Technische Universität Braunschweig

Klausur “Betriebssysteme”

(Name)	(Vorname)	(Matrikel-Nr.)
(HBK Matrikel-Nr.)	(Studiengang)	(Semester)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (20 Seiten inklusive Deckblatt),
- den Empfang der Manualseiten sem_init.3, sem_wait.3, sem_post.3, sem_get_value.3, sem_destroy.3, fork.2, open.2, exec.2, dup2.2, wait.2,
- die Kenntnisnahme der Hinweise auf Seite 2.

Braunschweig, 10.02.2025

(Unterschrift)

Hinweise

Bitte lesen Sie die folgenden Informationen aufmerksam und unterschreiben Sie die Erklärung auf der ersten Seite.

- Bearbeitungszeit: 90 Minuten.
- Zugelassene Hilfsmittel: Nicht-programmierbarer Taschenrechner; doppelseitig mit der Hand beschriebenes DIN A4 Blatt als “Spickzettel”.
- Manpages: Dienen nur als Orientierung, sind nicht vollständig und enthalten auch Funktionen, die nicht benötigt werden.
- Für die Aufgaben 2 bis 7 gibt es inoffizielle **Übersetzungen** ins Englische. Sie dienen nur zur Orientierung, maßgeblich ist die deutsche Aufgabenstellung!
- Die Lösung einer Aufgabe ist im dafür vorgesehenen Raum auf dem Aufgabenblatt zu schreiben. Der freigelassene Platz ist großzügig und entspricht nicht immer der erwarteten Antwortlänge. Bei Platzmangel können Rückseiten und Leerflächen genutzt werden. Kennzeichnen Sie die Zuordnung Ihrer Lösung zur jeweiligen Aufgabe. Zusätzliche weiße Lösungsblätter können bei Bedarf ausgegeben werden, bitte unbedingt den Klausurcode darauf vermerken!
- Die Lösungen müssen dokumentenecht in blau oder schwarz geschrieben werden. Als falsch Erkanntes muss deutlich durchgestrichen werden. Tintenkiller und andere Korrekturstifte dürfen nicht verwendet werden. Keinen Bleistift verwenden!
- Schmierpapier (farbige Blätter) darf **nicht** abgegeben werden. Bei Bedarf ist von der Aufsicht weiteres Schmierpapier (farbig) erhältlich.
- Fragen zu den Prüfungsaufgaben können grundsätzlich **nicht** beantwortet werden.
- Die Seiten 13, 14, 21, 22, 23 und 24 dürfen herausgetrennt werden. Bitte schreiben Sie keine Lösungen auf herausgetrennte Seiten. Alle anderen Seiten dürfen **nicht** herausgetrennt werden.
- Sie dürfen den Raum nicht verlassen, bevor Ihre Personalien überprüft wurden und Sie die Klausurunterlagen der Aufsicht zurückgegeben haben.
- Um Unruhe und die Störung Ihrer Mitstudierenden zu vermeiden, ist die vorzeitige Abgabe der Klausur ausgeschlossen. Bleiben Sie an Ihrem Platz sitzen, bis am Ende alle Klausurunterlagen eingesammelt sind und die Aufsicht das Zeichen zum Gehen gibt.

☐ **Aktion**: Tragen Sie Ihren Namen und Vornamen, Ihre Matrikelnummer, Studiengang und Fachsemesterzahl auf dem Deckblatt der Klausur ein.

☐ **Aktion**: Überprüfen Sie die Anzahl der Blätter Ihrer Klausur und unterschreiben Sie die Erklärung auf dem Deckblatt.

☐ **Aktion**: Bitte legen Sie Ihren gültigen Studierenden- und Lichtbildausweis zur Kontrolle bereit.

Korrektur – Zusammenfassung

	erreichbare Punkte	erhaltene Punkte				
Aufgabe 1	13					
Aufgabe 2	8	**1**	**2**			
Aufgabe 3	16	**M**	**MA**			
Aufgabe 4	10.5	**1**	**2**			
Aufgabe 5	28.5	**S**	**O**	**L**	**M**	
Aufgabe 6	7					
Aufgabe 7	7					
Summe	90					

Aufgabe 1: Ankreuzfragen (13 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 0,5 Punkte, jede falsche Antwort 0,5 Punkte Abzug. Nicht beantwortete 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 waagerechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

(A1.1) Bewerten Sie folgende Aussagen zu Betriebssystemen

2 Punkte-

Richtig	Falsch	
☐	☐	Das Betriebssystem ist ein Übersetzer von E_3-Programmen nach E_2-Programmen.
☐	☐	Eine atomare Aktion ist eine primitive oder komplexe Aktion, deren Einzelschritte nach außen sichtbar nur im Verbund stattfinden.
☐	☐	Multiplexing der Hardwareressourcen ist eine Kernaufgaben eines Betriebssystems.
☐	☐	Virtuelle Hardwareressourcen werden durch Schutzmechanismen (räumlich und zeitlich) voneinander isoliert.

