführung eines Programmes?*** *Die Initialisierung mit Anfangswerten findet durch Einträge im Programmimage statt. Hier gibt es die Sections .data und .data1, welche beide jeweils Informationen zur Inititalisierung beinhalten.*
- ***Woran erkennt man, um welchen Typ einer in ELF dargestellten Datei es sich handelt? Für welche Dateitypen ist ELF prinzipiell vorgesehen?*** *Man erkennt dies an dem Eintrag in e_type im ELF Header (NoFileType, Relocatable, Executable, Shared Object, CoreFile, ProcessorSpecific)*
- ***Unterscheiden sich ELF-Dateien für 32-Bit- und 64-Bit-Prozessorarchitekturen? Woran ist das gegebenenfalls erkennbar?*** *e_ident/EI-CLASS identifiziert die Kapazität oder die Dateiklasse. Falls der Wert auf 1 ist, so handelt es sich um 32 Bit Objekte, falls der Wert auf 2 gesetzt ist um 64 Bit Werte*
- ***Welchen Zweck haben die so genannten Sektionen (sections) bzw. die program headers?*** *Die Headertabelle einer Objektdatei ermöglicht es, alle Abschnitte der Datei zu finden. Die Headertabelle ist ein Array von $Elf32\_Shdr$-Strukturen. Ein Tabellenindex der Headertabelle ist ein Subskript in diesem Array. Das e_shoff-Mitglied des ELF-Headers gibt den Byte-Offset vom Anfang der Datei in die Sectionheadertable; e_shnum gibt an, wie viele Einträge die Sektionskopftabelle enthält; e_shentsize gibt die Größe jedes Eintrags in Bytes an. Die Programmkopftabelle einer ausführbaren oder gemeinsam genutzten Objektdatei ist eine Anordnung von Strukturen, die jeweils ein Segment oder andere Informationen beschreiben, die das System benötigt, um das Programm für die Ausführung vorzubereiten. Ein Objektdateisegment enthält einen oder mehrere Abschnitte. Programm-Header sind nur für ausführbare und gemeinsam genutzte Objektdateien von Bedeutung. Eine Datei-Spezifikation bestimmt seine eigene Programm-Header-Größe mit der e_phentsize des ELF-Headers und e_phnum-Mitglieder. Der ELF-Programmheader wird durch den Typ $Elf32\_Phdr$ oder $Elf64\_Phdr$ je nach Architektur gegeben.*

- ***Welche Bedeutung hat eine Symboltabelle als Teil einer in ELF dargestellten Datei?*** *Die Symboltabelle einer Objektdatei enthält Informationen, die benötigt werden, um die symbolischen Definitionen und Verweise eines Programms zu lokalisieren und zu verschieben. Ein Symboltabellenindex ist ein Subskript in diesem Array. Index 0 bezeichnet sowohl den ersten Eintrag in der Tabelle als auch den undefinierten Symbolindex.*

b) Untersuchen Sie experimentell Binärdateien hinsichtlich ihrer ELF-Metainformationen. Hierzu können Sie die Werkzeuge readelf und objdump verwenden, um zu ermitteln, welche konkreten Informationen eine Datei im ELF-Format enthalten kann. Als Beispiele sollen mindestens die folgenden ELF-Binärdateien dienen:

- *das ls-Utility, zu finden im Verzeichnis /bin,*
- *die Executable zum Programm p1.c (siehe Anlage zu Aufgabe 1),*
- *eine dynamisch ladbare Bibliothek aus dem Verzeichnis /lib oder /usr/lib.*

```
readelf -a p1
```

Übung 3

Repetitorium

- **Durch welche Ereignisse wechselt ein Thread vom Zustand aktiv in den Zustand blockiert? Durch welche in den Zustand suspendiert?**
  - Der Zustandswechsel geschieht beispielsweise dadurch, dass ein aktiver Prozess auf E/A Operationen oder Timer warten muss, aufgrund dessen wird er dann in den Zustand blockiert geschickt, weiterhin kann es an einem Mangel an Betriebsmitteln liegen.
  - Ein Prozess wechselt in den Zustand suspendiert/ready, wenn es initial rechenbereit war, allerdings aus dem Hauptspeicher geswapped wurde und auf einem externen Speicher ausgelagert (ßwapping") wurde. Der Prozess wird wieder aktiv, wenn er aus dem externen Speicher zurück in den Hauptspeicher geswappt wird.
  - Ein Prozess wechselt in den Zustand suspendiert/blockiert. Vom Konzpet ähnlich dem des Suspendiert/ready, mit dem Unterschied, dass der Prozess eine E/A Operation ausführte oder darauf wartete und ein Mangel an Hauptspeicher für die Auslagerung sorgte. Sobald allerdings die E/A Operation abgeschlossen ist, kann er in den Zustand suspendiert/bereit wechseln.
  - Der Nutzer kann selbst suspenderieren, allerdings geschieht es typischerweise eher durch das OS selbst. (z.b. durch Überlastungssituation)
- **Welche Auswirkung hat im Round-Robin-Schedulingalgorithmus die Veränderung der Größe der Zeitscheibe?**
  - Sollte eine große Zeitscheibe eingesetzt werden, so gibt es wenige Threadwechsel, dadurch einen geringen Schedulingoverhead, allerdings ist die Reaktivität eher schlecht, zudem entstehen bei nicht-preemptiver Implementierung viele Abschnitte mit Leerlaufsituationen.
  - Sollte eine kleine Zeitscheibe verwendet werden so hat man zwar einen sehr großen Overhead durch ständige Threadwechsel, allerdings eine sehr hohe Reaktivität.
  - In der Praxis sind typische Zeitscheiben zwischen 20-50ms lang.
- **Round-Robin-Scheduler verwalten normalerweise eine oder mehrere Listen von Prozessen, wobei jeder lauffähige Prozess genau einmal aufgeführt wird. Was würde passieren, wenn ein Prozess 2x in einer Liste stehen würde? Aus welchen Gründen könnte man so etwas erlauben?**
  - Wenn ein Prozess häufiger in der Liste stehen würde, so würde dieser anteilig gesehen häufiger zur Ausführung zugelassen werden, was also einer Erhöhung der Priorität gleich kommen würde, allerdings kommt er dadurch nicht unbedingt schneller zur Ausführung. Falls er n-mal in der Liste ist, dann bekommt er die n-fache Rechenzeit pro Listendurchlauf.
  - Unterschied zu mehreren Listen: Prozess mit hoher Prioriität rechnet solange, bis Prozess mit gleicher oder größerer Priorität existiert oder er terminiert.
  - Die Gründe dies zu erlauben sind offensichtlich. Die Einführung längerer Zeitscheiben für Prozesse höherer Priorität würden dazu führen, dass die Reaktivität anderer Threads verringert würde und es würde es dem Betriebssystem erschweren, bei einem Interrupt wieder einzuspringen. Die Aufrechterhaltung separater Listen für die Priorisierung von Prozessen würde einen viel komplexeren Scheduler erfordern, der

im Bezug auf die Zyklen teurer würde, bzw. höhere Kosten verursachen würde, dies lässt sich durch die mehrfache Einfügung eines Prozesses aber umgehen. Ein Problem könnte maximal beim Löschen von Threads auftrete, da ein Prozess nun häufiger in der Queue sein kann, müsste man alle Vorkommnisse finden und löschen, was eine Kostensteigerung des Löschens zur Folge hat.

- **Welche Form des Schedulings – preemptiv oder nicht preemptiv – führt aus grundsätzlichen Überlegungen zu robusteren Systemen?** Ein preemptives System kann einem Prozess Ressourcen wegnehmen und später wieder erneut zuweisen, um zwischenzeitlich andere Prozesse rechnen zu lassen. Da somit Fehlerquellen wie Endlosrekursionen verhindert oder gestoppt werden können. Somit sind preemptive Systeme als robuster zu bewerten.

