1 Prozesse und Scheduling (9 Punkte)

a) UNIX-Systemaufrufe (4 Punkte) Geben Sie die Ausgabe des folgenden C-Programms an.
Hinweis: Das Einbinden der Header-Dateien sowie ein Teil der Fehlerbehandlung wurden ausgelassen. Gehen Sie von einem fehlerfreien Ablauf aus. Das Symbol ␣ steht für ein Leerzeichen.

```
1  int x = 1;
2  void next() {
3          printf("%d␣", x++);
4  }
5  int main() {
6          next();
7          pid_t pid = fork();
8          if (pid > 0) { // Elternprozess
9                  wait(NULL);
10                 next();
11                 int x = 0;
12                 next();
13         } else if (pid == 0) { // Kindprozess
14                 next();
15         } else { // Fehlerfall
16                 next();
17                 printf("Fehler\n");
18         }
19         return 0;
20 }
```

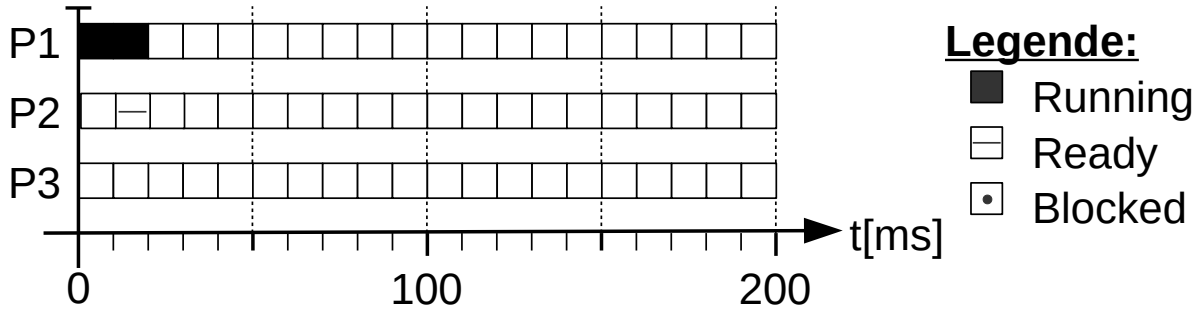
Ausgabe: _____

b) Prozess-Scheduling (Round Robin) (5 Punkte) Ein Betriebssystem verwaltet **drei** zyklisch arbeitende Prozesse P1, P2 und P3. Jeder Prozess führt zunächst Berechnungen auf der CPU aus. Sobald diese vollständig abgeschlossen sind, folgt ein E/A-Stoß, anschließend ist der Prozess wieder rechenbereit. Die Prozesse treffen zum Zeitpunkt der in der Tabelle angegebenen Ankunftszeit ein. In der folgenden Tabelle sind die Zeitangaben für die Ankunftszeit und Dauer von CPU- und E/A-Stößen jeweils in ms angegeben.

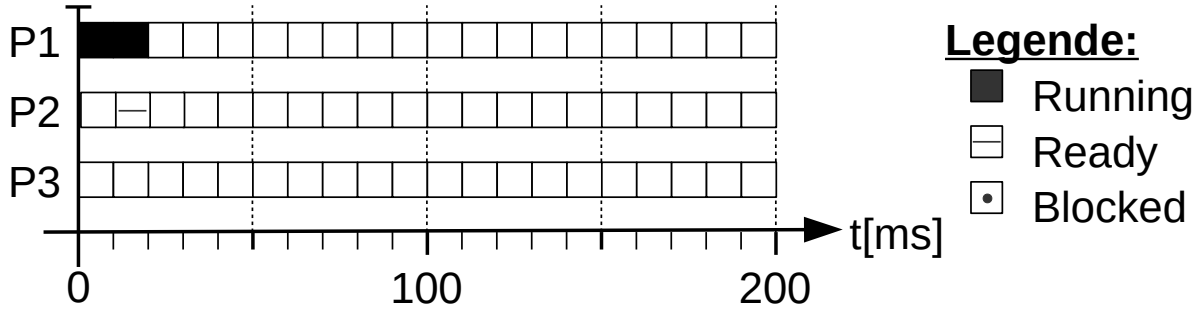
Prozess	Ankunftszeit	CPU-Zeit	E/A-Zeit
P1	0	20	20
P2	10	40	20
P3	30	40	30

