Inhaltsverzeichnis

Tim Nisslbeck (hu78sapy)
hu78sapy@stud.cs.fau.de — wwwcip.cs.fau.de/~hu78sapy

1 April 2010

1.1 Aufgabe 1 (30 Punkte)

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
3	2	2	2	2	4	2	3	4	2	1	4	3	4	3

a	b	c	d
6.6̄%	**40.00%**	26.6̄%	26.6̄%

a) Welche Aussage bezueglich der Freispeicherverwaltung mittels einer Bitliste ist falsch?

- Der zu verwaltende Speicher wird in Speichereinheiten gleicher Groesse unterteilt.
- Zur Suche nach freiem Speicher kann es noetig sein, die gesamte Bitliste zu durchsuchen.
- **Das Zusammenfassen von benachbarten freien Speichereinheiten ist besonders aufwaendig.**
- Je kleiner die Speichereinheiten sind, desto laenger ist die Bitliste.

b) Was versteht man unter einem Translation Lookaside Buffer (TLB)?

- Einen speziellen Cache der MMU, der den Inhalt der zuletzt angesprochenen Speicherzellen vorhaelt.
- **Einen speziellen Cache der MMU, der Informationen aus den zuletzt genutzten Seitendeskriptoren vorhaelt.**
- Einen Pufferspeicher des Compilers, um den Uebersetzungsvorgang zu beschleunigen (es werden die Codes der zuletzt uebersetzten Statements vorgehalten).
- Der TLB ist eine schnelle Umsetzeinheit der MMU, die physikalische in logische Adressen umsetzt.

c) Wie gross ist die Seitentabelle zur Adressabbildung von logischen auf physikalische Adressen in einem System mit Seitenadressierung wenn ein Eintrag in der Seitentabelle 32 Bit gross ist, der virtuelle Adressraum 4 GiB umfasst und eine Speicherseite 8192 Byte gross ist?

- 512 bis 8.192 Byte
- **2.097.152 Byte**
- 4.194.304 Byte
- 16.777.216 Byte

d) Was versteht man unter Virtuellem Speicher?

- Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.
- **Speicher, der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber moeglicherweise groesser als der verfuegbare physikalische Hauptspeicher ist.**
- Unter einem Virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht fuer einen Anwendungsprozess.

e) Welche Aussage ist bezueglich der Seitenersetzungsstrategie Least Recently Used (LRU) richtig?

- Die LRU-Strategie benoetigt ein Referenzbit in jedem Eintrag der Seitenkacheltabelle.
- **Als Auswahlkriterium fuer die Ersetzung einer Seite wird die Zeit seit dem letzten Zugriff auf die Seite verwendet.**
- Zur Implementierung von LRU benoetigt man eine sehr genaue Systemuhr.
- Die LRU-Strategie gewaehrleistet, dass immer die Seiten eingelagert sind, auf die in der Zukunft zugegriffen wird.

f) Namensraeume dienen u. a. der Organisation von Dateisystemen. Welche Aussage ist richtig?

- Flache Namensraeume sind besonders einfach implementierbar und damit vor allem fuer Mehrbenutzersysteme gut geeignet.
- Flache Namensraeume erlauben pro Benutzer nur einen Kontext.
- Der Nachteil von hierarchischen Namensraeumen besteht darin, dass das Dateisystem spezielle Funktionen zum Aufloesen von Namenskonflikten implementieren muss.
- **In einem hierarchisch organisierten Namensraum duerfen gleiche Namen in unterschiedlichen Kontexten enthalten sein.**

g) Beim Einsatz von RAID 5 wird durch eine zusaetzliche Festplatte Datensicherheit erzielt, so dass der Ausfall einer Festplatte den laufenden Betrieb nicht stoeren kann. Welche Aussage dazu ist richtig?

- Es sind mindestens 5 Festplatten noetig.
- **Die Paritaetsinformation wird gleichmaessig ueber alle beteiligten Platten verteilt.**
- Der Lesezugriff auf ein Plattensystem mit RAID 5 ist langsamer als bei normalen Plattenzugriffen, da der Zugriff auf die Platten komplexer ist.
- Es duerfen nicht mehr als 5 Festplatten beteiligt sein, da sonst die Paritaetsinformation nicht mehr gebildet werden kann.

h) Welches Attribut ist nicht im Inode eines UNIX-Dateisystems verzeichnet?

- Dateityp (Verzeichnis, normale Datei, Spezialdatei)
- Eigentuemer
- **Dateiname**
- Zeitpunkt des letzten Dateizugriffes

i) Nehmen Sie an, der Ihnen bekannte Systemaufruf stat(2) waere analog zu der Funktion readdir(3) mit folgender Schnittstelle implementiert: struct stat *stat(const char *path); Welche Aussage ist richtig?

- Der Systemaufruf liefert einen Zeiger zurueck, ueber den die aufrufende Funktion direkt auf eine Datenstruktur zugreifen kann, die die Dateiattribute enthaelt.
- Solch eine Schnittstelle ist nicht schoen, da dadurch die aufrufende Funktion auf internen Speicher des Betriebssystems zugreifen koennte.
- Der Aufrufer muss sicherstellen, dass er den zurueckgelieferten Speicher mit free(3) wieder freigibt, wenn er die Dateiattribute nicht mehr weiter benoetigt.
- **Ein Zugriff ueber den zurueckgelieferten Zeiger liefert voellig zufaellige Ergebnisse oder einen Segmentation fault.**

j) Was versteht man unter einem Interrupt?

- Eine Signalleitung teilt dem Prozessor mit, dass er den aktuellen Prozess anhalten und auf das Ende der Unterbrechung warten soll.
- **Mit einer Signalleitung wird dem Prozessor eine Unterbrechung angezeigt. Der Prozessor sichert den aktuellen Zustand bestimmter Register, insbesondere des Programmzaehlers, und springt eine vordefinierte Behandlungsfunktion an.**
- Der Prozessor wird veranlasst eine Unterbrechungsbehandlung durchzufuehren. Der gerade laufende Prozess kann die Unterbrechungsbehandlung ignorieren.
- Durch eine Signalleitung wird der Prozessor veranlasst, die gerade bearbeitete Maschineninstruktion abzubrechen.

k) Welche der folgenden Aussagen bezueglich Threads ist falsch?

- **Die Einlastung (das Dispatching) eines Threads ist eine privilegierte Operation und kann deshalb grundsaetzlich immer nur durch das Betriebssystem vorgenommen werden.**
- Das Betriebssystem ist nicht in der Lage, einen einzelnen User-Level-Thread bei einer fehlerhaften Operation (z. B. Segmentation fault) gezielt abzubrechen.
- Ein Anwendungsprogrammierer kann die Schedulingstrategie fuer seine UserLevel-Threads selbst programmieren.
- Wenn ein User-Level-Thread im Rahmen einer read-Operation warten muss (blockiert wird), kann das Betriebssystem nicht auf einen anderen User-LevelThread des gleichen Prozesses umzuschalten.

l) Ein Prozess wird in den Zustand bereit ueberfuehrt. Welche Aussage passt nicht zu diesem Vorgang?

- Der Prozess wartet auf eine Tastatureingabe.
- Der Prozess wurde von einem Prozess mit einer hoeherer Prioritaet verdraengt.
- Der Prozess hat auf Daten von der Festplatte gewartet und die Daten stehen nun zur Weiterbearbeitung bereit.
- **Der Prozess ist grundsaetzlich lauffaehig und wird im Rahmen der mittelfristigen Prozesseinplanung eingelagert.**

m) Es gibt verschiedene Ursachen, wie Nebenlaeufigkeit in einem System entstehen kann (gewollt oder auch ungewollt). Was gehoert nicht dazu?

- durch Interrupts
- durch preemptives Scheduling
- **durch Traps**
- durch Threads auf einem Multiprozessorsystem

n) Welches der folgenden Verfahren ist zur Synchronisation des Zugriffs auf gemeinsame Daten in einem Multiprozessorsystem nicht geeignet?

- Aktives Warten bis die Sperre aufgehoben wird
- Binaere Semaphore
- Spezialbefehle wie cas
- **Sperre der Interrupts**

Tim Nisslbeck (hu78sapy)
hu78sapy@stud.cs.fau.de — wwwcip.cs.fau.de/~hu78sapy

o) Was versteht man unter Verklemmungsvorbeugung?

- Das System ueberprueft vor dem Belegen von Betriebsmitteln, ob ein unsicherer Zustand eintreten wuerde.
- Bei einem Zyklus im Betriebsmittelbelegungsgraph wird einer der Prozesse aus dem Zyklus terminiert.
- **In einem System wird dafuer gesorgt, dass eine der vier Voraussetzungen fuer Verklemmungen nicht eintreten kann.**
- Wurde ein unsicherer Zustand im System erkannt, wird vorbeugend einer der beteiligten Prozesse abgebrochen, bevor die Verklemmung eintritt.

2 Juli 2010

2.1 Aufgabe 1 (30 Punkte)

a	b	c	d	e	f	g	h	i	j	k	l	m	n
3	1	1	4	3	2	3	4	3	2	3	3	3	4

a	b	c	d
14..29%	14.29%	**50.00%**	21.43%

a) Welche Aussage zum Thema Betriebsarten ist richtig?

- Beim Stapelbetrieb koennen keine globalen Variablen existieren, weil alle Daten im Stapel-Segment (Stack) abgelegt sind.
- Echtzeitsysteme findet man hauptsaechlich auf grossen Serversystemen, die eine enorme Menge an Anfragen zu bearbeiten haben.
- **Mehrzugangsbetrieb ist nur in Verbindung mit CPU- und Speicherschutz sinnvoll realisierbar.**
- Mehrprogrammbetrieb ermoeglicht die simultane Ausfuehrung mehrerer Programme innerhalb desselben Prozesses.

b) Welche Aussage zum Thema Adressraumschutz ist richtig?

- **Bei allen Verfahren des Adressraumschutzes fuehrt jeder Zugriff auf eine ungueltige Speicheradresse zu einem Trap.**
- Beim Adressraumschutz durch Abteilung wird der logische Adressraum in mehrere Segmente mit unterschiedlicher Semantik unterteilt.
- Beim Adressraumschutz durch Eingrenzung ist es prinzipbedingt nicht moeglich, dass mehrere Prozesse auf ein Stueck gemeinsamen Speichers zugreifen.
- In einem segmentierten Adressraum kann zur Laufzeit kein weiterer Speicher mehr dynamisch nachgefordert werden.

c) Welche Aussage zu Zeigern ist richtig?

- **Zeiger koennen verwendet werden, um in C eine call-by-reference Uebergabesemantik nachzubilden.**
- Die Uebergabesemantik fuer Zeiger als Funktionsparameter ist call-by-reference.
- Ein Zeiger kann zur Manipulation von schreibgeschuetzten Datenbereichen verwendet werden.
- Zeiger vom Typ void* existieren in C nicht, da solche SZeiger auf Nichts"keinen sinnvollen Einsatzzweck haetten.

d) Was ist ein Stack-Frame?

- Der Speicherbereich, in dem der Programmcode einer Funktion abgelegt ist.
- Ein spezieller Registersatz des Prozessors zur Bearbeitung von Funktionen.
- Ein Fehler, der bei unberechtigten Zugriffen auf den Stack-Speicher entsteht.
- **Ein Bereich des Speichers, in dem u.a. lokale automatic-Variablen einer Funktion abgelegt sind.**

e) Welche Aussage zum Thema Prozesszustaende ist falsch?

- Nach seiner Beendigung geht ein Prozess in den Zustand beendet ueber. In diesem Zustand wird ein Prozess auch als Zombie-Prozess bezeichnet.
- Es koennen sich maximal soviele Prozesse im Zustand laufend befinden, wie physikalische Prozessorkerne im System existieren.
- **Nach Wegfall der Blockadebedingung geht ein Prozess direkt in den Zustand laufend ueber.**
- Ein Prozess kann nur durch eigene Aktivitaet in den Zustand blockiert gelangen.

f) Welche Aussage zum Thema Prozesse/Threads ist richtig?

- Beim federgewichtigen Prozess bilden Adressraum und Prozess eine Einheit.
- **Leichtgewichtige Prozesse muessen vom Betriebssystem speziell unterstuetzt werden.**
- Eine gleichzeitige Ausfuehrung mehrere schwergewichtiger Prozesse auf verschiedenen Prozessoren ist nicht moeglich.
- Die Einlastung eines federgewichtigen Prozesses ist eine privilegierte Operation und erfordert Unterstuetzung der Betriebssystem.

g) Welche Information wird nicht im Dateikopf (Inode) eines typischen UNIX-Dateisystems gespeichert?

- Datum und Zeit der letzten Aenderung
- Datum und Zeit des letzten Zugriffs
- **Nummer des Inodes**
- Zugriffsrechte fuer den Dateibesitzer

h) Welche Aussage zum Thema Hard-Links ist falsch?

- Auf jede Datei existiert mindestens ein Hard-Link
- Auf jedes Verzeichnis existieren mindestens zwei Hard-Links
- Jeder Hard-Link besitzt in seinem Kontext einen eindeutigen Namen
- **Hard-Links duerfen nur vom Systemadministrator angelegt werden**

i) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

- Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzaehler, Register, Stack).
- Wenn ein Programm nur einen aktiven Ablauf enthaelt, nennt man diesen Prozess, enthaelt das Programm mehrere Ablaeufe, nennt man diese Threads.
- **Ein Prozess ist ein Programm in Ausfuehrung - ein Prozess kann aber auch mehrere verschiedene Programme ausfuehren**
- Ein Programm kann immer nur von einem Prozess ausgefuehrt werden

j) Sie kennen den Translation-Look-Aside-Buffer (TLB). Welche Aussage ist richtig?

- Der TLB puffert die Ergebnisse der Abbildung von physikalische auf logische Adressen, sodass eine erneute Anfrage sofort beantwortet werden kann.
- **Veraendert sich die Speicherabbildung von logischen auf physikalische Addressen aufgrund einer Adressraumumschaltung, so werden auch die Daten im TLB ungueltig.**
- Der TLB verkuerzt die Zugriffszeit auf den physikalischen Speicher, da ein Teil des moeglichen Speichers in einem schnellen Pufferspeicher vorgehalten wird.
- Der TLB puffert Daten bei der Ein-/Ausgabebehandlung und beschleunigt diese damit.

k) Welche Seitennummer und welcher Versatz gehoeren bei einer Seitengroesse von 2048 Bytes zu folgender logischer Adresse: 0xbabe?

- Seitennummer 0x11, Versatz 0xabe
- Seitennummer 0xb, Versatz 0xabe
- **Seitennummer 0x17, Versatz 0x2be**
- Seitennummer 0xba, Versatz 0xbe

l) Bei Demand-Paging kann der Effekt des Seitenflatterns (Thrashing) auftreten. Welche Aussage ist richtig?