Aufgabe 1: EDF

Die Scheduling-Strategie Earliest Deadline First (EDF) kommt dann zum Einsatz, wenn die Abarbeitung eines Prozesses bis zu einem definierten Zeitpunkt (Frist, Deadline) erfolgen muss. Wir wollen uns in dieser Aufgabe auf statische Deadlines beschränken, auch wenn diese in bestimmten, realen Anwendungen während der Abarbeitung von Prozessen (dynamisch) neu bestimmt werden können. Die Strategie ist nun wie folgt: Ein Prozess wird einem anderen vorgezogen, falls er eine frühere Deadline hat. Ein neu ankommender – und aufgrund einer zeitigeren Deadline höher priorisierter – Prozess verdrängt einen bereits rechnenden, aber niedriger priorisierten Prozess. Das Verfahren ist also präemptiv. a) Implementieren Sie EDF. Sie finden dazu eine

Musterklasse im Projekt unter Simulation → Source Packages → frm.pssav.sim → PreemptiveEDFScheduler.java.

```
package frm.pssav.sim;

import frm.pssav.sim.OperatingSystem.ProcessControlBlock;
import java.util.Comparator;
import java.util.concurrent.PriorityBlockingQueue;

/**
 * Scheduler using the preemptive earliest deadline first algorithm
 */
public class PreemptiveEDFScheduler extends Scheduler {

    PreemptiveEDFScheduler(OperatingSystem os, int queueCapacity) {
        super(os, new PriorityBlockingQueue<Integer>(
                queueCapacity, new PreemptiveEDFQueueComparator(os)));
    }

    /**
     * Methode zur Implementierung des präemptiven Teils des
     * Scheduling-Algorithmus.
     */
```

```
@Override
void doSchedule(int[] arrivals) {
    for (int pid : arrivals) {
        queue(pid);
    }

    if (isProcessTerminated()) {
        contextSwitch();
    }
    else {
        if (!isProcessRunning()) {
            contextSwitch();
        }
        else {
            /* @student
             * Diese Methode wird immer dann aufgerufen, wenn dem
             * Scheduler ein neuer Prozess bekannt wird. Sie können
             * hier also den präemptiven Teil ihres Algorithmus
             * implementieren.
             * Mittels getOS().getProcess([PID]) gelangen Sie an den
             * PCB des Prozesses mit dem Identifier PID. Ein kurzer Blick
             * in die Klasse ProcessControlBlock in "OperatingSystem.java"
             * gibt Ihnen Aufschluss darüber, wie Sie auf die Daten des
             * PCB zugreifen können. Mit getRunningPID() bekommen Sie
             * die ID des gerade laufenden Prozesses. getReadyQueue()
             * liefert Ihnen die Schlange der wartenden Prozesse.
             * Mittels contextSwitch() befehlen Sie dem Scheduler den
             * Prozess am Kopf der Schlange anstatt des aktuellen zu
             * rechnen. Dabei wird der aktuelle aber nicht automatisch wieder
             * in die Schlange eingereiht!
             * PS: Die Warteschlange enthält die PIDs der Prozesse.
             */
            if (getReadyQueue().isEmpty()) return;

            int runningPID = getRunningPID();
            int headPID = getReadyQueue().peek();
            ProcessControlBlock runningPCB = getOS().getProcess(runningPID);
            ProcessControlBlock headPCB = getOS().getProcess(headPID);

            if(runningPCB.getDeadline() > headPCB.getDeadline())
            {
                contextSwitch();
                getReadyQueue().add(runningPID);
            }
            else
            {
                // change nothing
            }
        }
```

```
    }
}

@Override
Comparator<Integer> getComparator(OperatingSystem os) {
    return new PreemptiveEDFQueueComparator(os);
}

/**
 * Verlgeichsmethode für die Priority-Queue.
 */
public static class PreemptiveEDFQueueComparator implements
        Comparator<Integer> {

    private OperatingSystem os;

    public PreemptiveEDFQueueComparator(OperatingSystem os) {
        this.os = os;
    }

    @Override
    public int compare(Integer o1, Integer o2) {
        ProcessControlBlock p1 = os.getProcess(o1);
        ProcessControlBlock p2 = os.getProcess(o2);
        /* @student
         * Diese Methode wird genutzt, um die Schlange der wartenden
         * Prozesse zu sortieren. Sie müssen hier also definieren, welche
         * Prozesse zunächst wichtiger als andere sind. Geben Sie -1 für
         * "o1 ist wichtiger als o2", 0 für "o1 ist genauso wichtig wie o2"
         * und 1 für "o1 ist weniger wichtig als o2" zurück.
         * Welche Methoden Ihnen für die Datenabfrage aus dem PCB zur
         * Verfügung stehen, erfahren Sie mit einem Blick in die Klasse
         * ProcessControlBlock in der Datei "OperatingSystem.java".
         */
        /*
        Hier kann ich p1.getDeadline() oder eventuell p1.getPriority() verwen
        */
        if(p1.getDeadline() > p2.getDeadline())
        {
            return 1;
        }
        else if(p1.getDeadline() < p2.getDeadline())
        {
            return -1;
        }
        else if(p1.getDeadline() == p2.getDeadline())
        {
            return 0;
        }
```

```
                return 0;
            }
        }
    }
```

b) Vergleichen Sie nun den einfachen Round-Robin-Algorithmus (eine Warteschlange) mit EDF. Überlegen Sie sich zwei Szenarien: Im ersten sollen beide Algorithmen die gesetzten Fristen einhalten. Im zweiten sollte ersichtlich werden, in welchen Fällen Round Robin gegenüber EDF versagt. Benutzen Sie ausreichend viele Prozesse, so dass die Simulation lange genug läuft, damit ihre Kommilitonen genügend Zeit haben, sich der dargestellten Probleme bewusst zu werden.

Zwei Prozesse definieren (0,4,4,10) und (2,5,5,8) (Arrival,Burst,Priority,Deadline). Mit RR können die Deadlines nicht eingehalten werden.

	Arrival Time	Burst Time	Priority	Deadline	
P1	0	4	4	10	
P2	2	5	5	8	

Aufgabe 2: Round Robin mit Prioritäten

Die Scheduling-Strategie Round Robin soll ein möglichst faires Scheduling mehrerer Prozesse ermöglichen. In der Vorlesung haben Sie die zusätzliche Möglichkeit kennengelernt, Round Robin mit einem Prioritätenschema zu kombinieren. Wir wollen uns in dieser Aufgabe auf statische Prioritäten beschränken, auch wenn diese in bestimmten, realen Anwendungen während der Abarbeitung von Prozessen (dynamisch) neu bestimmt werden können. a) Implementieren

Sie Round Robin mit Prioritäten. Sie finden dazu bereits eine Implementierung der Strategie ohne Berücksichtigung von Prioritäten unter Simulation → Source Packages → frm.pssav.sim → RoundRobinScheduler.java, die Sie entsprechend anpassen müssen. (siehe Netbeans)

b) Von welchen Faktoren ist die Länge einer Zeitscheibe in der Praxis abhängig? Von welchen die Priorität? Welcher Unterschied ergibt sich daraus für das Setzen dieser beiden Parameter?

Demonstrieren Sie die Auswirkungen unterschiedlich langer Zeitscheiben sowie unterschiedlicher Prioritäten anhand zweier Szenarien: Eines mit (genügend vielen) sehr kurzen Prozessen, ein anderes mit sehr viel länger rechnenden. Diskutieren Sie, welche realen Einflussfaktoren – die in PSSAV nicht berücksichtigt werden können – hier in der Praxis eine Rolle spielen müssen.

