B-BKSPP-PR (WS 2013/2014)
Einführung in die Systemprogrammierung mit C
Teil 2: Ausgewählte Techniken der UNIX-Systemprogrammierung

15. Dezember 2013

Im letzten fest vorgegebenen Teil des Praktikums behandeln wir nun verschiedene Techniken der Systemprogrammierung. In Abschnitt 0 der Aufgabenstellung erhalten Sie dazu einige Hintergrundinformationen, die sie bei Bedarf verwenden können.

Beachten Sie zur Übersetzung: Einige der Operationen, die wir hier verwenden, sind als Teil der UNIX-Spezifikation POSIX spezifiziert, oder GNU- bzw. Linux-spezifische Erweiterungen. Zur Übersetzung sollten Sie daher `-std=gnu99` oder besser `-std=gnu11` angeben, um die betreffenden Bibliotheksoperationen zu aktivieren.

0.1 Speicher

In diesem Abschnitt behandeln wir einige relevante Aspekte des Arbeitsspeichers, insbesondere:

- Virtuelle Speicherverwaltung, die Speicher größer und flexibler gestaltet
- Inhalte des Speichers zur Laufzeit des Programmes
- Die Speicherhierarchie, und speziell Caching, das Speicherzugriffe beschleunigt

0.1.1 Virtuelle Speicherverwaltung

In Multiprozeßsystemen müssen mehrere Programme gleichzeitig arbeiten können, ohne sich gegenseitig durch Programmierfehler oder Absicht beschädigen oder ausspähen zu können. Dies erreicht man in modernen Rechnern durch eine Abstraktionsschicht, die Prozessen nur indirekten Speicherzugriff erlaubt. Diese Abstraktionsschicht nennt sich die *virtuelle Speicherverwaltung*.

Wenn ein Programm[1] auf eine Speicheradresse zugreift, dann ist diese Speicheradresse nicht die wirkliche Adresse in der betreffenden Speicherhardware; daher bezeichnen wir sie auch als *virtuelle Adresse*. Stattdessen wird die Adresse zunächst übersetzt:

$$\text{virtuelle Adresse} \xrightarrow{\textit{Adreßübersetzung}} \text{physikalische Adresse}$$

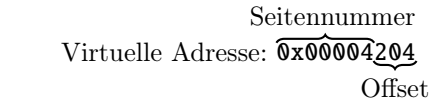
Diese Adreßübersetzung ist impliziter Teil jedes Speicherzugriffes. Mathematisch können wir sie als eine partielle Abbildung betrachten; manche Adressen sind also ungültig. Diese Abbildung muß nicht injektiv sein: Es ist durchaus möglich, daß verschiedene virtuelle Adressen die gleiche physikalische Adresse bezeichnen.

[1] Für den Betriebssystemkern gelten ggf. andere Regeln, je nach Prozessorarchitektur.

Die Seitentabelle. Die Implementierung dieser Adreßübersetzung liegt in den Händen der sogenannte *Speicherverwaltungseinheit* (*memory management unit*, MMU), die Adressen in der *Seitentabelle* (*page table*) nachschlägt. Jeder Prozeß (jedes laufende Programm) hat eine eigene Seitentabelle, die er auch zu eigenen Zwecken manipulieren kann, innerhalb eines engen, vom Betriebssystem erzwungenen Sicherheitsrahmens.

Die Seitentabelle merkt sich also, welche virtuellen Adressen auf welche physikalischen Adressen abgebildet werden. Sie merkt sich zusätzlich, welche Arten von Zugriffen erlaubt sind: Lesen, Schreiben, Ausführen, eine Kombination dieser Optionen, oder gar kein Zugriff. Wenn ein Programm einen ungültigen Zugriff ausführt, wird dieser dem Betriebssystem über eine Hardwareausnahme gemeldet. Das Betriebssystem kann dann entscheiden, wie es mit dieser Ausnahme umgehen will.

Seiten im Speicher. Der Begriff „Seitentabelle“ kommt daher, daß Speicher in modernen Systemen in sogenannten *Seiten* zusammengefaßt wird. Eine Seite im Speicher hat eine feste Größe, meist eine Zweierpotenz (4096 und 8192 Bytes sind typisch). Nehmen wir eine Seitengröße von 4096 ($= 2^{12}$) an, dann unterteilen sich Speicheradressen in *Seitenadressen* und *Offsets* wie folgt:



Die Adressen `0x4204` und `0x421c` hätten also die gleiche Seitennummer (`0x4`), aber unterschiedliche Offsets innerhalb der Seite.

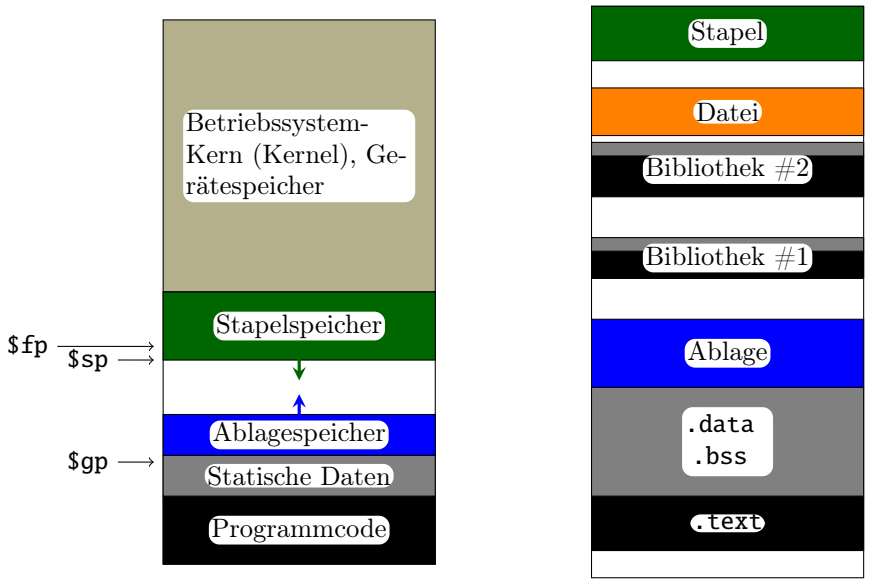
Um Speicherplatz zu sparen, merkt sich die Seitentabelle nur die Abbildung von virtuellen auf physikalische Seiten. Der Offset-Teil der Adresse wird unverändert übernommen. Daher kann die Seitentabelle z.B. bei 4096-Byte-Seiten verwendet werden, um die Inhalte der virtuellen Adressen `0x4204` und `0x5204` auszutauschen, aber nicht, um die Inhalte der virtuellen Adressen `0x4204` und `0x4205` auszutauschen.

Behandlung von ungültigen Speicherzugriffen. Die Seitentabelle wird meist primär vom Betriebssystem selbst genutzt, zu zwei hauptsächlichen Zwecken:

1. Trennung der Adreßbereiche unabhängiger Prozesse
2. Vergrößerung des verfügbaren Speichers

Virtueller Speicher erlaubt es Prozessen, mehr Speicher zu verwenden, als physikalisch im System vorhanden ist. Das Betriebssystem setzt dazu die Seitentabelle so auf, daß Seiten, die dem Programm „versprochen“ wurden, die aber noch nicht installiert sind, als ungültig markiert sind. Wenn nun ein ungültiger Speicherzugriff erfolgt, unterscheidet das Betriebssystem zwischen drei Fällen:

1. *Kleiner Seitenfehlzugriff* (*minor page fault*): Seite ist versprochen, aber noch nicht in der Seitentabelle. Dieser Fall tritt z.B. ein, wenn das Betriebssystem die Daten im Hintergrund schon vorgeladen hat. Hier muß meist nur die Seitentabelle aktualisiert
2. *Großer Seitenfehlzugriff* (*major page fault*) Seite ist versprochen, aber auf den Festspeicher ausgelagert. Dieser Fehlzugriff führt dazu, daß das ausführende Programm unterbrochen wird, bis die Daten vom Festspeicher nachgeladen werden konnten. Wenn das System zu wenig physikalischen Speicher für diese Inhalte hat, werden im gleichen Schritt andere Seiten aus dem Arbeitsspeicher in den Festspeicher ausgelagert (die dann wiederum einen großen Seitenfehlzugriff verursachen würden, wenn der Prozeß auf sie zugreifen würde).
3. *Segmentierungsverletzung* (*segmentation fault*, manchmal auch *bus error*): Das Programm hat versucht, auf eine Seite zuzugreifen, über die das Betriebssystem nichts weiß. Dies ist meist ein Programmierfehler, wird aber manchmal von Anwendungsprogrammen explizit genutzt, um Speicherzugriffe auf bestimmte Seiten zu erkennen und zu behandeln.



(a) MIPS32 Adreßraumkonvention

(b) Typische Adreßraumkonvention für C-Programme

Abbildung 1: Typische Adreßraumkonventionen.

Beachten Sie zur Adreßübersetzung auch Foliensatz 3, Folien 37–43 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-03.pdf`)

0.1.2 Der virtuelle Adreßraum in der Praxis

Wie beschrieben, weist das Betriebssystem jedem laufenden Benutzerprozeß die Bedeutung von Speicheradressen mit Hilfe eines Adreßübersetzungsverfahrens der Speicherverwaltungseinheit zu. Die dabei möglichen Speicheradressen variieren zwischen den verschiedenen Architekturen; so sind beispielsweise bei modernen 32-Bit-Architekturen meist alle 32-Bit-Zahlen gültige Adressen, aber die x86-64-Architektur erlaubt zur Zeit nur 2^{48} verschiedene Adressen.

Moderne Mehrbenutzerbetriebssysteme trennen meist die Adreßräume unterschiedlicher Nutzerprozesse vollständig (sofern nicht anders beantragt), und weisen den Prozessen dabei nur einen Bruchteil der von der Architektur erlaubten Adressen zu. Diese Trennung und räumliche Einschränkung bewirken, daß in drei Prozessen A, B, und C die gleiche Speicheradresse unterschiedliche Bedeutungen haben kann: In A könnte die Adresse eine globale Variable sein, in B der Körper einer ausführbaren Funktion, und in C könnte die Adresse ungültig sein.

