technische universität dortmund

fakultät für informatik

Alwin Berger

Dortmund, den 23. Juni 2025

Übungen zur Vorlesung Betriebssysteme Blatt 5

Es können 10 Punkte erreicht werden.

Abgabefrist Übungsabgabe 5

Die Lösung muss abhängig von der Tafelübung abgegeben werden, an der Sie teilnehmen:

- **W1** – Übung U5 in KW26 (23.06. - 26.09.): Abgabe bis **Mittwoch, den 02.07. 23:59**
- **W2** – Übung U5 in KW27 (30.06. - 03.07.): Abgabe bis **Montag, den 07.07. 14:00**

In darauffolgenden Tafelübungen werden teilweise einzelne abgegebene Lösungen besprochen, teilweise auch ein Lösungsvorschlag aus dem Tutorenteam.

Allgemeine Hinweise zu den BS-Übungen

- Die Aufgaben sind *in Dreiergruppen* zu bearbeiten. Der Lösungsweg und die Programmierung sind gemeinsam zu erarbeiten.
- Die Gruppenmitglieder **sollten nach Möglichkeit** gemeinsam an der gleichen Tafelübung teilnehmen. Es sind aber auch Abgaben mit Studierenden aus **verschiedenen Übungsgruppen** zulässig. Es gilt dabei immer die **früheste Abgabefrist**. Die Lösung wird jeweils komplett bewertet und den Gruppenmitgliedern gleichermaßen angerechnet.
- Die abgegebenen Antworten/Programme werden automatisch auf Ähnlichkeit mit anderen Abgaben überprüft. Wer beim Abschreiben[1] erwischt wird, verliert ohne weitere Vorwarnung die Möglichkeit zum Erwerb der Studienleistung in diesem Semester!
- Gelegentlich gibt es Zusatzaufgaben auf den Blättern. Diese sind ein Stück schwerer als die „normalen“ Aufgaben und geben zusätzliche Punkte.
- Die Aufgaben sind über AsSESS (`https://sys-sideshow.cs.tu-dortmund.de/ASSESS/`) abzugeben. Dort gibt **ein** Gruppenmitglied die erforderlichen Dateien ab und nennt dabei die anderen beteiligten Gruppenmitglieder (Matrikelnummer, Vor- und Nachname erforderlich!). Namen und Anzahl der abzugebenden C-Quellcodedateien[2] variieren und stehen in der jeweiligen Aufgabenstellung; Theoriefragen sind grundsätzlich in der Datei `antworten.txt`[3] zu beantworten. Bis zum Abgabetermin kann eine Aufgabe beliebig oft abgegeben werden – es gilt die letzte, vor dem Abgabetermin vorgenommene Abgabe.

[1] Da wir meist nicht unterscheiden können, wer von wem abgeschrieben hat, gilt das für Original **und** Plagiat.
[2] codiert in UTF-8
[3] reine Textdatei, codiert in UTF-8