Aufgabe 3: Linux-Scheduling

Moderne Universal-Betriebssysteme müssen heute mit Prozessen und Threads sehr unterschiedlichen Charakters umgehen können. Einerseits könnten z. B. Echtzeitprozesse zur Audio/Video-Verarbeitung ablaufen, andererseits gibt es auch zeitunabhängige aber rechenzeitintensive Prozesse wie beispielsweise das in der Vorlesung gezeigte Ray-Tracing-Programm. a) Recherchieren Sie

für das Betriebssystem Linux die für das Scheduling von Prozessen verantwortlichen funktionalen Komponenten (Scheduling-Subsystem) und ermitteln Sie, welche Schedulingstrategien Linux unterstützt.

- Linux verwendet seit Kernelversion 2.6.23 CFS (Completely Fair Scheduler)
- Es gibt verschiedene Schedulingstrategien wie:
  - SCHED_FIFO: First in First out Scheduling
  - SCHED_RR: Round-Robin Scheduling
  - SCHED_DEADLINE: Sporadic task model deadline scheduling
  - SCHED_OTHER: Default Linux time-sharing scheduling
  - SCHED_BATCH: Scheduling batch processes
  - SCHED_IDLE: Scheduling very low priority jobs
- Der Scheduler selbst ist eine Kernelkomponente welche entscheidet, welcher ausführbare Thread als nächster auf der CPU ausgeführt werden wird. Die von Linux verwendete Schedulingklasse basiert auf der POSIX Expansion für Echtzeitcomputersysteme.

Damit das Betriebssystem sinnvoll mit diesen unterschiedlichen Prozessen umgehen kann, müssen Prozesse selbst ihre Lastmerkmale dem Betriebssystem mitteilen (neben der Beobachtung und Klassifizierung der Prozesse durch das Betriebssystem selbst). Eine sehr einfache Möglichkeit, das Scheduling von Prozessen zu beeinflussen, besteht in der Vergabe unterschiedlicher (Basis-)Prioritäten, auf deren Grundlage die dynamische Änderung durch das Betriebssystem vorgenommen wird. Für jeden regulären Prozess steht hierzu der Systemaufruf nice zur Verfügung; ein Nutzer kann die Priorität seiner Prozesse mithilfe der Ausführung der Kommandos nice oder renice verschlechtern (der Name deshalb, weil er damit "nettßu anderen Prozessen ist). b)

Starten Sie zwei gleichartige rechenzeitintensive Prozesse (ggf. hierfür ein einfaches Programm schreiben), die lange genug rechnen. Demonstrieren und erläutern Sie deren Verhalten, wenn einerseits beide die gleiche Priorität haben und andererseits ein Prozess seine Priorität verringert hat. Überlegen Sie sich geeignete Demonstrationsmöglichkeiten.

- Verwendete taskset 0x1 ./simpleproc um die selben Prozessorcores zu verwenden
- über top fand ich die Prozessorauslastung in Prozent heraus
- Darüber hinaus zeigte sich bei meinen Programmen mit verschiedenen Prioritäten allerdiungs nicht wirklich ein Unterschied. Einzig für Prozess wie den NTP Deamon oder ASLA/-Pulseaudio würde es Sinn machen, denn wenn man beispielsweise der musikausgebenden Anwendung sehr viel Priorität wegnimmt und beispielsweise einem unbedeutenden Programm hinzufügen würde, so würde man hören, dass die Musik lückenhaft wird, bzw. das Programm unresponsiv wird.

Siehe auch Linux Manpages

Hinweise: Falls Sie nicht selbst geeignete Ideen haben: Beispielsweise lassen sich lange laufende Prozesse durch Hochzählen einer Variablen vom Typ long int (lange ganzzahlige Variable) erzeugen, oder durch Starten mehrerer Instanzen einer ausreichend anspruchsvollen Anwendung Ihrer Wahl (anspruchsvoll für den Hauptprozessor, nicht nur für die Grafikhardware).
Falls Sie an einem System mit einem Mehrkernprozessor arbeiten, könnten die zu zeigenden Effekte unsichtbar bleiben, wenn z. B. jeder Prozess auf einem eigenen Prozessorkern läuft. Beschäftigen Sie sich hierzu z. B. mit dem Systemaufruf $sched_setaffinity()$.

Übung 4

Repetitorium

Welche prinzipiellen Probleme entstehen, wenn parallel ablaufende Threads auf einen gemeinsamen Speicherbereich zugreifen? Welche beiden konkreten Synchronisationsprobleme bestehen beim Erzeuger/Verbraucher-Problem?

- Es entstehen verschiedene Probleme bei Verwendung eines gemeinsamen Speicherbereichs durch verschiedene Threads, hierzu zählen:
  - Zugriff auf Arbeitsspeicher ist bestenfalls wortweise atomar (4/8 Byte). Garantiet beim parallelen Lesen und Schreiben kein Ergebnis.
  - Das Grundproblem ist, dass zwei Threads gleichzeitig versuchen können Operationen auf dem Speicher auszuführen, wenn jetzt beide versuchen etwas zu speichern und zeitgleich ihre Daten schicken, ohne vorher in ausreichendem Maße zu prüfen, ob es zu diesem Zeitpunkt möglich ist, laufen sie in Gefahr entweder zeitglich zu schreiben (produziert Müll). Weiterhin können sie, falls der gemeinsame Speicher zur Synchronisation eingesetzt wird, durch Schedulingalgorithmen so beeinflusst werden, dass vorherige Tests auf Schreiberlaubnis nicht mehr gültig sind, auch hier würde sich ein Problem ergeben. Lösungsansätze müssen den folgenden Regeln genügen:
  - Korrektheit: D.h. im kritischen Lese/Schreib Abschnitt befinden sich zu jedem Zeitpunkt höchstens ein Thread
  - Lebendigkeit: Der Thread soll irgendwann die Möglichkeit haben, zur Ausführung zu kommen.
  - Verhungerungsfreiheit: Es gibt keinen Thread, der für immer vor einem kritischen Abschnitt wartet.
- Beim Erzeuger-Verbraucher-Problem zeigen sich zwei Synchronisationsprobleme:
  1. unterschiedliche Geschwindigkeiten von Erzeuger und Verbraucher führen zu leeren Puffern oder Pufferüberläufen, somit müssen hier die relativen Geschwindigkeiten synchronisiert werden. (Lösung über Zählsemaphoren)
  2. gleichzeitiges Lesen und Schreiben auf dem selben Pufferelement führt eventuell zu Inkonsistenzen, somit haben wir hier den Fall eines kritischen Abschnitts und müssen diesen durch Synchronisation, z.b. durch wechselseitigen Ausschluss, schützen.

In der Vorlesung haben wir binäre Semaphore kennen gelernt, die innerhalb eines kritischen Abschnitts strengen wechselseitigen Ausschluss garantieren. Eine allgemeinere Form der Semaphore lässt eine gewisse feste Anzahl von Aktivitäten in einen kritischen Abschnitt hinein, bevor die Sperrwirkung eintritt. Wie können derartige Semaphore implementiert werden?

- Ersetze Zustandsvaribale durch Zähler. (initialisiert mit maximaler Threadanzahl)
- P: Dekrementiert den Zähler, blockiert wenn Zähler = 0;
- V: Inkrementiert den Zähler (setzt wartenden Thread fort)
- Derartige Semaphore können mit mehreren gleichartigen Pufferelementen realisiert werden. Weiterhin könnte man eine Warteschlange verwenden um die wartenden Prozesse zu verwalten. Bei Eintritt eines Prozess in den kritischen Abschnitt würde eine

Zählvariable hochgezählt, welche bei Verlassen des kritischen Abschnitts wieder dekrementiert würde. Ist der Wert der Zählvariable gleich der maximal erlaubten Anzahl an laufenden Prozessen im kritischen Abschnitt, so werden neu anfragende Prozesse vorerst in die Warteschlange geschickt.