Die Organisation und Vergabe des Speichers erfolgt anhand von architektur- und betriebssystemspezifischen Konventionen, die teilweise beliebig gewählt wurden, sich teilweise aber auch aus Eigenschaften der zugrundeliegenden Hardware ergeben[2]. Abbildung 1(a) zeigt als Beispiel die Adreßraumkonvention für C- und Assemblerprogramme auf der 32-Bit MIPS32-Architektur. Die obere Hälfte des Adreßraumes (ab Adresse `0x80000000`) ist für Betriebssystemprozesse reserviert; die betreffenden Adressen sind für Benutzerprozesse ungültig. Diese Konvention ergibt sich zwingend aus der Konstruktion des MIPS-Prozessors. Eine andere Konvention ist, daß der nutzbare Adreßraum ab Adresse `0x40000` beginnt und Adressen im Intervall `0x0`–`0x3ffff` als ungültig markiert sind. Diese Konvention ergibt sich aus pragmatischen Überlegungen: viele Programme verwenden Null-Zeiger (Zeiger auf die Speicheradresse `0x0`) um das Fehlen von Informationen zu signalisieren; diese Null-Konvention ist z.B.

[2]Natürlich sind diese Eigenschaften von den zuständigen Entwicklern oft auch wieder recht „beliebig“ gewählt worden.

auch durch verbreitete Bibliotheken etabliert. Wenn ein Programm nun versehentlich auf einen solchen Zeiger zugreift, wird dies sofort einen Speicherfehler auslösen, anstatt zu einer subtileren Speicherkorruption zu führen.

Abbildung 1(b) zeigt den typischen Adreßraum eines C-Programmes. Auf MIPS entspricht dies dem Adreßintervall `0x0`–`0x7fffffff`. Wir sehen unten einen reservierten, freigelassenen Bereich, der vom Betriebssystem nach Möglichkeit als ungültig markiert wird, gefolgt von einem `.text`-Block, der Programmcode beinhaltet, gefolgt von sogenannten statischen Daten (`.data` und `.bss`). Statische Daten existieren normalerweise nur ein Mal pro Programm und werden für globale Variablen verwendet. Diese Bereiche stellen zusammen den *statischen* Teil des Programmspeichers, dessen Größe schon vor Programmstart feststeht und sich nicht mehr ändert.

Unmittelbar auf diesen Block folgt der *dynamische* Speicher. Im Diagramm besteht dieser aus vier wesentlichen Teilen:

- *Ablagespeicher*, der per Konvention explizit allozierte Speicherblöcke enthält,
- *Stapel-* bzw. *Kellerspeicher*, der per Konvention zum Speichern von Zwischenzuständen bei Subroutinenaufrufen verwendet wird, und
- *Dynamischen Bibliotheken*, die zur Laufzeit des Programmes geladen wurden. Dynamische Bibliotheken verwenden den gleichen Ablage- und Kellerspeicher wie das Hauptprogramm; ihr statischer Speicher (globale Variablen und Code) werden üblicherweise in der Region zwischen Keller- und Stapelspeicher abgelegt,
- *Weiteren in den Speicher abgebildeten Daten*: Das Betriebssystem kann Daten, die es ursprünglich anderen Adreßräumen zugeordnet hat, im Bereich zwischen Ablage- und Stapelspeicher abbilden. Dies wird insbesondere oft für Dateiinhalte verwendet (per **mmap(2)**).

Wir beschränken unsere Betrachtung hier auf den Ablage- und Stapelspeicher. Im Diagramm ist der Ablagespeicher 'unten' im Adreßraum und wächst nach 'oben', wenn zusätzlicher Speicher benötigt wird, während der Stapelspeicher 'oben' liegt und bei Bedarf nach 'unten' wächst. Diese Konvention gilt für MIPS und x86, aber auf anderen Architekturen ist sie teilweise umgekehrt.

0.1.3 Speicherallozierung

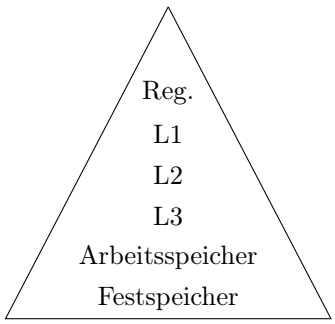
Wie wir gesehen haben, unterscheiden sich also die Orte, an denen die drei wesentlichen Speicherformen— also statischer Speicher, Stapelspeicher, und Ablagespeicher— alloziert werden, und es unterscheidet sich auch die Art und Weise, *wie* diese Speicherformen alloziert werden.

Statischer Speicher: Der statische Speicher ist schon beim Laden des Progammes bekannt. Die Programmdatei beinhaltet Kontrollinformationen, die angeben, wieviel Platz für statische Daten benötigt wird, und gegebenenfalls mit welchen Werten diese zu initialisieren sind. In C enspricht dies globalen (bzw. 'statischen') Variablen wie z.B. `static int c`.

Stapelspeicher: Der stapeldynamische Speicher (Stapelspeicher, Kellerspeicher) wird vom Betriebssystem zu Beginn des Programmes alloziert und auf eine bestimmte Maximalgröße festgesetzt. Ein *Stapelzeiger* merkt sich die untere Grenze[3] des Stapelspeichers.

Ablagespeicher: Der Ablagespeicher wird hauptsächlich vom C-Laufzeitsystem verwaltet (**malloc(3)**, **free(3)**). Das Betriebssystem ordnet dazu dem laufenden Programm einen bestimmten Speicherbereich zur 'beliebigen Verwendung' zu. Diesen Speicherbereich kann das C-Laufzeitsystem auf Anfrage (**brk(2)**, **sbrk(2)**) vergrößern. Dieser Schritt wird von **malloc** und ähnlichen Operationen automatisch ausgeführt, wenn der bisher verfügbare Ablagespeicher nicht ausreichen sollte.

[3] bzw. obere Grenze bei umgekehrter Organisation des dynamischen Teiles des Adreßraumes; wir verwenden in unserer Beschreibung die MIPS-Konventionen.



	Speicherplatz	Zugriffszeit
Reg.	≈ 512 B	sofort
L1	≈ 64 kiB	≈ 4 Zyklen
L2	≈ 256 kiB	≈ 10 Zyklen
L3	≈ 8 MiB	≈ 40–75 Zyklen
Arbeitsspeicher	≈ 128 GiB	≈ 200 Zyklen
Festspeicher	groß	> 120000 Zyklen

Abbildung 2: Die Speicherhierarchie. Zugriffszeiten typisch für Desktop-Prozessoren ca. 2013.

0.1.4 Die Speicherhierarchie

Speicher in modernen Systemen stellt sich Anwendungsprogrammen als eine Art Array dar, auf dessen Indizes (fast) beliebig zugegriffen werden kann. Diese Illusion bleibt so lange lebendig, bis wir die Zugriffszeiten messen: Tatsächlich sehen wir, daß das Lesen eines Wertes, den wir vor Kurzem schon einmal gelesen haben, meist deutlich schneller ist als das Lesen eines Wertes, den wir schon lange nicht mehr gelesen haben.

Ein laufendes Programm merkt sich die gerade verarbeitet werdenden Informationen in *Prozessorregistern*, die ohne Zeitverzögerung miteinander verknüpft werden können. Wenn es auf Informationen aus dem Speicher zugreifen muß, benötigt es evtl. mehrere Prozessortaktzyklen, bis es die betreffenden Daten gelesen hat. In einem modernen System kann solcher Arbeitsspeicherzugriff 200 oder mehr Prozessortaktzyklen verschlingen. Da Prozessorregister nur sehr wenig Platz stellen, sind Hauptspeicherzugriffe aber sehr oft nötig.

Um die Verzögerung durch Speicherzugriffe abzuschwächen, verwenden moderne Systeme sogenannte *Caches*: kleine, aber schnellere Zwischenspeicher. Diese merken sich Daten, die vor Kurzem gelesen oder geschrieben wurden und somit– heuristisch gesehen– wahrscheinlich bald wieder benötigt werden. Diese Caches geschickt zu füllen und zu leeren ist nichttrivial, liegt aber im Wesentlichen in den Händen der Rechnerhardware. In den vergangenen Jahren wurden wesentliche Performanzsteigerungen dadurch erzielt, daß Caches sich mehr Kontext merken, um diese Entscheidungen zu treffen, oder Werte vorladen, wenn sie bemerken, daß das laufende Programm einen Speicherblock in einem bestimmten Abstand durchschreitet.

Diese Caches bilden zusammen mit den anderen Speicherkomponenten die sogenannte *Speicherhierarchie* (Abbildung 2). Beachten Sie, daß die Mitglieder der Speicherhierarchie nach oben hin schneller und kleiner werden, dafür aber nach unten hin billiger.

In diesem Teil des Praktikums betrachten wir aus zeitlichen Gründen Caches nicht im Detail. Die obigen Informationen sind aber als Hintergrundwissen wichtig, insbesondere, wenn Sie in Ihrem Projekt an performanzkritischen Komponenten arbeiten sollten.

Um die Cachestruktur Ihres Systems zu untersuchen, können Sie (als Administrator) den Befehl `dmidecode -t cache` verwenden.

0.2 Programmausführung auf dem Stapel

Der Stapelspeicher erlaubt uns Funktionsaufrufe und Rekursion.

Er setzt sich aus mehreren *Ausführungsrahmen* zusammen, die jeweils einen Funktionsaufruf repräsentieren. Wenn eine Funktion einen Funktionsaufruf durchführt, wächst der Stapel nach unten. Wenn eine Funktion zurückkehrt, schrumpft der Stapel wieder nach oben.

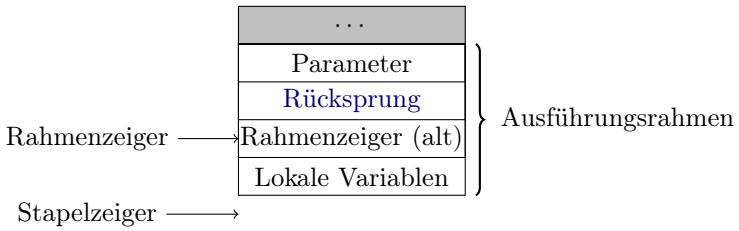


Abbildung 3: Ausführungsrahmen eines Programmes, auf dem Stapel

Ausführungsrahmen bestehen aus den in Abbildung 3 dargestellten Komponenten: Parametern, gefolgt von Verwaltungsinformationen, gefolgt von lokalen Variablen.