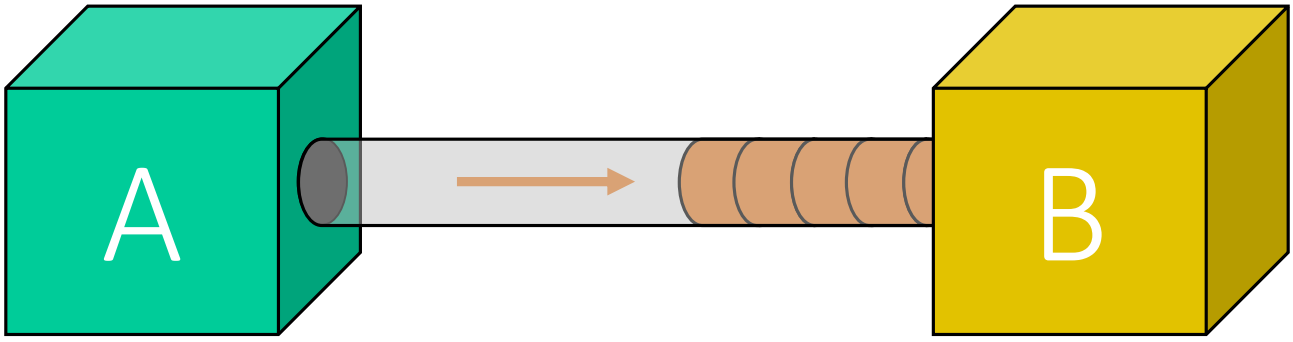
- Sobald eine Abgabe korrigiert wurde, kann die korrigierte Lösung ebenfalls in AsSESS eingesehen werden.
- Ihre Programme müssen von `gcc` (also kein C++) mit den Option `-std=c17 -Wall` kompilierbar sein (als Referenzumgebung dienen die Poolrechner. Wie empfehlen zusätzlich `-Werror`). Sollten Warnungen entstehen, können ihnen dafür Punkte abgezogen werden. Programm die nicht übersetzt werden können, werden in der Regel mit 0 Punkten bewertet.
- Nutzen Sie in ihrem Quelltext eine sinnvolle Einrückung für Blöcke und vermeiden sie mehrere Anweisungen in einer Zeile. In Fällen von unleserlicher Formatierung können ihnen Punkte abgezogen werden.
- Schreiben Sie nicht Ihre Namen und Matrikelnummern in die Abgabe. Generell erhalten nur die jenigen Gruppenmitglieder die Punkte, dessen Namen und Matrikelnummern im ASSESS vermerkt sind.

Aufgaben

Aufgabe 1: Prozesskommunikation durch FIFOs (5 Punkte)

In den folgenden Teilaufgaben implementieren Sie zwei getrennte Programme, welche über eine feststehende Programmverbindung (named pipe / FIFO) Nachrichten übertragen. Diese Nachrichten sind folgendermaßen aufgebaut: `0x1<length><id>0x2<text>0x4` (die Klammern `<>` dienen nur zur Verdeutlichung und sind nicht Teil der Nachricht). Dabei sind `length` und `id` jeweils vom Typ `uint32_t` und `text` ist eine Zeichenkette (ohne null-Terminator) der Länge `length`. Das Format ist binär codiert. Dementsprechend bedeuteen `0x1`,`0x2` und `0x4`, dass ein einzelnes Byte mit dem jeweiligen Wert übertragen wird. Die Variablen vom Typ `uint32_t` werden in vier Bytes übertragen.
Das erste Programm (`producer`) erzeugt zufällige Nachricht und schreibt diese in eine feststehende Programmverbindungen. Das zweite Programm (`consumer`) liest die Nachricht und gibt sie auf der Konsole aus.

Feststehende Programmverbindungen (FIFOs, siehe `fifo(7)`) sind nicht-reguläre Dateien im Dateisystem, über die Prozesse Daten austauschen können. Im Gegensatz zu anonymen Datenleitungen (`pipe(2)`), die nur zwischen verwandten Prozessen bestehen, ermöglichen feststehende Programmverbindungen die Kommunikation zwischen unabhängig gestarteten Programmen. Sie werden mit `mkfifo(3)` erzeugt und anschliessend mit `fopen(3)`, `fclose(3)`, `fread(3)`, `fwrite(3)` und `fflush(3)` verwendet. Zur Löschung wird `unlink(2)` benutzt. Nutzen Sie zur Benennung der Programmverbindung das Makro `FIFO_NAME` und für den Modus (Berechtigungen) das Makro `RWRWRW`.



a) Implementieren Sie das Programm `producer`, welches folgende Anforderungen erfüllt:

- `producer` benutzt eine FIFO-Datei `FIFO_NAME`, welche zu Beginn erstellt wird und zum Schluss wieder gelöscht wird (`mkfifo(3)` und `unlink(2)`). Die Löschung muss auch in Fehlerfällen erfolgen.
- `producer` erzeugt zufällige Zeichenketten (nur ASCII 32 bis 126) mit Längen im Bereich `[1..10]`. Diese Zeichenketten werden zusammen mit einer aufsteigenden id entsprechend des oben angegebenen Formats in die FIFO geschrieben. Nach zehn Nachrichten beendet sich das Programm ordnungsgemäß.