Wie viele Semaphore braucht man mindestens zur Lösung des allgemeinen Erzeuger/Verbraucher-Problems?

- Einen binären Semaphor für den wechselseitigen Ausschluss. Zwei Zählsemaphore (je für Anzahl leerer Pufferelemente und belegter Pufferelemente)
- Man benötigt mindestens 3 Semaphore um das Problem ausreichend gut lösen zu können. Hierzu zählen ein Semaphor A, welcher vor modifizierenden Zugriffen auf die Datenstruktur schützen soll. Wenn ein Prozess P1 gerade die Datenstruktur verändert, so zeigt er dies durch die P-Operation an. Wenn nun allerdings ein anderer Prozess P2 die Datenstrukur modifizieren, so wird dieser P2 solange blokiert, bis P1 die V-Operation aufruft. Semaphor B stellt sicher, dass Elemente zum Lesen verfügbar sind, wenn ein Verbraucher die Leseoperation ausfühhren möchte. Falls der Puffer leer ist, wird die P-Operation den aufrufenden Prozess blockieren, und ihn erst dann freigeben, wenn ein Erzeuger die V-Operation aufruft. Ist die Warteschlange gefüllt, so darf bei P-Operation eingetreten werden. Ein dritter Semaphor C, überwacht den Schreibezustand und blockiert Erzeuger, welche die P-OP aufrufen, wenn die Schlange schon voll ist. Ist sie nicht voll darf der Erzeuger weiter produzieren. Ein blockierter Prozess wird durch V-OP von einem Verbraucher wieder aufgeweckt.

Wir haben Bedingungsvariable im Kontext Hoare'scher Monitore kennen gelernt. Ist ein derartiges Synchronisationsmodell (Warten auf Erfüllung einer Bedingung) nicht auch außerhalb und losgelöst vom Monitormodell nützlich? Falls ja, worauf müsste man in einem solchen Fall achten?

- Klar, immer dann, wenn zustandslos ausreicht. Monitore garantieren wechselseitigen Ausschluss, losgelößt müssen wir das auch noch berücksichtigen. Somit: Bedingungsvariablen meist zusammen mit Locks/Mutex benutzt.

```
P(semaphore s) {
  atomicBegin(s);
  if (s.zustand = frei)
     s.zustand ← belegt;
  else
     s.warteliste ← aufrufer;
     scheduler.suspend(aufrufer);
  fi
  atomicEnd(s);
}
```

```
V(semaphore s) {
  atomicBegin(s);
  if (s.warteliste = leer)
     s.zustand ← frei;
  else
     scheduler.continue
        (s.warteliste.vorne)
  fi
  atomicEnd(s);
}
```

Aufgabe 1: Das Problem des schlafenden Barbiers

In einem Friseurladen gibt es einen Friseur, einen Frisierstuhl (an dem der Friseur arbeitet) und n Stühle für wartende Kunden. Entwickeln Sie einen Algorithmus, der unter den nachfolgenden

Annahmen Friseur und Kunden so synchronisiert, dass jeder wartende Kunde (irgendwann) bedient wird.

- Der Friseur und alle Kunden agieren parallel.
- Falls keine Kunden da sind, geht der Friseur schlafen.
- Wenn ein Kunde kommt, während der Friseur schläft, weckt der Kunde den Friseur und setzt sich in den Frisierstuhl (und wird bedient).
- Wenn ein Kunde eintrifft, während der Friseur arbeitet und ein freier Kundenstuhl vorhanden ist, setzt sich der Kunde und wartet.
- Trifft ein Kunde ein, während der Friseur arbeitet und alle Kundenstühle belegt sind, verlässt der Kunde den Friseurladen sofort wieder.
- Wenn der Friseur mit dem Bedienen eines Kunden fertig ist, verlässt dieser Kunde den Friseurladen und einer der wartenden Kunden (falls vorhanden) belegt den Frisierstuhl und wird bedient (sonst gilt Bedingung 2).

Idee: Es gibt vier Ressourcen um die sich die Kunden und der Barbier streiten. Dazu zählen: Der Friseurstuhl, der Barbier selbst, die Kunden selbst und das Wartezimmer. Es werden daher vier Semaphore verwendet. Der Warteraum sollte auf einen mehrwertigen Semaphoren gesetzt werden, ebenso wie der Barbier und die Kunden. Für Rasur-fertig reicht eine Einwertige.

```
Semaphor Barbier = 0
Semaphor Haarschnitt = 0 // benötigt man nicht zwingend
CountSemaphor Kunden = 0
Semaphor SitzZugriff = frei
int Sitze = N //Wartezimmer

Barbier()
{
    while(true)
    {
        DOWN(Kunden);
        P(SitzZugriff);
        freieSitze++;
        V(Barbier);
        V(SitzZugriff);
        Haarschnitt();
        V(Haarschnitt);
    }
}

Kunde()
{
    P(Sitzzugriff) // bei if-clause muss in allen Zweigen wieder freigegeben werd
    if(freieSitze > 0)
    {
        freieSitze--;
        UP(Kunden);
        V(SitzZugriff)
```

```
            P(Barbier);
            SchneideHaare();
            P(Haarschnitt);
        }
        else
        {
            gehe();
            V(SitzZugriff);
        }
    }
```

Aufgabe 2: Das Achterbahnproblem

Das Achterbahnproblem nach J.S. Herman wird durch das folgende Szenario beschrieben. Eine Anzahl n von "Vergnügungssüchtigen"(im Folgenden Passagiere genannt) versucht, möglichst oft eine Fahrt mit einem der m zur Verfügung stehenden Achterbahnwagen zu unternehmen. Dabei gelten allerdings die folgenden Bedingungen.

- *Ein Wagen darf nur losfahren, wenn er voll besetzt ist. (Dabei gilt: Jeder Wagen fasst c Passagiere, wobei die Gesamtzahl der beteiligten Passagiere mehr als nur einen Wagen füllt, d. h. $c < n$.)*
- *Wenn ein Wagen vollständig besetzt ist, fährt er los.*
- *Wenn der Wagen nach Abschluss der Fahrt wieder anhält, steigen alle Passagiere aus und bemühen sich erneut, in einem "neuen"Wagen eine weitere Fahrt zu unternehmen. (Unter entsprechenden Umständen kann es natürlich auch wieder der gleiche Wagen sein.)*
- *Wagen dürfen sich nicht überholen (was ja beim gegebenen Achterbahnproblem auch technisch nicht möglich ist), d. h. die Reihenfolge der Wagen bleibt immer gleich.*

Passagiere und Wagen sollen durch Aktivitäten simuliert werden, die synchronisiert werden müssen. Für eine der möglichen Lösungen könnten die folgenden Hinweise hilfreich sein: Bei der Ankunft eines Wagens sollte dieser eine Prozedur Einsteigen() aufrufen, danach sollten c Passagiere ihrerseits eine Prozedur $In_Wagen_Einsteigen()$ aufrufen. Nach beendeter Fahrt sollte ein anhaltender Wagen eine Prozedur Aussteigen() aufrufen, und die sich im Wagen befindenden c Passagiere sollten daraufhin eine Prozedur $Wagen_Verlassen()$ aufrufen.