(A1.2) Bewerten Sie folgende Aussagen zu Adressräumen

2 Punkte-

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 abzubilden.
☐	☐	Externe Fragmentierung kann die Anwendung durch Verschmelzung und Kompaktifizierung selbst lösen.
☐	☐	Der reale Adressraum kann größer sein als der logische Adressraum.

(A1.3) Bewerten Sie folgende Aussagen zu Dateisystemen

2 Punkte-

Richtig	Falsch	
☐	☐	Verzeichnisse sind spezielle Dateien, die Namen an Dateiobjekte binden.
☐	☐	Ein UNIX-Dateisystem ordnet seine Dateiobjekte stets in einer Baumstruktur an.
☐	☐	Die Zugriffsrechte eines Dateiobjekts wird im Verzeichnis gespeichert.
☐	☐	Im Gegensatz zu Symlinks, können Hardlinks Dateiobjekte auf anderen Dateisystemen referenzieren.

2 Punkte-

(A1.4) Bewerten Sie folgende Aussagen zum 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.
☐	☐	Round-Robin Scheduling ist ein kooperatives Scheduling-Verfahren.

2 Punkte-

(A1.5) Bewerten Sie die folgenden Aussagen zu `fork`.

Richtig	Falsch	
☐	☐	`fork` wird direkt vom Betriebssystem bereitsgestellt.
☐	☐	`fork` kehrt nur im Fehlerfall an die Aufrufstelle zurück.
☐	☐	`fork` führt das angegebene Programm in einem neuen Prozess aus.
☐	☐	`fork` hat drei verschiedene Gruppen von Rückgabewerten.

3 Punkte-

(A1.6) Betrachten Sie folgendes Programmfragment und bewerten Sie die Aussagen:

```
static int a = 0x20190308;
int* foo(int x) {
  char b[] = "Hello ";
  static int c;
  int *e = malloc(800*sizeof(int));
  strcat(b, "Paul");
  return e;
}
```

Richtig	Falsch	
☐	☐	`a` wird von Compiler und Linker in der `.bss`-Sektion abgelegt.
☐	☐	`c` ist uninitialisiert und enthält einen zufälligen Wert.
☐	☐	Die Lebenszeit von `b` endet beim Rücksprung aus `foo`.
☐	☐	`e` liegt auf dem Stack.
☐	☐	`strcat` erzeugt einen Pufferüberlauf und damit undefiniertes Verhalten.
☐	☐	Die Rückgabe von `e` ist ein Fehler, da der Speicher außerhalb der Funktion nicht mehr zugreifbar ist.

Aufgabe 2: Synchronisation, Allgemeines (8 Punkte)

(A2.1) Kann es bei dem hier gegebenen Beispiel zu einer Verklemmung kommen? Begründen Sie stichwortartig unter Verwendung aller Bedingungen für einen Deadlock.

5 Punkte-

Inofficial translation: Can a deadlock occur in the example given here? Give reasons in keywords using all conditions for a deadlock.

```
int N = 5;
sem_t needles[N];

void * go_knitting(void* param) {
  int me = (int) param;
  for(;;) {
    sem_wait(needles[me]);
    sem_wait(needles[(me+1)%N]);
    knit();
    sem_post(needles[(me+1)%N]);
    sem_post(needles[me]);
}
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_knitting, i);
  ...
}
```

(A2.2) Wie könnte in dem Beispiel der vorherigen Teilaufgabe die Verklemmung verhindert werden? Skizzieren Sie eine Lösung und begründen Sie stichwortartig.

3 Punkte-

Inofficial translation: How could a deadlock be prevented in the example from the previous subtask? Sketch a solution and justify it in keywords.

Aufgabe 3: Synchronisation, Reader-Writer Lock (16 Punkte)

Der Zugriff auf die Datenbank muss geschützt werden. Aus Konsistenzgründen sollen beliebig viele Leser gleichzeitig zugreifen können, jedoch darf nie mehr als ein Schreiber aktiv sein, und Leser und Schreiber dürfen nicht gleichzeitig zugreifen.

Verwenden Sie dazu:

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

Alle Zugriffe erfolgen ausschließlich über die vorgegebenen Funktionen. Die Initialisierungen sind in der `init()`-Funktion vorzunehmen. Es wird davon ausgegangen, dass keine Fehler auftreten. Leser sollen gegenüber Schreibern bevorzugt werden.

