

Aufgabe 2: (9 Punkte)

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

a) Vervollständigen Sie die gegebene Skizze zur Abbildung einer von Ihnen gewählten realen Adresse aus der gegebenen logischen Adresse 91c.

Logische Adresse

9	1	c
---	---	---

4 Punkte

Reale Adresse

--	--	--	--

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

3 Punkte

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

2 Punkte

Aufgabe 3: (11 Punkte)

Vervollständigen Sie das folgende Programm, welches eine Simulation einer Fabrik übernimmt. Die Fabrik hat eine Personalleiter **Christian**, der für das Einstellen neuer Mitarbeiter zuständig ist. Die eigentliche Herstellung von Waren wird von mehreren Arbeitern durchgeführt, die zufälligerweise alle **Lukas** heißen und die nach jeder hergestellten Ware mit einer Wahrscheinlichkeit von 1% kündigen. Die fertigen Waren werden von dem Verkaufschef **Daniel** verkauft. Zusammen haben sich alle Mitarbeiter der Fabrik darauf verständigt, die Arbeit wie folgt zu koordinieren:

- Christian sorgt dafür, dass immer genug Arbeiter eingestellt werden, wobei darauf geachtet wird, dass **maximal 35 Lukas** gleichzeitig angestellt sind.
- Immer wenn ein Lukas ein Produkt fertiggestellt hat, wird Daniel darüber benachrichtigt, damit dieses Produkt in der Verkauf gehen kann.
- Um die Kosten für die Ausgangsmaterialien für ein Produkt zu decken, zieht ein Arbeiter 5 € von der Bilanzsumme des Unternehmens ab.
- Sobald ein Lukas keine Lust mehr auf die Arbeit hat (er bricht seine Schleife ab), meldet er sich bei Christian ab, sodass ein neuer Arbeiter eingestellt werden kann.
- Wenn Daniel erfährt, dass ein Produkt fertig gestellt wurde, geht dieses in den Verkauf. Der dabei erzielten Gewinn von 15 € wird in der Bilanzsumme vermerkt.

Ergänzen sie den C-Code so, dass das beschriebene Verhalten erreicht wird. Hierfür müssen Sie die gegebenen Semaphoren so initialisieren und verwenden, dass die Arbeitsabläufe passend synchronisiert werden. Dabei kann es sein, dass einzelnen Felder leer bleiben können. **Streichen Sie in diesem Fall das Feld durch!**

Hinweise:

- Zu Beginn laufen bereits zwei Threads, von denen einer die Funktion `Christian()` ausführt und der andere die Funktion `Daniel()`.
- `hire_Lukas()` startet einen neuen Thread, der die Funktionen `Lukas()` ausführt.
- Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.

```
sem_t worker_sem;
sem_t product_sem;
sem_t bilanz_sem;
int bilanzsumme = 0;
```

```
void init() {
    _____;
    _____;
    _____;
}
```

```
void Daniel() {
    while (true) {
        _____;
        sell_good();
        _____;
        bilanzsumme += 15;
        _____;
    }
}
```

```
void Christian() {
    while (true) {
        _____;
        hire_Lukas();
    }
}
```

```
void Lukas() {
    while (random() % 100 != 0) {
        _____;
        bilanzsumme -= 5;
        _____;

        produce_good();
        _____;
    }
    _____;
}
```

Aufgabe 4: (6 Punkte)

Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 9 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 4 Byte groß, ein Block 5 KiByte.

a) Wieviele Blöcke werden benötigt, um eine Datei der Größe 25 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

2 Punkte

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

b) Wieviele Blöcke werden benötigt, um eine Datei der 210 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

2 Punkte

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

c) Wie groß kann eine Datei maximal in diesem Dateisystem sein?

2 Punkte

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Aufgabe 5: (30 Punkte)

Auf einem Rechner befinden sich diverse Dateien und Verzeichnisse in einer Verzeichnisstruktur. Aus diesen sollen regulären Dateien herausgesucht werden, deren Name `good` enthält. Für jede gefundene Datei soll das Programm `trumpet` mit dem Dateinamen als Parameter aufgerufen werden. Alle anderen Dateien sollen ignoriert werden.

Aufrufsyntax: `marder DIRECTORY...`

Implementieren Sie das Programm `marder`, welches mit einer Liste von Verzeichnissen aufgerufen wird, die rekursiv nach Dateien durchsucht werden sollen.