- Bei der Ersetzungsstrategie Second Chance (SC) wird bei einem Zugriff auf eine Seite ein Referenzbit gesetzt. Wird die Seite laengere Zeit nicht angesprochen, so wird dieses Bit geloescht. Da dieses Bit staendig den Wert aendert, spricht man von Seitenflattern.
- Seitenflattern tritt auf, wenn Seiten zur Defragementierung im Speicher verschoben werden.
- **Wird eine eben ausgelagerte Seite gleich wieder angesprochen, so muss diese wieder eingelagert werden. Tritt dieser Effekt haeufig auf, so spricht man von Seitenflattern.**
- Seitenflattern kann nur auftreten, wenn der dynamisch genutzte Speicher eines Prozesses groesser ist, als der physikalisch vorhandene Speicher des Systems.

m) Welche Aussage zu Speicherzuteilungsverfahren ist richtig?

- Die worst-fit-Strategie ist lediglich theoretisch interessant, da es in der Praxis nie sinnvoll ist, den am schlechtesten passenden Speicherplatz zuzuweisen.
- Best-fit ist in jedem Fall das beste Verfahren.
- **Bei zunehmenden Blockgroessen nimmt beim Buddy-Verfahren im Mittel auch der interne Verschnitt zu.**
- Buddy-Verfahren sind nur bei sehr leistungsfaehigen Rechnern und grossem Speicher einsetzbar, weil sie aufwaendig in der Berechnung sind.

n) Gegeben sei folgendes Szenario: zwei Faeden werden auf einem Monoprozessorsystem mit der Strategie First Come First Served"verwaltet. In jedem Faden wird die Anweisung ï++;äuf die gemeinsame, globale Variable i ausgefuehrt. Welche der folgenden Aussagen ist richtig?

- Waehrend der Inkrementoperation muessen Interrupts voruebergehend unterbunden werden.
- Die Inkrementoperation muss mit einer CAS-Anweisung nicht-blockierend synchronisiert werden.
- Die Operation i++ ist auf einem Monoprozessorsystem immer atomar.
- **In einem Monoprozessorsystem ohne Verdraengung ist keinerlei Synchronisation erforderlich.**

3 Februar 2011

3.1 Aufgabe 1.1 (22 Punkte)

a	b	c	d	e	f	g	h	i
3	4	1	1	1	3	1	2	4

a	b	c	d
44.$\overline{4}$%	11.$\overline{1}$%	22.$\overline{2}$%	22.$\overline{2}$%

a) Mit logischen Adressraeumen kann man mehrere Zwecke erreichen. Was gehoert nicht dazu?

- Schutzmechanismus: man kann die Auswirkungen von Berechnungsfehlern oder technischen Fehlern begrenzen.
- Sicherheit: man kann unbefugten Zugriff auf Daten verhindern.
- **Virtualisierung: man kann in einem Programmlauf mehr Speicher nutzen, als physikalisch vorhanden ist.**
- Bessere Verwaltung: Man kann Speicherbereiche mit unterschiedlicher Bedeutung voneinander abgrenzen.

b) Welche der Aussagen bzgl. eines logischen Adressraums, der auf dem Prinzip der Segmentierung aufgebaut wurde, ist richtig?

- Alle Segmente eines Prozesses haben die gleiche Laenge.
- Bei der Aus- und Einlagerung von Segmenten zwischen Hauptspeicher und Festplatte muss das Segment immer an die gleiche Hauptspeicherposition eingelagert werden, da sonst die Adressen nicht mehr gueltig sind.
- Die Segmentierung erlaubt bei der Abbildung eines logischen Adressraums keinen Zugriff auf Speicherzellen, die auch Bestandteil von anderen logischen Adressraeumen sind (Zugriffschutz).
- **Die Segmentierung schraenkt den logischen Adressraum derart ein, dass nur auf gueltige Speicheradressen erfolgreich zugegriffen werden kann.**

c) Welche Seitennummer und welcher Versatz gehoeren bei einer Seitengroesse von 1024 Bytes zu folgender logischer Adresse: 0xcafe?

- **Seitennummer 0x32, Versatz 0x2fe**
- Seitennummer 0xc, Versatz 0xafe
- Seitennummer 0x19, Versatz 0x2fe
- Seitennummer 0xca, Versatz 0xfe

d) Welche Antwort trifft fuer die Eigenschaften eines UNIX/Linux Filedeskriptors zu?

- **Ein Filedeskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei, ein Geraet, einen Socket oder eine Pipe benutzen kann.**
- Filedeskriptoren sind Zeiger auf Betriebssystemstrukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.
- Ein Filedeskriptor ist eine Integerzahl, die ueber gemeinsamen Speicher an einen anderen Prozess uebergeben werden kann und von letzterem zum Zugriff auf eine geoeffnete Datei verwendet werden kann.
- Beim Oeffnen ein und derselben Datei erhaelt ein Prozess jeweils die gleiche Integerzahl als Filedeskriptor zum Zugriff zurueck.

e) Gegeben sei folgendes Szenario: zwei Faeden werden auf einem Monoprozessorsystem mit der Strategie First Come First Served"verwaltet. In jedem Faden wird die Anweisung ï++;äuf die gemeinsame, globale Variable i ausgefuehrt. Welche der folgenden Aussagen ist richtig?

- **In einem Monoprozessorsystem ohne Verdraengung ist keinerlei Synchronisation erforderlich.**
- Waehrend der Inkrementoperation muessen Interrupts voruebergehend unterbunden werden.
- Die Inkrementoperation muss mit einer CAS-Anweisung nicht-blockierend synchronisiert werden.
- Die Operation i++ ist auf einem Monoprozessorsystem immer atomar.

f) In welcher der folgenden Situationen wird ein laufender Prozess in den Zustand blockiert ueberfuehrt?

- Ein Kindprozess des Prozesses terminiert.
- Der Prozess hat einen Seitenfehler fuer eine Seite, die bereits in den Freiseitenpuffer eingetragen, aber noch im Hauptspeicher vorhanden ist.
- **Der Prozess greift lesend auf eine Datei zu und der entsprechende Datenblock ist noch nicht im Hauptspeicher vorhanden.**
- Der Prozess ruft eine V-Operation auf einen Semaphor auf und der Semaphor hat gerade den Wert 0.

g) Fuer lokale Variablen, Aufrufparameter, etc. einer Funktion wird bei vielen Prozessoren ein sog. Aktivierungsblock (activation record oder stack frame) auf dem Stack angelegt. Welche Aussage ist richtig?

- **Ueber Zeiger kann man alle Daten des Aktivierungsblocks der aufrufenden Funktion veraendern.**
- Nach dem Ruecksprung aus einer Funktion sind Zeiger auf die Speicherzellen ihres Aktivierungsblocks nicht mehr gueltig. Ein Zugriff ueber solch einen Zeiger fuehrt dann zu einem Segmentation fault.
- Bei rekursiven Funktionsaufrufen kann der Aktivierungsblock in jedem Fall wiederverwendet werden weil die gleiche Funktion aufgerufen wird.
- Der Compiler legt zur Uebersetzungszeit fest, an welcher Position im Aktivierungsblock der main-Funktion die globalen Variablen angelegt werden.

h) Was versteht man unter der Second-Chance- (oder Clock-) Policy?

- Eine Seitenersetzungsstrategie, bei der jeweils die aelteste Seite ausgelagert wird.
- **Eine Seitenersetzungsstrategie, die mit Hilfe eines Referenz-Bits eine einfacher zu implementierende Annaeherung an LRU realisiert.**
- Eine Scheduling-Strategie, bei der Prozesse vor der Verdraengung eine zweite Chance erhalten.
- Eine Speicherallokationsstrategie, bei der im Fehlerfall ein zweiter Allokationsversuch stattfindet.

i) Was versteht man unter RAID 0?

- Ein auf Flash-Speicher basierendes, extrem schnelles Speicherverfahren.
- Datenbloecke werden ueber mehrere Platten verteilt und repliziert gespeichert.
- Auf Platte 0 wird Parity-Information der Datenbloecke der Platten 1 - 4 gespeichert.
- **Datenbloecke eines Dateisystems werden ueber mehrere Platten verteilt gespeichert.**

3.2 Aufgabe 1.2 (8 Punkte)

a	b
1257	124

a	b	c	d	e	f	g	h
100%	**100%**	0%	50%	50%	0%	50%	0%

Tim Nisslbeck (hu78sapy)
hu78sapy@stud.cs.fau.de — wwwcip.cs.fau.de/~hu78sapy

a) Welche der folgenden Aussagen zum Thema Threads sind richtig?

- **Bei User-Threads ist die Schedulingstrategie keine Funktion des Betriebssystemkerns.**
- **User-Threads blockieren sich bei blockierenden Systemaufrufen gegenseitig.**
- Kernel-Threads koennen Multiprozessoren nicht ausnutzen.
- Zur Umschaltung von Kernel-Threads ist kein Adressraumwechsel erforderlich.
- **Die Umschaltung von User-Threads ist sehr effizient.**
- Zu jedem Kern-Thread gehoert ein eigener Adressraum.
- **Bei Kernel-Threads ist die Schedulingstrategie meist durch das Betriebssystem vorgegeben.**
- Die Umschaltung von Threads muss immer im Systemkern erfolgen (privilegierter Maschinenbefehl).

b) Welche der folgenden Aussagen zum Thema Synchronisation sind richtig?

- **Semaphore sind fuer einseitige Synchronisation geeignet.**
- **Semaphore sind fuer mehrseitige Synchronisation geeignet.**
- Auch nicht-blockierende Synchronisation kann zu Verklemmungen fuehren.
- **Monitore sind Datentypen mit impliziten Synchronisationseigenschaften.**
- Einseitige Synchronisation erfordert immer Betriebssystem-Unterstuetzung.
- Schlossvariablen sind teuer, da lock() immer zu einem Prozesswechsel fuehrt.
- Die Synchronisation mit einem Signal-Handler unter LINUX erfolgt vorzugsweise ueber mehrseitige Synchronisationsverfahren.
- Sperren von Unterbrechungen ist das beste Verfahren zur Synchronisation mit Unterbrechungsbehandlungsfunktionen.

4 August 2011

4.1 Aufgabe 1.1 (22 Punkte)

a	b	c	d	e	f	g	h	i	j	k
1	4	4	3	2	1	3	3	1	4	2

a	b	c	d
27.$\overline{27}$%	18.$\overline{18}$%	**27.$\overline{27}$%**	**27.$\overline{27}$%**

a) Was versteht man unter Virtuellem Speicher?

- **Speicher der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber moeglicherweise groesser als der verfuegbare physikalische Hauptspeicher ist.**
- Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht fuer einen Anwendungsprozess.
- Unter einem Virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- Virtueller Speicher kann dynamisch zur Laufzeit von einem Programm erzeugt werden (Funktion valloc(3)).

b) Welche Aussage ueber das aktuelle Arbeitsverzeichnis (Current Working Directory) trifft zu?

- Jedem UNIX-Benutzer ist zu jeder Zeit ein aktuelles Verzeichnis zugeordnet.
- Besitzt ein UNIX-Prozess kein Current Working Directory, so beendet sich der Prozess mit einem Segmentation Fault.
- Mit dem Systemaufruf chdir() kann das aktuelle Arbeitsverzeichnis durch den Vaterprozess veraendert werden.
- **Pfadnamen, die nicht mit dem Zeichen '/' beginnen, werden relativ zu dem aktuellen Arbeitsverzeichnis interpretiert**

c) In einem UNIX-UFS-Dateisystem gibt es symbolische Namen/Verweise (Symbolic Links) und feste Links (Hard Links) auf Dateien. Welche Aussage ist richtig?

- Ein Symbolic Link kann nicht auf Dateien anderer Dateisysteme verweisen.
- Ein Hard Link kann nur auf Verzeichnisse nicht jedoch auf Dateien verweisen.
- Wird der letzte Symbolic Link auf eine Datei geloescht, so wird auch die Datei selbst geloescht.
- **Fuer jede regulaere Datei existiert mindestens ein Hard-Link im selben Dateisystem.**

d) Bei der Behandlung von Ausnahmen (Traps oder Interrupts) unterscheidet man zwei Bearbeitungsmodelle. Welche Aussage hierzu ist richtig?

- Das Wiederaufnahmemodell dient zur Behandlung von Interrupts (Fortfuehrung des Programms nach einer zufaellig eingetretenen Unterbrechung). Bei einem Trap ist das Modell nicht sinnvoll anwendbar, da ein Trap deterministisch auftritt und damit eine Wiederaufnahme des Programms sofort wieder den Trap verursachen wuerde.
- Nach dem Beendigungmodell werden Interrupts bearbeitet. Gibt man z. B. CTRL-C unter UNIX ueber die Tastatur ein, wird ein Interrupt-Signal an den gerade laufenden Prozess gesendet und dieser dadurch beendet.

- **Interrupts duerfen auf keinen Fall nach dem Beendigungsmodell behandelt werden, weil ueberhaupt kein Zusammenhang zwischen dem unterbrochenen Prozess und dem Grund des Interrupts besteht.**
- Das Betriebssystem kann Interrupts, die in ursaechlichem Zusammenhang mit dem gerade laufenden Prozess stehen, nach dem Beendigungsmodell behandeln, wenn eine sinnvolle Fortfuehrung des Prozesses nicht mehr moeglich ist.

e) Welche Aussage ist in einem Monoprozessor-Betriebssystem richtig?

- Ein Prozess im Zustand blockiert muss warten, bis der laufende Prozess den Prozessor abgibt und kann dann in den Zustand laufend ueberfuehrt werden.
- **Es befindet sich zu einem Zeitpunkt maximal ein Prozess im Zustand laufend.**
- Ist zu einem Zeitpunkt kein Prozess im Zustand bereit, so ist auch kein Prozess im Zustand laufend.
- In den Zustand blockiert gelangen Prozesse nur aus dem Zustand bereit.

f) Welche Antwort trifft fuer die Eigenschaften eines UNIX/Linux Filedeskriptors zu?

- **Ein Filedeskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei benutzen kann.**
- Filedeskriptoren sind Zeiger auf Betriebssystem-interne Strukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.
- Ein Filedeskriptor ist eine Integerzahl, die ueber gemeinsamen Speicher an einen anderen Prozess uebergeben werden kann, und von letzterem zum Zugriff auf eine geoeffnete Datei verwendet werden kann.
- Beim Oeffnen ein und derselben Datei erhaelt ein Prozess jeweils die gleiche Integerzahl als Filedeskriptor zum Zugriff zurueck.

g) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

- Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzaehler, Register, Stack).
- Wenn ein Programm nur einen aktiven Ablauf enthaelt, nennt man diesen Prozess, enthaelt das Programm mehrere Ablaeufe, nennt man diese Threads.
- **Ein Prozess ist ein Programm in Ausfuehrung - ein Prozess kann aber auch mehrere verschiedene Programme ausfuehren**
- Ein Programm kann immer nur von einem Prozess ausgefuehrt werden

h) Fuer welchen Zweck wird der Systemaufruf listen() benutzt?

- Der Aufruf von listen() wartet solange an einem Socket, bis eine einkommende Verbindungsanfrage vorliegt.
- Damit das Betriebssystem ueberhaupt Systemaufrufe annimmt, muss es erst mit listen() in einen Modus des ,,Zuhoerens" gebracht werden.
- **Mit listen() wird ein Socket fuer die Verbindungsannnahme vorbereitet. Ein Parameter gibt an, wieviele Verbindungsanfragen vor deren Annahme gepuffert werden koennen.**
- Mit listen() wird ein Socket fuer die Verbindungsannahme vorbereitet. Ein Parameter gibt an, wieviele laufende Verbindungen maximal moeglich sind.

i) Ein System setzt Segmentierung und Seitenadressierung ein. Zwei Prozesse sollen ein Segment gemeinsam benutzen. Wie geht das Betriebssystem vor, um das gemeinsame Segment einzurichten?

