Betriebssysteme (BS)

Probeklausur

https://sys.cs.tu-dortmund.de/de/lehre/ss23/bs

05.07.2023

Peter Ulbrich

peter.ulbrich@tu-dortmund.de
bs-problems@ls12.cs.tu-dortmund.de
https://sys.cs.tu-dortmund.de/de/lehre/kummerkasten

Basierend auf *Betriebssysteme* von Olaf Spinczyk, Universität Osnabrück

Organisatorisches

- Studienleistung
  - Ankündigung über Moodle, sobald sie eingetragen wurde
  - Vorher bitte **keine** Nachfragen via E-Mail
  - Falls nötig, wird die Anmeldefrist angepasst (→ Ankündigung)
- Klausur am Mittwoch, den 02.08.2023
  - Zeit: 8-10 Uhr
  - Dauer: 60 Min.
  - **Handgeschriebener** A4-Spickzettel ist erlaubt, wird eingesammelt
  - Weitere Details folgen nach Ablauf der Anmeldefirst (**vermutlich** am 18.07.)
- HelpDesk / Fragestunde
  - Mittwoch, 26.07.2023, von 10-12 Uhr s.t. im Raum E23 in der OH14
  - Mittwoch, 06.09.2023 von 10-12 Uhr s.t. im Raum E.003 in der OH12
  - Außerdem auch online – siehe Moodle

Ablauf

- Probeklausur (45 Minuten)
- Besprechung der Aufgaben
- Auswertung
- Weitere Hinweise zur Vorbereitung

Probeklausur

... in (fast) allen Belangen realistisch:

- Art der Aufgaben
  - Auswahl aus dem **gesamten** Inhalt der Veranstaltung
    - Betriebssystemgrundlagen **und** UNIX-Systemprogrammierung in C
    - alle Vorlesungen und Übungen sind relevant
- Umfang
  - kürzer als das „Original“: 45 (statt 60) Minuten
- Durchführung
  - **keine Hilfsmittel** erlaubt (ausser Spickzettel)
  - bitte **still arbeiten** (Raum wird nicht verlassen)
  - jeder für sich
- Die Klausur wird **nicht** eingesammelt.

1a) UNIX-Systemaufrufe (4 Punkte)

```
int x = 1;
void next() {
        printf("%d ", x++);
}
int main() {
        next(); ❶
        pid_t pid = fork();
        if (pid > 0) { // Elternprozess
                wait(NULL);
                next(); ❸
                int x = 0;  // lokal
                next(); ❹
        } else if (pid == 0) { // Kindprozess
                next(); ❷
        } else { // Fehlerfall
                next();
                printf("Fehler\n");
        }
        return 0;
}
```

Ausgabe: 1 2 2 3

1b) Fehlerbehandlung (2 Punkte)

```
int x = 1;
void next() {
        printf("%d ", x++);
}
int main() {
        next(); ❶
        pid_t pid = fork(); // Fehler
        if (pid > 0) { // Elternprozess
                wait(NULL);
                next();
                int x = 0;
                next();
        } else if (pid == 0) { // Kindprozess
                next();
        } else { // Fehlerfall
                next(); ❷
                printf("Fehler\n"); ❸
        }
        return 0;
}
```

Ausgabe: 1 2 Fehler

1c) UNIX Shell-Operatoren (3 Punkte)