Funktion: `main(int argc, char *argv[])`

- Prüfen der Anzahl der Parameter
- Nutzungsausgabe, wenn die Anzahl der Parameter falsch ist
- Iterieren über die angegebenen Verzeichnisse. Für jedes Verzeichnis die Funktion `recurse(char *path)` aufrufen.

Funktion: `recurse(char *path)`

- rekursiver Verzeichnisabstieg
- Aufruf von `iterate_current_dir()`, um das aktuelle Verzeichnis zu durchlaufen.

Funktion: `iterate_current_dir()`

- Iterieren über alle Einträge des Verzeichnisses
- versteckte Dateien und Sicherungskopien (Dateiname beginnt mit "." oder "~") sollen ignoriert werden
- reguläre Dateien an die Funktion `handle_file(char *filename)` übergeben
- in Verzeichnisse mittels `recurse(char *path)` absteigen

Funktion: `handle_file(char *filename)`

- Prüfen, ob der Dateiname `good` enthält
- Das Programm `trumpet` mit dem Dateinamen aufrufen
- Auf das Beenden des `trumpet`-Prozesses warten

Tritt ein nicht erwarteter Fehler auf, so ist das Programm mit Ausgabe einer Fehlermeldung zu beenden. Hierzu steht die Funktion `die(char *msg)` bereit.

Beispielfunktionsweise:

```
$ ls foobar/
dir/ file file1 file2 good test
```

Aufruf und Standard-Out:

```
$ ./marder foobar ... (eventuell >1 Pfade)
!Es gibt kein stdout von marder außer der Usage-Message!
```

Diese Seite darf herausgetrennt werden.

Aufgabe 6: (14 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 waagrechten Strichen durch (~~☒~~).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Bewerten Sie die folgenden Aussagen zu UNIX/Linux-Dateideskriptoren:

Richtig Falsch

- Ein Dateideskriptor ist eine Integerzahl, die über gemeinsamen Speicher an einen anderen Prozess übergeben werden kann und von letzterem zum Zugriff auf eine geöffnete Datei verwendet werden kann.
- Ein Dateideskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine geöffnete Datei benutzen kann.
- Dateideskriptoren sind Zeiger auf Betriebssystem-interne Strukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.
- Nur Dateideskriptoren mit dem Flag `FD_CLOEXEC` werden beim Prozessende mit `exit()` vom Betriebssystem geschlossen.

2 Punkte

b) Bereich: UNIX-Prozesse

Richtig Falsch

- Bei einem Einprozessor-Betriebssystem befindet sich zu jedem Zeitpunkt maximal ein Prozess im Zustand „laufend“.
- Ein Prozess im Zustand „beendet“ (Zombie) kann mit dem Systemaufruf `respawn()` neu gestartet werden.
- Ein Prozess wird durch seine Prozess-ID identifiziert.
- Ein Prozess wird durch seine Elter-Prozess-ID identifiziert.

2 Punkte

c) Bewerten Sie die folgenden Aussagen zum Thema Signale:

Richtig Falsch

- Signale koordinieren den Zugriff auf exklusive Betriebsmittel.
- Signale sind immer asynchrone Ereignisse.
- Es gibt synchrone und asynchrone Signale.
- Für jedes Signal kann ein Prozess einen Handler registrieren, um dieses zu behandeln.

2 Punkte

d) Scheduling

Richtig Falsch

- Ein Prozess, der sich in einem kritischen Abschnitt befindet, kann vom Betriebssystem unterbrochen werden.
- Kooperatives Scheduling und Mehrprogrammbetrieb schließen sich gegenseitig aus.
- Federgewichtige Prozesse (user-threads) können die Multiprozessorfähigkeit des Betriebssystems ausnutzen.
- Präemptives Scheduling ermöglicht es, die Monopolisierung der CPU zu verhindern.

2 Punkte

e) Welche der genannten Aussagen zum Thema Semaphoren sind richtig?

Richtig Falsch

- Ein Semaphore kann verwendet werden, um gegenseitigen Ausschluss zu implementieren.
- Ein Semaphore kann für die Verwaltung von wiederverwendbaren Betriebsmitteln verwendet werden.
- Ein Semaphore kann nicht zur Beschränkung von konkurrierenden Zugriffen verwendet werden.
- Die `up()` Funktion eines Semaphors erhöht den Zähler der Semaphore, falls möglich und wartet sonst, bis dies möglich ist.

