Aufgabe 1: Ankreuzfragen (16 Punkte)

In dieser Aufgabe sind jeweils m Aussagen angegeben. Davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie jeweils an, ob die entsprechende Aussage richtig oder falsch ist.
Jede korrekte Antwort gibt einen halben Punkt, jede falsche Antwort einen halben Minuspunkt. Nicht beantwortetete Aussagen gehen neutral in die Bewertung ein. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Bereich: Betriebssysteme allgemein

Richtig	Falsch	
☐	☐	Multiplexing und Isolation der Hardwareressourcen sind Kernaufgaben eines Betriebssystems.
☐	☐	Das Betriebssystem stellt eine virtuelle Maschine bereit, die von der Komplexität der Hardware abstrahiert.
☐	☐	Virtuelle Hardwareressourcen werden durch Schutzmechanismen (räumlich und zeitlich) voneinander isoliert.
☐	☐	Eine atomare Aktion ist eine primitive oder komplexe Aktion, deren Einzelschritte nach außen sichtbar nur im Verbund stattfinden.

b) Bereich: POSIX-Systemaufrufe

Richtig	Falsch	
☐	☐	Das Programm mit dem an `exec()` übergebene Programmpfad wird durch den aktuellen Prozess ausgeführt.
☐	☐	Der `pipe()`-Systemaufruf erzeugt einen bidirektionalen Kommunikationskanal.
☐	☐	Ein durch `fork()` erzeugter Prozess erbt alle Ressourcen des Elternprozesses.
☐	☐	Durch Ausführen eines Programms als Administrator (root) gelangt man in den privilegierten Modus des Prozessors.

c) Bereich: Adressräume

Richtig	Falsch	
☐	☐	Das Betriebssystem kann in die Adressräume aller Prozesse schreibend zugreifen.
☐	☐	Seitenadressierung (paging) ermöglicht es, die gleiche logische Adresse in unterschiedlichen logischen Adressräumen auf gleiche physikalische Adressen im realen Adressraum mit unterschiedlichen Zugriffsrechten abzubilden.
☐	☐	Externe Fragmentierung kann mit Verschmelzung und Kompaktifizierung durch die Anwendung aufgelöst werden.
☐	☐	Der reale Adressraum kann größer sein als der logische Adressraum.

d) Bereich: Dateisysteme

Richtig	Falsch	
☐	☐	Verzeichnisse sind spezielle Dateien des Dateisystems, die Namen an Dateiobjekte binden.
☐	☐	In einem UNIX-Dateisystem sind Dateiobjekte stets in einer Baumstruktur angeordnet.
☐	☐	Verzeichnisse definieren den Kontext für die (hierarchische) Namensauflösung.
☐	☐	Ein Dateideskriptor repräsentiert eine prozesslokale Zugriffsbefähigung auf eine Datei.

e) Bereich: UNIX-Prozesse

Richtig	Falsch	
☐	☐	Die Instruktionen der Ebene E_3 sind immer eine Obermenge der Ebene E_2.
☐	☐	Das Betriebssystem interpretiert den gesamten Programmcode des Programms.
☐	☐	Ein Prozess wird durch seine Elter-Prozess-ID identifiziert.
☐	☐	Ein Prozess im Zustand “beendet” (Zombie) kann mit dem Systemaufruf `respawn()` neu gestartet werden.

f) Bereich: Kommunikation, Signale und Fernaufrufe

Richtig	Falsch	
☐	☐	Ein asynchroner Auftrag blockiert den Sender bis zum Eintreffen des Ergebnisses der Berechnung.
☐	☐	Der Sender einer synchronen Meldung erhält eine Empfangsquittung.
☐	☐	Fernaufrufe ermöglichen vollständige Ortstransparenz von Client und Server.
☐	☐	Ein Prozess kann für jedes mögliche Signal eine Behandlungsfunktionen registrieren.

g) Bereich: Traps und Interrupts

Richtig	Falsch	
☐	☐	Ein Interrupt wird immer unmittelbar durch eine Aktivität des aktuell laufenden Prozesses ausgelöst.
☐	☐	Ein Trap führt zwingend zum Abbruch des laufenden Prozesses, da dieser einen schwerwiegenden Fehler darstellt.
☐	☐	Speicherzugriffe und Rechenoperationen können einen Trap auslösen.
☐	☐	Ein Interrupt führt zu der Unterbrechung des normalen Programmflusses

h) Scheduling

Richtig	Falsch	
☐	☐	Scheduling-Ziele können in der Regel nicht alle gleichzeitig erreicht werden.
☐	☐	Präemptives Scheduling ermöglicht es, die Monopolisierung der CPU zu verhindern.
☐	☐	Kooperatives Scheduling und Mehrprogrammbetrieb schließen sich gegenseitig aus.
☐	☐	Federgewichtige Prozesse (user-threads) können die Multiprozessorfähigkeit des Betriebssystems ausnutzen.