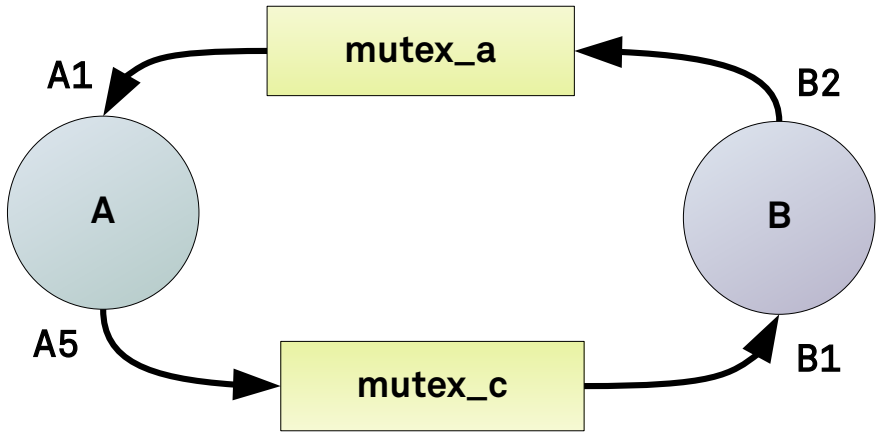
Geben Sie die Auswirkungen der beiden Befehle **„ls > wc“** sowie **„ls | wc“** an und erläutern Sie anhand dessen den Unterschied zwischen den Shell-Operatoren „>“ und „|“.:

- ls > wc schreibt Ausgabe von ls -l in die Datei „wc“ (1P)
- ls -l | wc leitet Ausgabe von ls an den Befehl „wc“ weiter (1P).
- Unterschied: > leitet in eine Datei um, | leitet an einen Befehl weiter (1P)

2a) Synchronisation (Insgesammt 5 Punkte)

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.

A1	...	B1	...	B2	...	A5



2) Vorbeugung (2 Punkte) Beschreiben Sie, wie der obige Programmcode geändert werden muss, so dass es nicht zu zyklischem Warten kommen kann.

- **Belegung der Ressourcen in identischer Reihenfolge. Z.B. B1 und B2 tauschen.**
- **Alternativ gleichzeitige Belegung von a und c z.B. mit weiterem Mutex.**

2b) Erzeuger/Verbraucher (5 Punkte)

```
Semaphore mutex = 1
Semaphore max_elements = 100
Semaphore elements_ready = 0;

/* Erzeuger */
Element *e = produce();


enqueue(queue, e );

v(&elements_ready); //Signal

/* Verbraucher */
Element *e;
p(&elements_ready); //Wait

e = dequeu(queue);


consume(e);
```

2b) Erzeuger/Verbraucher (5 Punkte)

```
Semaphore mutex = 1
Semaphore max_elements = 100
Semaphore elements_ready = 0;

/* Erzeuger */
Element *e = produce();
p(&max_elements);                    //wait(&max_elements)
p(&mutex);                           //wait(&mutex)
enqueue(queue, e );
v(&mutex);                           //signal(&mutex)
v(&elements_ready); //Signal

/* Verbraucher */
Element *e;
p(&elements_ready); //Wait

e = dequeu(queue);


consume(e);
```

2b) Erzeuger/Verbraucher (5 Punkte)

```
Semaphore mutex = 1
Semaphore max_elements = 100
Semaphore elements_ready = 0;

/* Erzeuger */
Element *e = produce();
p(&max_elements);                    //wait(&max_elements)
p(&mutex);                           //wait(&mutex)
enqueue(queue, e );
v(&mutex);                           //signal(&mutex)
v(&elements_ready); //Signal

/* Verbraucher */
Element *e;
p(&elements_ready); //Wait
p(&mutex);                           //wait(&mutex)
e = dequeu(queue);
v(&mutex);                           //signal(&mutex)
v(&max_elements);                    //signal(&max_elements)
consume(e);
```

3 – Speicherverwaltung + virt. Speicher

- b) LRU (Seitenersetzung)

Referenzfolge		5	3	5	1	2	5	4	6	1
	Kachel 1	5								
	Kachel 2									
	Kachel 3									
Kontrollzustände	Kachel 1	0								
	Kachel 2									
	Kachel 3									

3 – Speicherverwaltung + virt. Speicher

- b) LRU (Seitenersetzung)

