

Übungen zu Systemprogrammierung 2

ÜM – Besprechung der Miniklausur

Wintersemester 2025/26

Jürgen Kleinöder, Thomas Preisner, Tobias Häberlein, Ole Wiedemann

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät



M.1 Aufgabe 1

M.2 Aufgabe 2

M.3 Aufgabe 3



M.1 Aufgabe 1

M.2 Aufgabe 2

M.3 Aufgabe 3



a) Man unterscheidet zwischen Traps und Interrupts. Welche Aussage ist richtig?

2 Punkte

- Traps können nicht durch Speicherzugriffe ausgelöst werden.
- Bei der mehrfachen Ausführung eines unveränderten Programmes mit den selben Eingabedaten treten Interrupts immer an den selben Stellen auf.
- Traps stehen immer in ursächlichem Zusammenhang zu der Ausführung eines Maschinenbefehls.
- Ein Trap wird immer durch das Programm behandelt, welches den Trap ausgelöst hat, Interrupts werden hingegen immer durch das Betriebssystem behandelt.

b) Welche Aussage zu Semaphoren ist richtig?

2 Punkte

- Die Up-Operation eines Semaphors kann ausschließlich von einem Thread aufgerufen werden, der zuvor mindestens eine Down-Operation auf dem selben Semaphor aufgerufen hat.
- Die Down-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- Ein Semaphor kann nur zur Signalisierung von Ereignissen, nicht jedoch zum Erreichen gegenseitigen Ausschlusses verwendet werden.
- Die Up-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.



a) Man unterscheidet zwischen Traps und Interrupts. Welche Aussage ist richtig?

2 Punkte

- Traps können nicht durch Speicherzugriffe ausgelöst werden.
- Bei der mehrfachen Ausführung eines unveränderten Programmes mit den selben Eingabedaten treten Interrupts immer an den selben Stellen auf.
- Traps stehen immer in ursächlichem Zusammenhang zu der Ausführung eines Maschinenbefehls.
- Ein Trap wird immer durch das Programm behandelt, welches den Trap ausgelöst hat, Interrupts werden hingegen immer durch das Betriebssystem behandelt.

b) Welche Aussage zu Semaphoren ist richtig?

2 Punkte

- Die Up-Operation eines Semaphors kann ausschließlich von einem Thread aufgerufen werden, der zuvor mindestens eine Down-Operation auf dem selben Semaphor aufgerufen hat.
- Die Down-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- Ein Semaphor kann nur zur Signalisierung von Ereignissen, nicht jedoch zum Erreichen gegenseitigen Ausschlusses verwendet werden.
- Die Up-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.



a) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

3 Punkte

- Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- Der Prozess ist der statische Teil (Rechte, Speicher, etc.), das Programm der aktive Teil (Programmzähler, Register, Stack).
- Mit Hilfe des Systemaufrufs `execve(2)` (bzw. der Bibliotheksfunktion `exec(3)`) wird das bestehende Programm im aktuell laufenden Prozess ersetzt.
- Der C-Präprozessor übersetzt mehrere C-Quelldateien in Module die zu einem Programm gebunden werden können.
- Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausführen.



a) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig?

3 Punkte

- Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- Der Prozess ist der statische Teil (Rechte, Speicher, etc.), das Programm der aktive Teil (Programmzähler, Register, Stack).
- Mit Hilfe des Systemaufrufs `execve(2)` (bzw. der Bibliotheksfunktion `exec(3)`) wird das bestehende Programm im aktuell laufenden Prozess ersetzt.
- Der C-Präprozessor übersetzt mehrere C-Quelldateien in Module die zu einem Programm gebunden werden können.
- Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- Ein Prozess kann mit Hilfe von Threads mehrere Programme gleichzeitig ausführen.



M.1 Aufgabe 1

M.2 Aufgabe 2

M.3 Aufgabe 3



Zu programmieren war ein Programm `pdfsize`, welches einen Verzeichnispfad als Argument erhält und daraufhin alle `.pdf`-Dateien in dem übergebenen Verzeichnis inklusive deren Dateigröße ausgibt.



Zu programmieren war ein Programm `pdfsize`, welches einen Verzeichnispfad als Argument erhält und daraufhin alle `.pdf`-Dateien in dem übergebenen Verzeichnis inklusive deren Dateigröße ausgibt.