Aufgabe 2: Programmieraufgabe – Wetterstation (17.5 Punkte)

Sie arbeiten in einem Unternehmen, welches Wetterstationen herstellt. Diese übermitteln ihre Information im Klartext an die Auswertungssoftware. Um die Wetterstation eines Konkurrenten an ihre Auswertungssoftware anzubinden, wird ein Adapter benötigt, der mit Hilfe eines Übersetzers zwischen den Formaten übersetzt. Der Übersetzer ist bereits als fertiges Programm vorhanden. Er liest und schreibt auf die Standard Ein- und Ausgabekanäle.

Schreiben Sie die C-Funktion `adapter_plugin`, welche als Adapter eingesetzt werden kann. Diese soll das angegebene Übersetzerprogramm so starten, dass dieses die übergebenen Dateiedeskriptor als Ein- und Ausgabe verwendet. Tritt im Programmfluss ein Fehler auf, der nicht über den Rückgabewert übermittelt werden kann, so soll sich das Programm mittels der Funktion `die()` beenden. Alle nicht verwendeten Dateideskriptoren sollen geschlossen werden.

```
pid_t adapter_plugin(int in, int out, const char* translator)
```

`in` – Der Dateideskriptor, den der Übersetzer als Eingabe erhalten soll

`out` – Der Dateideskriptor, den der Übersetzer als Ausgabe erhalten soll

`translator` – Pfad des Übersetzer-Programms

`return` – Im Erfolgsfall die Prozess-ID des Übersetzers, -1 im Fehlerfall und `errno` gesetzt

Schreiben Sie die C-Funktion `adapter_unplug`, die nach Bearbeitung der Messdaten aufgerufen wird. Sie soll den Übersetzer beenden und aufräumen.

```
int adapter_unplug(pid_t translator_pid)
```

`translator_pid` – Prozess-ID des Übersetzers

`return` – 0 im Erfolgsfall, 1 im Fehlerfall und `errno` gesetzt

```c
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
#include<stdlib.h>

// Gegeben:

// Beenden mit Fehlerausgabe und Fehlerstatus
void die(char *msg);

//Zu implementieren:

// Den Übersetzer starten
pid_t adapter_plugin(int in, int out, char* translator);

// Übersetzer beenden und aufräumen
int adapter_unplug(pid_t translator_pid);
```

close(2) close(2)

NAME
close – close a file descriptor

SYNOPSIS
#include <unistd.h>

int close(int *fd***);**

DESCRIPTION
close() closes a file descriptor, so that it no longer refers to any file and may be reused.

RETURN VALUE
close() returns zero on success. On error, −1 is returned, and *errno* is set appropriately.

---

dup(2) dup(2)

NAME
dup, dup2 – duplicate a file descriptor

SYNOPSIS
#include <unistd.h>

int dup(int *oldfd***);**
int dup2(int *oldfd***, int** *newfd***);**

DESCRIPTION
The **dup**() system call duplicates a file descriptor.
The **dup2**() system call replaces the file descriptor specified in *newfd* with a copy of *oldfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused.

The steps of closing and reusing the file descriptor *newfd* are performed *atomically*.

RETURN VALUE
On success: the new file descriptor. On error, −1 is returned, and *errno* is set appropriately.

---

kill(2) kill(2)

NAME
kill – send signal to a process

SYNOPSIS
#include <sys/types.h>
#include <signal.h>

int kill(pid_t *pid***, int** *sig***);**

DESCRIPTION
The **kill**() system call can be used to send any signal to any process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*. If *pid* equals −1, then *sig* is sent to every process for which the calling process has permission to send signals.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed.

RETURN VALUE
On success (at least one signal was sent), zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS
EINVAL An invalid signal was specified.
EPERM The process does not have permission to send the signal to any of the target processes.
ESRCH The process or process group does not exist.

---

perror(3) perror(3)

NAME
perror – print a system error message

SYNOPSIS
#include <stdio.h>

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

#include <errno.h>

int *errno***;**

DESCRIPTION
The **perror**() function produces a message on standard error describing the last error encountered during a call to a system or library function.

First (if *s* is not NULL and **s* is not a null byte ('\0')), the argument string *s* is printed, followed by a colon and a blank. Then an error message corresponding to the current value of *errno* and a new-line.