- **Das Betriebssystem legt eine Seitentabelle fuer das gemeinsame Segment an und traegt diese bei beiden Prozessen in die jeweiligen Segmenttabellen ein.**
- Das Betriebssystem traegt jeweils eine Seitentabelle in die prozesseigene Segmenttabelle ein und sorgt dafuer, dass die Eintraege jeweils auf die gleichen Seitenrahmen im Hauptspeicher verweisen.
- Gemeinsame Segmente koennen nur durch den Prozess jedoch nicht vom Betriebssystem eingerichtet werden.
- Die beiden Prozesse bekommen die gleiche Segmenttabelle zugewiesen.

j) Welche Aussage zu einem RAID-1-Plattensystem ist richtig?

- Es muessen mindestens drei Festplatten an einem RAID-1-Verbund beteiligt sein.
- Der Schreibzugriff auf ein RAID-1-Plattensystem ist schneller, da mehrere Platten gleichzeitig beauftragt werden koennen.
- Die Paritaetsinformation wird gleichmaessig ueber alle beteiligten Platten verteilt.
- **Ein aus drei Festplatten bestehendes RAID-1-Plattensystem kann den Ausfall zweier Festplatten ohne Datenverlust verkraften**

k) Wozu benoetigt man Bedingungsvariablen (condition variables)?

- Bei if-Abfragen in C-Programmen.
- **Um in einem kritischen Abschnitt auf ein Ereignis zu warten und den kritischen Abschnitt waehrend der Wartezeit freizugeben.**
- Zur Vermeidung von aktivem Warten in kritischen Abschnitten.
- Zur Erkennung von Verklemmungen.

4.2 Aufgabe 1.2 (8 Punkte)

a	b
1578	3468

a	b	c	d	e	f	g	h
50%	0%	50%	50%	50%	50%	50%	**100%**

a) Welche der folgenden Aussagen zum Thema Threads sind richtig?

- **Bei User-Threads ist die Schedulingstrategie keine Funktion des Betriebssystemkerns.**
- Kern-Threads blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- Kern-Threads koennen Multiprozessoren nicht ausnutzen.
- Zur Umschaltung von User-Threads ist ein Adressraumwechsel erforderlich.
- **Die Umschaltung von User-Threads ist sehr effizient.**
- Zu jedem Kern-Thread gehoert ein eigener Adressraum.
- **Bei Kern-Threads ist die Schedulingstrategie meist durch das Betriebssystem vorgegeben.**
- **Die Umschaltung von Kern-Threads muss immer im Systemkern erfolgen.**

Tim Nisslbeck (hu78sapy)
hu78sapy@stud.cs.fau.de — wwwcip.cs.fau.de/~hu78sapy

b) Welche der folgenden Aussagen zum Thema Speicherverwaltung sind richtig?

- Das Buddy-Verfahren verhindert externen Verschnitt.
- Das Buddy-Verfahren verhindert internen Verschnitt.
- **Die Adressen zweier Buddies unterscheiden sich in genau einem Bit.**
- **Die Verschmelzung benachbarter Loecher ist beim Buddy-Verfahren besonders einfach.**
- Benachbarte Loecher gleicher Groesse koennen beim Buddy-Verfahren in jedem Fall verschmolzen werden.
- **Bei einer Speicheranforderung muss bei Worst-Fit u.U. die gesamte Freispeicherliste durchlaufen werden.**
- Der Verschmelzungsaufwand bei Best-Fit ist verglichen mit Worst-Fit erhoeht.
- **Ziel von First-Fit ist es, den Verwaltungsaufwand gering zu halten.**

4.3 Aufgabe 2 - pete (60 Punkte)

Schreiben Sie ein Programm **pete** (ParalEl Test Executor), das alle ausfuehrbaren Dateien in einem Verzeichnis (=Testfaelle) ausfuehrt und die Anzahl der erfolgreich ausgefuehrten Testfaelle ausgibt.

Das Programm bekommt auf der Befehlszeile vom Benutzer die maximale Anzahl an gleichzeitig laufenden Prozessen (**maxProcs**) und eines oder mehrere Verzeichnisse uebergeben. In jedem Verzeichnis befinden sich ausfuehrbare Dateien (Testfaelle), die durch `pete` parallel in Prozessen ausgefuehrt werden sollen. Mit Ruecksicht auf andere Benutzer des Systems wird die Anzahl der gleichzeitig laufenden Testfaelle auf `maxProcs` limitiert. Nach Abschluss aller Testfaelle eines Verzeichnisses gibt `pete` eine kurze Statistikmeldung aus. Um eine spaetere Analyse jedes Testfalles zu ermoeglichen, wird der Standardfehlerkanal jedes Testfalles in jeweils eine eigene Datei umgeleitet. Zum Schluss gibt `pete` eine Gesamtstatistik ueber alle Testfaelle aus.

Das Programm soll folgendermassen arbeiten:

- Die **main()**-Funktion ruft die Funktion **processDirContent** fuer jedes uebergebene Verzeichnis auf und wartet anschliessend passiv, bis sich alle gestarteten Prozesse beendet haben.

  Wenn `processDirContent` Testfaelle gestartet hat, wird eine Statistik mit der Anzahl der erfolgreich ausgefuehrten Testfaelle und der Gesamtzahl aller in diesem Verzeichnis gestarteten Testfaelle ausgegeben. Ein Testfall gilt als erfolgreich ausgefuehrt, wenn der zugehoerige Prozess sich mit dem Exitstatus 0 beendet hat. Die Ausgabe soll z. B. wie folgt aussehen:

  ‘9 of 11 testcases successful in directory foo‘

  Sollte die Funktion `processDirContent` einen Fehler melden, wird keine Statistik ausgegeben und das Verzeichnis wird in der Gesamtstatistik ignoriert.

  Danach wird mit dem naechsten Verzeichnis weitergemacht. Sobald alle Verzeichnisse abgearbeitet sind, wird die Gesamtzahl aller erfolgreich ausgefuehrten Testfaelle und die Gesamtzahl aller insgesamt gestarteten Testfaelle in folgendem Format ausgegeben:

  ‘27 of 30 testcases successful in all directories‘

- **Funktion int processDirContent(const char *dir)**: Durchsucht das uebergebene Verzeichnis nicht-rekursiv nach ausfuehrbaren, regulaeren Dateien (Testfaellen). Fuer jeden Testfall wird die Funktion **executeFile** aufgerufen. Gibt **executeFile** 0 zurueck, wird der Testfall als gestartet gezaehlt. Die Funktion processDirContent gibt die Anzahl der gestarteten Testfaelle zurueck. Sollte ein Fehler auftreten wird eine passende Fehlermeldung ausgegeben und -1 zurueckgegeben.
- Funktion **int executeFile(const char *fileName)**: Sollten aktuell mehr Prozesse gestartet worden sein als das Befehlszeilen-Argument **maxProcs** vorgibt, wird passiv gewartet, bis wieder neue Prozesse erzeugt werden koennen. Dann wird das uebergebene Programm (Testfall) in einem neuen Prozess ausgefuehrt. Hierbei wird der Standardfehlerkanal in die Datei filename**.err** (fileName durch den Namen des Testfalls ersetzen!) umgeleitet. Sollte in der Funktion `executeFile` ein Fehler auftreten, wird eine passende Fehlermeldung ausgegeben und -1 zurueckgegeben. Im Erfolgsfall wird 0 zurueckgegeben.

4.3.1 pete.c

```c
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

#define S_ISEXEC(mode) (((S_IXUSR|S_IXGRP|S_IXOTH)&mode) > 0 )

/* Funktions- und Strukturdeklarationen, globale Variablen*/
static volatile int curProcs, succProcs;
static int maxProcs;
static sigset_t old, block;

static void sighandler(int signal);
static int processDirContent(const char* dir);
static int executeFile(const char* fileName);

/* Funktion main */
int main(int argc, char* argv[]) {
  /* Argumente verarbeiten */
  if(3>argc) {
    fprintf(stderr,"Usage: %s <maxProcs> <dirs...>\n",argv[0]);
    return 1;
  }
  maxProcs=atoi(argv[1]);
  if(1>maxProcs) {
    fprintf(stderr,"invalid maxProcs\n");
    return 1;
  }

  /* Signalbehandlung und Signalmasken initialisieren */
  struct sigaction action = {
    .sa_handler=sighandler,
    .sa_flags=SA_RESTART | SA_NOCLDSTOP
  };
  sigemptyset(&action.sa_mask);
  sigaction(SIGCHLD,&action,NULL);
  sigemptyset(&block);
  sigaddset(&block,SIGCHLD);

  /* uebergebene Verzeichnisse abarbeiten */
  int allTests,allSuccess,started;
  allTests=allSuccess=0;
  for(int i=2;i<argc;i++) {
    /* Terminieren der Testfaelle abwarten */
    started=processDirContent(argv[i]);
    sigprocmask(SIG_BLOCK,&block,&old);
    while(0!=curProcs) sigsuspend(&old);
    sigprocmask(SIG_SETMASK,&old,NULL);
    if(-1==started) continue;

    /* Statistik fuer Directory ausgeben */
    printf("%d of %d testcases successful in %s\n",succProcs,started,argv[i]);
    allTests+=started;
```

```
59      allSuccess+=succProcs;
60    }
61
62    /* Gesamtstatistik ausgeben */
63    printf("%d of %d testcases successful in all dirs\n",allSuccess,allTests);
64    return 0;
65  }
66
67  /* Signalbehandlung fuer SIGCHLD */
68  static void sighandler(int signal) {
69    int old_errno,status;
70    old_errno=errno;
71
72    /* Prozesszahl und erfolgreich ausgef. Testfaelle erfassen */
73    while(0 < waitpid(-1,&status,WNOHANG)) {
74      curProcs--;
75      if(WIFEXITED(status) && 0==WEXITSTATUS(status)) succProcs++;
76    }
77    errno=old_errno;
78  }
79
80  /* Funktion processDirContent */
81  static int processDirContent(const char* dir) {
82    succProcs=0;
83
84    /* Verzeichnisse durchlaufen */
85    DIR* d=opendir(dir);
86    if(!d) { perror("opendir"); return -1; }
87    struct dirent* e;
88    int count=0;
89    while(errno=0,(e=readdir(d))) {
90      char path[strlen(dir)+2+strlen(e->d_name)];
91      sprintf(path,"%s/%s",dir,e->d_name);
92      struct stat stat;
93      if(lstat(path,&stat)) { perror("lstat"); continue; }
94
95      /* falls Datei=Testfall, Datei ausfuehren*/
96      if(S_ISREG(stat.st_mode) && S_ISEXEC(stat.st_mode)) {
97        if(!executeFile(path)) count++;
98        else { closedir(d); return -1; }
99      }
100   }
101   if(errno) { perror("readdir"); closedir(d); return -1; }
102
103   /* aufraeumen */
104   closedir(d); return count;
105 } /* Ende Funktion processDirContent */
106
107 /* Funktion executeFile */
108 static int executeFile(const char* fileName) {
109   sigprocmask(SIG_BLOCK,&block,&old);
110   while(curProcs>=maxProcs) sigsuspend(&old);
111
112   /* Prozess erzeugen und Testfall starten */
113   pid_t pid=fork();
114   if(-1==pid) {
115     sigprocmask(SIG_SETMASK,&old,NULL);
116     perror("fork"); return -1;
117   } else if(!pid) {
118     sigprocmask(SIG_SETMASK,&old,NULL);
119     char b[strlen(fileName)+strlen(".err")+1];
```

```
120     sprintf(b,"%s.err",fileName);
121     FILE* f=fopen(b,"w");
122     if(!f) { perror("fopen"); exit(1); }
123     if(-1==dup2(fileno(f),STDERR_FILENO)) { perror("dup2"); exit(1); }
124     execl(fileName,fileName,NULL);
125     perror("execl"); exit(1);
126   }
127
128   /* Fortsetzung */
129   curProcs++;
130   sigprocmask(SIG_SETMASK,&old,NULL);
131   return 0;
132 }
```

5 Februar 2012

5.1 Aufgabe 1.1 (22 Punkte)

a	b	c	d	e	f	g	h	i	j
3	3	4	1	2	2	4	1	4	3

a	b	c	d
20.00%	20.00%	**30.00%**	**30.00%**

a) Fuer welchen Zweck wird der Systemaufruf listen() benutzt?

- Der Aufruf von listen() wartet solange an einem Socket, bis eine einkommende Verbindungsanfrage vorliegt.
- Der Aufruf von listen() erzeugt eine leere verkettete Liste, die zum Speichern von Daten verwendet werden kann.
- **Mit listen() wird ein Socket fuer die Verbindungsannnahme vorbereitet. Ein Parameter gibt an, wieviele Verbindungsanfragen vor deren Annahme gepuffert werden koennen.**
- Mit listen() wird ein Socket fuer die Verbindungsannahme vorbereitet. Ein Parameter gibt an, wieviele laufende Verbindungen maximal moeglich sind.

b) Welche der genannten Attribute sind in einem Inode eines UNIX-Dateisystems gespeichert?

- Dateityp, Eigentuemer und Dateiname
- Gruppenzugehoerigkeit, Anzahl der Verweise und bei Verzeichnissen zusaetzlich die Anzahl der enthaltenen Unterverzeichnisse
- **Eigentuemer, Dateigroesse und Dateityp**
- Zeitpunkt des letzten Dateizugriffes, Erstellungszeitpunkt und aus Sicherheitsgruenden der Zeitpunkt der letzten Rechteaenderung.

c) In welcher der folgenden Situationen wird ein laufender Prozess in den Zustand blockiert ueberfuehrt?

- Ein Kindprozess des Prozesses terminiert.
- Der Prozess hat einen Seitenfehler fuer eine Seite, die bereits in den Freiseitenpuffer eingetragen, aber noch im Hauptspeicher vorhanden ist.
- Der Prozess ruft eine V-Operation auf einen Semaphor auf und der Semaphor hat gerade den Wert 0.
- **Der Prozess greift lesend auf eine Datei zu und der entsprechende Datenblock ist noch nicht im Hauptspeicher vorhanden.**

d) Welche Aussage zum Thema Ablaufplanung ist richtig?

- **Bei der FCFS-Strategie kann es aufgrund des Konvoieffekts zu hohen Antwortzeiten kommen.**
- Offline-Einplanungsverfahren eignen sich vor allem fuer den Einsatz auf mobilen Geraeten, da diese auch ohne Internetverbindung arbeiten koennen.
- In Echtzeitsystemen kommt es auf maximalen Durchsatz an, weshalb hier ausschliesslich nicht-unterbrechbare Schedulingverfahren verwendet werden.
- Asymmetrische Einplanungsverfahren zielen auf eine optimale Behandlung von Prozessmengen, die sich in E/A- und CPU-Intensitaet stark voneinander unterscheiden.

e) Welche Aussage bezueglich der Freispeicherverwaltung mittels einer Bitliste ist richtig?