Zeichnen Sie in das folgende Gantt-Diagramm ein, wie die drei Prozesse P1, P2 und P3 abgearbeitet werden, wenn das Scheduling nach der *Round-Robin*-Strategie mit einer Zeitscheibendauer von 30 ms vorgenommen wird. Die Prozessumschaltzeit kann vernachlässigt werden. E/A-Vorgänge können parallel ausgeführt werden. Markieren Sie in dem folgenden Diagramm die Prozesszustände entsprechend der Legende. Es genügt, die ersten 200ms anzugeben.

Hinweis: Die ersten zwei Zeiteinheiten sind bereits fertig ausgefüllt.


P1
P2
P3
t[ms]
0
100
200
Legende:
Running
Ready
Blocked

(Reserve)


P1
P2
P3
t[ms]
0
100
200
Legende:
Running
Ready
Blocked

2 Synchronisation und Verklemmungen (9 Punkte)

a) Synchronisation (insgesamt 5 Punkte) Im Folgenden sind C-Programmausschnitte von zwei nebenläufigen Programmen A und B gegeben, die auf einem gemeinsamen Speicher arbeiten. Die Funktion `arbeite(int *,int *)` verändert die referenzierten Parameter `a`, `b` und `c` und setzen dabei voraus, dass diese während der Ausführung nicht von anderen Prozessen verändert werden (gegenseitiger Ausschluss). Die Mutex-Variablen `mutex_a`, `mutex_b` und `mutex_c` sind vorgebenen um den Zugriff auf den gemeinsamen Speicher zu schützen. Mit `lock(Mutex *)` und `unlock(Mutex *)` werden diese gesperrt, bzw. wieder freigegeben.

```
/* gemeinsamer Speicher */
int a = 0, b = 0, c = 0;
Mutex mutex_a;
Mutex mutex_b;
Mutex mutex_c;
```

Zeile:	/* Programm A */
A1	lock(&mutex_a);
A2	lock(&mutex_b);
A3	arbeite(&a,&b);
A4	unlock(&mutex_b);
A5	lock(&mutex_c);
A6	arbeite(&a,&c);
A7	unlock(&mutex_a);
A8	unlock(&mutex_c);

Zeile:	/* Programm B */
B1	lock(&mutex_c);
B2	lock(&mutex_a);
B3	arbeite(&c,&a);
B4	unlock(&mutex_c);
B5	unlock(&mutex_a);

1) Verklemmungen (3 Punkte) Geben Sie einen konkreten Programmablauf der beiden Programme A und B an, bei dem es zu zyklischem Warten (*circular wait*) kommt. Tragen Sie schrittweise in die folgenden Tabelle ein welche Zeile (A1-A8, B1-B5) ausgeführt werden (inklusive Aufrufen die blockiert bleiben), bis es zur Verklemmung kommt. **Hinweis:** Im ersten Schritt ist also entweder A1 oder B1 einzutragen.

Schritt	1	2	3	4	5	6	7	8	9	10	11	12	13
Zeile													

(Ersatztabelle auf dem Reserveblatt: Streichen Sie ungültige Lösungen deutlich durch!)

2) Vorbeugung (2 Punkte) Beschreiben Sie, wie der obige Programmcode geändert werden muss, so dass es nicht zu zyklischem Warten kommen kann. Sie können dabei Bezug auf die Zeilen (A1-A8, B1-B5) nehmen.

b) Verklemmungen (4 Punkte) Erklären Sie kurz den Unterschied zwischen einem *deadlock* und einem *livelock*. Weshalb sind *deadlocks* das „geringere Übel“?