When a system call fails, it usually returns −1 and sets the variable *errno* to a value describing what went wrong.

---

exec(3) Linux Programmer's Manual exec(3)

NAME
execl, execlp, execv, execvp – execute a file

SYNOPSIS
#include <unistd.h>

int execl(const char ****pathname*, const char ****arg***, ... , NULL */);**
int execlp(const char * *file***, const char ****arg***, ... , NULL);**
int execv(const char * *pathname***, char *const** *argv***[]);**
int execvp(const char * *file***, char *const** *argv***[]);**

DESCRIPTION
The **exec**() family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

l - execl(), execlp()
The *const char *arg* and subsequent ellipses can be thought of as *arg0*, *arg1*, ..., *argn*. The list of arguments *must* be terminated by a null pointer.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

v - execv(), execvp()
The *char *const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

p - execlp(), execvp()
These functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character.

RETURN VALUE
The **exec**() functions return only if an error has occurred. The return value is −1, and *errno* is set to indicate the error.

---

fork(2) fork(2)

NAME
fork – create a child process

SYNOPSIS
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

DESCRIPTION
fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID.

* The child's parent process ID is the same as the parent's process ID.

RETURN VALUE
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, −1 is returned in the parent, no child process is created, and *errno* is set appropriately.

---

wait(2) wait(2)

NAME
wait, waitpid – wait for process to change state

SYNOPSIS
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int ****wstatus*);**

pid_t waitpid(pid_t *pid***, int ****wstatus***, int** *options***);**

DESCRIPTION
All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

wait() and waitpid()
The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

waitpid(−1, &wstatus, 0);

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

−1 meaning wait for any child process.

> 0 meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus***)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*wstatus***)**
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

RETURN VALUE
wait(): on success, returns the process ID of the terminated child; on error, −1 is returned. If no unwaited-for children exist, −1 is returned and *errno* is set to **ECHILD** ;

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On error, −1 is returned.

Each of these calls sets *errno* to an appropriate value in the case of an error.

```
pid_t adapter_plugin(int in, int out, char* translator) {
```


MS:


```
int adapter_unplug(pid_t translator_pid) {
```

MC:

Aufgabe 3: Synchronisation (14 Punkte)

Der Zugriff auf eine Datenbank soll geschützt werden. Aus konsistenzgründen dürfen hierbei beliebig viele Leser zeitgleich zugreifen, jedoch niemals mehr als ein Schreiber und keine Leser und Schreiber gleichzeitig.

Ergänzen sie die Funktionen `reader()` und `writer` um die nötigen Synchronisationsoperationen, um dieses Schema zu manifestieren.

Verwenden Sie dazu

- den Semaphor **`db_mutex`** um den Zugriff auf die Datenbank zwischen Lesern und Schreiber zu koordinieren.
- die Variable **`active_readers`** um zu zählen, wieviele Leser gerade aktiv auf der Datenbank lesen.
- den Semaphor **`ar_mutex`** um den Zugriff auf `active_readers` zwischen den Lesern zu koordinieren.

Es kann angenommen werden, dass alle Zugriffe über die gegebenen Funktionen durchgeführt werden. Initialisierungen sollen in der `init()`-Funktion durchgeführt werden. Leser sollen gegenüber Schreibern bevorzugt werden.

sem_destroy(3) sem_destroy(3)

NAME
sem_destroy – destroy a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_destroy(sem_t ****sem*);**

DESCRIPTION
sem_destroy() destroys the semaphore at the address pointed to by *sem*.

Destroying a semaphore that other processes or threads are currently blocked on (in **sem_wait**(3)) produces undefined behavior.

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using **sem_init**(3).

RETURN VALUE
sem_destroy() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

---

sem_getvalue(3) sem_getvalue(3)

NAME
sem_getvalue – get the value of a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_getvalue(sem_t ****sem*, int ****sval***);**

DESCRIPTION
sem_getvalue() places the current value of the semaphore pointed to *sem* into the integer pointed to by *sval*.

RETURN VALUE
sem_getvalue() returns 0 on success; on error, −1 is returned and *errno* is set appropriately.

---

sem_init(3) sem_init(3)

NAME
sem_init – initialize an unnamed semaphore

SYNOPSIS
#include <semaphore.h>

int sem_init(sem_t ****sem*, int** *pshared***, unsigned int** *value***);**

DESCRIPTION
sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process (0), or between processes (1). Initializing a semaphore that has already been initialized results in undefined behavior.

RETURN VALUE
sem_init() returns 0 on success; on error, −1 is returned, and *errno* is set appropriately.