Ergänzen Sie die Funktionen `reader()` und `writer()` um die notwendigen Synchronisationsoperationen, um dieses Zugriffsmodell sicherzustellen.

Inofficial translation: Access to the database must be protected. For reasons of consistency, any number of readers should be able to access the database at the same time, but never more than one writer may be active, and readers and writers must not have access at the same time. To do this, use

- the **`db_mutex`** semaphore to coordinate access to the database between readers and writers.
- the variable **`active_readers`** to count how many readers are actively reading the database.
- the semaphore **`ar_mutex`** to coordinate access to `active_readers` between the readers.

All accesses are made exclusively via the specified functions. The initializations are in the `init()` function. It is assumed that no errors occur. Readers should be given preference over writers. Complete the `reader()` and `writer()` functions with the necessary synchronization operations to ensure this access model.

11 Punkte-

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

void init(void) {
     ;
     ;
     ;
}

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
        ;
    }
}
```

(A3.2) Analyse des Reader-Writer Locks

5 Punkte-

Das beschriebene Synchronisationsprotokoll weist ein Problem auf. Benennen Sie dieses Problem und ordnen Sie es einer Kategorie (funktional/nicht-funktional) zu. Wie könnte das Problem verbessert werden? Begründen Sie Ihre Lösung stichwortartig!

Inofficial translation: There is a problem with the synchronization protocol described. Name this problem and assign it to a category (functional/non-functional). How could the problem be improved? Explain your solution in keywords!

Aufgabe 4: Prozesse (10,5 Punkte)

(A4.1) Definieren Sie die Begriffe “Programm” und “Prozess” und setzen Sie beide zueinander in Beziehung!

3 Punkte-

Inofficial translation: Define the terms “program” and “process” and relate them to each other!

(A4.2) Beschreiben Sie die Zustände eines Prozesses während des Schedulings, sowie die Ereignisse und die daraus resultierenden expliziten und impliziten Übergänge. (Sizze mit kurzer Benennung und klassifizierung der Zustände und Übergänge).

7,5 Punkte-

Inofficial translation: Describe the states of a process during scheduling, as well as the events and the resulting explicit and implicit transitions. (Sketch a picture with a brief description and classification of the states and transitions).

Aufgabe 5: Programmieraufgabe – parallel (28,5 Punkte)

Ihre Aufgabe ist es das Programm `parallel` zu implementieren, welches die Ausführung mehrerer Jobs als eigenständige Prozesse überwacht und koordiniert. `parallel` liest die auszuführenden Jobs von der Standardeingabe und startet jeden Job als einen eigenständigen Prozess.

Limitierung der aktien Jobs Dabei achtet `parallel` darauf, dass der Computer nicht überlastet wird und niemals mehr als vier (4) Jobs gleichzeitig laufen. Das bedeutet, laufen bereits 4 Prozesse, so wird der nächste erst dann gestartet, wenn einer laufenden Jobs sich beendet hat.

Ausgabeumleitung Jeder Job (`struct job_t`) besteht aus einem Befehl, der als ein eigener Prozess gestartet wird, und einem Dateinamen. Dieser Dateiname wird als Ausgabedatei verwendet. Leiten Sie die Standard-Ausgabe der Jobs in diese Ausgabedatei um Falls die Datei noch nicht existiert, erstellen Sie diese. Falls die Datei bereits existiert, wird die Ausgabe an den bisherigen Inhalt angehängt.

Implementieren sie dazu die genannten Funktionen, um die jeweils beschriebene Funktionalität.

Inofficial translation: Your task is to implement the program `parallel`, which monitors and coordinates the execution of several jobs as independent processes. `parallel` reads the jobs to be executed from the standard input and starts each job as an independent process.

Limiting the number of active jobs `parallel` ensures that the computer is not overloaded and that no more than four (4) jobs run simultaneously. This means that if 4 processes are already running, the next one is only started when one of the running jobs has finished.

Output redirection Each job (`struct job\_t`) consists of a command, which is started as a separate process, and a file name. This file name is used as the output file. Redirect the standard output of the jobs to this output file. If the file does not yet exist, create it. If the file already exists, the output is appended to the previous content.

Implement the described functionality in the following functions.

Diese Seite darf herausgetrennt werden.

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

////////////////////////////////////////////////////////////////
// Gegeben
void die(const char *msg) {
    perror(msg);
    exit(1);
}

typedef struct {
    char *  filename; // Name der Ausgabedatei
    char ** argv;     // geparster Programmaufruf
} job_t;


/* Die Funktion get_render_job liest den nächsten Job
 * aus dem angegebenen Dateideskriptor.
 *
 * Rückgabewert:
 *    -1 bei Fehlern, errno wird entsprechend gesetzt.
 *     0 wenn kein weiterer Job gelesen wurde.
 *     1 Es wurde ein Job gelesen und nach *next_job geschrieben
 */
int get_job(int fd, job_t *next_job);

////////////////////////////////////////////////////////////////
// TODO: Implementieren Sie die folgenden Funktionen
////////////////////////////////////////////////////////////////
void start_job(const job_t *job);         // A5.1
void redirect_output(const job_t *job);   // A5.2
void limit_jobs(int max_jobs);            // A5.3
int  main(int argc, char *argv[]);        // A5.4
int  active_jobs = 0;
```