- Autos: warten auf Passagiere (bis voll)
- Passagiere: warten auf Start/Ende

```
CountSem belegt = 0;
CountSem frei = 0;
Sem Start = 0;
Sem Ende = 0;

Auto()
{
        while(true)
        {
```

```
                for(j in 1...m)
                {
                        for(i in 1...c)
                        {
                                DOWN(belegt)
                        }
                        for(i in 1...c)
                        {
                                UP(start)
                        }
                        Fahre()
                        for(i in 1...c)
                        {
                                UP(Ende)
                        }
                        for(i in 1...c)
                        {
                                UP(frei)
                        }
                }
        }
}


Passagier()
{
        while(true)
        {
                UP(belegt)
                DOWN(Start)
                DOWN(ENDE)
                DOWN(frei)
        }
}


int count = 0;
semaphore queue = c;
semaphore boarding = 0;
semaphore riding = 0;
semaphore unloading = 0;
semaphore check-in = 1;

Prozess Take-Ride()
{
        down(Queue);
        down(Check-In);
        if(++count == Maximum)
        {
```

```
                up(Boarding);
        }
        up(Check-In);
        down(Riding);
        up(Unloading);
}

Prozess Car()
{
        int i,j;
        for(i = 0; i < NumberOfRides;i++)
        {
                for(j = 1; j <= Maximum; j++)
                {
                        up(Queue);
                }
        down(Boarding); // Jetzt sind alle Passagiere in einem Wagen.
        count = 0; //Passagiere sind in einem Auto
        for(j = 1; j <= Maximum, j++)   // ab hier ausladen
        {
                up(Riding);
                down(Unloading);
        }
        }
}
```

Aufgabe 3: Der Kaffeeautomat

Ein Kaffeeautomat, seine Kunden und ein Lieferant, der den Automaten regelmäßig mit Kaffee und Kaffeebechern auffüllt, sollen sich mittels Semaphoren synchronisieren. Synchronisieren Sie das Verhalten dieser Aktivitäten so, dass folgendes Verhalten realisiert wird.

- *Der Automat kann entweder einen Kunden bedienen oder durch den Lieferanten nachgefüllt werden. Beide Vorgänge sind nicht gleichzeitig möglich!*
- *Ein Kunde muss nach Aufforderung durch den Automaten eine 1-Euro-Münze als Bezahlung einwerfen, erst danach bekommt er seinen Kaffee. (Um eine ungeeignete Betriebsweise auszuschließen, soll angenommen werden, dass sich nur Kunden anmelden, die eine 1-Euro-Münze parat haben – und nach Aufforderung natürlich auch einwerfen!)*
- *Der Lieferant bekommt durch den Automaten mitgeteilt, dass dieser für den Auffüllvorgang bereit ist.*
- *Ein einmal gestarteter Vorgang (Bedienen bzw. Auffüllen) kann nicht mehr unterbrochen werden. Eine neue Anmeldung (durch den nächsten Kunden oder den Lieferanten) wird erst nach Abschluss dieses Vorgangs akzeptiert.*
- *Der nächste Kunde und der Lieferant können sich jeweils unabhängig voneinander beim Automaten anmelden. Die Reihenfolge der Bedienung hängt dann davon ab, in welcher Reihenfolge die Anmeldungen erfolgen.*
- *Lieferant und Kunde bekommen den Abschluss des jeweiligen Vorgangs durch den Automaten mitgeteilt.*
- *Falls der Kaffeevorrat verbraucht ist oder keine Becher mehr vorhanden sind, versetzt sich*

der Automat selbst in einen Wartezustand und wartet bis er durch den Lieferanten wieder befüllt ist.

- Lieferant: wartet auf Wartungsauftrag
- Kunden: wartet auf Kaffee
- Maschine: Portionen = 0? Wartet auf Wartung : warten auf Bestellung & Bezahlung

```
Semaphor Wartungsauftrag = 0
int Portionen = N
Semaphor PortionenZugriff  = 0
Semaphor Kaffee = 0
Semaphor Wartung = 0
Semaphor Bestellung = 0
Semaphor Bezahlung = 0

Lieferant()
{
        while(true)
        {
                P(Wartungsaufwand)
                P(PortionenZugriff)
                Portionen = N;
                V(Portionenzugriff)
                V(Wartung)
        }
}

Kunde()
{
        P(PortionenZugriff)
        if(Portionen > 0)
        {
                V(Bestellung)
                V(PortionenZugriff)
                V(Bezahlung)
                P(Kaffee)
        }
        else
        {
                gehe()
                V(PortionenZugriff)

        }
}
```

```
Kaffeemaschine()
{
        while(true)
        {
                P(Portionenzugriff)
                if(Portionen > 0)
                {
                        V(Portionenzugriff)
                        P(Bestellung)
                        P(Bezahlung)
                        MacheKaffee()
                        V(Kaffee)
                }
                else
                {
                        V(PortionenZUgriff)
                        V(Wartungsauftrag)
                        P(Wartung)
                }
        }

}

int Portions = n // Anzahl verfuegbarer Portionen
semaphore maintenanceDone = 0;
semaphore doMaintenance = 0;
semaphore request = 0;
semaphore pay = 0;
semaphore coffee = 0;

Machine {
        while(true)
        {
        if(Portions > 0)
                up(request);
                down(pay);
                makeCoffee();
                Portions = Portions -1;
                up(coffee);
        }
        else // Der Automat benoetigt einen Service
        {
                up(doMaintenance);
                down(MaintenanceDone);
        }
        }

}
```

```
Customer{
        while(true)
        {
                down(request);
                up(pay);
                down(coffee);
        }
}

ServiceMan{
        while(true)
        {
                down(doMaintenance)
                Portions = N; //nachfuellen
                up(maintenanceDone)
        }
}
```

Übung 5

Diskussionsfragen

Frage 1: Speicherbasierte vs. nachrichtenbasierte Interprozesskommunikation *Stellen Sie sich vor, Ihre Übungsgruppe müsste ein Videoschnittsystem entwickeln, welches aus Robustheitsgründen die Berechnung komplexer Effekte, die En- und Dekodierung verschiedener Formate sowie die Steuerung der gesamten Applikation in jeweils unterschiedlichen Prozessen implementieren soll. Diese Prozesse müssen natürlich miteinander kommunizieren und dabei neben Kontrollinformationen auch die zu verarbeitenden Video- und Audiodaten austauschen. Prinzipiell stehen dafür nachrichtenbasierte und speicherbasierte Kommunikationsmechanismen zur Verfügung. Für welche der existierenden Mechanismen würden sie sich entscheiden, um einerseits Kontrollinformationen und andererseits Mediendatenströme auszutauschen? Begründen Sie Ihre Antwort.*
Hinweis: Klären Sie zuerst, was die prinzipiellen Vor- und Nachteile dieser beiden Kommunikationsvarianten sind. Betrachten Sie anschließend die Kommunikationsmuster und Anforderungen der beiden Klassen (Kontroll- und Multimediadaten), bevor Sie eine Empfehlung geben.