Beispielaufruf:

```
$ ./pdfsize klausur/  
klausur/man.pdf: 43607  
klausur/2025w-SP-MKlausur.pdf: 229590  
klausur/2025w-SP-MKlausur_ml.pdf: 240367  
Gesamtgroesse: 513564
```



Zu programmieren war ein Programm `pdfsize`, welches einen Verzeichnispfad als Argument erhält und daraufhin alle `.pdf`-Dateien in dem übergebenen Verzeichnis inklusive deren Dateigröße ausgibt.

Beispielaufruf:

```
$ ./pdfsize klausur/  
klausur/man.pdf: 43607  
klausur/2025w-SP-MKlausur.pdf: 229590  
klausur/2025w-SP-MKlausur_ml.pdf: 240367  
Gesamtgroesse: 513564
```

Funktionen:

```
int main(int argc, char **argv);  
// Filterfunktion für scandir  
int filter(const struct dirent *ent);  
// Sortierfunktion für scandir  
int sizesort(const struct dirent **a, const struct dirent **b);
```



- Der Typ von `namelist` in `scandir` (3p)



- Der Typ von `namelist` in `scandir` (3p)
 - Ein einzelner Verzeichniseintrag ist ein Pointer zu `struct dirent`
→ `struct dirent *`



- Der Typ von `namelist` in `scandir` (3p)
 - Ein einzelner Verzeichniseintrag ist ein Pointer zu `struct dirent`
→ `struct dirent *`
 - Ein Array aus Verzeichniseinträgen ist dann vom Typ Pointer auf Pointer
→ `struct dirent **`



- Der Typ von `namelist` in `scandir` (3p)
 - Ein einzelner Verzeichniseintrag ist ein Pointer zu `struct dirent`
→ `struct dirent *`
 - Ein Array aus Verzeichniseinträgen ist dann vom Typ Pointer auf Pointer
→ `struct dirent **`
 - Damit `scandir` (3p) das Array befüllen kann, übergeben wir eine **Referenz** auf das Array
→ `struct dirent ***`

```
struct dirent **namelist;  
scandir(dir, &namelist, filter, sizesort);
```



- Der Typ von `namelist` in `scandir` (3p)
 - Ein einzelner Verzeichniseintrag ist ein Pointer zu `struct dirent`
→ `struct dirent *`
 - Ein Array aus Verzeichniseinträgen ist dann vom Typ Pointer auf Pointer
→ `struct dirent **`
 - Damit `scandir` (3p) das Array befüllen kann, übergeben wir eine **Referenz** auf das Array
→ `struct dirent ***`

```
struct dirent **namelist;  
scandir(dir, &namelist, filter, sizesort);
```

- Auf welchen Elementen muss `free` (3p) aufgerufen werden?
 - Auf jedem Element des `namelist`-Arrays
 - Auf dem `namelist`-Array selbst



- Fehlerbehandlung in `filter()`
 - Wenn `lstat(3p)` in `filter` fehlschlägt, sollte das Programm nicht beendet werden (kein `die()`)



- Fehlerbehandlung in `filter()`
 - Wenn `lstat(3p)` in `filter` fehlschlägt, sollte das Programm nicht beendet werden (kein `die()`)
 - Es kann sein, dass `lstat(3p)` aufgrund von fehlenden Berechtigungen auf einer Datei fehlschlägt
 - Deswegen sollte nicht direkt das Programm abbrechen → Fehlermeldung
Vergleiche mit `find(1p)`

```
$ find /proc/  
/proc/3995810/task/3995810/fd  
find: '/proc/3995810/task/3995810/fd': Permission denied  
/proc/3995810/task/3995810/fdinfo  
find: '/proc/3995810/task/3995810/fdinfo': Permission denied  
/proc/3995810/task/3995810/ns  
[...]
```



- *“Es sollen nur reguläre Dateien berücksichtigt werden, ...”*



- *“Es sollen nur reguläre Dateien berücksichtigt werden, ...”*
 - Triggerword: **reguläre Datei** → `stat(3p)/lstat(3p)` nötig
 - Prüfen, ob es sich um eine reguläre Datei handelt:
`S_ISREG(file_stat.st_mode)`



- *“Es sollen nur reguläre Dateien berücksichtigt werden, ...”*
 - Triggerword: **reguläre Datei** → `stat(3p)/lstat(3p)` nötig
 - Prüfen, ob es sich um eine reguläre Datei handelt:
`S_ISREG(file_stat.st_mode)`
- *“... symbolischen Verknüpfungen soll nicht gefolgt werden.”*