Die Bedeutung der vier Komponenten ist wie folgt:

- *Parameter* sind Speicherplätze, die für die Übergabe von tatsächlichen Funktionsparametern reserviert sind.
- *Rücksprung* ist die Speicheradresse eines ausführbaren Code-Stückes. Diese Rücksprungadresse ist die Programmstelle, von der aus die Funktion aufgerufen wurde. Beim Rücksprung (**`return`**) wird die Programmausführung an dieser Stelle wieder aufgenommen.
- *Rahmenzeiger (alt)* markiert die Position des vorherigen Ausführungsrahmens. Diese Information ist nötig, damit nach dem Rücksprung der Aufrufer wieder seinen eigenen Ausführungsrahmen finden kann.
- *Lokale Variablen* sind Speicherplätze, die für lokal allozierte Variablen reserviert sind (z.B. `y` und `tmp` im Beispiel unten).

Die genaue Struktur des Ausführungsrahmens (Größe und Typ der Parameter und lokalen Variablen) hängt dabei von der aufgerufenen Funktion ab.

Hier ist ein Beispiel eines verschachtelten Aufrufes:

```
int f(int x)
{
    int y = x + 2;
    // [A]
}

main(int argc, char **argv)
{
    int tmp = 0;
    return f(21);
}
```

`argc`
`argv`
Rücksprungadresse (Laufzeitumgebung)
Rahmenzeiger (Laufzeitumgebung)
`tmp = 0`
`x = 21`
Rücksprungadresse (`main`)
Rahmenzeiger (`main`)
`y = 23`

Wenn der Kontrollfluß des Programms die Position `[A]` erreicht, sieht der Ausführungsstapel ungefähr[4] wie die rechts angegebene Tabelle aus. Wir sehen zunächst die Parameter zu `main`, also `argc` und `argv`, dann die Rücksprungadresse und den Rahmenzeiger des C-Laufzeitsystems (das `main` aufruft). Es folgt die lokale Variable `tmp`. Diese fünf Komponenten bilden den Ausführungsrahmen von `main`.

Im Speicher folgt darauf unmittelbar der Parameter `x` def Funktion `f`, und der Rest des Ausführungsrahmens von `f`. Wenn `f` nun zurückkehrt, springt das Programm an die in der Rücksprungadresse gespeicherte Programmstelle und stellt den alten Ausführungsrahmen gemäß des gesicherten Rahmenzeigers wieder her.

[4] Die Details unterscheiden sich etwas zwischen Prozessorarchitekturen und Betriebssystemen.

0.3 Performanzzähler

Wie wir gesehen haben, kann die ungeschickte Nutzung des Arbeitsspeichers negative Konsequenzen auf die Laufzeit des Programmes haben. In den heutigen modernen Computersystem existieren aber noch zahlreiche weitere Effekte, die die Ausführung eines Programmes verlangsamen können.

Um feststellen zu können, ob diese Effekte sich auf ein bestimmtes Programm (oder einen bestimmten Teil eines Programmes) auswirken, wurden nach und nach in Betriebssysteme und Prozessoren sogenannte *Performanzzähler* integriert. Diese Zähler sind Register oder Speicherstellen, die sich merken, wie oft ein mit (hoher oder geringer) Performanz assoziiertes Ereignis stattgefunden hat.

Wir unterscheiden dabei zwischen *Software-Performanzzählern*, die vom Betriebssystem verwendet werden um z.B. die Anzahl der Seitenfehlzugriffe (*page faults*) zu zählen, und den *Hardware-Performanzzählern*, die auf dem Prozessor integriert sind und während der Ausführung eines Programmes nebenläufig die relevanten Effekte zählen.

0.3.1 Software-Performanzzähler

UNIX-Systeme haben eine einheitliche API zum Zugriff auf Performanzzähler über **getrusage(2)**.

```
#include <sys/time.h>
#include <sys/resource.h>

int getrusage(int who, struct rusage *usage);
```

Dabei ist `who` meist auf `RUSAGE_SELF` gesetzt (andere Werte können für spezifische Threads und die Kindprozesse verwendet werden). Der Zeiger `usage` verweist auf eine Struktur mit zahlreichen Feldern. Nicht jedes UNIX-System implementiert jedes Feld. Besonders interessante Felder sind folgende:

- `ru_minflt`: Kumulative Anzahl der 'kleinen' Seitenfehlzugriffe. Dies sind Seitenfehlzugriffe, die der Betriebssystemkern behandeln konnte, ohne auf den Festspeicher oder andere Peripheriegeräte (Netzwert etc.) zugreifen zu müssen.
- `ru_majflt`: Kumulative Anzahl der 'großen Seitenfehlzugriffe, die Ein-/Ausgabeoperationen nötig machten.
- `ru_nsignals`: Anzahl der empfangenen Signale (siehe auch Aufgabe 5)
- `ru_nivcsw`: Anzahl der unfreiwilligen Kontextwechsel, also die Anzahl der Male, die der Prozeß im Lauf durch den Betriebssystem-Scheduler unterbrochen wurde, um einem anderen Prozeß die Möglichkeit zu geben, zu arbeiten.
- `ru_nvcsw`: Anzahl der freiwilligen Kontextwechsel, z.B. durch `sched_yield(2)`.

Durch mehrere Aufrufe in Folge kann so überprüft werden, ob ein bestimmter Teil eines Programmes z.B. einen Seitenfehlzugriff ausgelöst hat.

0.3.2 Hardware-Performanzzähler

Hardwareperformanzzähler zählen als Implementierungsdetails von Prozessoren und unterscheiden sich daher sehr stark zwischen verschiedenen Prozessormodellen, selbst solchen, die gleiche oder ähnliche Befehlssätze unterstützen. Allerdings sind die Eigenschaften, die sie messen können, oft sehr ähnlich– so kann ein Prozessor z.B. die Gesamtzahl Cache-Lesefehlzugriffe messen, während ein anderer Prozessor diese nicht messen kann, dafür aber die Gesamtzahl der Cache-Fehlzugriffe und die Anzahl der Schreib-Fehlzugriffe, aus denen man die Lese-Fehlzugriffe berechnen kann.

Um den Zugriff auf diese Performanzzählregister zu vereinfachen, existiert eine Bibliothek namens PAPI, die 'Performance API', die auf zahlreichen, aber nicht allen UNIX-Systemen verfügbar ist[5].

[5]Da korrekte Unterstützung der Performanzzähler Unterstützung durch den Betriebssystemkern voraussetzt, kann PAPI in proprietären Betriebssystemen nur mit Unterstützung des Betriebssystemherstellers nachgerüstet werden.

Zum Einbinden können Sie den folgenden Header verwenden:

```
#include<papi.h>
```

Die notwendigen Bibliotheken werden mit `-lpapi` eingebunden.

PAPI bietet zwei APIs an: eine *high-level* API, die die meisten Gebrauchsfälle abdeckt, und eine *low-level* API, die prozessorspezifische Sonderfälle abdeckt. Beiden ist gemein, daß sie immer nur eine kleine Zahl von interessanten Kennwerten gleichzeitig messen können.

Der Grund dafür ist, daß die Prozessoren selbst meist eine beschränkte Anzahl von Zählerregistern (manchmal 4 oder auch nur 2) verfügen, die jeweils für verschiedene Ereignisse konfiguriert werden müssen. Folgendes Programm konfiguriert beispielsweise PAPI, um die Anzahl der Level-1, Level-2 und Level-3-Daten-Cache-Fehlzugriffe zu zählen:

```
// PAPI High-level API
const int PAPI_EVNR = 3;

int papi_events[PAPI_EVNR] = {
  PAPI_L1_DCM, // Level-1 Daten-Cache-Fehlzugriff
  PAPI_L2_DCM, // Level-2 Daten-Cache-Fehlzugriff
  PAPI_L3_DCM  // Level-3 Daten-Cache-Fehlzugriff
};

long long papi_results[PAPI_EVNR];

if (PAPI_start_counters(papi_events, PAPI_EVNR) != PAPI_OK) error();

run_benchmark(); // Zu messende Berechnung

if (PAPI_stop_counters(papi_results, PAPI_EVNR) != PAPI_OK) error();
```

PAPI-Operationen liefern `PAPI_OK` zurück, wenn sie erfolgreich waren, ansonsten kann nicht von einer erfolgreichen Messung ausgegangen werden. Wenn der Prozessor nur zwei Zählerregister hat oder die Messungen aus anderen Gründen nicht möglich sind (wenn der Prozessor z.B. keinen Level-3 cache verwendet oder gleichzeitige Messung von L2 und L1 nicht möglich ist), schlägt einer der Aufrufe entsprechend fehl.

Die relevanten PAPI-Schnittstellenoperationen sind wie folgt:

```
int PAPI_start_counters(int *zaehler, int zaehler_nr);
int PAPI_stop_counters(long long *ergebnisse, int zaehler_nr);
```

Die `zaehler` sind dabei ein Array mit mindestens `zaehler_nr` Einträgen, die angeben, welche Zähler beantragt werden. Die `ergebnisse` haben eine entsprechende Anzahl von Einträgen und werden mit dem Ergebnis der Messung zwischen `PAPI_start_counters` und `PAPI_stop_counters` beschrieben. Das Array von PAPI-Ereignissen kann verschiedene Eigenschaften messen, aber nicht alle sind auf allen Prozessoren vefügbar. Abbildung 4 führt einige der interessanteren Konstanten auf.

Eine der wichtigsten daraus Metriken sind die „Instruktionen pro Prozessorzyklus“ (IPC): dieser Kennwert wird aus `PAPI_TOT_INS` und `PAPI_TOT_CYC` berechnet und gibt an, wie gut die Recheneinheiten des Prozessors ausgelastet sind. Ein ‘guter’ Wert ist 1.0, aber in spezialisiertem Code können von superskalaren Prozessoren teilweise deutlich höhere Werte erreicht werden, wenn die separaten Ausführungseinheiten des Prozessor tatsächlich separat voneinander arbeiten können.

Beachten Sie, daß die Messungen nicht auf allen Prozessoren völlig exakt sein müssen, da die zugrundeliegenden Register nicht mit dem gleichen Maß an Genauigkeit wie der Rest des Prozessors getestet werden.