Diese Seite darf herausgetrennt werden.

(A5.1) Starten eines Jobs (8 Punkte)

Funktionalität: Starten des Jobs als ein eigener Prozess. Die Ausgabe wird durch `redirect_output()` in die Ausgabedatei umgeleitet.

Inofficial translation: Start the job as a separate process. The output is redirected to the output file by `redirect_output()`.

S:

(A5.2) Ausgabeumleitung (5,5 Punkte)

Funktionalität: Umleitung der Standard-Ausgabe des aktuellen Prozesses in die Datei, die in `job` angegeben ist. Falls die Datei bereits besteht, soll die Ausgabe hinten angehängt werden. Andernfalls soll diese mit den Rechten `0644` erstellt werden.

Inofficial translation: Redirects the standard output of the current process to the file specified in `job`. If the file already exists, the output should be appended at the end. Otherwise, it should be created with the access rights `0644`.

O:

(A5.3) Limitierung der Job-Anzahl (6 Punkte)

Funktionalität: Laufen mehr als `max_jobs` Prozesse, wird solange gewartet bis sich genug Kind-Prozess sich beendet haben bis nur noch `max_jobs` Prozesse laufen. Die Anzahl der aktiven Prozesse wird in der globalen Variable `active_jobs` in Zusammenarbeit mit `start_job()` verwaltet.

Inofficial translation: If more than `max_jobs` processes are running, the system waits until enough child processes have terminated until only `max_jobs` processes are still running. The number of active processes is managed in the global variable `active_jobs` in cooperation with `start_job()`.

L:

(A5.4) Funktion main (9 Punkte)

Funktionalität: Vollständiges Abarbeiten aller Jobs: Die existierende Funktion `get_job()` ließt den nächsten Job ein, welcher mit `start_job()` gestartet wird. Durch Aufruf der Funktion `limit_jobs()` wird die Anzahl an laufenden Jobs/Prozessen auf maximal 4 begrenzt. Am Ende wartet der Haupt-Prozess darauf, dass alle Jobs abgeschlossen wurden.

Inofficial translation: Complete processing of all jobs: The existing function `get_job()` starts the next job, which is started with `start_job()`. Calling the function `limit_jobs()` limits the number of running jobs/processes to a maximum of 4. At the end, the main process waits for all jobs to be completed.

M:

Aufgabe 6: Dateisystem (7 Punkte)

Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 6 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 4 Byte groß, ein Block 4 KiByte.

Inofficial translation: Given a file system with indexed storage. Each index node contains 6 direct references, and one single, double and triple indirect reference each. An address is 4 bytes in size, a block 4 KiByte.

(A6.1) Wieviele Blöcke werden benötigt, um eine Datei der Größe 21 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

2 Punkte-

Inofficial translation: How many blocks are needed to represent a file of size 21 KiByte? How are the data blocks addressed?

(A6.2) Wieviele Blöcke werden benötigt, um eine Datei der 56 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

2 Punkte-

Inofficial translation: How many blocks are needed to represent a file of size 56 KiByte? How are the data blocks addressed?

(A6.3) Wie groß kann eine Datei maximal in diesem Dateisystem sein?

3 Punkte-

Inofficial translation: What is the maximum size a file can be in this file system?

Aufgabe 7: Virtueller Speicher (7 Punkte)

Auf einem Byte-Adressierten Mikrocontroller ist seitenorientierter logischer Adressraum implementiert. Die 12 Bit breiten Adressen sind in 4-Bit Seitennummer und 8-Bit Offset geteilt. Es sind 12 Bit für Attribute im Seitendeskriptor vorgesehen.

Inofficial translation: Page-oriented logical address space is implemented on a byte-addressed microcontroller. The 12 bit wide addresses are divided into 4-bit page number and 8-bit offset. There are 12 bit for attributes in the page descriptor.