- Der zu verwaltende Speicher wird in Speichereinheiten unterschiedlicher Groesse unterteilt.
- **Zur Suche nach freiem Speicher kann es noetig sein, die gesamte Bitliste zu durchsuchen.**
- Das Zusammenfassen von benachbarten freien Speichereinheiten ist besonders aufwaendig.
- Je feiner die Granularitaet der Speichereinheiten ist, desto kuerzer ist die Bitliste.

f) Sie kennen den Begriff Seitenflattern (Thrashing). Welche Aussage ist richtig?

- Als Seitenflattern bezeichnet man das wiederholte Loeschen und Neuladen des Translation-Look-Aside-Buffer (TLB), ausgeloest durch haeufigen Prozesswechsel.
- **Als Seitenflattern bezeichnet man das wiederholte Einlagern einer erst vor kurzem verdraengten Speicherseite. Die Prozesse verbringen als Folge die meiste Zeit mit dem Warten auf die Behebung von Seitenfehlern.**
- Seitenflattern erkennt man an der starken Geraeuschentwicklung der Festplatte, da auf Grund haeufiger Seitenzugriffe der Lesekopf staendig neu positioniert wird. Bei Systemen ohne Festplatte (z. B. Thin-Clients) kann das Seitenflattern nicht auftreten.
- Bei Verwendung der LRU-Seitenersetzungstrategie kann Seitenflattern prinzipbedingt nicht auftreten.

g) Welche Aussage zum Thema Äktives Wartenïst richtig?

- Aktives Warten vergeudet gegenueber passivem Warten immer CPUZeit
- Auf Mehrprozessorsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen
- Aktives Warten sollte bei einer nicht-verdraengenden Scheduling-Strategie auf einem Monoprozessorsystem dem passiven Warten vorgezogen werden
- **Zur Implementierung einer Schlossvariable mit aktivem Warten ist keine Unterstuetzung durch das Betriebssystem notwendig.**

h) Nehmen Sie an, der Ihnen bekannte Systemaufruf stat(2) waere analog zu der Funktion readdir(3) mit folgender Schnittstelle implementiert: struct stat *stat(const char *path); Welche Aussage ist richtig?

- **Ein Zugriff ueber den zurueckgelieferten Zeiger liefert voellig zufaellige Ergebnisse oder einen Segmentation fault.**
- Der Systemaufruf liefert einen Zeiger zurueck, ueber den die aufrufende Funktion direkt auf eine Datenstruktur zugreifen kann, welche die Dateiattribute enthaelt.
- Solch eine Schnittstelle ist nicht schoen, da dadurch die aufrufende Funktion auf internen Speicher des Betriebssystems zugreifen koennte.
- Der Aufrufer muss sicherstellen, dass er den zurueckgelieferten Speicher mit free(3) wieder freigibt, wenn er die Dateiattribute nicht mehr weiter benoetigt.

i) Welche der folgenden Aussagen zum Thema Threads ist richtig?

- Zur Umschaltung von User-Threads verschiedener Prozesse ist kein Adressraumwechsel erforderlich.
- Kern-Threads teilen sich den kompletten Adressraum und verwenden daher den selben Stack.
- Auf Multiprozessorsystemen kann die Umschaltung von Kern-Threads ohne Mitwirken des Systemkerns erfolgen.
- **Der Synchronisationsbedarf im Anwendungsprogramm kann von der Ablaufplanung der Kernfaeden abhaengen.**

j) In einem UNIX-UFS-Dateisystem gibt es symbolische Namen/Verweise (Symbolic Links). Welche Aussage ist richtig?

- Der Systemaufruf stat() liefert im Gegensatz zum Systemaufruf lstat() die Dateiattribute des symbolischen Verweises und nicht die Attribute vom Ziel des Verweises.
- Ein symbolischer Verweis kann ausschliesslich auf regulaere Dateien verweisen.
- **Beim Zugriff auf einen Symbolic Link kann ein No such file or directory.Fehler auftreten (errno==ENOENT), obwohl der Symbolic Link existiert.**
- In jedem Inode ist ein Referenzzaehler gespeichert, welcher die Anzahl der Symbolic Links angibt, die auf ihn verweisen.

5.2 Aufgabe 1.2 (8 Punkte)

a	b
1358	1246

a	b	c	d	e	f	g	h
100%	50%	50%	50%	50%	50%	0%	50%

a) Welche der folgenden Aussagen zum Thema persistenter Datenspeicherung sind richtig?

- **Bei kontinuierlicher Speicherung von Daten ist es unter Umstaenden mit enormem Aufwand verbunden, eine bestehende Datei zu vergroessern.**
- Bei verketteter Speicherung dauert der wahlfreie Zugriff auf eine bestimmte Dateiposition immer gleich lang, wenn Cachingeffekte ausser Acht gelassen werden.
- **Bei indizierter Speicherung von Dateien muessen unter Umstaenden mehrere Bloecke geladen werden, bevor der Dateiinhalt gelesen werden kann.**
- Extents finden aus Performanzgruenden keine Anwendung in modernen Dateisystemen.
- **Journaling-Dateisysteme garantieren, dass auch nach einem Systemausfall alle Metadaten wieder in einen konsistenten Zustand gebracht werden koennen.**
- Journaling-Dateisysteme sind immun gegen defekte Plattenbloecke.
- Durch den Einsatz von mehreren Platten wird bei RAID 0 die Ausfallsicherheit des Gesamtsystems so stark erhoeht, dass Datensicherungen ueberfluessig sind.
- **Festplatten eignen sich besser fuer sequentielle als fuer wahlfreie Zugriffsmuster.**

b) Welche der folgenden Aussagen zum Thema Schedulingverfahren sind richtig?

- **Kooperative Schedulingverfahren ermoeglichen keinen wirksamen CPU-Schutz.**
- **Verdraengende Schedulingverfahren koennen nur mit Hilfe von Unterbrechungen realisiert werden.**
- Asymmetrische Schedulingverfahren sind nur bei Systemen mit heterogenen Prozessoren (z. B. Grafikprozessoren und CPUs) einsetzbar.
- **Symmetrische Schedulingverfahren sorgen fuer eine gleichmaessige Lastverteilung auf homogenen Multiprozessorsystemen.**
- Online-Schedulingverfahren sind fuer den Einsatz in Rechnern ohne Netzwerkschnittstelle ungeeignet.
- **Bei Offline-Schedulingverfahren wird der vollstaendige Ablaufplan bereits vor dem Start des Systems erstellt.**
- Probabilistische Schedulingverfahren eignen sich fuer Systeme mit harten Echtzeitanforderungen.
- Deterministische Schedulingverfahren sind nur in der Theorie relevant, da die genaue Laenge der CPU-Stoesse nie vorhergesagt werden kann.

5.3 Aufgabe 2 - msgpipe (60 Punkte)

Schreiben Sie ein Programm **msgpipe**, das ueber einen Netzwerk-Socket Nachrichten entgegennimmt und an den gewuenschten Empfaenger auf dem lokalen System ausliefert. Der Inhalt jeder Nachricht soll dabei durch ein vom jeweiligen Empfaenger definiertes Filter-Programm verarbeitet werden.

msgpipe wird mit einem optionalen Parameter aufgerufen, der angibt, auf welchem Port Verbindungen angenommen werden sollen. Gibt der Aufrufer kein Befehlszeilenargument mit, soll standardmaessig auf Port 2012 gelauscht werden. Bei einer ungueltigen Anzahl von Argumenten soll das Programm den Benutzer auf die korrekte Verwendung hinweisen und sich beenden.

Im Konfigurationsverzeichnis **/etc/msgpipe** liegt fuer jeden dem System bekannten Empfaenger eine gleichnamige (d. h. Name des Empfaengers) Datei, die eine Zeile mit dem Namen des von ihm gewuenschten Filter-Programms (Maximallaenge: 256 Zeichen) enthaelt. Existiert fuer einen Empfaenger keine Konfigurationsdatei oder schlaegt das Lesen der Datei fehl, wird die Nachricht abgewiesen.

Jedes Filter-Programm erwartet den Namen des Empfaengers als Befehlszeilenargument und den Nachrichtentext auf dem Standardeingabe-Kanal.

Die Kommunikation laeuft wie folgt ab: Pro Verbindung wird genau eine Nachricht uebertragen. Der Client sendet zunaechst eine Zeile mit dem Namen des Empfaengers (Maximallaenge: 256 Zeichen; es kann davon ausgegangen werden, dass der Name nur aus alphanumerischen Zeichen besteht).

Wird die Nachricht abgewiesen, so antwortet der Server mit der Zeile **Rejected** und beendet die Verbindung, andernfalls wird die Antwortzeile **Accepted** an den Client gesendet. Dieser schickt daraufhin einen beliebig langen Nachrichtentext, der durch das Filter-Programm bearbeitet wird.

Nach erfolgreicher Abarbeitung durch das Filter-Programm sendet der Server die Antwortzeile OK zurueck, ansonsten **Failed**.

Das Programm soll im Detail folgendermassen arbeiten:

- Das Hauptprogramm nimmt auf dem angegebenen Port Verbindungen an. Die Abarbeitung jeder Anfrage erfolgt innerhalb eines eigenen Prozesses durch Aufrufen der Funktion **handleRequest()**. Hierbei sollen nicht mehr als 16 Kindprozesse gleichzeitig aktiv sein. Ist die Maximalanzahl bereits erreicht und eine neue Verbindung wird akzeptiert, wird passiv gewartet, bis ein Kindprozess sich beendet hat. Falls waehrenddessen noch weitere Verbindungsanfragen von Clients eintreffen, sollen nicht mehr als 8 von ihnen zwischengespeichert werden - darueber hinausgehende werden abgewiesen.
- Funktion void **handleRequest(FILE *client)**: Fuehrt die Anfragebearbeitung durch. Der Name des Empfaengers wird gelesen, anschliessend wird die zugehoerige Datei aus dem Konfigurationsverzeichnis eingelesen und dann das Filter-Programm ausgefuehrt (Funktion **runFilterApp()**). Das Trennen der Verbindung ist Aufgabe des Aufrufers.
- Funktion int **runFilterApp(const char program[], const char recipient[], FILE *client)**: Startet einen Prozess, in dem das angegebene Programm mit dem Namen des Empfaengers als Argument ausgefuehrt wird, und wartet auf seine Beendigung. Der Nachrichtentext, der vom Client kommt, wird in den Standardeingabe-Kanal des Filter-Programms geleitet. Hat sich der Prozess regulaer mit Exit-Status 0 beendet, gibt die Funktion 0 zurueck, sonst -1.

5.3.1 msgpipe.c

```c
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

/* Konstanten, Funktions- und Strukturdeklarationen, globale Variablen */
static volatile int curProcs;

static void sighandler(int signal);
static void handleRequest(FILE* client);
static int runFilterApp(const char program[], const char recipient[], FILE* client);

static void die(const char *x) { perror(x); exit(1); }

/* Funktion main */
int main(int argc, char* argv[]) {
  int port=2012;
  sigset_t old,block;

  /* Argumente auswerten und weitere allgemeine Vorbereitungen */
  if(1!=argc && 2!=argc) { fprintf(stderr,"Usage: %s [port]\n",argv[1]); return 1;}
  if(2==argc) port=atoi(argv[1]);
  if(0>port) { fprintf(stderr,"negative port!\n"); return 1; }

  /* Signalbehandlung und Signalmasken initialisieren */
  struct sigaction action = {
    .sa_handler=sighandler,
    .sa_flags=SA_RESTART | SA_NOCLDSTOP
  };
  sigemptyset(&action.sa_mask);
  sigaction(SIGCHLD,&action,NULL);

  sigemptyset(&block);
  sigaddset(&block,SIGCHLD);

  sigprocmask(SIG_UNBLOCK,&block,NULL);

  /* Socket erstellen und auf die Verbindungsannahme vorbereiten */
  int fd=socket(AF_INET6,SOCK_STREAM,0);
  if(-1==fd) { die("socket"); }
  struct sockaddr_in6 addr = {
    .sin6_family=AF_INET6,
    .sin6_port=htons(port),
    .sin6_addr=in6addr_any
  };
  int flag;
  if(0!=setsockopt(fd,SOL_SOCKET,SO_REUSEADDR,&flag,sizeof(flag))) die("setsockopt");
  if(-1==bind(fd,(struct sockaddr*)&addr,sizeof(addr))) die("bind");
  if(-1==listen(fd,8)) die("listen");

  /* Verbindungen annehmen */
  int cs;
  while(-1!=(cs=accept(fd,NULL,NULL))) {
```

```
59      /* Warten, falls schon zu viele Kindprozesse laufen */
60      sigprocmask(SIG_BLOCK,&block,&old);
61      while(16==curProcs) sigsuspend(&old);
62      sigprocmask(SIG_SETMASK,&old,NULL);
63
64      /* Anfrage abarbeiten */
65      pid_t p=fork();
66      if(0>p) { die("fork"); }
67      if(!p) {
68        close(fd);
69        FILE* f=fdopen(cs,"a+");
70        if(!f) { die("ffdopen"); }
71        handleRequest(f);
72        fclose(f);
73        exit(0);
74      }
75      close(cs);
76    }
77    die("accept");
78  }
79
80  /* Signalbehandlung SIGCHLD */
81  static void sighandler(int signal) {
82    int old_errno = errno;
83    while(0>waitpid(-1,NULL,WNOHANG)) curProcs--;
84    errno=old_errno;
85  }
86
87  /* Funktion Handlerequest */
88  static void handleRequest(FILE* client) {
89    char name[257];
90    if(!fgets(name,sizeof(name),client)) { if (ferror(client)) perror("fgets"); return; }
91    strtok(name, "\r\n");
92    char file[strlen("msgp")+2+strlen(name)];
93    sprintf(file,"%s/%s","msgp",name);
94    FILE* r=fopen(file,"r");
95    if(!r) { fputs(client, "Rejected"); return; }
96    char fname[257];
97    if(!fgets(fname,sizeof(fname),r)) { if (ferror(r)) perror("fgets"); return; }
98    if(fclose(r)) { perror("fclose"); return; }
99    strtok(fname, "\r\n");
100   fputs(client, "Accepted");
101   if(-1==runFilterApp(fname,name,client)) fputs(client, "Failed"); else fputs(client, "OK");
102 }
103
104 /* Funktion runFilterApp */
105 static int runFilterApp(const char program[], const char recipient[], FILE* client) {
106   struct sigaction action = {
107     .sa_handler=SIG_DFL
108   };
109   sigemptyset(&action.sa_mask);
110   sigaction(SIGCHLD,&action,NULL);
111
112   int status;
113   pid_t p=fork();
114   if(0>p) { perror("fork"); return -1; }
115   if(!p) {
116
117     if(-1==dup2(fileno(client),STDIN_FILENO)) die("dup2");
118
119     execl(program,program,recipient,NULL);
```

```
120     die("execl");
121   }
122   if(-1==waitpid(p,&status,0)) { perror("waitpid"); return -1; }
123   if(WIFEXITED(status) && 0==WEXITSTATUS(status)) return 0;
124   return -1;
125 }
```

6 Juli 2012

6.1 Aufgabe 1.1 (22 Punkte)

a	b	c	d	e	f	g	h	i	j
2	1	4	2	3	2	4	3	1	4

a	b	c	d
20.00%	**30.00%**	20.00%	**30.00%**

a) Welche Aussage zum Thema Adressraumverwaltung ist richtig?