PAPI_TOT_INS	Gesamtzahl der ausgeführten Maschinensprachebefehle
PAPI_TOT_CYC	Gesamtzahl der abgeschlossenen Prozessorzyklen
API_L1_DCM	Gesamtzahl der Daten-Cache-Fehlzugriffe für den L1-Cache (analog für L2, L3)
API_L1_DCH	Gesamtzahl der Daten-Cache-Treffer für den L1-Cache (analog für L2, L3)
API_L1_DCR	Gesamtzahl der Daten-Cache-Lesezugriffe für den L1-Cache (analog für L2, L3)
API_L1_DCW	Gesamtzahl der Daten-Cache-Schreibzugriffe für den L1-Cache (analog für L2, L3)
API_L1_DCA	Gesamtzahl der Daten-Cache-Zugriffe (insgesamt) für den L1-Cache (analog für L2, L3)
API_L1_ICM	Gesamtzahl der Instruktions-Cache-Fehlzugriffe für den L1-Cache (analog für L2, L3 und für ICR, ICW, ICA, ICH)
PAPI_BR_CN	Anzahl der ausgeführten bedingten Sprünge
PAPI_BR_TKN	Anzahl der bedingten Sprünge, die genommen wurden
PAPI_BR_NTK	Anzahl der bedingten Sprünge, die nicht genommen wurden
PAPI_BR_MSP	Anzahl der falsch vorhergesagten bedingten Sprünge
PAPI_BR_PRC	Anzahl der korrekt vorhergesagten bedingten Sprünge
PAPI_INT_INS	Anzahl der ausgeführten Ganzzahl-Operationen
PAPI_FP_INS	Anzahl der ausgeführten Fließßkomma-Operationen
PAPI_LD_INS	Anzahl der ausgeführten Lade-Operationen
PAPI_SR_INS	Anzahl der ausgeführten Schreib-Operationen
PAPI_VEC_INS	Anzahl der ausgeführten Vektor-Operationen

Abbildung 4: Wichtige PAPI-Konstanten zur Konfiguation des Meßsystems. Beachten Sie, das nicht alle dieser Optionen auf allen Prozessoren unterstützt werden.

0.3.3 Kommandozeilenwerkzeug

Das Programm **perf(1)** bietet eine integriertes Interface, um die Performanzeigenschaften eines kompletten Programmdurchlaufes oder des Gesamtsystems zu untersuchen. Wir werden es hier allerdings nicht weiter betrachten.

0.4 Punkte

In diesem Teil des Praktikums sind 11,5 Punkte und 10 Bonuspunkte erreichbar; Notation (x + yB), wobei x Punkte und y Bonuspunkte sind. Dabei gilt:

- Der Praktikumsteil gilt als bestanden, wenn die Summe der Punkte und Bonuspunkte insgesamt mindestens 9,5 erreicht.
- Die Anzahl der Bonuspunkte für eine Aufgabe ist begrenzt durch die Anzahl der Punkte für die gleiche Aufgabe. Eine Aufgabe mit (**2** + 2B) Punkten, deren Bonusaufgaben komplett korrekt bearbeitet sind, deren Hauptaufgaben aber fehlen, wird somit mit 0 Punkten angerechnet.

  Ausgenommen davon sind Aufgaben 4 und 12, die reine Bonusaufgaben sind.

Beachten Sie: in diesem Teil *alle* Speicherfehler zu Punktabzug führen können, sofern sie nicht gezielt von der Aufgabenstellung provoziert wurden. Verwenden Sie `valgrind`, um nach solchen Fehlern zu suchen; wenn Ihr Programm gemäß `valgrind` keine Speicherfehler hat, führen eventuell verbleibende Fehler nicht zu Punktabzug.

0.5 Werkzeuge

In Teil 1b haben Sie mehrere Werkzeuge kennengelernt. Verwenden Sie diese Werkzeuge nach Bedarf auch für diesen Teil. Sie haben Zugriff auf folgende **git(1)**-repositories:

`git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/2-${n}`

wobei `${n}` die Nummer der betreffenden Aufgabe ist.

...

0.6 Einsendung

Das Einsendeformat entspricht dem für Teil 1b. Zur (alternativen) Abgabe per `git` ist folgendes Repository eingerichtet:

`git@web.sepl.cs.uni-frankfurt.de:${GRUPPENNAME}/abgabe-2`

1 Prozesse ausführen (1 + 1B)

UNIX-Prozesse werden über den Systemaufruf **execve(1)** gestartet. Dabei ersetzt **execve(1)** den Inhalt des aktuell laufenden Prozesses durch einen anderen Programmaufruf. Die Prozessnummer bleibt dabei erhalten, aber die Inhalte von Ablagespeicher, Stapel etc. werden nach Maßgaben des neuen Programmaufrufes ersetzt.

Lesen Sie bei Bedarf zunächst folgende Abhandlung über Prozesse:
`http://openbook.galileocomputing.de/linux_unix_programmierung/Kap07-000.htm`
Insbesondere die folgenden Kapitel:

- Kapitel 7.1
- Kapitel 7.2.1
- Kapitel 7.3
- Kapitel 7.4
- Kapitel 7.10 bis einschließlich 7.10.2

Aufgaben.

1. Schreiben Sie einen universellen Dateibetrachter. Ihr Programm soll genau einen Parameter nehmen und anhand der Dateiendung entscheiden, wie das Programm zu betrachten ist:

   - `.png`, `.jpeg`, `.jpg`: Per **display(1)**
   - `.txt`: Per **less(1)**

   Verwenden Sie **execve(1)**, um das betreffende Programm aufzurufen; Sie können dazu auch Kapitel 7.10 lesen.

   Sie können vereinfachend davon ausgehen, daß die verwendeten Helferprogramme in bestimmten Verzeichnissen liegen, die denen Ihres Rechners entsprechen. Sie können **which(1)** verwenden, um zu einem Programm aus Ihrem Aufrufpfad die vollständige Position im Dateisystem zu bestimmen.

   execve verwendet einen Parameter, um *Umgebungsvariablen* zu übernehmen. Diese merken sich bestimmte Informationen über den Benutzer und das verwendete Graphik- und Textinterface. Wir betrachten diese hier nicht im Detail; Sie können für diesen Parameter einfach die globale Variable **environ(7)** einsetzen, die die Umgebungsvariablen Ihres eigenen Programmes enthält.

(Bonus) (Nach Aufgabe 3) Erweitern Sie Ihr Programm so, daß es das Programm **file(1)** verwendet, um die Datei zu identifizieren. Unterstützen Sie dazu auch die Ausgabe von mindestens folgendem:

- Textdateien, auch wenn sie nicht per Dateiendung gekennzeichnet sind
- Objektdateien (also z.B. `.o` und `.so`-Dateien, wie wir sie mit `gcc` und ähnlichen Werzeugen erzeugen), dargestellt mit den Werkzeugen aus Teil 1b

2 Ein- und Ausgabe (1)

Jeder UNIX-Prozeß erhält zu Beginn einen Eingabekanal, **stdin(3)**, und zwei Ausgabekanäle, **stdout(3)** und **stderr(3)**. Auf diese Kanäle kann man auf zwei Arten und Weisen zugreifen: mit einer Schnittstelle auf höherer Ebene, die im C-Sprachstandard festgelegt ist, und einer Schnittstelle auf niedriger Ebene, die Teil der UNIX-Spezifikation POSIX ist.

Lesen Sie dazu zunächst Kapitel 2.1 und 2.2 (2.2.1ff sind nicht nötig) des Buchs:
`http://openbook.galileocomputing.de/linux_unix_programmierung/Kap02-000.htm`

Kapitel 2.3 und 2.4 beschreiben Ein- und Ausgabefunktionen für die beiden API-Ebenen. Wir verwenden in dieser und den folgenden Aufgaben beide Ebenen. Für diese Aufgabe ist die höhere Ebene (Kapitel 2.4) ausreichend, insbesondere durch die Operationen **fgets(3)** und **fputs(3)**.

Aufgaben.

1. Schreiben Sie ein Programm, das ISO-8859-1-Umlaute aus der Standardeingabe entfernt und durch 'ae', 'oe' etc. ersetzt. Testen Sie das Programm mit Büchern aus `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/buecher.tar.gz`. Das Programm soll seine Ausgabe nach **stdout(3)** schreiben. Allozieren Sie zur Bearbeitung insgesamt nur eine geringe Menge (4–5 kiB) an Ablage– und Stapelspeicher.

   Wenn das Programm mit Kommandozeilenparametern aufgerufen wird, soll es eine Fehlermeldung nach **stderr(3)** schreiben. Um eine Datei `buch` nach **stdin(3)** zu schicken, können Sie auf der Kommandozeile z.B.

   ```
   cat buch | programm
   ```

   oder

   ```
   programm <buch
   ```

   verwenden.

2. Schreiben Sie ein Programm, das ASCII-Großbuchstaben in Kleinbuchstaben umwandelt. Wie im vorherigen Programm soll es von **stdin(3)** lesen, normale Ausgabe nach **stdout(3)** schreiben, 4–5 kiB Zwischenspeicher verwenden, und angegebene Kommandozeilenparameter in **stderr(3)** monieren.

3. Verwenden Sie die Ausgabeweiterleitung ('pipe', `|`) der Shell, um die beiden Programme zu kombinieren. Sie können die Ausgabe eines Programmes oder Programmteiles `a` nach `b` senden, indem Sie folgendes schreiben:

   ```
   a | b
   ```

4. Untersuchen Sie: leitet die Ausgabeweiterleitung auch **stderr** um?

5. Untersuchen Sie: wird die Ausgabe ihres ersten Programmes vollständig berechnet, bevor sie weitergeleitet wird, oder wird inkrementell weitergeleitet?

3 Prozesse und Datenleitungen (1 + $\frac{1}{2}$B)

Die Ausgabeumleitung aus dem vorherigen Beispiel kann natürlich auch ohne Hilfe der Shell erreicht werden. Dazu müssen wir allerdings selbst die relevanten Programme in separate Umgebungen starten und ihre Eingabe und Ausgabe miteinander verbinden. Ersteres erreichen wir durch die Prozeß-API des Betriebssystems, letzteres durch Interprozeßkommunikation (IPC)– hier verwenden wir die wahrscheinlich einfachste Form der Interprozeßkommunikation, nämlich *Datenleitungen* (*pipes*).

Falls Ihnen diese Konzepte nicht geläufig sind, finden sie eine Einführung in die relevanten Punkte in `http://openbook.galileocomputing.de/linux_unix_programmierung/Kap07-000.htm`, und `http://openbook.galileocomputing.de/linux_unix_programmierung/Kap09-000.htm`, in den folgenden Kapiteln:

- Kapitel 7.8
- Kapitel 7.9
- Kapitel 9.3 bis mindstens 9.3.2

Sie benötigen also insbesondere **fork(2)** und **pipe(2)**. Die allgemeinen Konzepte zu Datenleitungen werden auch auf **pipe(7)** erläutert.

Aufgaben.

