Übungsblatt 9

Aufgabe 1 (Interprozesskommunikation)

1. Beschreiben Sie, was ein kritischer Abschnitt ist.
2. Beschreiben Sie, was eine Race Condition ist.
3. Beschreiben Sie, warum Race Conditions schwierig zu lokalisieren und zu beheben sind.
4. Nennen Sie eine Möglichkeit, um Race Conditions zu vermeiden.

Aufgabe 2 (Synchronisation)

1. Beschreiben Sie den Vorteil von Signalisieren und Warten gegenüber aktivem Warten (Warteschleife).
2. Geben Sie an, welche beiden Probleme durch Blockieren entstehen können.
3. Beschreiben Sie den Unterschied zwischen Signalisieren und Blockieren.
4. Kreuzen Sie vier Bedingungen an, die gleichzeitig erfüllt sein müssen, damit ein Deadlock entstehen kann?

☐ Rekursive Funktionsaufrufe	☐ Anforderung weiterer Betriebsmittel
☐ Wechselseitiger Ausschluss	☐ > 128 Prozesse im Zustand `blockiert`
☐ Häufige Funktionsaufrufe	☐ Iterative Programmierung
☐ Geschachtelte `for`-Schleifen	☐ Zyklische Wartebedingung
☐ Ununterbrechbarkeit	☐ Warteschlangen

5. Führen Sie die Deadlock-Erkennung mit Matrizen durch und geben Sie an, ob es zum Deadlock kommt.

$$\text{Ressourcenvektor} = \begin{pmatrix} 8 & 6 & 7 & 5 \end{pmatrix}$$

$$\text{Belegungsmatrix} = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 3 & 1 & 0 & 4 \\ 0 & 2 & 1 & 1 \end{bmatrix} \qquad \text{Anforderungsmatrix} = \begin{bmatrix} 3 & 2 & 4 & 5 \\ 1 & 1 & 2 & 0 \\ 4 & 3 & 5 & 4 \end{bmatrix}$$

Aufgabe 3 (Kommunikation von Prozessen)

1. Geben Sie an, was bei Interprozesskommunikation über gemeinsame Speichersegmente (Shared Memory) zu beachten ist.

2. Beschreiben Sie die Aufgabe der Shared Memory Tabelle im Linux-Kernel.

3. Kreuzen Sie an, welche Auswirkungen ein Neustart (Reboot) des Betriebssystems auf die bestehenden gemeinsamen Speichersegmente (Shared Memory) hat.
   (Nur eine Antwort ist korrekt!)

   ☐ Die gemeinsamen Speichersegmente werden beim Neustart erneut angelegt und die Inhalte werden wieder hergestellt.
   ☐ Die gemeinsamen Speichersegmente werden beim Neustart erneut angelegt, bleiben aber leer. Nur die Inhalte sind also verloren.
   ☐ Die gemeinsamen Speichersegmente und deren Inhalte sind verloren.
   ☐ Nur die gemeinsamen Speichersegmente sind verloren. Die Inhalte speichert das Betriebssystem in temporären Dateien im Ordner `\tmp`.

4. Geben Sie an, nach welchem Prinzip Nachrichtenwarteschlangen (Message Queues) arbeiten.
   (Nur eine Antwort ist korrekt!)

   ☐ Round Robin ☐ LIFO ☐ FIFO ☐ SJF ☐ LJF

5. Geben Sie an, wie viele Prozesse über eine Pipe miteinander kommunizieren können.

6. Beschreiben Sie was passiert, wenn ein Prozess in eine volle Pipe schreiben will.

7. Beschreiben Sie was passiert, wenn ein Prozess aus einer leeren Pipe lesen will.

8. Geben Sie an, welche zwei Arten von Pipes existieren.

9. Geben Sie an, welche zwei Arten von Sockets existieren.

10. Kommunikation via Pipes funktioniert...
    (Nur eine Antwort ist korrekt!)

    ☐ speicherbasiert ☐ nachrichtenbasiert

11. Kommunikation via Nachrichtenwarteschlangen funktioniert...
    (Nur eine Antwort ist korrekt!)

    ☐ speicherbasiert ☐ nachrichtenbasiert

12. Kommunikation via gemeinsamen Speichersegmenten funktioniert...
(Nur eine Antwort ist korrekt!)

☐ speicherbasiert ☐ nachrichtenbasiert

13. Kommunikation via Sockets funktioniert...
(Nur eine Antwort ist korrekt!)

☐ speicherbasiert ☐ nachrichtenbasiert

14. Geben Sie an, welche zwei Formen der Interprozesskommunikation bidirektional funktionieren.

☐ Gemeinsame Speichersegmente ☐ Nachrichtenwarteschlangen
☐ Anonyme Pipes ☐ Benannte Pipes
☐ Sockets