- Da das Laufzeitsystem auf die Betriebssystemschnittstelle zur Speicherverwaltung zurueckgreift, ist die Granularitaet der von malloc() zurueckgegebenen Speicherbloecke vom Betriebssystem vorgegeben.
- **Ein Speicherbereich, der mit Hilfe der Funktion free() freigegeben wurde, verbleibt im logischen Adressraum des zugehoerigen Prozesses.**
- Mit Hilfe des Systemaufrufes malloc() kann ein Programm zusaetzliche Speicherbloecke von sehr feinkoerniger Struktur vom Betriebssystem anfordern.
- Mit malloc() angeforderter Speicher, welcher vor Programmende nicht freigegeben wurde, kann vom Betriebssystem nicht mehr an andere Prozesse herausgegeben werden und ist damit bis zum Neustart des Systems verloren.

b) Man unterscheidet Traps und Interrupts. Welche Aussage ist richtig?

- **Der Zugriff auf eine virtuelle Adresse kann zu einem Trap fuehren.**
- Bei der mehrfachen Ausfuehrung eines unveraenderten Programms mit gleicher Eingabe treten Interrupts immer an den gleichen Stellen auf.
- Der Zeitgeber (Systemuhr) unterbricht die Programmbearbeitung in regelmaessigen Abstaenden. Die genaue Stelle der Unterbrechungen ist damit vorhersagbar. Somit sind solche Unterbrechungen in die Kategorie Trap einzuordnen.
- Wenn ein Interrupt einen schwerwiegenden Fehler signalisiert, muss das unterbrochene Programm abgebrochen werden.

c) Welche der folgenden Informationen wird typischerweise in dem Seitendeskriptor einer Seite eines virtuellen Adressraums gehalten?

- die Position der Seite im virtuellen Adressraum
- die Identifikation des Prozesses, dem die Seite zugeordnet ist
- die Zuordnung zu einem Segment (Text, Daten, Stack, ...)
- **Zugriffsrechte (z. B. lesen, schreiben, ausfuehren)**

d) Wodurch kann es zu Seitenflattern kommen?

- wenn sich zu viele Prozesse im Zustand blockiert befinden
- **wenn bei einer nicht-verdraengenden Scheduling-Strategie die Zahl der von den Prozessen aktiv genutzten Seiten die Zahl der verfuegbaren Seitenrahmen uebersteigt**
- durch Programme, die eine Defragmentierung auf der Platte durchfuehren
- wenn zu viele Prozesse im Rahmen der mittelfristigen Einplaung ausgelagert (swap-out) wurden

e) Welche Aussage zu Speicherzuteilungsverfahren ist richtig?

- best-fit ist in jedem Fall das beste Verfahren
- buddy-Verfahren sind nur bei sehr leistungsfaehigen Rechnern einsetzbar, weil sie aufwaendig in der Berechnung sind.
- **die worst-fit-Strategie kann mit der kuerzesten Antwortzeit einen ausreichend grossen Speicherbereich liefern**
- die worst-fit-Strategie ist lediglich theoretisch interessant, da es in der Praxis nie sinnvoll ist, den am schlechtesten passenden Speicherplatz zuzuweisen.

f) Beim Einsatz von RAID-Systemen kann durch zusaetzliche Festplatten Fehlertoleranz erzielt werden. Welche Aussage dazu ist richtig?

- Bei RAID 4 Systemen wird Paritaetsinformation gleichmaessig ueber alle beteiligten Platten verteilt.
- **Bei allen RAID-Systemen ist ein hoeherer Lese-Durchsatz als bei einer einzelnen Platte moeglich, da mehrere Platten gleichzeitig beauftragt werden koennen.**
- Bei RAID 4 und 5 darf eine bestimmte Menge von Festplatten nicht ueberschritten werden, da es sonst nicht mehr moeglich ist, die Paritaetsinformation zu bilden.
- RAID 0 erzielt Fehlertoleranz durch das Verteilen der Daten auf mehrere Platten.

g) Wodurch kann Nebenlaeufigkeit in einem System entstehen?

- durch Seitenflattern
- durch langfristiges Scheduling
- durch Compiler-Optimierungen
- **durch Threads auf einem Monoprozessorsystem mit praeemptivem Scheduling**

h) Welche Aussage zum Thema Äktives Wartenïst richtig?

- Aktives Warten vergeudet gegenueber passivem Warten immer CPUZeit
- Auf Mehrprozessorsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen
- **Aktives Warten darf bei nicht-verdraengenden Scheduling-Strategien auf einem Monoprozessorsystem nicht verwendet werden**
- Bei verdraengenden Scheduling-Strategien verzoegert aktives Warten nur den betroffenen Prozess, behindert aber nicht andere

i) Wozu dient der Maschinenbefehl cas (compare-and-swap)?

- **Um auf einem Multiprozessorsystem einfache Modifikationen an Variablen ohne Sperren implementieren zu koennen.**
- Um bei Monoprozessorsystemen Interrupts zu sperren.
- Um bei der Implementierung von Schlossvariablen (Locks) aktives Warten zu vermeiden.
- Zur Realisierung einer effizienten Verdraengungssteuerung bei einseitiger Symchronisation.

j) Virtualisierung kann als Massnahme gegen Verklemmungen genutzt werden. Warum?

- Im Fall einer Verklemmung koennen zusaetzliche virtuelle Betriebsmittel neu erzeugt werden. Diese koennen dann eingesetzt werden, um die fehlenden physikalischen Betriebsmittel zu ersetzen.
- Durch Virtualisierung kann man ueber Abbildungsvorgaenge Zyklen, die auf der logischen Ebene vorhanden sind, auf der physikalischen Ebene aufloesen.
- Eine Verklemmungsaufloesung ist einfacher, weil virtuelle Betriebsmittel jederzeit ohne Schaden entzogen werden koennen.
- **Durch Virtualisierung ist ein Entzug von physikalischen Betriebsmitteln moeglich, obwohl dies auf der logischen Ebene unmoeglich ist.**

6.2 Aufgabe 1.2 (8 Punkte)

a	b
235	4568

a	b	c	d	e	f	g	h
0%	50%	50%	50%	**100%**	50%	0%	50%

a) Welche der folgenden Aussagen zum Thema Einplanung sind richtig?

- Einplanungsverfahren lassen sich in drei Kategorien einteilen: federgewichtig, leichtgewichtig und schwergewichtig.
- **Ein Prozess, der sich im Zustand laufend befindet, kann nicht direkt in den Zustand schwebend blockiert ueberfuehrt werden.**
- **Prozesse im Zustand gestoppt sind der langfristigen Einplanung zuzuordnen.**
- Ein Prozess kann sich in realen Systemen nie im Zustand beendet befinden, da bei seiner Terminierung saemtliche Betriebsmittel freigegeben werden und damit auch der Prozess selbst verschwindet.
- **Fuer die mittelfristige Einplanung muss das Betriebssystem die Umlagerung (engl. swapping) von kompletten Programmen bzw. logischen Adressraeumen unterstuetzen.**
- Ein Prozess im Zustand erzeugt kann sich selbst durch die Ausfuehrung des Systemaufrufes exec() in den Zustand bereit ueberfuehren.
- Prozesse im Zustand blockiert oder bereit koennen unmittelbar in den Zustand gestoppt ueberfuehrt werden.
- Verdraengende Prozesseinplanung bedeutet, dass das Eintreten des erwarteten Ereignisses unmittelbar die Einlastung des wartenden Prozesses bewirkt.

b) Welche der folgenden Aussagen zum Thema Adressraumkonzepte sind richtig?

- Da virtuelle Adressraeume direkt durch die MMU unterstuetzt werden, koennen zur Laufzeit keine zusaetzlichen Kosten gegenueber logischen Adressraeumen auftreten.
- Bei einer seitenorientierten Adressraumorganistaion muss die Groesse jeder Seite in der Seitentabelle gespeichert werden.
- Der Adressraumschutz durch Abteilung ermoeglicht die Isolation mehrerer Prozesse voneinander.
- **Bei Adressraumschutz durch Abteilung werden die Betriebssystemdaten vor Zugriffen aus dem Anwendungsprogramm geschuetzt.**
- **Ein Seitenfehler wird abhaengig von der Ursache nach dem Fortsetzungs- oder dem Beendigungsmodell behandelt.**
- **Ein Adressraumwechsel ist eine privilegierte Operation und kann daher nur durch das Betriebssystem vorgenommen werden.**
- Fuer den Adressraumschutz durch Eingrenzung ist zwingend eine MMU erforderlich.
- **Zur Implementierung von logischen Adressraeumen ist spezielle Hardwareunterstuetzung noetig.**

6.3 Aufgabe 2 - chatserver (60 Punkte)

Schreiben Sie ein mehrfaediges Programm **chatserver**, das einen Gruppenkommunikationsdienst anbietet. Teilnehmer (Clients) koennen sich ueber den TCP-Port 6670 mit dem Dienst verbinden und untereinander Textnachrichten austauschen. Der Server fungiert als sogenanntes Relay: Jede Nachricht, die er entgegennimmt, leitet er an alle aktuell verbundenen Teilnehmer weiter.

Das Kommunikationsprotokoll ist wie folgt definiert:

Der Dienst kann von maximal 100 (**MAX_CLIENTS**) Clients gleichzeitig genutzt werden. Ein Client muss hierfuer eine TCP-Verbindung auf dem angegebenen Port herstellen und an den Server eine Zeile mit seinem Nutzernamen (Maximallaenge: 8 Zeichen, **MAX_USER_LEN**) senden. Zur Vereinfachung kann angenommen werden, dass der Server nicht auf Namenskollisionen oder ueberlange Zeilen prueft. Der Client kann nun an der Gruppenkommunikation teilnehmen, bis die SocketVerbindung getrennt wird. Jede weitere Textzeile, die er an den Server sendet (Maximallaenge: 1024 Zeichen, **MAX_MSG_LEN**), wird von diesem mit einem Praefix versehen und an alle Teilnehmer (auch an den Autor selber) weitergeleitet. Das Praefix enthaelt den Namen des Autors in spitzen Klammern, gefolgt von einem Tabulator-Zeichen. Eine Zeile, die vom Server versendet wird, koennte zum Beispiel so aussehen:

<Faust> Das also war des Pudels Kern!

Das Programm (**chatserver.c**) soll folgendermassen strukturiert sein:

- Das Hauptprogramm nimmt auf dem angegebenen Port Verbindungen an. Jede akzeptierte Verbindung wird in Form eines **FILE *** in einen Ringpuffer ("Verbindungs-Puffer") mit einer Kapazitaet von 10 Elementen (**CONN_BUF_SIZE**) eingetragen. Die weitere Abarbeitung wird durch separate Threads durchgefuehrt. Hierfuer werden zu Beginn **MAX_CLIENTS** LeseThreads und ein Sende-Thread gestartet, die dann endlos laufen.
- Lese-Thread (Funktion **void *receive(void *arg)**): Entnimmt jeweils eine Client-Verbindung aus dem Verbindungs-Puffer und bedient diese. Jede gelesene Textzeile wird wie im Protokoll spezifiziert mit einem Praefix versehen und zur Ausgabe an die anderen Teilnehmer in einen zweiten Ringpuffer ("Nachrichten-Puffer") der Groesse 128 (**LINE_BUF_SIZE**) geschrieben.

  Die Client-Verbindungen werden in einem globalen Feld mit einem Eintrag pro LeseThread abgelegt. Ist ein Lese-Thread nicht aktiv, traegt er **NULL** ein.
- Sende-Thread (Funktion **void *broadcast(void *arg)**): Entnimmt jeweils eine Nachricht aus dem Nachrichten-Puffer und sendet sie an alle aktuell verbundenen Clients.

Implementieren Sie ferner ein generisches FIFO-Ringpuffer-Modul (**bbuffer.c**), das Elemente vom Typ **void *** verwaltet. Der Puffer soll so synchronisiert sein, dass Ueber- und Unterlaufsituationen verhindert werden und dass seine Operationen im gegebenen Anwendungskontext threadsicher sind.

Der Ringpuffer soll die folgenden Funktionen anbieten:

`BNDBUF *bb_init(size_t size);`	Konstruktur (liefert im Fehlerfall **NULL**)
`void bb_add(BNDBUF *bb, void *value);`	fuegt ein Element in den Ringpuffer ein
`void *bb_get(BNDBUF *bb);`	entfernt ein Element aus dem Ringpuffer

Sie koennen fuer Synchronisationszwecke in allen Programmteilen die existierenden Funktionen

SEM *sem_init(int initVal); void P(SEM *sem); void V(sem *sem);

eines **nicht** selbst zu implementierenden Semaphor-Moduls benutzen. Dabei koennen Sie davon ausgehen, dass **sem_init()** nie fehlschlaegt.