4 Punkte-

(A7.1) Vervollständigen Sie die gegebene Skizze zur Abbildung einer von Ihnen gewählten realen Adresse aus der gegebenen logischen Adresse `2fc` .

Inofficial translation: Complete the given sketch to map a real address of your choice from the given logical address `2fc` .

Logische Adresse | 2 | f | c |

Reale Adresse | | | |

3 Punkte-

(A7.2) Bestimmen Sie die folgenden Größen: Größe einer Seitentabelle; Größe einer Seite; maximale Größe des logischen Adressraums.

Inofficial translation: Determine the following sizes: size of a page table; size of a page; maximum size of the logical address space.

sem_init(3) sem_init(3)

NAME
sem_init − initialize an unnamed semaphore

SYNOPSIS
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* ar gument 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_init(3) sem_init(3)

NAME
sem_init − ein unbenanntes Semaphore initialisieren

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

BESCHREIBUNG
Die Funktion **sem_init**() initialisiert das unbenannte Semaphore an der Adresse, auf die *sem* zeigt. Das Argument *value* gibt den Anfangswert für das Semaphore an.

Das Argument *pshared* gibt an, ob dieses Semaphore zwischen den Threads eines Prozesses (0) oder zwischen Prozessen (1) geteilt werden soll. Das Initialisieren eines bereits initialisierten Semaphores führt zu undefiniertem Verhalten.

RÜCKGABEWERT
Die Funktion **sem_init**() gibt bei Erfolg 0 zurück; im Fehlerfall wird −1 zurückgegeben, und *errno* entsprechend gesetzt.

sem_wait(3) sem_wait(3)

NAME
sem_wait, sem_timedwait − lock a semaphore

SYNOPSIS
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_wait(3) sem_wait(3)

NAME
sem_wait, sem_timedwait − ein Semaphore sperren

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

BESCHREIBUNG
Die Funktion **sem_wait**() verringert (sperrt) das Semaphore, auf das *sem* zeigt. Wenn der Wert des Semaphores größer als null ist, wird die Verringerung durchgeführt, und die Funktion kehrt sofort zurück. Wenn das Semaphore aktuell den Wert null hat, blockiert der Aufruf, bis entweder die Verringerung möglich ist (d. h. der Semaphore-Wert steigt über null), oder ein Signal-Handler den Aufruf unterbricht.

Die Funktion **sem_trywait**() entspricht **sem_wait**(), außer dass der Aufruf bei fehlender unmittelbarer Möglichkeit zur Verringerung einen Fehler zurückgibt (*errno* wird auf **EAGAIN** gesetzt), anstatt zu blockieren.

RÜCKGABEWERT
Bei Erfolg: 0; im Fehlerfall bleibt der Wert des Semaphores unverändert, −1 wird zurückgegeben, und *errno* wird gesetzt, um den Fehler anzuzeigen.

FEHLER
EINTR Der Aufruf wurde durch einen Signal-Handler unterbrochen.
EINVAL *sem* ist kein gültiges Semaphore.
EAGAIN Die Operation konnte nicht ohne Blockierung durchgeführt werden (nur bei sem_trywait()).

sem_post(3) sem_post(3)

NAME
sem_post − unlock a semaphore

SYNOPSIS
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_post(3) sem_post(3)

NAME
sem_post − ein Semaphore freigeben

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

BESCHREIBUNG
Die Funktion **sem_post**() erhöht (gibt frei) das Semaphore, auf das *sem* zeigt. Wenn der Wert des Semaphores dadurch größer als null wird, wird ein anderer Prozess oder Thread, der in einem **sem_wait**(3) -Aufruf blockiert ist, aufgeweckt und fährt fort, das Semaphore zu sperren.

RÜCKGABEWERT
Die Funktion **sem_post**() gibt bei Erfolg 0 zurück; im Fehlerfall bleibt der Wert des Semaphores unverändert, −1 wird zurückgegeben, und *errno* wird gesetzt, um den Fehler anzuzeigen.

sem_getvalue(3) sem_getvalue(3)

NAME
sem_getvalue − get the value of a semaphore

SYNOPSIS
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_getvalue(3) sem_getvalue(3)

NAME
sem_getvalue − den Wert eines Semaphores abfragen

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

BESCHREIBUNG
Die Funktion **sem_getvalue**() schreibt den aktuellen Wert des Semaphores, auf den *sem* zeigt, in die Ganzzahl, auf die *sval* zeigt.

RÜCKGABEWERT
Die Funktion **sem_getvalue**() gibt bei Erfolg 0 zurück; im Fehlerfall wird −1 zurückgegeben, und *errno* entsprechend gesetzt.