(3 Punkte)

⇒ `producer.c`

b) Implementieren Sie das Programm `consumer`, welches folgende Anforderungen erfüllt:

- `consumer` versucht bis zu 10 Sekunden lang, die gemeinsame FIFO-Datei zu öffnen. Falls dies nicht erfolgreich ist, beendet sich das Programm mit einer Fehlermeldung, welche die Ursache beschreibt.
- `consumer` liest kontinuierlich Nachrichten (s.o. für das Format) aus der FIFO. Der Puffer für die Texte wird dynamisch mit `malloc(3)` alloziert. Achten Sei darauf, dass die angegebene Länge beim Lesen nicht überschritten wird.
- Jede gelesene Nachricht wird in der Form `<id>, <length>: <text>` auf der Konsole ausgegeben.
- Sobald der Eingabestrom vor Beginn einer neuen Nachricht endet (end of file), wird das Lesen eingestellt und das Programm ordnungsgemäß beendet (s.o. für FIFO Schließung und Löschung).

(2 Punkte)

⇒ `consumer.c`

Aufgabe 2: Additives Mischen von .WAV-Dateien (5 Punkte)

Digitale Audiodaten werden häufig in verschiedenen Formaten gespeichert, wobei das WAV-Format (Waveform Audio File Format) eines der ältesten und gebräuchlichsten für unkomprimierte Audiodaten ist. Es basiert auf dem Resource Interchange File Format (RIFF) von Microsoft und IBM. Eine WAV-Datei ist strukturell in Chunks (Datenblöcke) unterteilt, die jeweils eine spezifische Art von Daten enthalten:

- **`RIFF`-Header (12 Bytes):**
  - `ChunkID` (4 Bytes): Enthält immer die ASCII-Zeichen `RIFF` (Magische Zahl).
  - `ChunkSize` (4 Bytes): Die Größe der gesamten Datei in Bytes minus 8 Bytes (für `ChunkID` und `ChunkSize` selbst).
  - `Format` (4 Bytes): Enthält immer die ASCII-Zeichen `WAVE`.
- **`fmt`-Chunk (Format-Block, ca. 24 Bytes):** Dieser Chunk beschreibt das Format der Audiodaten.
  - `Subchunk1ID` (4 Bytes): Enthält immer `fmt ` .
  - `Subchunk1Size` (4 Bytes): Die Größe des restlichen fmt-Chunks (typischerweise 16 für PCM).
  - `AudioFormat` (2 Bytes): Gibt das Audioformat an (1 für PCM = Pulse Code Modulation, d.h. unkomprimiert).
  - `NumChannels` (2 Bytes): Anzahl der Kanäle (1 für Mono, 2 für Stereo).
  - `SampleRate` (4 Bytes): Abtastrate in Hz (z.B. 44100 für CD-Qualität).
  - `ByteRate` (4 Bytes): `SampleRate * NumChannels * BitsPerSample / 8`.
  - `BlockAlign` (2 Bytes): `NumChannels * BitsPerSample / 8` (Größe eines Frames).
  - `BitsPerSample` (2 Bytes): Anzahl der Bits pro Sample pro Kanal (z.B. 16 für 16-Bit-Audio).
- **`Data`-Chunk (Abtastwerte):** Dieser Chunk enthält die eigentlichen Audiodaten.
  - `Subchunk2ID` (4 Bytes): Enthält immer den String `data`.
  - `Subchunk2Size` (4 Bytes): Die Größe der Audiodaten in Bytes.
  - `Data`: Die eigentlichen Audiodaten, typischerweise als aufeinanderfolgende Samples (z.B. 16-Bit Signed Integers für PCM).