6.3.1 chatserver.c

```c
/** TODO:
  * 1. Lieber BNDBUFs ueber void* arg uebergeben?
  * 2. Wie die client FILE* ARray verwalten und die curProcs hochzaehlen - In der Main etc?
  * 3. Die Threads nach create in den jeweiligen Funktionen detachen?
  * 4. Makefile error... has an incomplete type der struct, wsch wgn header
  **/

#include "bbuffer.h"
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>

#include <unistd.h> //TODO: Wird fuer close gebraucht gibts aber nicht in der angabe??

/* Konstanten */
#define MAX_CLIENTS 100
#define MAX_USER_LEN 8
#define MAX_MSG_LEN 1024
#define CONN_BUF_SIZE 10
#define LINE_BUF_SIZE 128

/* Globale Variablen, Funktionsdeklarationen usw. */
static BNDBUF* cbuf, mbuf;
static volatile int curProcs=0; //TODO: USE IT!
static FILE* carr[MAX_CLIENTS];

static void die(const char* name) { perror(name); exit(1); }
static void* pdie(const char* name) { perror(name); return NULL; }

static void* receive(void* arg);
static void* broadcast(void* arg);

/* Funktion main() */
int main(int argc, char* argv[]) {
  /* Allgemeine Vorbereitungen */
  cbuf=(BNDBUF*) bb_init(CONN_BUF_SIZE);
  if(!cbuf) die("bb_init");
  mbuf=(BNDBUF*) bb_init(LINE_BUF_SIZE);
  if(!mbuf) die("bb_init");

  /* Lese-Threads und Sende-Thread starten */
  pthread_t p;
  for(int i=0;i<MAX_CLIENTS;i++) {
    errno=pthread_create(&p,NULL,&receive,NULL);
    if(errno) die("pthread_create");
  }
  errno=pthread_create(&p,NULL,&broadcast,NULL);
  if(errno) die("pthread_create");

  /* Socket erstellen und auf Verbindungsannahme vorbereiten */
  int ss = socket(PF_INET6,SOCK_STREAM,0);
  if(-1==ss) die("socket");
  struct sockaddr_in6 a = {
    .sin6_family=AF_INET6,
```

```
59      .sin6_port=htons(6670),
60      .sin6_addr=in6addr_any
61    };
62    if(bind(ss,(struct sockaddr *)&a,sizeof(a))) die("bind");
63    if(listen(ss,MAX_CLIENTS)) die("listen"); //TODO: -1==func ist genauso wie if(func) ???
64
65    /* Client-Verbindungen annehmen */
66    int cs;
67    while(-1!=(cs=accept(ss,NULL,NULL))) {
68      FILE* f=fdopen(cs,"a+");
69      if(!f) { perror("fdopen"); close(cs); continue; }
70      bb_add(cbuf,f);
71    }
72    die("accept");
73  } /* Ende Funktion main() */
74
75  /* Lese-Thread: Funktion receive() */
76  static void* receive(void* arg) {
77    errno=pthread_detach(pthread_self());
78    if(errno) pdie("pthread_detach");
79
80    while(1) {
81      FILE* c=(FILE*) bb_get(cbuf);
82
83      char name[MAX_USER_LEN+1]; //TODO: +1 oder nicht??
84      if(!fgets(name,sizeof(name),c)) {
85        if(ferror(c)) perror("fgets");
86        if(fclose(c)) perror("fclose");
87        continue;
88      }
89      strtok(name,"\r\n");
90
91      char cmsg[MAX_MSG_LEN+1]; //TODO: +1 oder nicht?
92      while(fgets(cmsg,sizeof(cmsg),c)) {
93        char* msg =(char*) malloc((strlen(name)+4+MAX_MSG_LEN)*sizeof(char));
94        if(!msg) die("malloc"); //TODO: Soll das hier zum Absturz vom ganzen System fuehren??
95        sprintf(msg,"<%s>\t%s",name,cmsg);
96        bb_add(mbuf,msg);
97      }
98
99      if(ferror(c)) perror("fgets");
100     if(fclose(c)) perror("fclose");
101   }
102   fprintf(stderr,"IILB\n");
103   return NULL;
104 }
105
106 /* Sende-Thread: Funktion broadcast() */
107 static void* broadcast(void* arg) {
108   errno=pthread_detach(pthread_self());
109   if(errno) die("pthread_detach"); //Wenn der einzige Sender Thread stirbt, macht das Program keinen Si
110
111   while(1) {
112     char* msg=(char*) bb_get(mbuf);
113
114     for(int i=0;i<MAX_CLIENTS;i++) {
115       if(!carr[i]) continue;
116       fputs(msg,carr[i]);
117       if(feof(carr[i])) perror("fputs"); //TODO: Ist das richtig so? Oder lieber pdie benutzen | ferror
118     }
119
```

```
120     free(msg);
121     continue;
122   }
123   fprintf(stderr,"IILB\n");
124   return NULL;
125 }
```

6.3.2 jbuffer.c

```
1  #include "bbuffer.h"
2  #include "sem.h"
3  #include <stdlib.h>
4
5  typedef struct BNDBUF {
6    void** data;
7    int size; //sollte das hier nicht size_t sein???? fehler in angabe?
8    volatile int readPos,writePost;
9  //  SEM* full,free,readLock,writeLock;
10   SEM* readLock,writeLock;
11 } BNDBUF;
12
13 /* Funktion bb_init() */
14 BNDBUF* bb_init(size_t size) {
15   if(size>=INT_MAX) return NULL;
16   BNDBUF* bb=(BNDBUF*) malloc(sizeof(BNDBUF));
17   if(!bb) return NULL;
18
19   bb->data=(void**) malloc(sizeof(*bb->data));
20   if(!bb->data) return NULL;
21
22   bb->size=size;
23   bb->readPos=buf->writePos=0;
24   bb->readLock=sem_init(0);
25   bb->writeLock=sem_init(size);
26   return bb;
27 }
28
29 /* Funktion bb_add() */
30 void bb_add(BNDBUF* bb, void* value) {
31   P(bb->writeLock);
32   bb->data[bb->write%bb->size]=value;
33   bb->write++;
34   V(bb->readLock);
35 }
36
37 /* Funktion bb_get() */
38 void* bb_get(BNDBUF* bb) {
39   P(bb->readLock);
40   void* get;
41   int old=0, new=1;
42   do {
43     old=bb->readPos;
44     new=old+1;
45     get=bb->data[old];
46   } while(-1==__sync_bool_compare_and_swap(&bb->readPos,old,new%bb->size));
47   V(bb->writeLock);
48   return get;
49 }
```

7 Februar 2013

7.1 Aufgabe 1.1 (22 Punkte)

a	b	c	d	e	f	g	h	i	j	k
2	4	1	3	3	1	3	3	3	4	4

a	b	c	d
18.18%	9.09%	**45.45%**	27.27%

a) Beim Einsatz von RAID-Systemen kann durch zusaetzliche Festplatten ein fehlertolerierendes Verhalten erzielt werden. Welche Aussage dazu ist richtig?

- Bei RAID-4-Systemen wird die Paritaetsinformation gleichmaessig ueber alle beteiligten Platten verteilt.
- **Der Lesezugriff auf ein gestreiftes Plattensystem, insbesondere auch auf ein RAID-5 System, ist schneller, da mehrere Platten gleichzeitig beauftragt werden koennen.**
- Wenn bei einem RAID-4-System die Parity-Platte ausfaellt, sind alle Daten verloren.
- Bei einem RAID-5-System kommen immer genau 5 Platten zum Einsatz. Die Parity-Information wird dabei auf alle 5 Platten gleichmaessig verteilt.

b) Welche Aussage zu den Eigenschaften eines Journaling-Filesystems ist richtig?

- Alle Aenderungen am Dateisystem werden in Form von Transaktionen in einer Log-Datei mitprotokolliert. Wird nach einem Systemabsturz festgestellt, dass eine Transaktion in der Log-Datei unvollstaendig ist, muss die betroffene Datei geloescht werden.
- Es wird immer zuerst die Aenderung im Dateisystem auf der Platte duchgefuehrt und anschliessend zur Absicherung ein entsprechender Eintrag in die Log-Datei geschrieben.
- Je groesser eine Platte ist, desto laenger dauert der Reparaturvorgang eines Journaling-Filesystems nach einem Systemabsturz.
- **Die Eintraege in der Protokolldatei muessen immer auch Informationen zu einem Undo und Redo der Transaktion enthalten.**

c) Wie werden im Rahmen von Seitenersetzungsverfahren Freiseitenpuffer genutzt?

- **Wenn zu wenige freie Seitenrahmen im System vorhanden sind (low water mark erreicht) werden prophylaktisch Seiten ausgelagert, selbst wenn kein akuter Bedarf besteht.**
- Wird ein Seitenrahmen in den Freiseitenpuffer eingetragen, wird seine bisherige Zuordnung aus dem enstsprechenden Seitendeskriptor entfernt.
- Auf eine Seite im Freiseitenpuffer darf von dem bisherigen "Besitzer"(Prozess) noch weiterhin zum Lesen und Schreiben zugegriffen werden, bis der Seitenrahmen tatsaechlich fuer eine neue Seite genutzt wird.
- Auf eine Seite im Freiseitenpuffer darf von dem bisherigen "Besitzer"(Prozess) noch weiterhin, allerdings nur zum Schreiben, zugegriffen werden, bis der Seitenrahmen tatsaechlich fuer eine neue Seite genutzt wird.

d) Welche Aussage zum Thema Adressraumschutz ist richtig?

- Beim Adressraumschutz durch Eingrenzung ist es prinzipbedingt nicht moeglich, dass mehrere Prozesse auf ein Stueck gemeinsamen Speichers zugreifen.

- Beim Adressraumschutz durch Abteilung wird der logische Adressraum in mehrere Segmente mit unterschiedlicher Semantik unterteilt.
- **Bei allen Verfahren des Adressraumschutzes fuehrt jeder Zugriff auf eine ungueltige Speicheradresse zu einem Trap.**
- In einem segmentierten Adressraum kann zur Laufzeit kein weiterer Speicher mehr dynamisch nachgefordert werden.

e) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

- Der Prozess ist der statische Teil (Rechte, Speicher, etc.), das Programm der aktive Teil (Programmzaehler, Register, Stack).
- Wenn ein Programm nur einen aktiven Ablauf enthaelt, nennt man diesen Prozess, enthaelt das Programm mehrere Ablaeufe, nennt man diese Threads.
- **Ein Prozess ist ein Programm in Ausfuehrung - ein Prozess kann aber auch mehrere verschiedene Programme ausfuehren.**
- Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausfuehren.

f) Welche Aussage zum Thema Prozesse/Threads ist richtig?

- **Die Umschaltung schwergewichtiger Prozesse kann nur mit Hilfe des Betriebssystems erfolgen.**
- Eine gleichzeitige Ausfuehrung von mehreren schwergewichtigen Prozessen auf unterschiedlichen Prozessoren ist nicht moeglich.
- Die Einlastung eines federgewichtigen Prozesses ist eine privilegierte Operation und erfordert Unterstuetzung des Betriebssystems.
- Leichtgewichtige Prozesse bedingen den Einsatz von verdraengenden Schedulingverfahren.

g) Welche der folgenden Aussagen zum Thema Adressraeume ist richtig?

- Der logische Adressraum ist ebenso wie der physikalische Adressraum durch die gegebene Hardwarekonfiguration definiert.
- Der virtuelle Adressraum eines Prozesses kann nie groesser sein als der physikalisch vorhandene Arbeitsspeicher.
- **Der physikalische Adressraum ist durch die gegebene Hardwarekonfiguration definiert.**
- Die maximale Groesse des virtuellen Adressraums kann unabhaengig von der verwendeten Hardware frei gewaehlt werden.

h) Welche der folgenden Aussagen zum Thema Seitenfehler (page fault) ist richtig?

- Ein Seitenfehler zieht eine Ausnahmebehandlung nach sich. Diese wird dadurch ausgeloest, dass die MMU das Signal SIGSEGV an den aktuell laufenden Prozess schickt.
- Wenn der gleiche Seitenrahmen in zwei verschiedenen Seitendeskriptoren eingetragen wird, loest dies einen Seitenfehler aus (Gefahr von Zugriffskonflikten!).
- **Seitenfehler koennen auch auftreten, obwohl die entsprechende Seite gerade im physikalischen Speicher vorhanden ist.**
- Ein Seitenfehler wird ausgeloest, wenn der Offset in einer logischen Adresse groesser als die Laenge der Seite ist.

i) Bei der Behandlung von Ausnahmen (Traps oder Interrupts) unterscheidet man zwei Bearbeitungsmodelle. Welche Aussage hierzu ist richtig?

- Das Wiederaufnahmemodell dient zur Behandlung von Interrupts (Fortfuehrung des Programms nach einer zufaellig eingetretenen Unterbrechung). Bei einem Trap ist das Modell nicht sinnvoll anwendbar, da ein Trap deterministisch auftritt und damit eine Wiederaufnahme des Programms sofort wieder den Trap verursachen wuerde.
- Nach dem Beendigungmodell werden Interrupts bearbeitet. Gibt man z. B. CTRL-C unter UNIX ueber die Tastatur ein, wird ein Interrupt-Signal an den gerade laufenden Prozess gesendet und dieser dadurch beendet.
- **Interrupts duerfen auf keinen Fall nach dem Beendigungsmodell behandelt werden, weil ueberhaupt kein Zusammenhang zwischen dem unterbrochenen Prozess und dem Grund des Interrupts besteht.**
- Das Betriebssystem kann Interrupts, die in ursaechlichem Zusammenhang mit dem gerade laufenden Prozess stehen, nach dem Beendigungsmodell behandeln, wenn eine sinnvolle Fortfuehrung des Prozesses nicht mehr moeglich ist.

j) Welche Aussage zu nicht-blockierender Synchronisation ist richtig?

- Bei allen nicht-blockierenden Verfahren tritt das ABA-Problem auf.
- Verfahren zur nicht-blockierenden Synchronisation benoetigen keine spezielle Hardware-Unterstuetzung.
- In vielen Faellen sind die Algorithmen bei Verwendung nicht-blockierender Synchronisation einfacher zu beschreiben als bei blockierender Synchronisation.
- **Bei nicht-blockierenden Verfahren koennen keine Verklemmungen auftreten.**

k) Wodurch kann Nebenlaeufigkeit in einem System entstehen?

- durch Seitenflattern
- durch langfristiges Scheduling
- durch Compiler-Optimierungen
- **durch Interrupts**

7.2 Aufgabe 1.2 (8 Punkte)

a	b
4678	247

a	b	c	d	e	f	g	h
0%	50%	0%	**100%**	0%	50%	**100%**	50%

a) Welche der folgenden Aussagen zu UNIX-Dateisystemen sind richtig?

- Dateien (Name, Attribute und Inhalt) werden in Dateikatalogen abgespeichert.
- Beim Anlegen einer Datei wird die maximale Groesse festgelegt. Wird sie bei einer Schreiboperation ueberschritten, wird ein Fehler gemeldet.
- Ein Dateikopf ist eine Verwaltungsstruktur, die vorne in der Datei gespeichert wird.
- **Ein hard link ist ein Verweis aus einem Katalog auf eine Dateiverwaltungsstruktur (Inode).**
- Auf einen Katalog darf immer nur ein hard link existieren.
- **Man darf einen hard link auf eine Datei erzeugen, auch wenn man keine Zugriffsrechte auf diese Datei hat.**
- **In einem Verzeichnis darf es nicht mehrere Eintraege mit gleichem Namen geben, selbst wenn diese auf verschiedene Inodes verweisen wuerden.**
- **Obwohl eine Datei geloescht wurde, kann es symbolic links geben, die noch auf sie verweisen.**

b) Welche der folgenden Aussagen zum Thema Einplanungsverfahren sind richtig?

- First-Come-First-Served ist nur im Stapelbetrieb bei lang laufenden Auftraegen sinnvoll einsetzbar.
- **Round-Robin benachteiligt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.**
- Virtual-Round-Robin benachteiligt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.
- **Prioritaeten-basierte Verfahren sind auch fuer interaktiven Betrieb gut geeignet.**
- Zur Realisierung von verdraengenden Einplanungsverfahren wird Hardwareunterstuetzung durch eine MMU benoetigt.
- Shortest-Process-Next ist nur theoretisch interessant, weil die Laenge der naechsten Rechenphase (CPU-Stoss) in der Praxis nicht abgeschaetzt werden kann.
- **Feedback-Strategien sind eine Erweiterung von Round-Robin um Prioritaetsebenen.**
- Echtzeitverarbeitung ist nur mit geeigneten Prioritaeten-setzenden Verfahren realisierbar.

7.3 Aufgabe 2 - videostreamer (60 Punkte)

Schreiben Sie ein mehrfaediges Programm **videostreamer**, das auf Anfrage ueber eine TCP-Verbindung transcodierte Videodateien ausliefert beziehungsweise die vorhandenen Videodateien auflistet. Die Videodateien liegen in unkomprimierter Form im aktuellen Arbeitsverzeichnis.

Beim Start des Programms wird ein Thread-Pool mit 16 Arbeiter-Threads erzeugt, die die Funktion **doTranscoding()** ausfuehren. Diese Threads erhalten ihre Auftraege in **VideoFrame**-Strukturen ueber einen Ringpuffer (Implementierung vorgegeben) mit einer Kapazitaet von 32 Elementen.