3 Speicherverwaltung und Virtueller Speicher (5 Punkte)

a). Seitenersetzung (3 Punkte) Erläutern Sie, wofür im Kontext von *Demand Paging* eine Seitenersetzungsstrategie benötigt wird. Geben Sie außerdem als Beispiel den Namen für eine Ersetzungsstrategie an, allerdings nicht *FIFO*.

b). Freispeicherbelegung (2 Punkte) Nach welchem Kriterium wird eine freie Kachel im Hauptspeicher für die Einlagerung einer Seite ausgewählt? Erläutern Sie ihre Antwort.

4 E/A und Dateisysteme (7 Punkte)

a) E/A-Scheduling (4 Punkte) Gegeben sei ein Plattenspeicher mit 16 Spuren. Der E/A-Scheduler bekommt immer wieder Aufträge für eine bestimmte Spur. Die Leseaufträge in L_0 sind dem E/A-Scheduler bereits bekannt. Nach drei bearbeiteten Aufträgen erhält er die Aufträge in L_3. Nach drei weiteren (d.h. nach insgesamt sechs) bearbeiteten Aufträgen erhält er die Aufträge in L_6. Zu Beginn befindet sich der Schreib-/Lesekopf über Spur 0.

$L_0 = \{1, 4, 7, 2\},\ L_3 = \{3, 6, 0\},\ L_6 = \{5, 2\}$

Tragen Sie hier die Reihenfolge der gelesenen Spuren für einen E/A-Scheduler ein, der nach der **Fahrstuhlstrategie** (***Elevator***) arbeitet.

b) Geräteklassen (3 Punkte) Nennen Sie die aus der Vorlesung bekannten Geräteklassen von E/A-Geräten, und erläutern Sie kurz den Unterschied zwischen ihnen.

5 Programmieraufgabe (15 Punkte)

Implementieren Sie das Programm `countfiles`, welches mit einer beliebigen Anzahl an Verzeichnissen, jedoch mindestens einem, aufgerufen wird:

Schnittstelle: `./countfiles` [Verzeichnis1] [Verzeichnis2] [Verzeichnis3] ...

(Beispiel: `./countfiles` foo/ bar/)

Funktionsweise:

a) main-Funktion (5 Punkte)

- Zunächst soll die Argumentenliste auf den korrekten Aufruf hin überprüft werden.
- Anschließend soll ihr Programm für jedes Argument die Funktion `int count_files(char *dirName)` aufrufen.
- Für jeden Aufruf sollen die Rückgabewerte aufsummiert werden.
- Zum Abschluss soll die Summe zusammen mit dem Text **„Sum of all files“** ausgeben werden.

b) count_files-Funktion (10 Punkte)

- Die Funktion `int count_files(char *dirName)` soll das Verzeichnis `dirName` öffnen, über alle Einträge iterieren und anschließend wieder schließen.
- Der übergebene Pfad `dirName` muss nicht nochmal explizit auf seine Richtigkeit oder Fehler überprüft werden. Es genügt, dies über eine passende Behandlung des Aufrufs von `opendir` zu erledigen.
- Für jeden Eintrag soll mit Hilfe der gegebenen Funktion `int dir_entry_type(char *path)` geprüft werden, ob es sich um eine Datei (Rückgabewert 1) oder ein Verzeichnis (Rückgabewert 2) handelt.
- Handelt es sich um eine Datei, so soll deren Vorkommen gezählt werden.
- Handelt es sich um ein Verzeichnis, soll nichts geschehen.
- Die Anzahl aller Dateien soll am Ende der Funktion zurückgegeben werden.
- Beachten Sie die bereitgestellten Hilfsfunktionen `char* concat_paths(const char *first, const char *second)` und `int is_dot_dir(char *entryName)`.
  Beide müssen an geeigneten Stellen verwendet werden.

Fehlerbehandlung: Beachten Sie für Ihr gesamtes Programm, dass Fehler auftreten können und entsprechend behandelt werden müssen.