- **Message Passing vs. Shared Memory** Shared Memory hat die wünschenswerte Eigenschaft, dass die gesamte Kommunikation über implizites Laden und Speichern in einem globalen Adressraum erfolgt. Eine weitere grundlegende Eigenschaft von Shared Memory ist, dass Synchronisation und Kommunikation getrennt sind. Zusätzlich zu den Lade- und Speicheroperationen müssen spezielle Synchronisationsoperationen (Mechanismen) verwendet werden, um zu erkennen, wann Daten produziert und/oder konsumiert wurden. Im Gegensatz dazu wird beim Message Passing ein explizites Kommunikationsmodell verwendet. Explizite Nachrichten werden zwischen den Prozessen ausgetauscht. Synchronisation und Kommunikation sind im Message Passing vereint. Die Erzeugung von entfernten, asynchronen Ereignissen ist ein integraler Bestandteil des Message-Passing-Kommunikationsmodells. Es ist jedoch wichtig, darauf hinzuweisen, dass Shared-Memory- und Message-Passing-Kommunikationsmodelle universell sind, d.h., es ist möglich, das eine zu verwenden, um das andere zu simulieren. Es ist jedoch zu beobachten, dass es einfacher ist, Shared Memory mit Message Passing zu simulieren als umgekehrt. Das liegt im Wesentlichen an der asynchronen Ereignissemantik von Message Passing im Vergleich zur Polling-Semantik des Shared Memory. Das Shared-Memory-Kommunikationsmodell ermöglicht es dem Programmierer, sich auf die mit der Parallelität verbundenen Probleme zu konzentrieren, indem er von den Details der Interprozessorkommunikation entlastet wird. In diesem Sinne stellt das Shared-Memory-Kommunikationsmodell eine geradlinige Erweiterung des Uniprozessor-Programmierparadigmas dar. Darüber hinaus ist die Shared-Memory-Semantik unabhängig vom physikalischen Speicherort und daher offen für die dynamische Optimierung, die das zugrunde liegende Betriebssystem bietet. Auf der anderen Seite ist das Shared-Memory-Kommunikationsmodell im Wesentlichen eine Polling-Schnittstelle. Dies ist ein Nachteil, was die Synchronisation betrifft. Die Nachrichtenweitergabe kann als ein interruptgesteuertes Kommunikationsmodell charakterisiert werden. Bei der Nachrichtenübermittlung enthalten die Nachrichten sowohl Daten als auch Synchronisation in einer einzigen Einheit. Als solches eignet sich das Message-Passing-Kommunikationsmodell für Betriebssystemaktivitäten, bei denen die Kommunikationsmuster im Voraus explizit bekannt sind, z. B. E/A, Interprozessor-Interrupts und Task- und Datenmigration. Auf der anderen Seite leidet das Message Passing unter der Notwendigkeit von Marshaling-Kosten, d. h. den Kosten für das Assemblieren und Disassemblieren der Nachricht. Eine natürliche Schlussfolgerung aus der obigen Diskussion ist, dass sich Shared-Memory- und Message-Passing-

Kommunikationsmodelle jeweils für bestimmte Anwendungsdomänen eignen. Shared Memory bietet sich für Anwendungsentwickler an, während Message Passing sich für Betriebssystementwickler anbietet. Es ist daher naheliegend, die Kombination von Shared Memory und Message Passing in Mehrzweck-Multiprozessorsystemen in Betracht zu ziehen. Dies war die Hauptantriebskraft hinter Systemen wie dem Stanford FLexible Architecture for SHared memory (FLASH) System. Dabei handelt es sich um ein Multiprozessorsystem, das die Unterstützung für Shared Memory und Message Passing effizient integriert und dabei sowohl den Hardware- als auch den Software-Overhead minimiert.

- **Nachrichtenbasiert**: + kein Blockieren bei asynchroner Variante, einfache, einheitliche Nutzung (Architekturunabhängig), keine Synchronisation nötig, - ggf. kopieren der Daten (ineffizient und redundant), Blockieren bei der synchronen Variante, unidirektional (geht nur in eine Richtung)
- **Speicherbasiert**: + Ideal für große Dateien, da annähernd Verzögerungsfrei, nicht blockierend (im allgemeinen), kann bidirektional verwendet werden - Synchronisation notwenig, gemeinsamer Speicher, also nicht für verteilte Systeme geeignet, aufwändiger bei korrekter Implementierung
- Idee: Verwenden von Speicherbasierterkommunikation für die großen Video & Audiodateien, da man diese nur schwer & insbesondere langsam per Nachrichten verschicken kann. Verwendung von Messaepassing für Kontroolldatenströme

Frage 2: Synchronisation durch Semaphore *Bei asynchroner nachrichtenbasierter Kommunikation kommen stets Warteschlangen zum Einsatz, um unterschiedliche Geschwindigkeiten der Sender- und Empfängerprozesse auszugleichen. Der Zugriff auf diese Warteschlangen muss aus verschiedenen Gründen durch Synchronisationsmechanismen (z. B. Semaphore) geregelt werden. Was sind diese Gründe und weshalb sind insgesamt drei Semaphore pro Warteschlange notwendig?*

Benötigen drei Semaphore: Schreibzugriff auf volle Schlange, Lesezugriff auf leere Schlange, und Zugriffskontrolle.

Frage 3: Synchronisationsvarianten bei nachrichtenbasierter Kommunikation *Welche Nachteile asynchroner Kommunikation treten beim Einsatz synchroner Varianten der Sende- und Empfangsoperationen nicht auf? Warum ist es trotzdem manchmal sinnvoll oder unumgänglich, die asynchronen Varianten einzusetzen? Nennen Sie mindestens drei Beispiele realer Applikationen, in denen asynchron kommuniziert wird.*

- Pufferspeicher: Größe?
- eventuell Synchronisation notwendig. (z.B. für Datenströme), eventuell extra Techniken zur Benachrichtigung oder Synchronisation
- Trotzdem notwenig wenn:
  - Hoher Grad an Parallelität notwendig

  - Ereignisse sporadisch oder unvorhersehbar (hier synchrones Warten ineffizient)
  - Auf Ereignisse zeitnah reagiert werden muss (z.B. in Echtzeitsystemen)
- Es wird beispielsweise in E-Mails, Whatsapp, SMS asynchron kommuniziert

Frage 4: Management asynchroner Ereignisse *Welche Alternativen haben die Entwickler von Betriebssystemen, um mit asynchron auftretenden Ereignissen (Mausbewegungen, Einstecken von USB-Geräten etc.) umzugehen? Welche Technik erlaubt es auch einem Benutzerprozess, auf asynchrone Ereignisse zu reagieren, ohne direkten Hardwarezugriff zu haben?*

- Busy Waiting (Warte in Endlosschleife / sehr ineffizient)
- Polling (Wahl der Zykluszeit)
- Interrupts (HW-Signal, Behandlung über Routine aus IVT)
  - inline-Prozeduraufruf
  - IPC über Botschaften
  - pop-up threads
- Hierzu gibt es die Möglichkeit, dass der Prozessor alle Eingabe-Geräte zyklisch abfragt (Polling). Was bei der Vielzahl an Komponenten in einem Computer bedeuten würde, dass der Prozessor mit nichts anderem mehr beschäftigt wäre.
- Eine Alternative ist die sogenannten Unterbrechungsanforderung (to interrupt, unterbrechen), die dann eintritt, wenn Daten von außen anstehen. Dazu wurde die Möglichkeit geschaffen den Hauptprozessor auf definierte Weise bei der laufenden Arbeit zu unterbrechen.
- Auf Anwendungsebene: Registrieren von eigenen Signalhandlern

Aufgabe 1: Nachrichtenwarteschlangen (Message Queues)

a) Recherchieren Sie die Funktionsweise, Charakteristiken und Eigenschaften von Message Queues. Wie wird der Kontrollfluss der Prozesse dabei durch das Betriebssystem gesteuert (z. B. Synchronisation durch Blockierungen, durch die Ankunft von Daten usw.)?

- Hier werden Nachrichten von einem Prozess in eine Nachrichtenschlange (Message Queue) geschickt, welche typischerweise nach dem FIFO Prinzip arbeitet. Jede Messagequeue ist durch einen eindeutigen Bezeichner gekennzeichnet, unidirektional und mit festgelegtem Format.
- MessageType: Unterscheidung innerhalb der Warteschlange möglich.
- send blockiert, wenn die Queue voll ist.
- receive blockiert, wenn keine Nachricht mit dem spezifizierten Type in der Queue ist.

- Es gibt auch eine nichtblockierende Variante. (IPC-NOWAIT)
- Das Einsatzgebiet der Warteschlangen ist typischerweise die Datenübergabe zwischen asynchronen Prozessen in verteilten Systemen.

b) In der Anlage zu dieser Übungsaufgabe (u4-a1-anlage) befinden sich ein Server- und ein Client-Programm, die beide Lücken enthalten. Vervollständigen und übersetzen Sie die Programme. Starten Sie anschließend zuerst den Server und dann den Client. Falls Sie die Lücken richtig ausgefüllt haben, muss das Client-Programm ein "Passwortän den Server senden und anschließend ein Geheimnis ausgeben, das es vom Server als Antwort erhalten hat.