In dieser Aufgabe sollen Sie ein Programm implementieren, welches als Argumente die Pfade von WAV-Dateien entgegennimmt, diese ausliesst, additiv mischt und das signalgemsich in eine neue WAV-Datei schreibt. Das Letzte Argument fungiert als Dateipfad der Ausgabedatei: `wavmix input1.wav input2.wav ... output.wav` Additiv mischen bedeutet, dass die Abtastwerte der einzelnen Eingangskanäle für jeden Zeitpunkt summiert und dann durch die Anzahl der gemischten Dateien geteilt werden (Durchschnittsbildung). Hierbei ist es entscheidend, dass alle Eingabedateien über kompatible Audioformate verfügen (gleiche Abtastrate, Anzahl der Kanäle und Bittiefe). Wenn die Länge der Eingabedateien variiert, wird die Mischerfunktion nur so lange Samples lesen, wie alle Dateien gültige Daten liefern können. Wir gehen der Einfachheit halber davon aus, dass alle Eingabedateien den selben Header besitzen, gleich lang sind und im Format PCM 16-Bit gespeichert sind. Im Moodle finden Sie außerdem beispielhafte Audiospuren zum mischen.

Nach Bearbeitung jedes `TODO`s entfernen Sie den jw. Kommentar `///TODO` aus Ihrem Code.

a) Prüfen Sie, ob die Ausgabedatei bereits existiert (`TODO1`). Falls ja, soll das Programm mit einer Fehlermeldung beendet werden. Nutzen Sie dazu das vorgegebene Makro `ERR0(PATHNAME)`.

(0.5 Punkte)

b) Implementieren Sie das öffnen der Eingabedateien. Überlegen Sie, wie die vordefinierten lokalen Variablen sinnvoll genutzt werden können:

- `2`: Prüfen Sie vorher, ob diese geöffnet werden kann. Falls die Eingabedatei nicht sinnvoll geöffnet werden kann, nutzen Sie das Makro `ERR1` um die entsprechende Fehlermeldung auszugeben.
- `3`: Setzen Sie den Dateizeiger an den Anfang der Datei und lesen Sie den WAV-Header ein. Prüfen Sie, ob der Header gültig ist (z.B. `RIFF` und `WAVE` Signaturen). Verwenden Sie `ERR2` bei Fehlern.
- `4`: Vergleichen Sie die Header der einzelnen Eingabedateien miteinander und mit einem Referenzheader (z.B. dem ersten erfolgreich gelesenen Header), um sicherzustellen, dass sie kompatibel sind (gleiche `SampleRate`, `NumChannels`, `BitsPerSample`). Bei Inkompatibilität verwenden Sie `ERR3`.

(1.5 Punkte)

c) Bereiten Sie die Ausgabedatei vor.

- `5`: Erstellen Sie die Ausgabedatei für das Schreiben der gemischten Audiodaten. Sollte dies nicht gelingen soll mittels `ERR4` ein Fehler gemeldet werden.
- `6`: Schreiben Sie den Referenzheader als Platzhalter an den Anfang der Ausgabedatei. Dieser Header wird später aktualisiert, wenn die Größe der Audiodaten berechnet wurde.

(1 Punkt)

d) Implementieren Sie die Mischlogik für die Abtastwerte.

- `7`: Lesen Sie Sample für Sample von allen Eingabedateien, bilden Sie für jeden Zeitschritt und für jeden Kanal den Durchschnitt der entsprechenden Samples aller Dateien und schreiben Sie das Ergebnis in die Ausgabedatei. Berücksichtigen Sie, dass die Dateien unterschiedliche Längen haben können. Überlegen Sie, wie die vordefinierten lokalen Variablen sinnvoll genutzt werden können.
- `8`: Aktualisieren Sie am Ende den Header der Ausgabedatei mit der tatsächlich geschriebenen Abtastungsmenge.
- `10`: Schliessen Sie alle Dateien und vergessen Sie nicht Ihren Heap aufzuräumen!
- `11`: Nutzen Sie das Makro `MIXING_COMPLETED` um zu signalisieren, dass das Mischen erfolgreich war.