15. Geben Sie an, welche Form der Interprozesskommunikation nur zwischen Prozessen funktioniert die eng verwandt sind.

☐ Gemeinsame Speichersegmente ☐ Nachrichtenwarteschlangen
☐ Anonyme Pipes ☐ Benannte Pipes
☐ Sockets

16. Geben Sie an, welche Form der Interprozesskommunikation über Rechnergrenzen hinweg funktioniert.

☐ Gemeinsame Speichersegmente ☐ Nachrichtenwarteschlangen
☐ Anonyme Pipes ☐ Benannte Pipes
☐ Sockets

17. Geben Sie an, bei welcher Form der Interprozesskommunikation die Daten auch ohne gebundenen Prozess erhalten bleiben.

☐ Gemeinsame Speichersegmente ☐ Nachrichtenwarteschlangen
☐ Anonyme Pipes ☐ Benannte Pipes
☐ Sockets

18. Geben Sie an, bei welcher Form der Interprozesskommunikation das Betriebssystem nicht die Synchronisierung garantiert.

☐ Gemeinsame Speichersegmente ☐ Nachrichtenwarteschlangen
☐ Anonyme Pipes ☐ Benannte Pipes
☐ Sockets

Aufgabe 4 (Kooperation von Prozessen)

1. Beschreiben Sie was eine Semaphore ist und ihren Einsatzzweck.

2. Geben Sie die beiden Zugriffsoperationen auf eine Semaphore an.
   Gesucht sind die Bezeichnungen und eine (kurze) Beschreibung der Funktionsweise.

3. Beschreiben Sie den Unterschied zwischen Semaphoren und Blockieren (Sperren und Freigeben).

4. Beschreiben Sie was eine binäre Semaphore ist.

5. Beschreiben Sie was ein Mutex ist und seinen Einsatzzweck.

6. Geben Sie an, welche Form der Semaphoren die gleiche Funktionalität wie der Mutex.

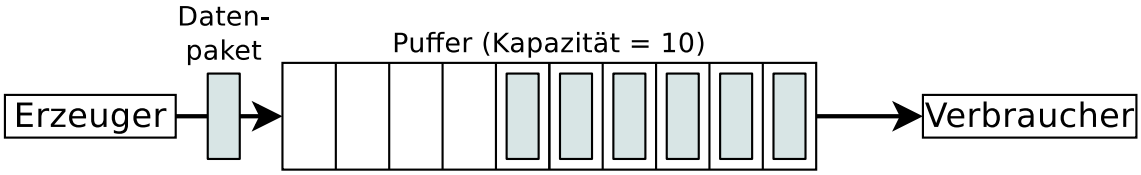
7. Geben Sie die möglichen Zustände eines Mutex an.

8. Geben Sie das Linux/UNIX-Kommando an, das Informationen zu bestehenden gemeinsamen Speichersegmenten, Nachrichtenwarteschlangen und Semaphoren liefert.

9. Geben Sie das Linux/UNIX-Kommando an, das es ermöglicht, bestehende gemeinsame Speichersegmente, Nachrichtenwarteschlangen und Semaphoren zu löschen.

Aufgabe 5 (Erzeuger/Verbraucher-Szenario)

Ein Erzeuger soll Daten an einen Verbraucher schicken. Ein endlicher Zwischenspeicher (Puffer) soll die Wartezeiten des Verbrauchers minimieren. Daten werden vom Erzeuger in den Puffer gelegt und vom Verbraucher aus diesem entfernt. Gegenseitiger Ausschluss ist nötig, um Inkonsistenzen zu vermeiden. Ist der Puffer voll, muss der Erzeuger blockieren. Ist der Puffer leer, muss der Verbraucher blockieren.