Diese Seite darf herausgetrennt werden.

sem_destroy(3) sem_destroy(3)

NAME
sem_destroy − destroy a semaphore

SYNOPSIS
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_destroy(3) sem_destroy(3)

NAME
sem_destroy − ein Semaphore zerstören

SYNOPSIS
int sem_destroy(sem_t *_sem_);

BESCHREIBUNG
Die Funktion **sem_destroy**() zerstört das Semaphore an der Adresse, auf die *sem* zeigt.

Das Zerstören eines Semaphores, auf dem andere Prozesse oder Threads derzeit blockiert sind (z. B. in **sem_wait**(3)), führt zu undefiniertem Verhalten.

Die Verwendung eines zerstörten Semaphores führt zu undefinierten Ergebnissen, bis das Semaphore mit **sem_init**(3) neu initialisiert wurde.

RÜCKGABEWERT
Die Funktion **sem_destroy**() gibt bei Erfolg 0 zurück; im Fehlerfall wird −1 zurückgegeben, und *errno* wird gesetzt, um den Fehler anzuzeigen.

fork(2) fork(2)

NAME
fork − create a child process

SYNOPSIS
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.

fork(2) fork(2)

NAME
fork − erstellt einen Kind-Prozess

SYNOPSIS
pid_t fork(void);

BESCHREIBUNG
fork() erzeugt einen neuen Prozess, indem der aufrufende Prozess dupliziert wird. Der neue Prozess wird als *Kind* Prozess bezeichnet. Der aufrufende Prozess wird als *Eltern* Prozess bezeichnet.

Der Kind-Prozess ist eine exakte Kopie des Eltern-Prozesses mit Ausnahme der folgenden Punkte:

* Der Kind-Prozess hat eine eigene, einzigartige Prozess-ID.
* Die Eltern-Prozess-ID des Kind-Prozesses ist die gleiche wie die des Eltern-Prozesses.

RÜCKGABEWERT
Im Erfolgsfall wird die PID des Kind-Prozesses im Eltern-Prozess zurückgegeben, und 0 wird im Kind-Prozess zurückgegeben. Im Fehlerfall wird −1 im Eltern-Prozess zurückgegeben, es wird kein Kind-Prozess erstellt, und *errno* wird entsprechend gesetzt.

open(2) open(2)

NAME
open, creat − open and possibly create a file

SYNOPSIS
int open(const char *_pathname_, int _fla gs_);
int open(const char *_pathname_, int _fla gs_, mode_t _mode_);

int creat(const char *_pathname_, mode_t _mode_);

DESCRIPTION
The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more flags can be bitwise-*or*'d in *flags*. The*file cr eation flags* are

O_APPEND
The file is opened in append mode.

O_CREAT
If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective user ID of the process.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O_CREAT** is specified in *flags*; otherwise *mode* is ignored.

creat()
A call to **creat**() is equivalent to calling **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

RETURN VALUE
open(), and **creat**() return the new file descriptor, or −1 if an error occurred (in which case, *errno* is set appropriately).

open(2) open(2)

NAME
open, creat − öffnet und erstellt möglicherweise eine Datei

SYNOPSIS
int open(const char *_pathname_, int _fla gs_);
int open(const char *_pathname_, int _fla gs_, mode_t _mode_);

int creat(const char *_pathname_, mode_t _mode_);

BESCHREIBUNG
Der **open**() Systemaufruf öffnet die Datei, die durch *pathname* angegeben ist. Wenn die angegebene Datei nicht existiert, kann sie optional (wenn **O_CREAT** in *flags* angegeben ist) von **open**() erstellt werden.

Der Rückgabewert von **open**() ist ein Dateideskriptor.

Das Argument *flags* muss einen der folgenden *Zugriffsmodi* enthalten: **O_RDONLY**, **O_WRONLY**, oder **O_RDWR**. Diese fordern an, die Datei nur lesend, nur schreibend oder zum Lesen/Schreiben zu öffnen.

Zusätzlich können null oder mehr Flags bitweise in *flags* kombiniert werden. Die*Dateier stellungs-Flags* sind:

O_APPEND
Die Datei wird im Anhängemodus geöffnet.

O_CREAT
Wenn *pathname* nicht existiert, wird sie als reguläre Datei erstellt.

Der Besitzer (Benutzer-ID) der neuen Datei wird auf die effektive Benutzer-ID des Prozesses gesetzt.

Das *mode* Argument gibt die Datei-Modusbits an, die angewendet werden, wenn eine neue Datei erstellt wird. Dieses Argument muss angegeben werden, wenn **O_CREAT** in *flags* angegeben ist; ansonsten wird *mode* ignoriert.

creat()
Ein Aufruf von **creat**() entspricht einem Aufruf von **open**() mit *flags* gleich **O_CREAT|O_WRONLY|O_TRUNC**.