- *“Es sollen nur reguläre Dateien berücksichtigt werden, ...”*
 - Triggerword: **reguläre Datei** → `stat(3p)/lstat(3p)` nötig
 - Prüfen, ob es sich um eine reguläre Datei handelt:
`S_ISREG(file_stat.st_mode)`
- *“... symbolischen Verknüpfungen soll nicht gefolgt werden.”*
 - `stat(3p)` folgt symbolischen Verknüpfungen
 - `lstat(3p)` greift auf die Datei zu, welche die Verknüpfung enthält



- *“Es sollen nur reguläre Dateien berücksichtigt werden, ...”*
 - Triggerword: **reguläre Datei** → `stat(3p)/lstat(3p)` nötig
 - Prüfen, ob es sich um eine reguläre Datei handelt:
`S_ISREG(file_stat.st_mode)`
- *“... symbolischen Verknüpfungen soll nicht gefolgt werden.”*
 - `stat(3p)` folgt symbolischen Verknüpfungen
 - `lstat(3p)` greift auf die Datei zu, welche die Verknüpfung enthält
- Pfad für `stat(3p)/lstat(3p)` aus Verzeichnis und Dateiname zusammenbauen
 - `namelist[i]->d_name` enthält nur den Dateinamen **ohne** Verzeichnisnamen



- *“Es sollen nur reguläre Dateien berücksichtigt werden, ...”*
 - Triggerword: **reguläre Datei** → `stat(3p)/lstat(3p)` nötig
 - Prüfen, ob es sich um eine reguläre Datei handelt:
`S_ISREG(file_stat.st_mode)`
- *“... symbolischen Verknüpfungen soll nicht gefolgt werden.”*
 - `stat(3p)` folgt symbolischen Verknüpfungen
 - `lstat(3p)` greift auf die Datei zu, welche die Verknüpfung enthält
- Pfad für `stat(3p)/lstat(3p)` aus Verzeichnis und Dateiname zusammenbauen
 - `namelist[i]->d_name` enthält nur den Dateinamen **ohne** Verzeichnisnamen

```
struct dirent *entry = namelist[i];
char path[strlen(dir) + 1 + strlen(entry->d_name) + 1];
sprintf(path, "%s/%s", dir, entry->d_name);
```



M.1 Aufgabe 1

M.2 Aufgabe 2

M.3 Aufgabe 3



Folgendes Programm enthält Programmierfehler. Nennen Sie diese und beschreiben Sie die möglichen Auswirkungen.

```
int main(void) {  
    // Reserviere Speicher für 64 Integer  
    int *p = malloc(64);  
    for (int i = 0; i < 64; i++) {  
        p[i] = i;  
    }  
    free(p);  
    return 0;  
}
```



Folgendes Programm enthält Programmierfehler. Nennen Sie diese und beschreiben Sie die möglichen Auswirkungen.

```
int main(void) {  
    // Reserviere Speicher für 64 Integer  
    int *p = malloc(64);  
    for (int i = 0; i < 64; i++) {  
        p[i] = i;  
    }  
    free(p);  
    return 0;  
}
```

- **Problem 1:** `malloc(64)` reserviert 64 Byte
Größe eines `ints` ist implementierungsabhängig (aber **min. 2 Byte**)
→ `sizeof`-Operator verwenden!



Folgendes Programm enthält Programmierfehler. Nennen Sie diese und beschreiben Sie die möglichen Auswirkungen.

```
int main(void) {  
    // Reserviere Speicher für 64 Integer  
    int *p = malloc(64);  
    for (int i = 0; i < 64; i++) {  
        p[i] = i;  
    }  
    free(p);  
    return 0;  
}
```

- **Problem 1:** `malloc(64)` reserviert 64 Byte
Größe eines `ints` ist implementierungsabhängig (aber **min. 2 Byte**)
→ `sizeof`-Operator verwenden!
- **Problem 2:** Fehlende Fehlerbehandlung für `malloc(3p)`
Falls kein Speicher mehr verfügbar ist, wird `NULL` zurückgegeben



```
int main(void) {  
    // Reserviere Speicher für 64 Integer  
    int *p = malloc(64 * sizeof(int));  
    if (p == NULL) {  
        perror("malloc");  
        return 1;  
    }  
  
    for (int i = 0; i < 64; i++) {  
        p[i] = i;  
    }  
    free(p);  
    return 0;  
}
```



Schreiben Sie den vorherigen Programmcode so um, dass nur Speicher auf dem Stack verwendet wird. Wie unterscheidet sich der Parameter eines `malloc`-Aufrufs von der Größendefinition eines Arrays?