Das Programm nimmt Verbindungen auf Port 7083 entgegen. Jede angenommene Verbindung wird durch einen neu zu startenden Thread bedient (Thread-Funktion **void *serve(FILE *client))**. Der Thread liest von der Verbindung eine Zeile mit dem Kommando des Clients (Maximallaenge: 2048 Zeichen), bearbeitet es und schliesst die Verbindung.

Wird das Kommando **"LIST"** empfangen, ruft der Thread die Funktion **int listDir(FILE *client)** auf, die die Namen aller Eintraege im aktuellen Arbeitsverzeichnis zeilenweise auf den Socket ausgibt. Eintraege, deren Name mit einem Punkt beginnt, werden ignoriert. Im Erfolgsfall liefert **listDir()** 0 zurueck, im Fehlerfall -1.

Wird das Kommando **"PLAY Dateiname"** empfangen, ruft der Thread die Funktion **streamVideo()** mit dem angeforderten Dateinamen auf. Jedes andere Kommando ist ungueltig und wird mit der Zeile **"INVALID"** beantwortet. Tritt bei der Anfrageverarbeitung ein Fehler auf, erhaelt der Client die Antwortzeile **"FAILED"** und die Verbindung wird getrennt.

- Funktion **int streamVideo(const char fileName[], FILE *client)**:

  Erstellt und initialisiert zunaechst eine Struktur vom Typ **VideoJob**, die die Anfrage repraesentiert. Anschliessend wird die Videodatei geoeffnet und nacheinander die Video-Frames (Einzelbilder) eingelesen, wobei alle Frames gleich gross sind (Groesse: **VID_FRAME_SIZE**). Eventuelle unvollstaendige Frames am Dateiende werden verworfen. Jeder eingelesene Video-Frame erhaelt eine laufende Nummer und wird in eine Kontrollstruktur vom Typ **VideoFrame** eingehaengt. Die Kontrollstrukturen werden zur Weiterverarbeitung ueber den Ringpuffer an den Thread-Pool uebergeben. Nachdem alle Frames eingelesen und uebergeben wurden, wird gewartet, bis der letzte Frame transcodiert und ausgegeben ist, und danach die **VideoJob**-Struktur zerstoert. Der Rueckgabewert 0 signalisiert Erfolg, -1 einen Fehler.

- In der Funktion **void *doTranscoding(void *arg)** entnimmt der Arbeiter-Thread eine **VideoFrame**-Kontrollstruktur aus dem Ringpuffer und ruft die im Modul **transcode.o** vorgegebene Funktion **size_t transcode(void *outData, const void *inData, size_t inSize)** auf. Diese Funktion fuehrt das Transcodieren der uebergebenen Frame-Daten durch, schreibt das Ergebnis in den Puffer **outData** und gibt die tatsaechliche Groesse des Ausgabe-Frames zurueck. Es ist garantiert, dass die Ausgabe-Groesse die Eingabe-Groesse nicht uebersteigt.

  Der Ausgabe-Frame wird nun auf der Socketverbindung ausgegeben. Dabei ist darauf zu achten, dass die Frames in der richtigen Reihenfolge ausgegeben werden.

Fuer die Reihenfolge-Synchronisation soll ein Modul **atomiccounter** benutzt und implementiert werden, das die folgende Schnittstelle bereitstellt:

int acInit(AtomicCounter *c, int i);	- initialisiert den Zaehler; liefert 0 bzw. bei Fehler -1
void acDestroy(AtomicCounter *c);	- zerstoert den atomaren Zaehler
void acIncrement(AtomicCounter *c);	- inkrementiert den Zaehlerwert atomar um 1
void acWait(AtomicCounter *c, int v);	- blockiert den aufrufenden Thread so lange, bis der Zaehler den Wert v erreicht hat.

7.3.1 videostreamer.c

```c
#include "atomiccounter.h"
#include "transcode.h"
#include <dirent.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>

// Konstanten
#define MAX_REQUEST_LEN 2048
#define VID_FRAME_SIZE 256

// Struktur-Deklarationen
typedef struct VideoJob {
  FILE *out; // Ausgabe-Stream
  AtomicCounter processedFrames; // Schon verarbeitete Frames
} VideoJob;
typedef struct VideoFrame {
  int id; // Fortlaufende Frame-Nummer
  char data[VID_FRAME_SIZE]; // Frame-Daten
  VideoJob *job; // Job, zu dem der Frame gehoert
} VideoFrame;

// Externe Funktionen des Ringpuffer-Moduls
BNDBUF *bbCreate(size_t size);
void bbDestroy(BNDBUF *bb);
void bbPut(BNDBUF *bb, VideoFrame *frame);
VideoFrame *bbGet(BNDBUF *bb);

// Hilfsfunktion
static int die(const char message[]) { perror(message); exit(EXIT_FAILURE); }

// Globale Variablen, Funktionsdeklarationen usw.

// Funktion main()
  // Allgemeine Vorbereitungen
  // Verbindungen annehmen und in neuem Thread abarbeiten
  // Socket erstellen und auf Verbindungsannahme vorbereiten
// Ende Funktion main()

// Thread-Funktion serve()

// Funktion listDir()

// Funktion streamVideo()

// Thread-Pool-Funktion doTranscoding()

/*************************************************************
  * Modul atomiccounter
  *************************************************************/
typedef struct AtomicCounter {
  volatile int value;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
```

```
59 } AtomicCounter;
60
61 // Funktion acInit()
62
63 // Funktion acIncrement()
64
65 // Funktion acWait()
66
67 // Funktion acDestroy()
```

8 Juli 2013

8.1 Aufgabe 1.1 (22 Punkte)

a	b	c	d	e	f	g	h	i	j	k
4	2	3	1	1	3	3	1	1	2	4

a	b	c	d
36.$\overline{36}$%	18.$\overline{18}$%	27.$\overline{27}$%	18.$\overline{18}$%

a) Welche der folgenden Aussagen zum Thema Adressraeume ist richtig?

- Der virtuelle Adressraum eines Prozesses kann nie groesser sein als der physikalisch vorhandene Arbeitsspeicher.
- Die maximale Groesse des virtuellen Adressraums kann unabhaengig von der verwendeten Hardware frei gewaehlt werden.
- Virtuelle Adressraeume sind Voraussetzung fuer die Realisierung logischer Adressraeume.
- **Logische Adressraeume bieten Schutz vor Berechnungsfehlern.**

b) Der Speicher eines UNIX-Prozesses ist in Text-, Daten- und Stack-(Stapel-)Segment untergliedert. Welche Aussage zur Platzierung von Daten in diesen Segmenten ist richtig?

- Alle lokalen Variablen werden im Stack-Segment abgelegt.
- **Variablen der Speicherklasse *static* liegen im Daten-Segment.**
- Bei einem malloc-Aufruf wird das Stack-Segment dynamisch erweitert.
- Dynamisch allozierte Zeichenketten liegen im Text-Segment.

c) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

- Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzaehler, Register, Stack).
- Der Compiler erzeugt aus mehreren Programmteilen (Module) einen Prozess.
- **Ein Programm kann durch mehrere Prozesse gleichzeitig ausgefuehrt werden.**
- Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausfuehren.

d) Welche Aussage ueber exec() ist richtig?

- **Das im aktuellen Prozess laufende Programm wird durch das angegebene Programm ersetzt.**
- Der an exec() uebergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgefuehrt.
- exec() erzeugt einen neuen Kind-Prozess und startet darin das angegebene Programm.
- Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurueckgeliefert.

e) Bei der Behandlung von Ausnahmen (Traps oder Interrupts) unterscheidet man zwei Bearbeitungsmodelle. Welche Aussage hierzu ist richtig?

- **Das Wiederaufnahmemodell ist fuer Interrupts und Traps gleichermassen geeignet.**
- Das Beendigungsmodell ist fuer Interrupts und Traps gleichermassen geeignet.
- Bei der Behandlung einer Ausnahme nach dem Wiederaufnahmemodell wird der unterbrochene Prozess neu gestartet.
- Das Beendigungsmodell sieht das Herunterfahren des Betriebssystems im Falle eines schwerwiegenden Fehlers vor.

f) Welche Aussage zum Thema RAID ist richtig?

- Bei RAID 5 liegen die Paritaetsinformationen auf einer dedizierten Platte.
- Bei RAID 4 werden alle im Verbund beteiligten Platten gleichmaessig beansprucht.
- **Bei RAID 1 wird beim Lesen ein Geschwindigkeitsvorteil erzielt.**
- Bei RAID 0 fuehrt der Ausfall einer der beteiligten Platten nicht zu Datenverlust.

g) Welche Aussage zum Thema Seitenersetzungsstrategien ist richtig?

- Bei der Ersetzungsstrategie LRU wird diejenige Seite ersetzt, seit deren letztem Zugriff die geringste Zeit vergangen ist.
- Bei der Ersetzungsstrategie LFU ist der Zeitpunkt des letzten Zugriffes das ausschlaggebende Kritierum fuer die Ersetzung einer Seite.
- **Die Ersetzungsstrategie OPT ist in der Praxis nur schwer realisierbar, weil Wissen ueber das zukuenftige Verhalten des Gesamtsystems notwendig ist.**
- Die Ersetzungsstrategie FIFO eignet sich vor allem fuer Programme, die nichtsequentielle Speicherzugriffsmuster aufweisen.

h) Welche Aussage zu Prozesszustaenden ist richtig?

- **Ein Prozess kann nur durch seine eigene Aktivitaet vom Zustand laufend in den Zustand blockiert ueberfuehrt werden.**
- Ein Prozess, der sich im Zustand gestoppt befindet, kann sich selbst durch den Aufruf der Funktion fork() in den Zustand bereit ueberfuehren.
- Ein Prozess, der sich im Zustand laufend befindet, kann im Rahmen der mittelfristigen Einplanung in den Zustand schwebend-laufend ueberfuehrt werden.
- Ein Prozess kann sich nicht selbst in den Zustand beendet ueberfuehren.

i) Welche Aussage zu nicht-blockierender Synchronisation ist richtig?

- **Verfahren zur nicht-blockierenden Synchronisation greifen auf spezielle Prozessor-Instruktionen wie CAS zurueck und lassen sich daher nicht in reinem C99 implementieren.**
- Nicht-blockierende Synchronisationsverfahren setzen besondere Unterstuetzung durch das Betriebssystem voraus.
- Auch bei nicht-blockierender Synchronisation kann es zu Verklemmungen (Deadlocks) kommen.
- Nicht-blockierende Verfahren sind intransparent fuer den Scheduler und daher anfaellig fuer Prioritaetsverletzung und -umkehr.

j) In welcher der folgenden Situationen wird ein Prozess vom Zustand laufend in den Zustand bereit ueberfuehrt?

- Der Prozess greift lesend auf eine Datei zu und der entsprechende Datenblock ist noch nicht im Hauptspeicher vorhanden.
- **Der Scheduler bewirkt, dass der Prozess durch einen anderen Prozess verdraengt wird.**
- Der Prozess ruft eine P-Operation auf einen Semaphor auf, welcher den Wert 0 hat.
- Der Prozess ruft die Bibliotheksfunktion exit(3) auf.

k) Welche Aussage zum Aufbau einer Kommunikationsverbindung zwischen einem Client und Server ueber eine Socket-Schnittstelle ist richtig?

- Der Server erzeugt einen Socket und ruft anschliessend listen(2) auf - der Client kann daraufhin mit connect(2) eine Verbindung herstellen und sofort Daten uebertragen.
- Der Client kann erst connect(2) aufrufen, nachdem der Server accept(2) aufgerufen hat - vorher wuerde der Verbindungsversuch mit der Meldung connection rejectedäbgewiesen werden.
- Der Server signalisiert durch den Aufruf von connect(2), dass er zur Annahme von Verbindungen bereit ist; ein Client kann dies durch accept(2) annehmen.
- **Der Server richtet am Socket eine Warteschlange fuer ankommende Verbindungen ein und kann dann mit accept(2) eine konkrete Verbindung annehmen. accept(2) blockiert so lange die Warteschlange leer ist.**

8.2 Aufgabe 1.2 (8 Punkte)

a	b
25678	1245

a	b	c	d	e	f	g	h
50%	**100%**	0%	50%	**100%**	50%	50%	0%

a) Welche der folgenden Aussagen zu UNIX-Dateisystemen sind richtig?

- Zur Anzeige des Inhaltes einer Datei ist es notwendig, das Leserecht auf dem uebergeordneten Verzeichnis zu besitzen.
- **Fuer das Loeschen einer Datei sind die Rechte-Informationen im Dateikopf (Inode) der Datei irrelevant.**
- Nach dem Loeschen eines Dateikopfes (Inode) und der dazugehoerigen Datenbloecke ist es moeglich, dass weiterhin hard links auf den Inode verweisen.
- Ein Pfadname, der nicht mit einem '/'-Zeichen beginnt, wird relativ zum HomeVerzeichnis des Benutzers interpretiert.
- **Innerhalb eines UNIX-Dateisystembaumes koennen die Inhalte mehrerer Festplatten eingebunden sein.**
- **In einem Namensraum mit hierarchischer Struktur ist die Verwendung von gleichen Namen in unterschiedlichen Kontexten moeglich.**
- **Der Name einer Datei wird getrennt von ihrem Dateikopf (Inode) gespeichert.**
- **Auf jedes Verzeichnis verweisen immer mindestens zwei hard-links.**

b) Welche der folgenden Aussagen zum Thema Speicherverwaltung sind richtig?

- **Bei Segmentierung kann externe Fragmentierung des Arbeitsspeichers auftreten.**
- **Bei der Adressumsetzung in einem seitennummerierten Adressraum kann ein Zugriffsfehler auftreten, obwohl das present-bit im Seitendeskriptor gesetzt ist.**

- Bitkarten eignen sich zur Verwaltung von segmentierten Adressraeumen besser als zur Verwaltung von seitenbasierten Adressraeumen.
- **Bei der Platzierungsstrategie best-fit muss die Freispeicherliste bei einer Allokation gegebenenfalls zweimal durchlaufen werden.**
- **Bei der Platzierungsstrategie first-fit ist die Verschmelzung von freien Speicherbereichen einfacher als bei worst-fit.**
- Beim Buddy-Verfahren koennen zwei aneinandergrenzende Bloecke gleicher Groesse immer verschmolzen werdenr
- Der Systemaufruf free(2) sorgt dafuer, dass die angegebene Seite im Freiseitenpuffer landet.
- Bei der Auslagerung einer Seite ist keine Anpassung des TLBs erforderlich.

8.3 Aufgabe 2 - tesa (62 Punkte)

Schreiben Sie ein Programm **tesa** (Testsuite Shell Advanced), welches eine interaktive Shell zum Einlesen und Ausfuehren einer Testfolge (Testsuite) anbietet. Die unterstuetzten Kommandos lauten wie folgt:

- **readTestsuite**: Sucht im aktuellen Arbeitsverzeichnis nach Testfaellen und fuegt diese der Testfolge hinzu.
- **runTestsuite**: Fuehrt die vorher eingelesene Testfolge aus und gibt zu jedem Testfall eine kurze Zusammenfassung aus.