(2 Punkte)

⇒ `wavmix.c`

Bonusaufgabe: Aufzugsteuerung

(2 Bonuspunkte) In Aufzügen moderner Gebäuden mit vielen Etagen oder bei der Steuerung von Festplattenleseköpfen ist es wichtig, dessen Bewegung so effizient wie möglich zu gestalten. Das Ziel ist es, die Anzahl der Etagenwechsel oder Spurwechsel zu minimieren, um die Bearbeitungszeit einer Menge von Anfragen (Anfragemenge L_t) die zu unterschiedlichen Zeitpunkten t gestellt werden, so kurz wie möglich zu halten.

Um diesen Ziel möglichst nahe zu kommen, werden spezielle Algorithmen eingesetzt. Einer der bekanntesten und effektivsten Algorithmen für diese Art von Problemstellung ist der sogenannte **SCAN-Algorithmus**, den wir auch als **Fahrstuhlalgorithmus** oder **Aufzugalgorithmus** kennengelernt haben.

Eine Anfragemenge L_t sei hier als eine Struktur definiert, die den Zeitpunkt des Eintreffens (`arrivaltime`), die Anzahl der enthaltenen Anfragen (`count`) und ein Array von Etagen- oder Spurnummern (`unsigned int requests[]`) enthält.

Der Algorithmus funktioniert wie folgt:

1. **Ausgangssituation:** Der Aufzug befindet sich in einer bestimmten Etage (`current_track`) und hat eine aktuelle Bewegungsrichtung (`direction`: aufwärts (`UP`) $\uparrow = 1$ oder abwärts (`DOWN`) $\downarrow = -1$). Eine Menge L enthält anstehende Anfragen.

2. **Bewegungsrichtung festlegen:** Wenn der Aufzug stillsteht (keine aktive Bewegung), wählen Sie die Richtung zur nächsten anstehenden Anfrage. Ansonsten behalten Sie die aktuelle Bewegungsrichtung bei.

3. **Anfragen sortieren:** Sortieren Sie die anstehenden Anfragen in der aktuellen Bewegungsrichtung, um die nächste anzufahrende Etage zu identifizieren.

4. **Abarbeiten der Anfragen:** Bewegen Sie den Aufzug in der festgelegten Richtung. Halten Sie an jeder Etage an, für die eine Anfrage vorliegt. Nehmen Sie während der Fahrt neue Anfragen auf, die in dieselbe Bewegungsrichtung gehen.

5. **Richtungswechsel:** Wenn keine weiteren Anfragen in der aktuellen Richtung vorliegen, oder ein Ende des Fahrbereichs (`end_of_simulation`) erreicht wurde und keine weiteren Anfragen in dieser Richtung ausstehen, wechseln Sie die Richtung. Beginnen Sie dann erneut mit Schritt 3.

6. **Leerlauf:** Wenn keine Anfragen mehr vorliegen, wartet der Aufzug auf neue Anfragen. Bei Eintreffen einer neuen Anfrage beginnt der Prozess wieder mit Schritt 2.

Ein Beispiel zur Veranschaulichung: Stellen Sie sich ein Hochhaus vor, das neben dem Erdgeschoss (Etage 0) 15 weitere Etagen besitzt. Alle 16 Etagen (0-15) werden von einem Aufzug bedient. Der Aufzug startet im Erdgeschoss (Etage 0) mit einer anfänglichen Bewegungsrichtung UP (aufwärts).
Zu verschiedenen Zeitpunkten t gehen folgende Anfragemengen ein:

- $t = 0$: Im Erdgeschoss steigen vier Personen ein, die in die Etagen $5, 15, 2$ und 9 möchten: $L_0 = \{5, 15, 2, 9\}$
- $t = 2$: Auf der Etage 5 steigen Personen zu, die auf den Etagen $4, 10$ und 1 aussteigen wollen. Die entsprechenden Tasten im Aufzug betätigen diese alle simultan zum Zeitschritt 2: $L_2 = \{4, 10, 1\}$
- $t = 6$: Auf dem Weg zur Etage 1 steigen weitere Personen zu, die auf den Etagen $8, 6$ und 14 aussteigen wollen. Das passiert zum Zeitschritt 6: $L_6 = \{8, 6, 14\}$
- $t = 10$: Eine weitere Person möchte ins Erdgeschoss. Aus Brandschutz und Effizienzgründen verbleibt der Aufzug bei weiterer nichtnutzung im Erdgeschoss: $L_{10} = \{0\}$