Schreiben Sie den vorherigen Programmcode so um, dass nur Speicher auf dem Stack verwendet wird. Wie unterscheidet sich der Parameter eines `malloc`-Aufrufs von der Größendefinition eines Arrays?

- Definition eines `int`-Arrays als lokale Variable: `int arr[64];`
 - Reserviert Platz für 64 Ganzzahlen auf dem Stack



Schreiben Sie den vorherigen Programmcode so um, dass nur Speicher auf dem Stack verwendet wird. Wie unterscheidet sich der Parameter eines `malloc`-Aufrufs von der Größendefinition eines Arrays?

- Definition eines `int`-Arrays als lokale Variable: `int arr[64];`
 - Reserviert Platz für 64 Ganzzahlen auf dem Stack
 - Umrechnung in Bytes übernimmt der Compiler, anders als bei `malloc(3p)`, automatisch



Schreiben Sie den vorherigen Programmcode so um, dass nur Speicher auf dem Stack verwendet wird. Wie unterscheidet sich der Parameter eines `malloc`-Aufrufs von der Größendefinition eines Arrays?

- Definition eines `int`-Arrays als lokale Variable: `int arr[64];`
 - Reserviert Platz für 64 Ganzzahlen auf dem Stack
 - Umrechnung in Bytes übernimmt der Compiler, anders als bei `malloc(3p)`, automatisch
- Falsche Lösungen:
 - `int[64] arr`
Eckige Klammern gehören nach den Bezeichner



Schreiben Sie den vorherigen Programmcode so um, dass nur Speicher auf dem Stack verwendet wird. Wie unterscheidet sich der Parameter eines `malloc`-Aufrufs von der Größendefinition eines Arrays?

- Definition eines `int`-Arrays als lokale Variable: `int arr[64];`
 - Reserviert Platz für 64 Ganzzahlen auf dem Stack
 - Umrechnung in Bytes übernimmt der Compiler, anders als bei `malloc(3p)`, automatisch
- Falsche Lösungen:
 - `int[64] arr`
Eckige Klammern gehören nach den Bezeichner
 - `int arr = new int[64]`
In C gibt es das Schlüsselwort `new` nicht!

Schreiben Sie den vorherigen Programmcode so um, dass nur Speicher auf dem Stack verwendet wird. Wie unterscheidet sich der Parameter eines `malloc`-Aufrufs von der Größendefinition eines Arrays?

- Definition eines `int`-Arrays als lokale Variable: `int arr[64];`
 - Reserviert Platz für 64 Ganzzahlen auf dem Stack
 - Umrechnung in Bytes übernimmt der Compiler, anders als bei `malloc(3p)`, automatisch
- Falsche Lösungen:
 - `int[64] arr`
Eckige Klammern gehören nach den Bezeichner
 - `int arr = new int[64]`
In C gibt es das Schlüsselwort `new` nicht!
 - `int[] arr = int(64)`





Nennen Sie den Bereich, in dem der von `malloc` allokierte Speicher verwaltet wird, sowie einen Nachteil im Vergleich zur Nutzung von Stack-Speicher.



Nennen Sie den Bereich, in dem der von `malloc` allokierte Speicher verwaltet wird, sowie einen Nachteil im Vergleich zur Nutzung von Stack-Speicher.

- `malloc(3p)` allokiert Speicher auf dem **Heap**



Nennen Sie den Bereich, in dem der von `malloc` allokierte Speicher verwaltet wird, sowie einen Nachteil im Vergleich zur Nutzung von Stack-Speicher.

- `malloc(3p)` allokiert Speicher auf dem **Heap**
- Nachteil: die Allokation des Speichers (Ausführungszeit von `malloc(3p)`) dauert länger als auf dem Stack



Nennen Sie den Bereich, in dem der von `malloc` allokierte Speicher verwaltet wird, sowie einen Nachteil im Vergleich zur Nutzung von Stack-Speicher.

- `malloc(3p)` allokiert Speicher auf dem **Heap**
- Nachteil: die Allokation des Speichers (Ausführungszeit von `malloc(3p)`) dauert länger als auf dem Stack
- Nicht richtig ist: der **Zugriff** auf Heap-Speicher dauert länger
 - Wurde Speicher auf dem Heap erst einmal zugeteilt, unterscheidet sich der Zugriff darauf nicht vom Zugriff auf Stack-Speicher