1. Installieren Sie das Programm `html2text`, falls nicht auf Ihrem System installiert, aus der folgenden Quelle:

   `http://www.mbayer.de/html2text/downloads/`

   Das Programm wandelt HTML-Dateien in einfache Textdateien um. Ähnlich wie die meisten Kommandozeilenprogramme liest es (wenn nicht anders angegeben) von der Standardeingabe und schreibt in die Standardausgabe.

2. Schreiben Sie ein Programm, das einen Kindprozeß erzeugt, der das Programm **wget(1)** aufruft, um Daten von einer als Parameter angegebenen URL zu lesen. Unterdrücken Sie bei Ihrem Aufruf die Zustandsausgabe von **wget** (Parameter `-q`), und erzwingen Sie, daß die Ausgabe nach **stdout** geschrieben wird (Parameter `-O -`).

3. Verwenden Sie **dup2(2)**, um die Ausgabe von **wget** in die Eingabe von `html2text` umzuleiten. Ihr fertiges Programm sollte eine Webseite von einer gegebene URL in reinen Text konvertiert nach **stdout** schreiben.

(Bonus) Erlauben Sie die Weitergabe von zusätzlichen Kommandozeilenparametern an die beiden Prozesse. Denken Sie sich dazu ein geeignetes Schema aus, um die Parameter an den geeigneten Prozeß zu leiten.

4 Kommunikation mit sockets: Ein kleiner Chat-Server (4B)

Wir haben nun gesehen, wie Datenleitungen zur Kommunikation zwischen Prozessen verwendet werden können. Dieser mächtige Mechanismus erreicht jedoch seine Grenzen, wenn die beteiligten Prozesse keine Möglichkeit haben, Dateideskriptoren per **pipe** zu erzeugen und auszutauschen. Das ist der Fall, wenn die Prozesse keinen Adreßraum teilen und dazu auch nicht gezwungen werden können (z.B. weil ein Prozeß dauerhaft als „Serverprozeß“ läuft oder die Prozesse auf getrennten Rechnern arbeiten.)

Wir wollen nun einen Chatserver bauen. Zur Vereinfachung bauen wir diesen nur lokal, also ohne Internetzugang. Dabei sollen mehrere Personen, die (z.B. per **ssh(1)**) auf dem gleichen Rechner eingeloggt sind, miteinander kommunizieren können.

In diesem Fall verwenden wir zur Kommunikation sogenannte *sockets*. Ein *socket* (Steckdose) ist Start- und Endpunkt für Kommunikation zwischen unabhängigen Prozessen. Sockets können *benannt* werden, so daß andere Prozesse sie anhand eines Namens finden können. Sie können dann mit anderen Sockets verbunden werden und Daten austauschen.

4.1 Fehlerbehandlung

Beim Erzeugen und Verwenden von Sockets kann es schnell zu Fehlern kommen. Wenn eine der Socket-Operationen einen Fehlercode zurückliefert, können wir uns mit der Funktion

```
void perror(const char *s);
```

eine Fehlerbeschreibung ausgeben lassen; die Zeichenkette `s` wird dabei als Präfix vor der Fehlerbeschreibung ausgedrukt. Stellen Sie im Folgenden sicher, daß Sie alle Rückgabewerte der Systemaufrufe auf Fehler überprüfen!

4.2 Socket-Erzeugung

Ein Socket kann mit dem Befehl **socket(2)** erzeugt weden. Das Betriebssystem unterstützt Sockets in verschiedenen *Domänen*; die wichtigsten davon sind `AF_INET` und `AF_INET6` (für das Inernet-Protokoll IP) sowie `AF_UNIX` für lokale Kommunikation auf UNIX-Rechnern (ohne Netzwerk). Internet-Sockets haben eine Socketnummer (*port*), die explizit beantragt oder vom Betriebssystem zugewiesen werden kann. Der Name des Sockets ergibt sich dann aus der IP-Adresse des Rechners zusammen mit dem gewählten Port.

UNIX-Sockets werden stattdessen durch Namen im Dateisystem bezeichnet. Wir verwenden hier `AF_UNIX` mit dem Namen `/tmp/server`[6].

Beim Aufruf von **socket** zum Erzeugen eines Sockets wird die Domäne als erster Parameter angegeben:

```
int socket(int domain, int type, int protocol);
```

`protocol` ist üblicherweise 0, und `type` gibt an, welche Art von Verbindung in dieser Domäne hergestellt werden soll. Die beiden wichtigsten Optionen sind `SOCK_STREAM` und `SOCK_DGRAM`:

- `SOCK_DGRAM` sendet Nachrichten in beliebiger Reihenfolge. Wenn Sendungen verlorengehen, wird dies nicht gemeldet. Für IP wird dies durch das UDP-Protokoll implementiert.
- `SOCK_STREAM` sendet Nachrichten in garantierter Reihenfolge und meldet Fehler, wenn die Nachrichten nicht angekommen sind. Für IP wird dies durch das TCP-Protokoll implementiert.

In den meisten Fällen ist `SOCK_DGRAM` einfacher zu verwenden[7].

Falls die Erzeugung des Sockets erfolgreich war, liefert **socket** einen Dateideskriptor größer/gleich 0 zurück.

[6] Ihnen ist freigestellt, diesen Namen zu ändern, aber er wird überall in der Aufgabenstellung verwendet

[7] `SOCK_DGRAM` findet seine Anwendung, wenn eine schnelle Übertragung wichtiger ist als eine fehlerfreie Übertragung.

4.3 Server: Socket-Benennung

Ein Socket selbst ist zunächst ein rein prozeßlokales Konzept. Um den Socket nach außen sichtbar zu machen, verwenden wir den Befehl **bind(2)**, der dem Socket einen globalen Namen verleiht. **bind** hat folgenden Prototypen:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

1. `sockfd` ist der zu benennende Socket.
2. `addr` zeigt auf eine Struktur, die den Namen des Sockets kodiert. Die genaue Struktur variiert je nach verwendeter Domäne. Wie in **unix(7)** erklärt, wird bei `AF_UNIX` folgende Struktur verwendet:

   ```
   #include <sys/socket.h>
   #include <sys/un.h>

   struct sockaddr_un {
       sa_family_t sun_family;               /* AF_UNIX */
       char        sun_path[UNIX_PATH_MAX];  /* pathname */
   };
   ```

   Dabei muß `sun_family` immer `AF_UNIX` sein, und `sun_path` enthält eine Zeichenkette mit einem Dateinamen, der dem Socket im Dateisystem zugewiesen wird (z.B. `/tmp/server`).
3. `addrlen` ist die Größe des Speicherbereiches von `addr`. Da verschiedene Socket-Arten verschiedene Socket-Strukturen verwenden, kann diese Größe variieren. Wir könnnen wie üblich **`sizeof`** verwenden, um die Größe der Struktur zu berechnen.

Im Erfolgsfall liefert die Funktion 0; andere Werte signalisieren einen Fehler.

4.4 Server: Socket zum Horchen konfigurieren

Um unseren Socket serverseitig verwenden zu können, müssen wir ihn noch mit dem Aufruf **listen(2)** als einen Server-Socket konfigurieren:

```
int listen(int sockfd, int backlog);
```

Der Parameter `backlog` gibt dabei an, wieviel Platz das Betriebssystem als Puffer für eingehende Verbindungsanfragen reservieren soll, also wieviele eingehende Verbindungen in eine Warteschleife gesteckt werden können, während der Server arbeitet. Desto größer die Zahl, desto mehr Speicher wird benötigt. Kleine Zahlen sorgen hingegen dafür, daß Verbindungsgesuche abgelehnt werden können, wenn der Server gerade stark belastet ist.

Im Erfolgsfall liefert die Funktion 0; andere Werte signalisieren Fehler.

4.5 Server: Warten auf Verbindungen

Um nun auf eingehende Verbindungen zu warten, müssen wir warten, bis es von unserem Socket etwas zu lesen gibt. Dazu verwenden wir die Funktion **select(2)** mit ihren Helferfunktionen:

```
void FD_ZERO(fd_set *set);

void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);

int  FD_ISSET(int fd, fd_set *set);

int select(int nfds, fd_set *readfds, fd_set *writefds,
```

```
            fd_set *exceptfds, struct timeval *timeout);
```

select ist eine Funktion, die auf mehrere Dateideskriptoren gleichzeitig warten kann. Das heißt, **select** nimmt eine Menge von Dateideskriptoren und unterbricht die Ausführungen des aktuellen Prozesses so lange, bis mindestens einer der Dateideskriptoren gelesen ode beschrieben werden kann, oder einen Ausnahmezustand angenommen hat. Um diese drei Fälle zu unterscheiden, werden **select** drei verschiedene Dateideskriptormengen übergeben.

Eine Dateideskriptormenge hat den Typ `fd_set`. Wir operieren auf diesen Typen mit den vier Helferfunktionen:

- `FD_ZERO(fd_set *set)` setzt `*set` auf die leere Menge.
- `FD_SET(int fd, fd_set *set)` fügt `fd` `*set` hinzu.
- `FD_CLR(int fd, fd_set *set)` entfernt `fd` aus `*set`.
- `FD_ISSET(int fd, fd_set *set)` überprüft, ob `fd` in `*set` ist.

Wir werden in unserem Server nur auf die Lesbarkeit von Dateideskriptoren warten. Wenn wir passende Deskriptormengen erzeugt haben, können wir **select(2)** aufrufen:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Die Parameter sind wie folgt:

1. `ndfs` ist der höchste Dateideskriptor aus den drei folgenden Menge, plus eins.
2. `readfds` ist die Menge aller Dateideskriptoren, auf deren Lesebereitschaft wir warten.
3. `writefds` ist die Menge aller Dateideskriptoren, auf deren Schreibbereitschaft wir warten.
4. `exceptfds` ist die Menge aller Dateideskriptoren, auf deren Ausnahmezustände wir warten.
5. `timeout` zeigt auf ein `struct timeval` (die gleiche Struktur, die auch von **gettimeofday(2)** verwendet wird), mit dem man angeben kann, daß **select** nach einem bestimmten Zeitraum zu warten aufhören soll. Wir können stattdessen auch `NULL` angeben, um unendliches Warten zu konfigurieren.

   select verringert `timeout` um die verstrichene Zeit, bevor es zurückkehrt.

Die Rückgabe von **select** ist die Anzahl der Dateideskriptoren, die lesbar/schreibbar/in Ausnahmezustand sind. Die genauen betroffenen Deskriptoren werden in `readfds` `writefds` und `exceptfds` gesetzt und können mit `FD_ISSET` geprüft werden. `select` kann aber auch einen negativen Wert liefern, um einen Fehler oder eine Unterbrechung anzuzeigen, wie z.B. `EINTR`, falls ein Signal empfangen wurde (siehe auch Aufgabe 5). Bei einem Timeout wird der Wert 0 zurückgeliefert.