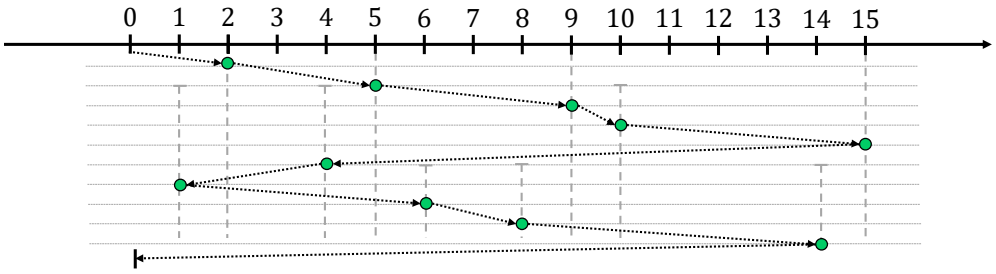


Abbildung 1: Die Anwendung des SCAN-Algorithmus auf diese Anfragen (beginnend im Erdgeschoss) sollte die folgende Abfolge liefern: `2, 5, 9, 10, 15, 4, 1, 6, 8, 14, 0`

Die Datei `scan.c` enthält bereits einen Simulationsrahmen für den Aufzugbetrieb sowie einen Parser für die Eingabe der Anfragemengen. Sie können das Programm im Terminal mit folgender Syntax aufrufen: `./scan Lx = {..., ...}, Ly = {..., ...}, ...` Wobei `x, y` etc. die Ankunftszeiten der Anfragemengen sind und die geschweiften Klammern die angefragten Etagenlisten enthalten.

Die eigentliche Aufzugsteuerung soll in `SCAN(unsigned int* queue, int* queue_size, int* current_track, int* direction, unsigned int end_of_simulation)` implementiert werden. Diese wird in jedem Zeitschritt aufgerufen und berechnet jeweils, wo der Aufzug als nächstes hinfahren soll.

Die Parameter der `scan`-Funktion haben folgende Bedeutung:

- `unsigned int* queue`: Ein Zeiger auf das Array, das alle aktuell anstehenden Anfragen (Etagen) enthält.
- `int* queue_size`: Ein Zeiger auf eine Ganzzahl, die die aktuelle Anzahl der Anfragen im `queue`-Array angibt.
- `int* current_track`: Ein Zeiger auf die aktuelle Etage, auf der sich der Aufzug befindet.
- `int* direction`: Ein Zeiger auf die aktuelle Bewegungsrichtung des Aufzugs (`UP` für aufwärts, `DOWN` für abwärts).

Implementieren Sie nun die Aufzugsteuerung aus `scan.c` wie folgt. Nach Bearbeitung jedes `TODO`s entfernen Sie den jw. Kommentar `///TODO` aus Ihrem Code.

- `1`: Sammeln Sie alle Anfragen aus der Warteschlange (`queue`) in ein temporäres Array (`candidates`), die sich in der aktuellen Bewegungsrichtung (`direction`) befinden und sich auf oder nach der aktuellen Etage (`current_track`) befinden.
- `2`: Verbleibt `count` bei dem Wert 0, soll die Richtung `direction` geändert werden.
- `3`: Sortieren Sie die Anfragen aus `queue` basierend auf der aktuellen Richtung (`direction`). Wenn die Richtung `UP` ist, sortieren Sie aufsteigend. Wenn die Richtung `DOWN` ist, sortieren Sie absteigend. Beachten Sie dabei `current_track`. Nutzen Sie dazu die Bibliotheksfunktion `qsort` sowie die vordefinierten Funktionen `compare_up` sowie `compare_down`.
- `4`: Aktualisieren Sie den `current_track`.
- `5`: Suchen und entfernen Sie die besuchte Etage aus `queue`.

⇒ `scan.c`