Aufgabe 2: Gemeinsamer Speicher (Shared Memory)

a) Recherchieren Sie die Funktionsweise, Charakteristiken und Eigenschaften von Shared Memory. Wie wird der Kontrollfluss der Prozesse dabei durch das Betriebssystem gesteuert? (Wann und wodurch erfolgt eine Synchronisation?)

- Shared Memory ist eine durch das Betriebssystem bereitgestellte Möglichkeit, bei welcher mehrere Prozesse gemeinsam auf einen gemeinsamen Speicher zugreifen können. Um dies zu erreichen muss zuerst ein gemeinsamer Datenspeicher angelegt werden. Nachdem dies geschehen ist, muss der Datenspeicher den Prozessen bekanntgemacht werden (einfügen in deren Adressraum), welche darauf zugreifen sollen dürfen.
- Wichtig hierbei: Shared Memory ist eine der schnellsten, wenn nicht die schnellste Art der Interprozesskommunikation, da das Kopieren/Versenden zwischen Clients/Server, bzw. verschiedenen Prozessen entfällt.
- Da man allerdings gemeinsam auf Speicher zugreift, ist eine Synchronisation, meist durch Semaphore oder Monitore unumgänglich.
- shmat() fügt das durch shmid identifizierte Speichersegment an den Adressraum des aufrufenden Prozesses an. Die Anfügeadresse wird durch shmaddr spezifiziert.
- Weiterhin gibt es Einschränkungen bezüglich der Rechte im shmflg Bitmaskargument:
  - SHM_EXEC: erlaubt eine Ausführung der Segmentinhalte
  - SHM_RDONLY: Fall gesetzt, so ist das Segment nur für den Lesezugriff angehängt, ist es nicht gesetzt, so ist Lesen und Schreiben erlaubt.
  - SHM_REMAP. Dieses Flag spezifiziert, dass das Mapping des Segments alle bisherigen Mappings im Bereich von shmaddr und den folgenden dateigrößelangen Segment ersetzt.
- Ist shmat() erfolgreich, so wird *shm_atime* auf die jetzige Zeit gesetzt, *shm_lpid* auf die ProzessID des aufrufenden Prozess gesetzt und *shm_nattch* wird um 1 erhöht.

b) In der Anlage zu dieser Übungsaufgabe (u4-a2-anlage) befinden sich ein Server- und ein Client-Programm, die beide Lücken enthalten. Vervollständigen und übersetzen Sie die Programme. Starten Sie anschließend zuerst den Server und dann den Client. Falls Sie die Lücken richtig

ausgefüllt haben, muss das Client-Programm ein "Passwortän den Server senden und anschließend ein Geheimnis ausgeben, das es vom Server als Antwort erhalten hat.
Hinweis: An den verwendeten Semaphoroperationen sind keine Änderungen notwendig.

Aufgabe 3: Benannte Pipes (Named Pipes, FIFOs)

a) Recherchieren Sie die Funktionsweise, Charakteristiken und Eigenschaften von Pipes und Named Pipes. Wie wird der Kontrollfluss der Prozesse dabei durch das Betriebssystem gesteuert? (Wann und wodurch erfolgt eine Synchronisation?)

- Pipes stellen einen unidirektionalen Kommunikationskanal zwischen zwei Prozessen dar. Ein Pipe hat ein Lese- und ein Schreibende, wobei man am Leseende diejenigen Daten lesen kann, welche zuvor auf das Schreibende geschrieben wurden.
- Wenn ein Prozess von einer leeren Pipe lesen will, so blockiert read() bis Daten vorhanden sind. Wenn ein Prozess versucht auf eine volle Pipe zu schreiben, dann wird write() solange blockieren, bis wieder genügend Platz auf der Pipe frei ist.
- Die über Pipes ablaufende Kommunikation ist nachrichtenfrei und bytestromorientiert.
- Eine Pipe hat eine begrenzte Kapazität
- POSIX besagt, dass write() von weniger als $PIPE\_BUF$ Bytes atomar sein muss, bei mehr als $Pipe\_BUF$ Bytes kann es auch nichtatomar sein.

```
O_NONBLOCK disabled, n <= PIPE_BUF
       All n bytes are written atomically; write(2) may block if
       there is not room for n bytes to be written immediately

O_NONBLOCK enabled, n <= PIPE_BUF
       If there is room to write n bytes to the pipe, then write(2)
       succeeds immediately, writing all n bytes; otherwise write(2)
       fails, with errno set to EAGAIN.

O_NONBLOCK disabled, n > PIPE_BUF
       The write is nonatomic: the data given to write(2) may be
       interleaved with write(2)s by other process; the write(2)
       blocks until n bytes have been written.

O_NONBLOCK enabled, n > PIPE_BUF
       If the pipe is full, then write(2) fails, with errno set to
       EAGAIN.  Otherwise, from 1 to n bytes may be written (i.e., a
       "partial write" may occur; the caller should check the return
       value from write(2) to see how many bytes were actually
       written), and these bytes may be interleaved with writes by
       other processes.
```

- Named Pipes / FIFO sind einer Pipe sehr ähnlich, mit dem kleinen Unterschied, dass darauf als Teil des Dateisystems zugegriffen wird. Wenn Prozesse jetzt Daten über FIFO austauschen, dann behandelt der Kernel alle Daten intern ohne sie in das Dateisystem zu schreiben. Somit dienen die FIFO Dateien im Filesystem nur als Referenzpunkt und Namen und geben Prozessen Informationen darüber, an welcher Stelle Prozesse auf die Pipe zugreifen können. Dies bedeutet aber auch, dass hier im Gegensatz zu unbenannten Pipes Prozesse miteinander kommunizieren können, die nicht miteinander verwandt sind.
- Der Kernel verwaltet genau ein Pipe-Objekt für jede FIFO-Spezialdatei, die von mindestens einem Prozess geöffnet wird. Das FIFO muss an beiden Enden (lesend und schreibend) geöffnet werden, bevor Daten übergeben werden können. Normalerweise wird auch das Öffnen der FIFO-Blöcke bis zum anderen Ende geöffnet.

b) In der Anlage zu dieser Übungsaufgabe (u4-a3-anlage) befinden sich ein Server- und ein Client-Programm, die beide Lücken enthalten. Vervollständigen und übersetzen Sie die Programme. Starten Sie anschließend zuerst den Server und dann den Client. Falls Sie die Lücken richtig ausgefüllt haben, muss das Client-Programm ein "Passwortän den Server senden und anschließend ein Geheimnis ausgeben, das es vom Server als Antwort erhalten hat.

Übung 6

Repetitorium

Wodurch sind die Mehrkosten eines Systemaufrufs im Vergleich zu einem regulären Prozeduraufruf

Für die Behandlung asynchroner Unterbrechungen (Interrupts) gibt es mehrere Modelle, u.a. auch d

Vor welchem fehlerhaften Verhalten von Anwendungsprozessen schützen private virtuelle Adressräu

Warum ist es nur schwer möglich, schon während der Übersetzung eines Programms dafür zu sorgen

Warum ist eine optimale Pagingstrategie im Allgemeinen nicht erreichbar? Unter welcher speziellen

Wie viele Seitentabellen gibt es in einem anlaufenden Betriebssystem mit virtueller Speicherverwaltu

Was ist die Arbeitsmenge (Working Set) eines Prozesses? Wozu dient sie? Zu welchen Zeitpunkten b

Welche Aufgaben hat eine MMU? Wozu dient ein TLB (Translation Look-aside Buffer)? Wann wird

Seitentabellen können je nach Adressraumgröße sehr groß werden. Mit diesem Problem gehen mehrs

Wie groß ist die Seitentabelle? Wie groß ist die Seitenrahmen- bzw. Kachelgröße?