In einigen Fällen ist auch die allgemeinere Funktion **pselect(2)** nützlich.

4.6 Klient: Verbinden mit dem Server

Wir wissen nun genug, um den Server dazu zu bringen, auf Klienten zu warten. Um den Klienten allerdings dazu zu bringen, den Server zu kontaktieren, müssen wir Sockets etwas anders verwenden. Wir erzeugen dazu (wie zuvor) einen Socket per **socket(2)** und rufen dann **connect(2)** auf:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Die Parameter entsprechen denen von **bind(2)** (Abschnitt 4.3).

connect wartet, bis der Server die Verbindung akzeptiert hat, und liefert bei Erfolg 0 zurück. Aus Sicht des Klienten können wir nun an den Socket schreiben und von ihm lesen.

4.7 Server: Akzeptieren einer Verbindung

Auf der Server-Seite müssen wir also nun eingehende Verbindungen akzeptieren, sobald **select** die Kontrolle an das Programm zurückgibt und signalisiert, daß eine eingehende Verbindung vorliegt. Dazu erzeugen wir aus dem Socket, den wir auf die Adresse `/tmp/server` gebunden haben, einen neuen Socket, der uns die Kommunkation mit dem Klienten erlaubt. Die dafür zuständige Funktion ist **accept(2)**:

```
int accept(int sockfd, const struct sockaddr *addr, socklen_t *addrlen);
```

Die Parameter entsprechen wieder denen von **bind(2)** (Abschnitt 4.3). `addr` und `addrlen` beziehen sich aber hier auf die Verbindungsdaten des Klienten, die in lokale Strukturen eingetragen werden. Auf diese Weise kann man bei einer `AF_INET`-Verbindung zum Beispiel die IP-Adresse des Klienten bestimmen. Wenn der Server sich nicht für diese Details interessiert, können die beiden Zeiger auf NULL gesetzt werden.

Im Erfolgsfall liefert **accept(2)** eine nichtnegative Zahl, die ein Socket-Dateideskriptor ist, von dem wir lesen und auf den wir schreiben können.

Auf der Server-Seite erzeugen wir für jede eingehende Verbindungsanfrage einen solchen Socket per **accept(2)**. Jeder derart akzeptierte Socket ist mit dem korrespondierenden Socket des Klienten direkt verbunden. Auf diese Weise kann sich eine große Menge von Sockets ansammeln, auf deren Eingänge wir per **select** warten.

4.8 Daten Senden

Daten können Sie mit den üblichen Low-Level Lese- und Schreiboperationen **write(2)** und **read(2)** versenden und lesen. Beachten Sie dabei, daß die Sockets nicht-blockierend sind, also beim Lesen 0 zurückliefern, wenn es zur Zeit nichts zu lesen gibt.

Beachten Sie dazu evtl. Kapitel 2.3.4 und 2.3.5 von `http://openbook.galileocomputing.de/linux_unix_programmierung/Kap02-002.htm`.

Die traditionellen Operationen zur Socketkommunikation sind **send(2)** und **recv(2)**, die noch einige zusätzliche Parameter verwenden. Sie sind allerdings für diese Aufgabe nicht nötig.

4.9 Socket schließen

Sockets können genau wie Dateien auch mit **close(2)** geschlossen werden. Wenn ein Lese- oder Schreibbefehl auf einem Socket einen Fehler oder ein end-of-file (`EOF`) meldet, liegt dies meist daran, daß sich der Klient abgemeldet hat oder abgebrochen wurde; in diesem Fall ist es wichtig, den nicht mehr benötigten Socket freizugeben, damit er in späteren Verbindungen wiederverwertet werden kann– ansonsten gehen dem Prozeß früher oder später die Socketnummern aus[8].

4.10 Unterbrechungen

Wenn ein verbundener Socket geschlossen wird, markiert dies den Partner-Socket so, als ob er gelesen werden könnte. Wenn ein Klient seinen Socket schließt, markiert dies **select** also in der Menge der lesbaren Dateideskriptoren.

Zusammengefaßt ist der Ablauf einer einfachen Verbindung (ohne Warten und Wiederholung) wie folgt:

[8] Das betreffende Limit können Sie per **getrlimit(2)** auslesen; der Bezeichner des Limits für Dateideskriptoren ist `RLIMIT_NOFILE`. Unter Linux ist das Limit auch unter `/proc/${PID}/limits` einsehbar, wobei `${PID}` die Prozeß-ID des fraglichen Prozesses ist.

Server		Klient
socket(2)		**socket(2)**
bind(2)		
listen(2)		
select(2)		
		connect(2)
accept(2)		
read(2)	$\Longleftarrow$	**write(2)**
write(2)	$\Longrightarrow$	**read(2)**
close(2)		**close(2)**
unlink(2)		

Die temporär erzeugte Verbindungsdatei `/tmp/server` sollte beim Beenden des Servers per **unlink(2)** gelöscht werden (`unlink("foo")` 'löscht'[9] die Datei mit Namen `foo`).

Aufgaben.

1. Schreiben Sie einen Chat-Server nach den obigen Angaben. Der Server soll zunächst nur auf Verbindungen warten.

2. Schreiben Sie einen Chat-Zuhör-Klienten, der sich mit dem Server verbindet, von da an ununterbrochen auf ankommende Daten wartet und diese softort ausdruckt, wenn sie ankommen. Vebinden Sie die beiden Prozesse über `/tmp/server`.

3. Ändern Sie den Server, so daß er einem sich verbindenden Klienten eine kurze Nachricht (z.B. „Hallo“) sendet, die der Klient dann ausgibt. Stellen Sie sicher, daß der Server mit mehreren verbundenen Klienten umgehen kann.

   Nehmen Sie in dieser und der folgenden Aufgabe an, daß Sie maximal 5 Klienten unterstützen müssen.

4. Schreiben Sie einen Chat-Sende-Klienten, der sich mit dem Server verbindet, Eingaben vom Benutzer liest (z.B. per **getline(3)**) und diese an den Server sendet. Damit der Server zwischen den beiden Klienten unterscheiden kann, sollen sie bei ihrer ersten Verbindung eine kurze Nachricht an den Server schicken, die sie unterscheidet. Der Server soll nur an Zuhör-Klienten senden.

5. Erweitern Sie den Chat-Klienten so, daß er beim Verbinden einen (als Kommandozeilenparameter angegebenen) Chatnamen mit angibt. Wenn der Klient einen Text sendet, soll der Server den Chatnamen des Benutzers an die Zuhör-Klienten senden, bevor er den tatsächlichen Text sendet, so daß alle Zuhör-Klienten wissen, wer die Chatnachricht gesendet hat.

   Sie können der Einfachheit halber annehmen, daß der Betnuzername nicht länger als 20 Zeichen und die Chatnachrichten nicht länger als 128 Zeichen sind.

6. Beschreiben Sie das *Protokoll*, das Sie entworfen haben, um die Kommunikation zwischen Server und Klienten zu ermöglichen, also die verschiedenen Formen von Nachrichten, die Sie zwischen Server und Klienten hin- und hersenden.

7. Erweitern Sie den Chat-Server so, daß Benutzer beliebige Befehle ausführen können, deren Ausgabe dann an alle Chatteilnehmer ausgegeben wird (**popen(3)** kann ihnen dabei die wesentliche Arbeit abnehmen). Welche Zugriffsrechte haben Sie dabei?

8. Stellen Sie den Chat-Server so um, daß er Internet-Verbindungen akzeptiert. Deaktivieren Sie dabei die Änderungen der vorherigen Aufgabe aus Sicherheitsgründen.

[9] Genauer gesagt wird der betreffende Eintrag aus dem Dateisystem entfernt. Ob die zugehörigen Daten gelöscht werden, hängt von anderen Faktoren ab. Dies ist aber gleichwertig mit der umgangssprachlichen Bedeutung des „Löschens“ einer Datei.

5 Signale und Signalbehandlung (1 + 1B)

Signale sind eine asynchrone Form der Kommunikation mit Prozessen. Im Gegensatz zu anderen Kommunikationsformen erlauben sie keinen beliebigen Datentransfer. Dafür können sie ein laufendes Programm aber zu einem fast beliebigen Zeitpunkt unterbrechen, um die Abarbeitung des Signals zu erzwingen. Signale werden insbesondere vom Betriebssystem verwendet, um kritische Situationen zu melden; so lösen z.B. ungültige Speicherzugriffe, ungültige Maschinenbefehle, oder Unterbrechungen des kontrollierenden Terminals Signale aus.

Beachten Sie zunächst Foliensatz 9, Folien 19–23. (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-09.pdf`)

Signale können programmatisch per **signal(2)** gesendet werden; **sigaction(2)** erlaubt das Abfangen von Signalen. **signal(7)** beinhaltet eine Liste aller verwendeten Signale.

Aufgaben.

1. Wählen Sie einen Prozeß in Ihrem System und senden Sie ihm das Signal 11 per **kill(1)**. Was passiert?
2. Schreiben Sie eine Signalbehandlung für Signal 11 (SIGSEGV) in Ihrem Programm. Die Signalbehandlung soll die Speicheradresse ausgeben, auf die fehlerhaft zugegriffen wurde, und das Programm beenden. Demonstrieren Sie, daß die Signalbehandlung korrekt funktioniert.
3. Was passiert, wenn Ihre Signalbehandlung zum Programm zurückkehrt (`return`), anstatt es zu beenden?

(Bonus) Falls Sie eine Lösung für Ihren Chat-Server konstruiert haben (egal ob lokal oder per Internet):

Ändern Sie Ihren Chat-Server so, daß er beim Erhalten von `SIGUSR1` den Server deaktiviert und sich von allen Chat-Zuhör-Klienten freundlich verabschiedet, und beim Erhalten von `SIGUSR2` die aktuelle Uhrzeit an alle Chat-Teilnehmer sendet. Sie können serverseitig **getpid(2)** zur Bestimmung der Prozeß-ID verwenden.