Synchronisieren Sie die beiden Prozesse, indem Sie die nötigen Semaphoren erzeugen, diese mit Startwerten versehen und Semaphor-Operationen einfügen.

```
typedef int semaphore;



void erzeuger (void) {
  int daten;
  while (TRUE) {                  // Endlosschleife
    erzeugeDatenpaket(daten);     // erzeuge Datenpaket


    einfuegenDatenpaket(daten);   // Datenpaket in Puffer schreiben


  }
}

void verbraucher (void) {
  int daten;
  while (TRUE) {                  // Endlosschleife


    entferneDatenpaket(daten);    // Datenpaket aus Puffer holen


    verbraucheDatenpaket(daten); // Datenpaket nutzen
  }
}
```

Aufgabe 6 (Semaphoren)

In einer Lagerhalle werden ständig Pakete von einem Lieferanten angeliefert und von zwei Auslieferern abgeholt. Der Lieferant und die Auslieferer müssen dafür ein Tor durchfahren. Das Tor kann immer nur von einer Person durchfahren werden. Der Lieferant bringt mit jeder Lieferung 3 Pakete zum Wareneingang. An der Ausgabe holt ein Auslieferer jeweils 2 Pakete ab, der andere Auslieferer 1 Paket.

```
Lieferant                  Auslieferer_X              Auslieferer_Y
{                          {                          {
  while (TRUE)               while (TRUE)               while (TRUE)
  {                          {                          {

    <Tor durchfahren>;         <Tor durchfahren>;         <Tor durchfahren>;


    <Wareneingang betreten>;   <Warenausgabe betreten>;   <Warenausgabe betreten>;


    <3 Pakete entladen>;       <2 Pakete aufladen>;       <1 Paket aufladen>;


    <Wareneingang verlassen>; <Warenausgabe verlassen>; <Warenausgabe verlassen>;


    <Tor durchfahren>;         <Tor durchfahren>;         <Tor durchfahren>;

    }                          }                          }
}                          }                          }
```

Es existiert genau ein Prozess `Lieferant`, ein Prozess `Auslieferer_X` und ein Prozess `Auslieferer_Y`.

Synchronisieren Sie die beiden Prozesse, indem Sie die nötigen Semaphoren erzeugen, diese mit Startwerten versehen und Semaphor-Operationen einfügen.

Folgende Bedingungen müssen erfüllt sein:

- Es darf immer nur ein Prozess das Tor durchfahren.
- Es darf immer nur einer der beiden Auslieferer die Warenausgabe betreten.
- Es soll möglich sein, dass der Lieferant und ein Auslieferer gleichzeitig Waren entladen bzw. aufladen.
- Die Lagerhalle kann maximal 10 Pakete aufnehmen.
- Es dürfen keine Verklemmungen auftreten.
- Zu Beginn sind keine Pakete in der Lagerhalle vorrätig und das Tor, der Wareneingang und die Warenausgabe sind frei.

Quelle: TU-München, Übungen zur Einführung in die Informatik III, WS01/02

Aufgabe 7 (Interprozesskommunikation)

Entwickeln Sie einen Teil eines Echtzeitsystems, das aus vier Prozessen besteht:

1. **Conv**. Dieser Prozess liest Messwerte von A/D-Konvertern (Analog/Digital) ein. Er prüft die Messwerte auf Plausibilität und konvertiert sie gegebenfalls. Wir lassen Conv in Ermangelung eines physischen A/D-Konverters Zufallszahlen erzeugen. Diese müssen in einem bestimmten Bereich liegen, um einen A/D-Konverter zu simulieren.

2. **Log**. Dieser Prozess liest die Messwerte des A/D-Konverters (Conv) aus und schreibt sie in eine lokale Datei.

3. **Stat**. Dieser Prozess liest die Messwerte des A/D-Konverters (Conv) aus und berechnet statistische Daten, unter anderem Mittelwert und Summe.

4. **Report**. Dieser Prozess greift auf die Ergebnisse von Stat zu und gibt die statistischen Daten in der Shell aus.

Bezüglich der Daten in den gemeinsamen Speicherbereichen gelten als Synchronisationsbedingungen:

- **Conv** muss erst Messwerte schreiben, bevor **Log** und **Stat** Messwerte auslesen können.

- **Stat** muss erst Statistikdaten schreiben, bevor **Report** Statistikdaten auslesen kann.

Entwerfen und implementieren Sie das Echtzeitsystem in C mit den entsprechenden Systemaufrufen und realisieren Sie den Datenaustausch zwischen den vier Prozessen einmal mit **Pipes**, **Message Queues** und **Shared Memory mit Semaphore**. Am Ende der praktischen Übung müssen drei Implementierungsvarianten des Programms existieren. Der Quellcode soll durch Kommentare verständlich sein.

Vorgehensweise

Die Prozesse Conv, Log, Stat, und Report sind parallele Endlosprozesse. Schreiben Sie ein Gerüst zum Start der Endlosprozesse mit dem Systemaufruf `fork`. Überwachen Sie mit geeigneten Kommandos wie `top`, `ps` und `pstree` Ihre parallelen Prozesse und stellen Sie die Vater-Kindbeziehungen fest.