---

sem_post(3) sem_post(3)

NAME
sem_post – unlock a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_post(sem_t ****sem*);**

DESCRIPTION
sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a **sem_wait**(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE
sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

---

sem_wait(3) sem_wait(3)

NAME
sem_wait, sem_timedwait – lock a semaphore

SYNOPSIS
#include <semaphore.h>

int sem_wait(sem_t ****sem*);**
int sem_trywait(sem_t ****sem*);**

DESCRIPTION
sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

RETURN VALUE
on success: 0; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS
EINTR The call was interrupted by a signal handler
EINVAL *sem* is not a valid semaphore.
EAGAIN The operation could not be performed without blocking (sem_trywait() only).

```
sem_t db_mutex;
sem_t ar_mutex;
int active_readers;

void init(char *path) {
    [                                    ];
    [                                    ];
    [                                    ];
}

void reader() {
    while(1) {
        my_data data;
        [                                    ];
        active_readers++;
        if (active_readers == 1) {
            [                                    ];
        }
        [                                    ];
        data = do_read(); // kritisches lesen
        [                                    ];
        active_readers--;
        if (active_readers == 0) {
            [                                    ];
        }
        [                                    ];
        process_read_data(data); // unkritisches verarbeiten
    }
}

void writer(void) {
    while(1) {
      create_data(); // unkritisches generieren
        [                                    ];
        do_write(); // kritisches schreiben
        [                                    ];
    }
}
```


b) Kann bei diesem Zugriffsprotokoll ein Problem auftreten? Wenn ja welches? Wie könnte man dieses verbessern? Begründen Sie stichwortartig.

Aufgabe 4: Virtueller Speicher (14 Punkte)

Auf einem Byte-adressierten Mikrocontroller ist seitenorientierter logischer Adressraum implementiert. Die 20 Bit breiten Adressen sind in 4-Bit Seitennummer und 16-Bit Offset geteilt. Es sind 8 Bit für Attribute im Seitendeskriptor vorgesehen.

a) Vervollständigen Sie die gegebene Skizze zur Abbildung auf eine von Ihnen gewählte reale Adresse aus der gegebenen logischen Adresse `0x2b01c`.

Logische Adresse	2	b	0	1	c

Reale Adresse					

b) Bestimmen Sie die folgenden Größen: Größe einer Seitentabelle; Größe einer Seite; maximale Größe des logischen Adressraums

c) Nennes Sie ein alternatives Verfahren, um einen logischen Adressraum zu implementieren. Beschreiben Sie einen Vor/Nachteil gegenüber der seitenorientierten Implementierung.

d) Beschreiben Sie stichwortartig den Ablauf, wenn eine Anwendung auf momentan in den Hintergrundspeicher ausgelagerte Daten des virtuellen Adressraums zugreift.

Aufgabe 5: Textfragen (15 Punkte)

a) Kann es bei dem hier gegebenen Beispiel zu einer Verklemmung kommen? Begründen Sie stichwortartig unter Verwendung der Bedingungen für einen Deadlock.

```
void * go_stitching(void* param) {
  int me = (int) param;
  for(;;) {
    sem_wait(needles[me]);
    sem_wait(needles[(me+1)%N]);
    stitch();
    sem_post(needles[(me+1)%N]);
    sem_post(needles[me]);
  }

int N = 5;
sem_t needles[N];
void main() {
  for (int i=0; i < N; ++i) sem_init(&needles[i], 1);
  for (int i=0; i < N; ++i) pthread_create(NULL, NULL, go_stitching, i);
  ...
}
```

b) Wie könnte in dem Beispiel der vorherigen Teilaufgabe die Verklemmung verhindert werden? Skizzieren Sie eine Lösung und begründen Sie stichwortartig.

c) Warum dürfen nicht alle Bibliotheksfunktionen gefahrlos im Zusammenhang mit Signalen verwendet werden? Begründen Sie stichwortartig und nennen Sie ein Beispiel.

d) Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 4 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 8 Byte groß, ein Block 4 KiByte. Wieviele Blöcke werden benötigt, um je eine Datei der Größe 8 KiByte und 23 KiByte darzustellen? Wie werden die Datenblöcke jeweils adressiert?

Aufgabe 6: Prozesse (13.5 Punkte)

a) Beschreiben Sie die Zustände eines Prozesses während der Einplanung, sowie die Ereignisse und die daraus resultierenden expliziten und impliziten Übergänge. (Sizze mit kurzer Benennung und klassifizierung der Zustände und Übergänge).

b) Erklären Sie folgende Begriffe stichwortartig und setzen Sie diese in Beziehung zueinander: Prozess, Betriebssystem, Adressraum