6 Speicherabbildungen und Speicherschutz (2)

Der Befehl **mmap(2)** erlaubt es, die Seitentabelle eines laufenden Prozesses direkt zu manipulieren. Damit kann man mehrere nützliche Effekte erreichen:

- Speicher gezielt auf bestimmten Speicheradressen allozieren
- Inhalte von Dateien (die ja im Arbeitsspeicher zwischengespeichert werden) im Speicher direkt abzubilden, ohne, daß die Dateien dabei per **read(2)** oder **fread(3)** stückweise eingelesen werden müssen
- Kommnikation zwischen Prozessen, indem man eine Datei als geteilten Speicherbereich verwendet

Wir werden in dieser Aufgabe **mmap(2)** nur zur Allozierung einsetzen. In späteren Aufgaben wird es uns aber auch beim Dateizugriff helfen.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap alloziert Speicher der angegebenen Länge, bevorzugt an einer angegebenen Adresse:

1. `addr` ist eine gewünschte Speicheradresse zur Abbildung, oder `NULL`, falls der Aufrufer keinen besonderen Wunsch hat. Wunschadressen sind besonders interessant in Verbindung mit `MAP_FIXED` (s.u.). Die Adresse muß ein Vielfaches der Größe einer Speicherseite sein.
2. `length` ist die Anzahl der zu allozierenden bzw. abzubildenden Bytes.
3. `prot` ist der anzuwendende Speicherschutz. Dies ist entweder `PROT_NONE` (nichts ist erlaubt) oder eine Bitweise-Oder-Kombination der folgenden Flags:
   - `PROT_READ`: Lesezugriff erlaubt
   - `PROT_WRITE`: Schreibzugriff erlaubt
   - `PROT_EXEC`: Inhalte der Seite dürfen ausgeführt werden (z.B. für dynamisch erzeugten Maschinensprachecode oder 'von Hand' nachgeladene Bibliotheken).

   Wenn **mmap** verwendet wird, um eine Datei im Speicher abzubilden, müssen die Speicherschutzeinstellungen den Einstellungen der Datei beim Öffnen entsprechen (so kann z.B. eine nur-Lesen geöffnete Datei nicht mit `PROT_WRITE` schreibbar gemacht werden).

   Die Speicherschutzeinstellungen können später per **mprotect(2)** geändert werden.
4. `flags` sind zusätzliche Optionen, die durch Bitweise-Oder verknüpft werden können. Dabei ist es nötig, genau eine der ersten beiden Optionen zu wählen:
   - `MAP_SHARED`: Die Seite kann mit anderen Prozessen geteilt werden.
   - `MAP_PRIVATE`: Die Seite ist privat im aktuellen Prozeß. Jegliche Änderungen behält der Prozeß in einer privaten Kopie bei.
     Entweder `MAP_SHARED` oder `MAP_PRIVATE` muß angegeben werden.
   - `MAP_ANONYMOUS`: bilde keine Datei ab, sondern alloziere 'frischen' Speicher. Diese Option ist nicht Teil der POSIX-Spezifikation, ist aber auf den meisten Systemen verfügbar.
   - `MAP_FIXED`: erzwinge die Allozierung auf `addr`. **mmap** deallociert überlappende Bereiche, falls nötig, und schlägt fehl, falls die gegebene Adresse nicht alloziert werden kann (üblicherweise dann, wenn die Adresse kein Vielfaches der Seitengröße ist.) Dies ist ebenfalls eine Erweiterung jenseits des POSIX-Standards.
5. `fd` ist ein optionaler Dateideskriptor einer Datei, die im Speicher abzubilden ist. Wenn keine Datei abgebildet wird, sollte hier -1 übergeben werden.

6. `offset` wird nur beim Abbilden von Dateien verwendet; in diesem Fall gibt es an, welche Stelle in der Datei den Anfang des abgebildeten Speicherbereiches darstellen soll. Auf diese Weise kann der Speicherbereich ein Fenster in die Mitte der Datei sein.

Im Erfolgsfall liefert **mmap** einen Zeiger auf den allozierten Speicher, sonst `MAP_FAILED`.

munmap entfernt den betreffenden Eintrag wieder aus der Seitentabelle.

Die Funktion **mprotect(2)** kann nachträglich verwendet werden, um Speicherschutzeinstellungen zu ändern:

```
int mprotect(void *addr, size_t len, int prot);
```

Im Erfolgsfall wird 0 zurückgeliefert.

Unser Ziel ist nun, leicht verschlüsselten Speicher zu verwenden. Moderne Rechner sind oft mit externen Schnittstellen (wie z.B. Thunderbolt) ausgestattet, die direkten Zugriff auf den Speicher erlaubten. Dies kann zu einer sogenannten *DMA attack* verwendet werden, bei der ein externes Gerät den kompletten Inhalt des Arbeitsspeichers ausliest.

Unser Ziel ist es, bestimmte Speicherinhalte gegen einen solchen Zugriff (schwach) zu schützen. Dazu wollen wir soviel Speicher wie möglich verschlüsselt halten und nur bei Bedarf ein Minimum davon entschlüsseln.

Aufgaben.

1. Bestimmen Sie per `sysconf(_SC_PAGE_SIZE)` die Größe einer Speicher-Seite auf Ihrem System.

2. Schreiben Sie eine Funktion `cryptalloc`, die wie **malloc** funktioniert, aber ihren Speicher per **mmap** alloziert. Zur Vereinfachtung können wir davon ausgehen, daß `cryptalloc` nur ein Mal pro Programmaufruf verwendet wird.

3. Schreiben Sie einfache Verschlüsslungs- und Entschlüsslungsfunktionen. Sie können für unsere Zwecke den Bitumkehroperator (~) verwenden; diese „Verschlüsselung“ wäre zwar trivial zu brechen, ist aber ausreichend für unsere Demonstrationszwecke.

4. Im Gegensatz zu **malloc** soll `cryptalloc`-allozierter Speicher nun verschlüsselt sein, allerdings transparent: der Rest des Programmes soll den Speicher wie **malloc**-allozierten Speicher verwenden können.

   Dies erreichen Sie, indem Sie immer genau eine (oder maximal zwei Seiten) entschlüsselt halten, auf die das Programm gerade zugreift. Alle anderen Seiten sind verschlüsselt.

   Verwenden Sie dazu **mprotect** und eine Signalbehandlung für `SIGSEGV`. Das erwünschte Verhalten ist folgendes:

   Wenn der Benutzer auf eine verschlüsselte Seite zugreift, wird/werden die aktuell entschlüsselte(n) Seite(n) verschlüsselt, die beantragte Seite und optional die folgende Seite entschlüsselt, und der Kontrollfluß zurück an das Programm gegeben.

   Ihnen ist freigestellt, ob Sie mit genau einer entschlüsselten Seite oder mit 1-2 entschlüsselten Seiten arbeiten möchten. Letztere Variante erlaubt Ihrem Anwendungsprogramm nicht-ausgerichtete Speicherzugriffe, die Seitengrenzen überlappen.

5. Demonstrieren Sie in einem geeigneten Programm, daß Ihre Implementierung funktioniert.

6. Demonstrieren Sie, daß der Speicher verschlüsselt ist: deaktivieren Sie die Signalbehandlung kurzzeitig und geben Sie den gesamten Speicherbereich aus. Ihr Demonstrationsprogramm sollte entsprechend mehr als eine Seite Speicher allozieren.

7 Das `.text`-Segment im Speicher (1)

Wir betrachten nun die Stellen im Speicher, in die der Lader den übersetzten Programmcode läd. Wie alle anderen Daten sind auch die Maschinenbefehle lesbar und, unter bestimmten Umständen, schreibbar.

Aufgaben.

1. Schreiben Sie ein kleines Programm mit einer Funktion, die den Wert `0x12345678` per **`return`** zurückliefert. Geben Sie das Ergebnis dieser Funktion durch eine geeignete Funktion (z.B. **printf(3)**) aus.

2. Der Maschinencode Ihrer Funktion liegt an einer bestimmten Adresse im Speicher, im sogenannten `.text`-Segment. Von dort aus kann der Prozessor die Maschinenbefehle laden und ausführen. Teil dieser Maschinenbefehle ist nun die Zahl `0x12345678`; sie ist allerdings von Maschinenbefehlen umgeben, deren Größe schwer vorhersagbar ist.

   Wieviele Bytes hinter der Startadresse der Funktion finden Sie den Wert `0x12345678` im Speicher?

   Beachten Sie zur Speicherrepräsentierung von Zahlen auch Foliensatz 0, Folien 38–40 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-00.pdf`)

3. Ändern Sie die gefundene Speicheradresse. Verwenden Sie bei Bedarf **mprotect(2)**.

4. Wie wirkt sich Ihre Änderung bei Übersetzeroptimierungsstufe `-O3` aus? Bei `-O0`?

8 Dynamisches Binden (1 + 1B)

Lesen Sie zunächst folgendes: `http://openbook.galileocomputing.de/linux_unix_programmierung/Kap17-001.htm#RxxKap170010400061A1F0291A1`

Aufgaben.

1. Schreiben Sie ein Programm, das eine beliebige Datei einliest und über einen globalen Zeiger namens `d` (`unsigned char *`) verfügbar macht. Verwenden Sie dazu **mmap(2)**. Legen Sie die Größe der Datei in einer globalen Variable `size` ab.

   Die Dateigröße könnnen Sie per **fstat(2)** bestimmen:

   ```
   struct stat dateistatus;
   fstat(offener_dateideskriptor, &dateistatus);
   int size = dateistatus.st_size;
   ```

2. Ihr Programm soll nun dem Benutzer erlauben, die Datei zu inspizieren. Dazu soll der Benutzer beliebige C-Anweisungen verwenden können, die Sie z.B. per **getline(3)** einlesen. Sie können davon ausgehen, daß im Falle von mehreren Anweisungen (z.B. einer Schleife, die alle Elemente ausgibt) alle relevanten Teile in eine Eingabezeile gepackt werden.

   Damit Sie nicht selbst einen C-Interpreter implementieren müssen, scheiben Sie die Benutzereingabe in eine geeignete Datei und rufen Sie von Ihrem Programm aus **gcc** auf, um die Datei in eine dynamisch ladbare Bibliothek zu überstzen, die Sie dann laden und ausführen können.

3. Testen Sie Ihr Programm mit geeigneten Eingaben. Bei ungültigen Eingaben (z.B. zu vielen/zu wenigen geschweifte Klammern) soll Ihr Programm vermelden, daß es einen Fehler gab, diesen aber nicht weiter beschreiben.

(Bonus) Erlauben Sie auch die Modifizierung der Datei durch in-den-Speicher-Schreiben. Zum Beispiel sollte `d[0] = 'A';` das erste Zeichen der Datei auf der Festplatte auf 'A' ändern.