RÜCKGABEWERT
open() und **creat**() geben den neuen Dateideskriptor zurück, oder −1, wenn ein Fehler aufgetreten ist (in diesem Fall wird *errno* entsprechend gesetzt).

Diese Seite darf herausgetrennt werden.

exec(2) exec(2)

NAME
exec, execl, execv, execle, execve, execlp, execvp − execute a file

SYNOPSIS
int execlp(const char ****file*, const char *****ar g0***, ..., const char *****argn***, char *** /*NULL*/**);**

int execvp (const char ****file*, char *const** *ar gv[ ]***);**

DESCRIPTION
Each of the functions in the **exec** family overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

When a C program is executed, it is called as follows:

int main (int argc, char ∗**argv[]);**

where *argc* is the argument count, and *argv* is an array of character pointers to the arguments themselves. As indicated,*ar gc* is at least one, and the first member of the array points to a string containing the name of the file.

The *argv* argument is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). The *argv* argument is terminated by a null pointer.

The *path* argument points to a path name that identifies the new process file.

The *file* argument points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)).

File descriptors open in the calling process remain open in the new process.

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal**(3C)). Otherwise, the new process image inherits the signal dispositions of the calling process.

RETURN VALUES
If a function in the **exec** family returns to the calling process, an error has occurred; the return value is **−1** and **errno** is set to indicate the error.

exec(2) exec(2)

NAME
exec, execl, execv, execle, execve, execlp, execvp − führt eine Datei aus

SYNOPSIS
int execlp(const char ****file*, const char *****ar g0***, ..., const char *****argn***, char *** /*NULL*/**);**

int execvp (const char ****file*, char *const** *ar gv[ ]***);**

BESCHREIBUNG
Jede der Funktionen aus der **exec** Familie überschreibt das aktuelle Prozessbild mit einem neuen. Das neue Prozessbild wird aus einer gewöhnlichen, ausführbaren Datei erstellt. Diese Datei ist entweder eine ausführbare Objektdatei oder eine Datei mit Daten für einen Interpreter. Es kann keine Rückkehr von einem erfolgreichen Aufruf dieser Funktionen erfolgen, da das aufrufende Prozessbild durch das neue Prozessbild ersetzt wird.

Wenn ein C-Programm ausgeführt wird, wird es wie folgt aufgerufen:

int main (int argc, char ∗**argv[]);**

wobei *argc* die Argumentanzahl ist und *argv* ein Array von Zeigern auf die Argumente selbst ist. Wie angegeben, *argc* ist mindestens 1, und das erste Element des Arrays zeigt auf eine Zeichenkette, die den Namen der Datei enthält.

Das *argv* Argument ist ein Array von Zeichenkettenzeigern auf null-terminierte Strings. Diese Strings stellen die Argumentliste dar, die dem neuen Prozessbild zur Verfügung steht. Nach Konvention muss *argv* mindestens ein Element haben, und es sollte auf eine Zeichenkette zeigen, die mit *path* identisch ist (oder dessen letztem Bestandteil). Das *argv* Argument wird durch einen Null-Zeiger beendet.

Das *path* Argument zeigt auf einen Pfadnamen, der die neue Prozessdatei identifiziert.

Das *file* Argument zeigt auf die neue Prozessdatei. Wenn *file* kein Schrägstrich-Zeichen enthält, wird das Pfadpräfix für diese Datei durch eine Suche in den Verzeichnissen ermittelt, die in der **PATH** Umgebungsvariable angegeben sind (siehe **environ**(5)).

Offene Dateideskriptoren im aufrufenden Prozess bleiben im neuen Prozess offen.

Signale, die vom aufrufenden Prozess abgefangen werden, werden im neuen Prozessbild auf die Standardbehandlung gesetzt (siehe **signal**(3C)). Ansonsten erbt das neue Prozessbild die Signalbehandlungen des aufrufenden Prozesses.