Das Programm soll folgendermassen strukturiert sein:

- Das Hauptprogramm liest wiederholt Zeilen von der Standardeingabe ein. Sie koennen davon ausgehen, dass keine der eingegebenen Zeilen mehr als 100 (Nutz-)Zeichen beeinhaltet;

  siehe **MAX_LINE_LEN**. Enthaelt die Zeile ein unbekanntes Kommando, so soll sie ignoriert werden und die Fehlermeldung **"Unknown command"** ausgegeben werden. Leere Zeilen werden ohne Fehlermeldung ignoriert. Ist die Eingabe ein unterstuetztes Kommando, so wird die entsprechende Funktion aufgerufen.

- Kommando **readTestsuite**, implementiert durch die Funktion **void readTestsuite(void)**: Die Funktion extrahiert aus dem aktuellen Arbeitsverzeichnis die Namen aller regulaeren Dateien, die ausfuehrbar (siehe Makro **S_ISEXEC**) sind, und traegt den Testfall (`struct testcase`) in die Verwaltungsstruktur der Testfolge (`struct testsuite`) ein. Sie koennen davon ausgehen, dass die Namen der Testfaelle kuerzer sind als 256 (=**MAX_PROGNAME**) Zeichen. Sollten mehr als 100 (=**MAX_TESTCASES**) Testfaelle im Verzeichnis vorhanden sein, werden die ueberzaehligen ohne Fehlermeldung ignoriert. Sollte bei der Bearbeitung in Fehler auftreten, ist eine aussagekraeftige Fehlermeldung auszugeben und der Verzeichniseintrag zu ignorieren.

- Kommando **runTestsuite**, implementiert durch die Funktion void **runTestsuite(void)**: Die Funktion startet alle Testfaelle der vorher eingelesenen Testfolge und wartet anschliessend passiv auf die Beendigung aller Testfaelle. Alle Zombies sollen sofort aufgesammelt werden. Durch Druecken von Strg-C (Signal **SIGINT**) kann der Benutzer die Ausfuehrung der Testfaelle vorzeitig abbrechen. Dies soll allerdings nicht zum Abburch der **tesa** fuehren. Nach Auftreten des Signals **SIGINT** haben die Testfaelle 3 (=**TESTCASE_TIMEOUT**) Sekunden Zeit, sich zu beenden. Testfaelle, die nach Ablauf dieser Frist noch nicht terminiert sind, sollen durch das Signal **SIGKILL** unverzueglich beendet werden. Zur Realisierung dieses Timeouts steht die Funktion **alarm(2)** zur Verfuegung, deren genaue Funktionsweise der beigefuegten Manual-Page zu entnehmen ist. Nachdem alle Testfaelle beendet wurden, wird zu jedem Testfall eine kurze Zusammenfassung abhaengig von der Art der Prozessbeendigung ausgegeben. Dies kann durch Aufruf der vorgegebenen Funktion **void printSummary(const struct testsuite *)** erfolgen.

- Die als implementiert anzunehmende Funktion **void printSummary(const struct testsuite[])** gibt fuer alle (Eintrag `numberOfTestcases` der `struct testsuite`) Testfaelle der Testfolge den Status (Eintrag status in der `struct testcase`) des Kindprozesses aus.

Tipp: Die Verwendung von Funktionen der dynamischen Speicherverwaltung ist bei dieser Aufgabe nicht notwendig. Auf den folgenden Seiten finden Sie ein Geruest fuer das beschriebene Programm. In den Kommentaren sind nur die wesentlichen Aufgaben der einzelnen zu ergaenzenden Programmteile beschrieben, um Ihnen eine gewisse Leitlinie zu geben. Es ist ueberall sehr grosszuegig Platz gelassen, damit Sie auch weitere notwendige Anweisungen entsprechend Ihrer Programmierung einfuegen koennen. Einige wichtige Manual-Seiten liegen bei - es kann aber durchaus sein, dass Sie bei Ihrer Loesung nicht alle diese Funktionen oder gegebenenfalls auch weitere Funktionen benoetigen.

8.3.1 tesa.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <signal.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>

// Konstanten
#define MAX_LINE_LEN 100+2
#define MAX_PROGNAME 256
#define MAX_TESTCASES 100
#define TESTCASE_TIMEOUT 3

// Struktur-Deklarationen
struct testcase {
  char progName[MAX_PROGNAME];
  int status;
  pid_t pid;
};

struct testsuite {
  int numberOfTestcases;
  struct testcase testcases[MAX_TESTCASES];
};

// Hilfsfunktion die
static int die(const char message[]) {
  perror(message);
  exit(EXIT_FAILURE);
}

// Prueft, ob Datei ausffuehrbar ist
#define S_ISEXEC(mode) (((S_IXUSR|S_IXGRP|S_IXOTH)&mode) > 0 )

// Hilfsfunktion zur Ausgabe der Testfaelle einer Testfolge
static void printSummary(const struct testsuite ts[]) {
  if(!ts) { printf("You failed! Testsuite must be nonnull :D\n"); return; }
  for(int i=0;i<ts->numberOfTestcases;i++) printf("Testcase [%d]: [%s] [%d] [%d]",i,ts->testcases[i].pr
}

// Globale Variablen, Funktionsdeklarationen usw.
static void readTestsuite(void);
static void runTestsuite(void);
static void sigchld(int signal);
static void sigint(int signal);
static void sigalarm(int signal);

static struct testsuite tests;
static volatile int curt = 0; // TODO LEARN VOLATILE!!

static char own[MAX_PROGNAME]; // TODO DELME

// Funktion main()
int main(int argc, char** argv) {
```

```
59    strcpy(own,argv[0]);
60    // Allgemeine Vorbereitungen
61    struct sigaction schild = {.sa_handler=sigchld, .sa_flags = SA_RESTART | SA_NOCLDWAIT | SA_NOCLDSTOP};
62    sigemptyset(&schild.sa_mask);
63    sigaction(SIGCHLD,&schild,NULL);
64
65    struct sigaction sint = {.sa_handler=sigint, .sa_flags = SA_RESTART};
66    sigemptyset(&sint.sa_mask);
67    sigaction(SIGINT,&sint,NULL);
68
69    struct sigaction salarm = {.sa_handler=sigalarm, .sa_flags = SA_RESTART};
70    sigemptyset(&salarm.sa_mask);
71    sigaction(SIGALRM,&salarm,NULL);
72
73    // Zeile einlesen und Kommando interpretieren
74    char buf[MAX_LINE_LEN];
75    while(fgets(buf,sizeof(buf),stdin)) {
76      if(!strcmp(buf,"\n")) continue;
77      if(!strcmp(buf,"readTestsuite\n")) { readTestsuite(); continue; }
78      if(!strcmp(buf,"runTestsuite\n")) { runTestsuite(); continue; }
79      fprintf(stderr,"Unknown command\n");
80    }
81    if(ferror(stdin)) die("fgets");
82    exit(EXIT_SUCCESS);
83  }
84  // Ende Funktion main()
85
86  // Funktion readTestsuite
87  static void readTestsuite() {
88    // Pruefen, ob Verzeichniseintrag ein Testfall ist
89    DIR* d=opendir(".");
90    if(!d) { perror("opendir"); return; }
91
92    struct dirent* e; tests.numberOfTestcases=0; struct stat buf;
93    while(errno=0,(e=readdir(d)) && tests.numberOfTestcases<MAX_TESTCASES) {
94      if(-1==lstat(e->d_name,&buf)) { perror("lstat"); continue; }
95      if(!S_ISREG(buf.st_mode) || !S_ISEXEC(buf.st_mode)) continue;
96      if(!strcmp(e->d_name,own)) continue; // TODO del me... no recursion please...
97      strcpy(tests.testcases[tests.numberOfTestcases++].progName,e->d_name); // TODO ++ :)
98    }
99    if(errno) perror("readdir");
100   if(closedir(d)) perror("closedir");
101 }
102 // Ende Funktion readTestsuite
103
104 // Funktion runTestsuite
105 static void runTestsuite() {
106   sigset_t old,block;
107   sigemptyset(&block); sigaddset(&block,SIGCHLD);
108
109   // Testfaelle starten
110   for(int i=0;i<tests.numberOfTestcases;i++) {
111     pid_t p = fork();
112     tests.testcases[i].pid=p;
113     if(p<0) { perror("fork"); continue; }
114     if(!p) {
115       execl(tests.testcases[i].progName,tests.testcases[i].progName,NULL);
116       die("execl");
117     }
118     sigprocmask(SIG_BLOCK,&block,&old); // TODO LEARN
119     curt++;
```

```
120     sigprocmask(SIG_SETMASK,&old,NULL);
121   }
122   // Auf die Beendigung der Testfaelle warten
123   sigprocmask(SIG_BLOCK,&block,&old);
124   while(curt>0) sigsuspend(&old);
125   sigprocmask(SIG_SETMASK,&old,NULL);
126 
127   // Zusammenfassung ausgeben
128   printSummary(&tests);
129 }
130 // Ende der Funktion runTestsuite
131 
132 // Signalbehandlung SIGCHLD
133 static void sigchld(int signal) {
134   int serrno=errno, status;
135   pid_t p;
136   while((p=waitpid(-1,&status,WNOHANG))>0) {
137     curt--;
138     for(int i=0;i<tests.numberOfTestcases;i++) {
139       if(p==tests.testcases[i].pid) {
140         tests.testcases[i].pid=p; break;
141       }
142     }
143   }
144 // TODO kann man das nicht aehnlich wie hier machen?
145 //  for(int i=0;i<tests.numberOfTestcases;i++) {
146 //    if(-1==waitpid(tests.testcases[i].pid,&tests.testcases[i].status,NULL)) perror("waitpid");
147 //    curt--;
148 //  }
149   errno=serrno;
150 }
151 
152 // Signalbehandlung SIGINT
153 static void sigint(int signal) { alarm(TESTCASE_TIMEOUT); }
154 
155 // Signalbehandlung SIGALRM
156 static void sigalarm(int signal) {
157   for(int i=0;i<tests.numberOfTestcases;i++) {
158     if(-1==kill(tests.testcases[i].pid,SIGKILL)) perror("kill");
159   }
160 }
```

Tim Nisslbeck (hu78sapy)
hu78sapy@stud.cs.fau.de — wwwcip.cs.fau.de/~hu78sapy

9 Februar 2014

9.1 Aufgabe 1.1 (20 Punkte)

9.2 Aufgabe 1.2 (8 Punkte)

9.3 Aufgabe 2 - mops (62 Punkte)

a) Schreiben Sie ein Programm mops (**M**ulti-**O**utput **P**rint **S**erver), das als Druck-Server eingehende Druckauftraege auf mehrere physische Drucker verteilt.

mops ermittelt beim Start die im System verfuegbaren Drucker und startet fuer jeden Drucker einen Arbeiter-Thread, der Auftraege auf diesem Drucker ausgibt. Der Haupt-Thread nimmt auf einen TCP/IPv6-Socket eingehende Verbindungen an und uebergibt diese ueber einen Ringbuffer, der Platz fuer **BUFFER_SIZE** (=64) Eintraege bietet, an die Arbeiter-Threads.

Das Programm soll folgendermassen arbeiten:

- Das Programm erhaelt als Parameter auf der Befehlszeile die Portnummer, auf der Verbindungen angenommen werden sollen.
- Zum Erzeugen der Arbeiter-Threads wird die Funktion

  unsigned int spawnThreads(void);

  aufgerufen. Diese sucht im Verzeichnis **/dev** nach Eintraegen vom Typ **Character Device**, deren Name mit der Zeichenkette "lp" beginnt (also beispielsweise **/dev/lp0** und **/dev/lp1**). Fuer jede gefundene Drucker-Geraetedatei erzeugt die Funktion einen Arbeiter-Thread mit der Startfunktion **feedPrinter()**. Die Funktion **spawnThreads()** gibt die Anzahl der gefundenen Drucker (=Anzahl der gestarteten Threads) zurueck.
- Nach dem Starten der Arbeiter-Threads gibt das Hauptprogramm auf der Standardausgabe die Anzahl der gefundenen Drucker im folgenden Format aus:

  2 printer devices found

  Falls im System kein Drucker gefunden werden konnte, wird nach Ausgabe dieser Meldung der Prozess beendet.

  Ansonsten erzeugt das Programm einen TCP/IPv6-Socket, auf der es Verbindungen entgegennimmt. Nach jeder Verbindungsannahme wird der dazugehoerige Socket-Deskriptor in den Ringbuffer eingetragen. Ist der Puffer bereits voll, soll mit dem EInfuegen solange gewartet werden, bis wieder ein freier Platz vorhanden ist.
- Jeder der Arbeiter-Threads startet in der Funktion

  void *feedPrinter(const char device[]);

  die als Parameter den Namen der Drucker-Geraetedatei erhaelt. Der Arbeiter-Thread oeffnet das Geraet zum Schreiben im **Append**-Modus (wie eine normale Datei). Anschliessend werden in einer Endlosschleife Druckauftraege abgearbeitet. Zur Annahme eines Auftrags wird ein Socket-Deskriptor aus dem Ringpuffer entnommen. Ist der Ringpuffer leer, blockiert die Funktion solange, bis ein Auftrag eintrifft.

  Alle Daten, die der Client ueber die Socket-Verbindung sendet, werden auf den Drucker ausgegeben. Danach wird die Verbindung beendet, womit die Bearbeitung des Druckauftrags abgeschlossen ist.

Synchronisieren Sie den Ringpuffer mit zaehlenden Semaphoren. Schnittstelle:

SEM* semCreate(int initValue);
void P(SEM* sem);
void V(SEM* sem);

Die Semaphor-Funktionalitaet ist in ein separates Modul **sem.c** ausgegliedert, das ebenfalls von Ihnen zu implementieren ist. Die Schnittstelle ist in der existierenden Datei **sem.h** deklartiert.

9.3.1 mops.c

9.3.2 sem.c

```c
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>

/** Opaque type of a semaphore. */
typedef struct SEM{
  volatile int a;
  pthread_mutex_t m;
  pthread_cond_t c;
} SEM;

SEM *semCreate(int initVal) {
  SEM* sem = (SEM*) malloc(sizeof(SEM));
  if(NULL==sem) return NULL;
  if((errno=pthread_mutex_init(&sem->m,NULL))) {
    free(sem);
    return NULL;
  }
  if((errno=pthread_cond_init(&sem->c,NULL))) {
    pthread_mutex_destroy(&sem->m);
    free(sem);
    return NULL;
  }
  sem->a = initVal;
  return sem;
}

void semDestroy(SEM *sem) {
  if((errno=pthread_mutex_destroy(&sem->m))) {
    pthread_cond_destroy(&sem->c);
    free(sem);
    return;
  }
  errno=pthread_cond_destroy(&sem->c);
  free(sem);
}

void P(SEM *sem) {
  pthread_mutex_lock(&sem->m);
  while(0==sem->a) pthread_cond_wait(&sem->c,&sem->m);
  --sem->a; pthread_mutex_unlock(&sem->m);
}

void V(SEM *sem) {
  pthread_mutex_lock(&sem->m);
  ++sem->a; pthread_cond_broadcast(&sem->c);
  pthread_mutex_unlock(&sem->m);
}
```