(Bonus) Erlauben Sie das Vergrößern und Verkleinern der Datei durch Zuweisung an `size`. Gehen Sie vereinfachend davon aus, daß beim Vergrößern der Datei erst im nächsten eingelesenen Befehl auf den neuen Speicherbereich zugegriffen wid.

9 Vektoroperationen mit `gcc` ($1\frac{1}{2}$ + 1B)

Einer der Gründe für die Verwendung systemnaher Programmierung ist das Erlangen hoher Abarbeitungsgeschwindigkeit. Auf modernen Prozessoren sind die wesentlichsten Schlüssel dazu die folgenden:

- Graphik-Koprozessoren
- Nebenläufige Prozesse
- Vektoroperationen

Wir betrachten in dieser und der folgenden Aufgabe die letzten beiden Punkte dieser Liste. Hier untersuchen wir zunächst Verktoroperationen.

Eine Vektoroperation ist eine SIMD[10]-Operation, die mehrere gleichartige Operationen auf einem Vektor von Daten durchführt. So kann eine einzelne solche Operation zum Beispiel mehrere unabhängige Werte um eins erhöhen. Solche Operationen sind in Hardware effizient realisierbar und werden von allen neueren Prozessoren unterstützt.

Nur wenige Programmiersprachen sehen direkte Unterstützung für Vektoroperationen vor. Der C-Übersetzer `gcc`[11] erlaubt Vektoroperationen mit einer Spracherweiterung, in der einem Typen ein spezielles *Attribut* verliehen wird:

```
typedef float float4 __attribute__ ((vector_size (16)));
```

In obigem Beispiel definieren wir einen Typen `float4`, der Vektoren von vier `float`-Werten beschreibt. Die Zahl `16` hier gibt die Speichergröße an, die der Vektor belegt; diese muß eine Zweierpotenz sein. Da ein `float` eine Größe von 4 hat, kommen wir auf 4 floats in den 16 Bytes.

Der Typ `float4` kann nun wie ein normaler C-Typ verwendet werden. Ähnlich wie bei einem Array können Elemente des Vektors einzeln ausgelesen und manipuliert werden:

```
float4 v = {0.1f, 0.2f, 0.3f, 0.4f}; // Initialisierung
float4 z = v;                        // Kopieren
z[2] += 10.0f;  // Dritten Eintrag (Index 2) erhoehen
```

Der Nutzen eines Vektors ergibt sich aus arithmetischen Operationen. Der Übersetzer erlaubt im Wesentlichen zwei Arten von Verknüpfungen:

- Verknüpfungen mit 'Skalarwerten'. Beispiel `v4 += n`:

$$\langle v_0, v_1, v_2, v_3\rangle + n = \langle v_0 + n, v_1 + n, v_2 + n, v_3 + n\rangle$$

- Verknüpfungen mit anderen Vektoren des gleichen Typs. Beispiel `v *= z`:

$$\langle v_0, v_1, v_2, v_3\rangle \times \langle z_0, z_1, z_2, z_3\rangle = \langle v_0 \times z_0, v_1 \times z_1, v_2 \times z_2, v_3 \times z_3\rangle$$

Eine wichtige Einschränkung der Vektoroperatoren ist, daß Vektoren nur *ausgerichtet* aus dem Speicher gelesen bzw. in diesen geschrieben werden können. Bei einem Vektor mit einer Größe von 16 Bytes muß also die Speicheradresse einer solchen Speicheroperation immer durch 16 teilbar sein. Anders ausgedrückt:

```
float data[32]; // Annahme:  &data[0] ist durch 16 teilbar und somit
                // ausgerichtet (muss in der Praxis nicht so sein!)
float4 *p;

p = &data[8];
```

[10] *Single Instruction, Multiple Data*– Einzelne Instruktion, mehrere Daten

[11] Ebenso der Intel-C-Übersetzer `icc`

```
float4 v = *p;  // Erlaubt:  &data[0] + (8 * sizeof(float)) ist durch
                // 16 teilbar, also ausgerichtet

p = &data[3];
float4 w = *p;  // Speicherzugriffsfehler!
                // &data[0] + (3 * sizeof(float)) ist nicht durch 16 teilbar
```

Um diese Ausrichtung zu erzwingen, kann seit C11[12] folgende Funktion verwendet werden:

```
void *aligned_alloc(size_t alignment, size_t size);
```

wobei `alignment` die gewünschte Ausrichtung (hier 16) und `size` die gewünschte Speichergröße ist.

Anmerkungen zur Übersetzung: Um den Übersetzer möglichst guten Code erzeugen zu lassen, sollten Sie für diese Aufgabe folgende Übersetzerparameter setzen:

- `-O3` (Alle Optimierungen aktiviert)
- `-mcpu=native` (Optimiere für den übersetzenden Rechner und nutze all dessen Vektoroperationen)
- `-std=gnu11` (Verwende den C11-Standard mit GNU-Erweiterungen)

Aufgaben.

1. Installieren Sie eine Implementierung einer trivialen Matrixmultiplikation, die als Benchmark konfiguriert ist:

   `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/matrix-multiply.c`. Überprüfen Sie, daß Sie das Programm übersetzen und ausführen können. Die Ausgaben sind:

   - Ergebnismatrix einer Matrixpotenzberechnung einer 200×200-Matrix
   - Median-Laufzeit
   - Abweichung der besten Laufzeit vom Median
   - Abweichung der schlechtesten Laufzeit vom Median

   Wenn Ihr System nicht zu ausgelastet ist, sollten die Abweichungswerte nahe bei 1 liegen.

2. Vektorisieren Sie die Matrixmultiplikation so, daß sie vier Floats als Vektor verwendet.
3. Wieviel schneller ist ihre vektorisierte Variante?
4. Wird Ihre Implementierung auch mit Matrizen korrekt funktionieren, die kein Vielfaches von 4 als Breite haben? Falls nein, warum nicht?

(Bonus) Stellen Sie sicher, daß Ihre vektorisierte Fassung mit Matrizen beliebiger positiver Breite arbeiten kann. Erklären Sie Ihre diesbezüglichen Änderungen.

(Bonus) Wieviel schneller sind Varianten mit 2 bzw. 8 floats? Geben Sie den verwendeten Prozessortyp in Ihrer Antwort mit an (`cat /proc/cpuinfo`).

[12] Übersetzung mit `-std=gnu11`

10 POSIX Threads (1 + 1B)

Threads (Ausführungsstränge) sind eine Alternative zu Prozessen, um Nebenläufigkeit in der Ausführung zu ermöglichen. Wir verwenden hier die POSIX Thread-API. Die nötige Include-Datei ist `#include<pthread.h>`. Binden Sie die pthreads-Bibliothek mit `-lpthread` ein.

Um Threads zu starten, verwenden Sie

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Hierbei ist `thread` ein Speicherbereich, der von der Funktion selbst beschrieben und initialisiert wird, sie müssen also nur Speicher zur Verfügung stellen (z.B. in Form einer Variable). `attr` sind Thread-Attribute, die wir hier auf den Standardwerten (`NULL`) lassen können. `start_routine` ist die Funktion, die im Thread ausgeführt wird, und `arg` das an diese Funktion übergebene Argument.

```
int pthread_join(pthread_t thread, void **retval);
```

Diese Funktion wartet auf das Ende der Laufzeit des gegebenen Threads. Der Rückgabewert der Thread-Funktion wird dabei in `*retval` abgelegt.

Aufgaben.

1. Verwenden Sie zwei Threads, um die Matrixmultiplikation des Programmes aus Aufgabe 9 zu parallelisieren (vor oder nach der Vektorisierung).
2. Wie viel schneller ist Ihre parallele Fassung?

(Bonus) Könnten Sie statt Threads auch mit **fork(2)** erzeugte Kindprozesse zur nebenläufigen Berechnung verwenden? Erklären Sie Ihre Antwort. Beschreiben Sie Experimente, die Sie zur Beantwortung durchgeführt haben, falls nötig.

Welche Unterschiede sehen Sie zwischen Threads und Prozessen?

11 Performanzzähler (1 + $\frac{1}{2}$B)

Verwenden Sie Performanzzähler, um die folgenden Fragen zu beantworten:

Aufgaben.

1. Wieviele Prozessorinstruktionen führt die nicht-vektorisierte Fassung der Matrixmultiplikation pro Prozessorzyklus aus? (IPC = *instructions per cycle*)
2. Wieviele Instruktionen pro Zyklus sehen wir in der vektorisierten Fassung?
3. Schreiben Sie ein Programm mit einem statisch allozierten `int`-Array in der Speichergröße von einem Megabyte (gehen Sie von 32-Bit-Integer-Zahlen aus). Summieren Sie die Inhalte des Arrays auf, ohne es zu beschreiben (alle Elemente sollten 0 sein.) Messen Sie dabei die Anzahl der Seitenfehlzugriffe. Wann beobachten Sie jeweils einen solchen?

(Bonus) Messen Sie: wie lange dauert ein Aufruf von `PAPI_start_counters` und `PAPI_stop_counters` ungefähr? Hängt das von der Anzahl der gemessenen Eigenschaften ab, und wenn ja, wie?

12 Bonusaufgabe: Der Stapelspeicher (2B)

Aufgaben.

1. Installieren Sie folgendes Programm: `http://web.sepl.cs.uni-frankfurt.de/2013-ws/b-bkspp-pr/resources/p.c`
2. Übersetzen Sie das Programm mit `-O0`.
3. Bringen Sie das Programm dazu, "`Zugriff erfolgreich.`" nach **stderr(3)** zu schreiben. Sie dürfen dazu das Programm nicht verändern, weder vor noch nach der Übersetzung. Sie dürfen auch den Übersetzer nicht verändern. Das Programm muß dabei mit dem normalen Binder, Lader, den unveränderten Standard-Bibliotheken, und dem Standard-Systemkern ausgeführt werden. Erklären Sie Ihr Vorgehen.

13 Literatur

- "The C Programming Language" (Brian W. Kernighan, Dennis M. Ritchie) (auch auf Deutsch in der Bibliothek)
- C99–Spezifikation (draft) `http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/C99.pdf`
- "The Linux Programming Interface: A Linux and UNIX System Programming Handbook" (Michael Kerrisk)
- „Linux– Unix- Systemprogrammierung" (Helmut Herold)
- „Linux-Unix-Programmierung", Jürgen Wolf, Galileo Computing `http://openbook.galileocomputing.de/linux_unix_programmierung/index.htm`
- Foliensätze der Veranstaltung B-SYSP vom Sommersemester 2013 `http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/veranstaltungen.de.html`