Das Programm kann mit der Tastenkombination `Ctrl-C` abgebrochen werden. Dazu müssen Sie einen Signalhandler für das Signal `SIGINT` implementieren. Beachten Sie bitte, dass beim Abbruch des Programms alle von den Prozessen belegten Betriebsmittel (Pipes, Message Queues, gemeinsame Speicherbereiche, Semaphoren) freigegeben werden.

Entwickeln und implementieren Sie die drei Varianten, bei denen der Datenaustausch zwischen den vier Prozessen einmal mit **Pipes**, **Message Queues** und **Shared Memory mit Semaphore** funktioniert.

Überwachen Sie die Message Queues, Shared Memory Bereiche und Semaphoren mit dem Kommando `ipcs`. Mit `ipcrm` können Sie Message Queues, Shared Memory Bereiche und Semaphoren wieder freigeben, wenn Ihr Programm dieses bei einer inkorrekten Beendigung versäumt hat.

Aufgabe 8 (Shell-Skripte, Datenkompression)

1. Schreiben Sie ein Shell-Skript, dass eine Datei `testdaten.txt` erzeugt.
   - Die Datei soll mit Nullen gefüllt werden.
   - Die Nullen liefert die virtuelle Gerätedatei `/dev/zero`.
     (Beispiel: `dd if=/dev/zero of=/pfad/zur/datei bs=512 count=1`
   - Die Dateigröße soll mindestens 128 und maximal 512 kB sein.
   - Wie groß die Datei wird, soll mit `RANDOM` zufällig festgelegt werden.
2. Schreiben Sie ein Shell-Skript, das als Kommandozeilenargument einen Dateinamen einliest.
   - Die Datei soll das Shell-Skript dahingehend untersuchen, ob es sich um eine Datei, einen Link oder ein Verzeichnis handelt.
   - Wenn es sich um eine Datei handelt, soll der Benutzer mit Hilfe von `select` folgende Auswahlmöglichkeiten haben:

     ```
     1) ZIP
     2) ARJ
     3) RAR
     4) GZ
     5) BZ2
     6) Alle
     7) Beenden
     ```

   - Wählt der Benutzer einen Kompressionsalgorithmus, soll mit diesem die Datei komprimiert werden und der Dateiname entsprechend angepasst werden. Die Dateigröße der originalen und der komprimierten Datei soll das Skript zum Vergleich ausgeben. z.B:

     ```
     Testdatei.txt        <Dateigröße>
     Testdatei.txt.rar    <Dateigröße>
     ```

- Wählt der Benutzer die Auswahlmöglichkeit (`Alle`), soll das Skript die Datei mit allen Kompressionsalgorithmen komprimieren und die Dateigrößen der originalen und der komprimierten Dateien zum Vergleich ausgeben.

  ```
  Testdatei.txt        <Dateigröße>
  Testdatei.txt.zip    <Dateigröße>
  Testdatei.txt.arj    <Dateigröße>
  Testdatei.txt.rar    <Dateigröße>
  Testdatei.txt.gz     <Dateigröße>
  Testdatei.txt.bz2    <Dateigröße>
  ```

3. Testen Sie das Shell-Skript mit der generierten Datei `testdaten.txt`. Was ist das Ergebnis?

Aufgabe 9 (Shell-Skripte, Datei-Browser)

Schreiben Sie ein Shell-Skript, das via `select` einen Datei-Browser realisiert.

- Die Liste der Dateien und Verzeichnisse im aktuellen Verzeichnis soll ausgegeben und die einzelnen Einträge sollen auswählbar sein.
- Wird eine Datei ausgewählt, soll der Dateiname mit Endung, die Anzahl der Zeichen, Wörter und Zeilen sowie eine Information über den Inhalt der Datei ausgegeben werden. z.B:

  ```
  <Dateiname>.<Dateiendung>
  Zeichen: <Anzahl>
  Zeilen:  <Anzahl>
  Wörter:  <Anzahl>
  Inhalt:  <Angabe>
  ```

  Informationen zur Anzahl der Zeichen, Wörter und Zeilen einer Datei liefert das Kommando `wc`. *Information über den Inhalt einer Datei liefert das Kommando* `file`.

- Wird ein Verzeichnis ausgewählt, soll das Skript in dieses Verzeichnis wechseln und die Dateien und Verzeichnisse im Verzeichnis ausgeben.
- Es soll auch möglich sein, im Verzeichnisbaum nach oben zu gehen (`cd ..`).