2 Punkte

f) Man unterscheidet Traps und Interrupts (*Unterbrechungen*). Bewerten Sie die folgenden Aussagen:

Richtig Falsch

- Systemaufrufe sind im Programmfluss für das Betriebssystem nicht vorhersagbar und sind deshalb in die Kategorie Interrupt einzuordnen.
- Ein Zugriff auf eine reale Speicheradresse kann zu einem Trap führen.
- Ein Trap wird immer unmittelbar durch eine Aktivität des aktuell laufenden Prozesses ausgelöst.
- Der Zeitgeber (Systemuhr) unterbricht die Programmbearbeitung in regelmäßigen Abständen. Die genaue Stelle der Unterbrechungen ist damit vorhersagbar. Somit sind solche Unterbrechungen in die Kategorie Trap einzuordnen.

2 Punkte

g) Bereich: Dateisysteme

Richtig Falsch

- Für jede reguläre Datei existiert mindestens ein Hardlink im selben Dateisystem.
- Verzeichnisse sind spezielle Dateien des Dateisystems, die Namen an Dateiobjekte binden.
- Symlinks beeinflussen den Linkzähler im Indexknoten der Zielfeile.
- Ein Hardlink kann auf andere Dateisysteme verweisen.

2 Punkte

Diese Seite darf herausgetrennt werden.

```
childr(2)
exec(3)
exec(3)

NAME
  execl, execlp, execlv, execlvp – 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 execlv(const char *pathname, char *const argv[]);
int execlvp(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 - execlv, execlvp
  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(), execlvp()
  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.
```

17

```
childr(2)
childr(2)

NAME
  chdir, chdirr – change working directory

SYNOPSIS
#include <unistd.h>
int chdir(const char *path);
int chdirr(int fd);

DESCRIPTION
chdir() changes the current working directory of the calling process to the directory specified in path. chdirr() is identical to chdir(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE
  On success, zero is returned. On error, -1 is returned, and errno is set appropriately.
```

```
closedir(3)
closedir(3)

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.

RETURN VALUE
  The closedir() function returns 0 on success. On error, -1 is returned, and errno is set appropriately.
```

Diese Seite darf herausgetrennt werden.

Diese Seite darf herausgetrennt werden.

```
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.
```

```
opendir(3)
opendir(3)

NAME
  opendir, fdopendir – open a directory

SYNOPSIS
DIR *opendir(const char *name);

DESCRIPTION
  The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream.

RETURN VALUE
  The opendir() function returns a pointer to the directory stream. On error, NULL 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.
```

```
read(3)
read(3)

NAME
  readr – 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 pointed to by dirp. It returns NULL on reaching the end of the directory or if an error occurred.
  The dirent structure is defined as follows:


```
struct dirent {
 int d_ino; /* i-node number */
 unsigned short d_reclen; /* Length of this record */
 unsigned char d_type; /* Type of file */
 char d_name[256]; /* Null-terminated filename */
};
d_type
 This field indicates the file type:
 DT_DIR This is a directory.
 DT_FIFO This is a named pipe (FIFO).
 DT_LNK This is a symbolic link.
 DT_REG This is a regular file.
 DT_SOCK This is a UNIX domain socket.

RETURN VALUE
 On success, readdir() returns a pointer to a dirent structure.
 If the end of the directory is reached, NULL is returned and errno is not changed. If an error occurs, NULL is returned and errno is set appropriately.
```


```

18

Diese Seite darf herausgetrennt werden.

```
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_timedwait(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 process, and the function returns immediately. If the semaphore currently has zero or a negative value, then the calling process will be blocked until it can perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.
sem_timedwait() 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.
```

19

```

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.

```
strstr(3)
```

```

NAME
    strstr and manipulate strings

SYNOPSIS
#include <string.h>
char *strstr(const char *haystack, const char *needle);
size_t strlen(const char *s);

```

DESCRIPTION

The **strstr()** function finds the first occurrence of the substring *needle* in the string *haystack*.

The **strlen()** function calculates the length of the string pointed to by *s*, excluding the terminating null byte (**'0'**).

RETURN VALUE

The function **strstr()** returns a pointer to the beginning of the located substring, or NULL if the substring is not found.

The function **strlen()** returns the number of bytes in the string pointed to by *s*.