Rückgabewert

0 Die Ausführung war korrekt.

1 Es gab einen Fehler bei der Ausführung.

Relevante Manual-Seiten: opendir, readdir, closedir

Hinweis: Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die sematische Umsetzung.

```c
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>

static int count_files(const char *dirName);

/*
 * Alloziert dynamisch Speicher und
 * konkanteniert beide Parameter in einem String, inkl. Slash.
 * Es wird ein Zeiger auf den allozierten Speicher zurückgegeben.
 */
static char* concat_paths(const char *first, const char *second) {
        /*  Aus Platzgründen weggelassen */
}

/*
 * Bestimmt für den Parameter pfath, ob es eine Datei (Rückgabewert 1)
 * oder ein Verzeichnis (Rückgabewert 2) ist.
 */
static int dir_entry_type(char *path) {
        /*  Aus Platzgründen weggelassen */
}

static int is_dot_dir(char *entryName) {
        return !strncmp(entryName, ".", 1) || !strncmp(entryName, "..", 2);
}

/*
 * Ab hier den Code für die main-Funktion
 * sowie für count_files-Funktion einfügen.
 */
int main(int argc, const char *argv[]) {
```

OPENDIR

NAME

opendir, fdopendir − open a directory

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

fdopendir():
Since glibc 2.10:
_POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_GNU_SOURCE

DESCRIPTION

The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The fdopendir() function is like opendir(), but returns a directory stream for the directory referred to by the open file descriptor fd. After a successful call to fdopendir(), fd is used internally by the implementation, and should not otherwise be used by the application.

RETURN VALUE

The opendir() and fdopendir() functions return a pointer to the directory stream. On error, NULL is returned, and errno is set to indicate the error.

ERRORS

EACCES Permission denied.

EBADF fd is not a valid file descriptor opened for reading.

EMFILE The per-process limit on the number of open file descriptors has been reached.

ENFILE The system-wide limit on the total number of open files has been reached.

ENOENT Directory does not exist, or name is an empty string.

ENOMEM Insufficient memory to complete the operation.

ENOTDIR

name is not a directory. [...]

NOTES

Filename entries can be read from a directory stream using readdir(3). [...]

CLOSEDIR

NAME

closedir − close a directory

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);

DESCRIPTION

The closedir() function closes the directory stream associated with dirp. A successful call to closedir() also closes the underlying file descriptor associated with dirp. The directory stream descriptor dirp is n available after this call.

RETURN VALUE

The closedir() function returns 0 on success. On error, −1 is returned, and errno is set to indicate the error.

ERRORS

EBADF Invalid directory stream descriptor dirp.

[...]

READDIR

NAME

readdir − read a directory

SYNOPSIS

#include <dirent.h>

struct dirent *readdir(DIR *dirp);

DESCRIPTION

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or an error occurred.

In the glibc implementation, the dirent structure is defined as follows:

```
struct dirent {
ino_t d_ino; /* Inode number */
off_t d_off; /* Not an offset; see below */
unsigned short d_reclen; /* Length of this record */
unsigned char d_type; /* Type of file; not supported by all filesystem types */
char d_name[256]; /* Null−terminated filename */
};
```

The only fields in the dirent structure that are mandated by POSIX.1 are d_name and d_ino. [...]

The fields of the dirent structure are as follows:

d_ino This is the inode number of the file.

d_name This field contains the null terminated filename. See NOTES.

[...]

The data returned by readdir() may be overwritten by subsequent calls to readdir() for the same directory stream.

RETURN VALUE

On success, readdir() returns a pointer to a dirent structure. (This structure may be statically allocatec do not attempt to free(3) it.)

If the end of the directory stream is reached, NULL is returned and errno is not changed. If an error occurs, NULL is returned and errno is set to indicate the error. To distinguish end of stream from an error, set errno to zero before calling readdir() and then check the value of errno if NULL is returned.

ERRORS

EBADF Invalid directory stream descriptor dirp.

[...]

NOTES

A directory stream is opened using opendir(3).

The order in which filenames are read by successive calls to readdir() depends on the filesystem implementation; it is unlikely that the names will be sorted in any fashion. [...]