RÜCKGABEWERT
Wenn eine Funktion aus der **exec** Familie zum aufrufenden Prozess zurückkehrt, ist ein Fehler aufgetreten; der Rückgabewert ist **−1** und **errno** ist gesetzt, um den Fehler anzuzeigen.

dup(2) dup(2)

NAME
dup, dup2 − duplicate a file descriptor

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

DESCRIPTION
The **dup**() system call creates a copy of the file descriptor *oldfd*, using the lowest-numbered unused file descriptor for the new descriptor.

dup2()
The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in *newfd*. If the file descriptor*ne wfd* 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, these system calls return the new file descriptor. On error, −1 is returned, and *errno* is set appropriately.

dup(2) dup(2)

NAME
dup, dup2 − dupliziert einen Dateideskriptor

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

BESCHREIBUNG
Der **dup**() Systemaufruf erstellt eine Kopie des Dateideskriptors *oldfd*, indem der niedrigste nicht verwendete Dateideskriptor für den neuen Deskriptor verwendet wird.

dup2()
Der **dup2**() Systemaufruf führt die gleiche Aufgabe wie **dup**() aus, verwendet jedoch anstelle des niedrigsten nicht verwendeten Dateideskriptors die im *newfd* angegebene Dateideskriptor-Nummer. Wenn der Dateideskriptor *newfd* bereits geöffnet war, wird er stillschweigend geschlossen, bevor er wiederverwendet wird.

Die Schritte zum Schließen und Wiederverwenden des Dateideskriptors *newfd* werden *atomar* ausgeführt.

RÜCKGABEWERT
Im Erfolgsfall geben diese Systemaufrufe den neuen Dateideskriptor zurück. Im Fehlerfall wird −1 zurückgegeben, und *errno* wird entsprechend gesetzt.

Diese Seite darf herausgetrennt werden.

wait(2) wait(2)

NAME
wait − wait for a process to change state

SYNOPSIS
pid_t wait(int *_wstatus_);

DESCRIPTION
The **wait**() system call is used to wait for a state change 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 no wait is performed, the terminated child remains in a "zombie" state.

wait()
The **wait**() system call suspends the execution of the calling thread until one of its children terminates.

If *wstatus* is not NULL, **wait**() stores the status of the terminated child in the *int* variable it points to. This value can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it):

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 used only if **WIFEXITED** returned true.

WIFSIGNALED(*wstatus***)**
returns true if the child was terminated by a signal, i.e., if the process was terminated due to a signal like SIGSEGV or SIGKILL.

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

wait(2) wait(2)

NAME
wait − wartet auf eine Zustandsänderung eines Prozesses

SYNOPSIS
pid_t wait(int *_wstatus_);

BESCHREIBUNG
Der **wait**() Systemaufruf wird verwendet, um auf eine Zustandsänderung eines Kindes des aufrufenden Prozesses zu warten und Informationen über das Kind zu erhalten, dessen Zustand sich geändert hat. Im Falle eines beendeten Kindes ermöglicht ein Aufruf von wait, dass das System die mit dem Kind verbundenen Ressourcen freigibt; wenn kein wait aufgerufen wird, bleibt das beendete Kind im „Zombie“-Zustand.

wait()
Der **wait**() Systemaufruf blockiert die Ausführung des aufrufenden Threads, bis eines seiner Kinder beendet wird.

Wenn *wstatus* nicht NULL ist, speichert **wait**() den Status des beendeten Kindes in der *int*-Variable, auf die es zeigt. Dieser Wert kann mit den folgenden Makros überprüft werden (die die Integer-Variable selbst als Argument nehmen, nicht einen Zeiger darauf):

WIFEXITED(*wstatus***)**
gibt true zurück, wenn das Kind normal beendet wurde, das heißt, durch Aufruf von **exit**(3) oder **_exit**(2), oder durch Rückkehr von main().

WEXITSTATUS(*wstatus***)**
gibt den Exit-Status des Kindes zurück. Dies besteht aus den niedrigstwertigen 8 Bits des *status* Arguments, das das Kind in einem Aufruf von **exit**(3) oder **_exit**(2) oder als Argument für eine Rückgabe in main() angegeben hat. Dieses Makro sollte nur verwendet werden, wenn **WIFEXITED** true zurückgegeben hat.

WIFSIGNALED(*wstatus***)**
gibt true zurück, wenn das Kind durch ein Signal beendet wurde, d.h. wenn der Prozess aufgrund eines Signals wie SIGSEGV oder SIGKILL beendet wurde.

RÜCKGABEWERT
Im Erfolgsfall gibt der **wait**() Systemaufruf die Prozess-ID des beendeten Kindes zurück. Im Fehlerfall wird −1 zurückgegeben, und *errno* wird auf **ECHILD** gesetzt, wenn keine unbeachteten Kinder existieren.

Diese Seite darf herausgetrennt werden.

Entfernen Sie dieses Blatt vor der Bearbeitung der Klausur und bewahren Sie es auf! Mit Hilfe des Codes können Sie Ihr persönliches Ergebnis online abfragen.

Klausur **Klausur “Betriebssysteme”**, 2025-02-10.

Ergebnisse online: https://studip.tu-braunschweig.de