- Um welchen Faktor reduziert sich die Größe der Seitentabelle, die ständig im Hauptspeicher zu halten ist, bei einem zweistufigen Seitentabellenverfahren gegenüber einer einstufigen Seitentabelle?
- Welche Größe haben die Seitentabellen bei einstufigen bzw. zweistufigen Verfahren bei einer Eintragsbreite von jeweils 4 Byte?

Bei einer Seitengröße von 8 KiByte und einem virtuellen Adressraum der Größe 2^{64}: Wie groß wäre

Aufgabe 1: Systemaufrufe

Dienstleistungen des Betriebssystems (z. B. Erzeugen von Prozessen, Threads oder Dateien) werden unter Zwischenschaltung von Stellvertreterprozeduren aufgerufen (z. B. für Programme in der Sprache C finden diese sich bei Linux-Systemen meist in der C-Standardbibliothek libc), die dann über einen aufwändigeren als den Prozedurmechanismus den tatsächlichen Sprung ins Betriebssystem implementieren. a) Warum kann man aus einem Anwendungsprogramm nicht direkt

eine im Betriebssystem implementierte Prozedur aufrufen? Erläutern Sie die gängige alternative Verfahrensweise. Welche prinzipiellen Bestandteile enthalten die genannten Stellvertreterprozeduren?

b) In der Anlage zur Aufgabe finden Sie das Programm syscall.c; dort ist beispielhaft ein Systemaufruf manuell implementiert. Warum enthält das Unterprogramm Assemblerbefehle? Erweitern Sie nach obigem Muster Ihren ëigenen"exit()-Systemaufruf my_exit(), indem Sie das begonnene Fragment ergänzen. Demonstrieren Sie dessen korrekte Funktionsweise.
Hinweis: Mit dem Systemaufruf wait() kann man den Terminierungsstatus von Kindprozessen

überprüfen. Um das Programm zu kompilieren, auszuführen und seinen Rückgabewert anzuzeigen, können Sie alternativ das beigefügte Shell-Skript run.sh benutzen.

c) Vor der Einführung des Maschinenbefehls syscall in modernen 64-Bit-Architekturen musste zum Auslösen eines Systemaufrufs durch ein Anwendungsprogramm ein spezieller Interrupt (trap) ausgelöst werden. Recherchieren Sie, welche Vorteile die Verwendung von syscall gegenüber dieser konventionellen Verfahrensweise hat. Auch heute noch wird aus Gründen der Abwärtskompatibilität und zur Unterstützung spezieller Prozessorarchitekturen der trap-Mechanismus durch den Linux-Kernel unterstützt. Schaffen Sie es, Ihre my_exit()-Implementierung mittels Ersetzen der syscall Instruktion durch einen Trap-Interrupt (int$0x80) entsprechend zu portieren? Beachten Sie, dass dabei Systemaufruf-Nummern für 32-Bit-Hardwarearchitekturen und andere Register zu verwenden sind.

Tipp zur gesamten Aufgabenstellung: Unter Linux können Sie mit strace < Programm > u.a. die Systemaufrufe eines Programms verfolgen.

Aufgabe 2: Ereignismanagement mit Linux-Signalen

In Linux-Systemen kann mittels des Signalmechanismus eine Behandlung bestimmter Ereignisse auf Prozessebene stattfinden (aus der Vorlesung als Ïnterruptbehandlung auf Prozessebene"bekannt).
Dieses Prinzip soll nun mit mittels sogenannter Dämon-Prozesse (daemons) veranschaulicht werden. Dabei handelt es sich um Hintergrundprozesse, die i. d. R. im Verlauf des Boot-Vorgangs gestartet werden und keine direkte Benutzerinteraktion vorsehen. So kümmert sich beispielsweise der Line Printer Daemon (lpd) oder der Common Unix Printing System Daemon (cupsd) darum, Druckaufträge nebenläufig zu anderen Benutzerprozessen abzuwickeln. Eine typische Verwendung von Signalen ist beispielsweise, einem Dämon die Änderung seiner Konfigurationsdatei(en) zur Laufzeit mitzuteilen, woraufhin diese erneut eingelesen werden sollen. Hierfür hat sich die Benutzung des Signals SIGHUP etabliert.

a) Recherchieren Sie, welche Signale der für Unix-Systeme etablierte POSIX-Standard vorsieht und wie diese einem Prozess zugestellt werden. Demonstrieren Sie in der Übung, wie man auf der Kommandozeile einem beliebigen Prozess ein Signal (z. B. SIGHUP oder SIGKILL) senden kann. Informationen hierzu finden Sie wie immer in den Linux-Manpages oder im Handbuch zur C-Standardbibliothek libc.

In der Anlage zu dieser Aufgabe stellen wir Ihnen den Quellcode eines kleinen Dämon-Prozesses zur Verfügung. Zur besseren Demonstration haben wir darauf verzichtet, ihn als Hintergrundprozess zu initialisieren; er läuft daher wie ein Nutzerprozess und kann so seine Ausgaben auf der Kommandozeile sichtbar machen.

b) Starten Sie den mitgelieferten Dämon und demonstrieren Sie, wie dieser auf verschiedene Signale reagiert. Erklären Sie das Verhalten mithilfe Ihrer Recherche aus Teilaufgabe a).

c) Erweitern Sie den Dämon nun um die Fähigkeit, seine Konfigurationsdateien neu zu laden, wann immer er das Signal SIGHUP empfängt. Sie können dazu als Reaktion auf das Signal die bereits vorhandene Funktion $load_config()$ aufrufen.

d) Schaffen Sie es, durch Reaktion auf die Signale SIGTERM und SIGKILL ein explizites Terminieren des Prozesses durch den Benutzer zu verhindern? Demonstrieren und erklären Sie Ihre Ergebnisse.

Aufgabe 3: Virtuelle Speicherverwaltung von Linux-Systemen

Stellen Sie das virtuelle Speichermanagement von Linux-Systemen vor. Gehen Sie dabei insbesondere auf die Struktur der Seitentabellen und die Seitengröße ein. Virtuelle Speicherverwaltung bietet auch eine elegante Möglichkeit zur dynamischen Speicherverwaltung, d. h. je nach Bedarf den von einem Prozess belegten Speicherplatz zu vergrößern bzw. wieder zu verkleinern. Die Motivation zur Integration dieser Technik in den Linux-Kernel war die Einführung dynamischer Objekte in Programmiersprachen, die z. B. mittels new() während des Programmablaufs bei Bedarf neue Variablen, insbesondere große Arrays, erzeugen und auch wieder löschen können. Zum Ansprechen der entsprechenden Implementierungen in Linux, werden auf Nutzerebene die Funktionen malloc() ("memory allocation") und free() (Wiederfreigeben von Speicher) bereitgestellt. Beide Funktionen sind über Eintrittspunkte in die C-Standard-Bibliothek implementiert. Freier Speicher wird dabei vom Heap besorgt.

a) Erklären Sie diesen Begriff im Linux-Kontext. Stellen Sie dann die beiden obigen Funktionen vor und demonstrieren Sie ihre Benutzung.

Im Linux-Kern benutzen malloc() und $free()$ den Systemaufruf $brk()$ ("break"), der ebenso freien Speicher requiriert, aber für die Verwendung als Programmierschnittstelle nicht empfohlen wird. Dieser ändert den so genannten Programm-"break". b) Was bedeutet dies? Wie kann man

dessen aktuellen Wert sowie dessen Maximalwert feststellen?

c) Wodurch unterscheiden sich die Funktionen malloc()/$free()$ und $brk()$?