Referenzfolge		5	3	5	1	2	5	4	6	1
	Kachel 1	5	5	5	5	5	5	5	5	1
	Kachel 2		3	3	3	2	2	2	6	6
	Kachel 3				1	1	1	4	4	4
Kontrollzustände	Kachel 1	0	1	0	1	2	0	1	2	0
	Kachel 2		0	1	2	0	1	2	0	1
	Kachel 3				0	1	2	0	1	2

4a) IO-Scheduling (4 Punkte)

$$L_1=\{1,4,7,2\} \quad L_2=\{3,6,0\} \quad L_3=\{5,2\}$$

Sofort bekannt

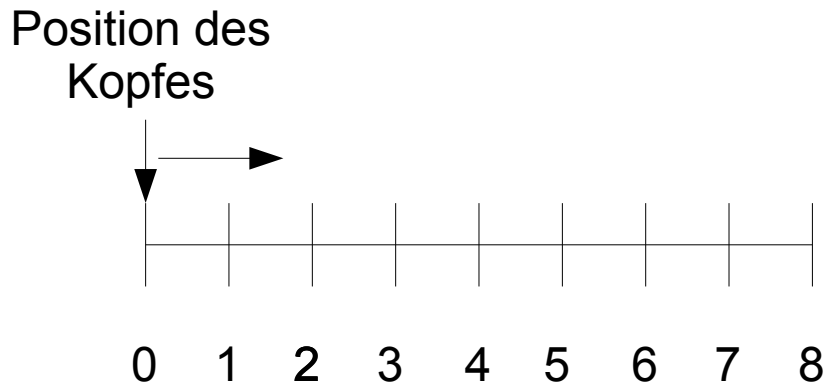
Nach 3 Ops bekannt

Nach 6 Ops bekannt

- Bitte tragen Sie hier die Reihenfolge der gelesenen Spuren für einen I/O-Scheduler, der nach der **Fahrstuhl (Elevator)** Strategie arbeitet, ein:

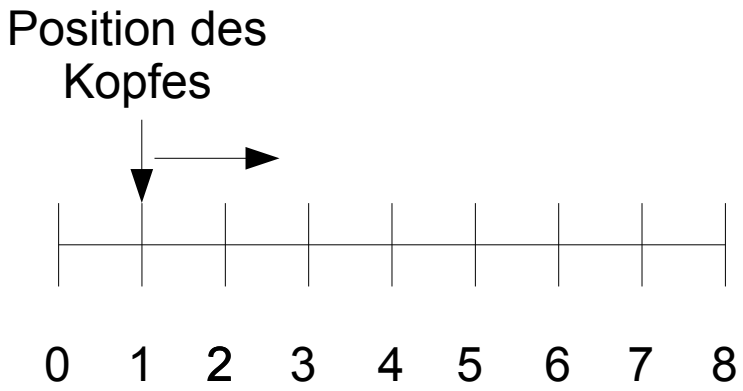
4a) IO-Scheduling (4 Punkte)

T = 0 I/O-Anfragen:
1, 4, 7, 2



4a) IO-Scheduling (4 Punkte)

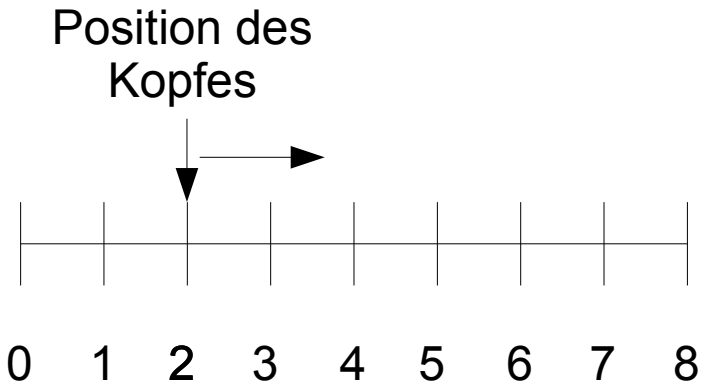
T = 1 I/O-Anfragen:
4, 7, 2



1								

4a) IO-Scheduling (4 Punkte)

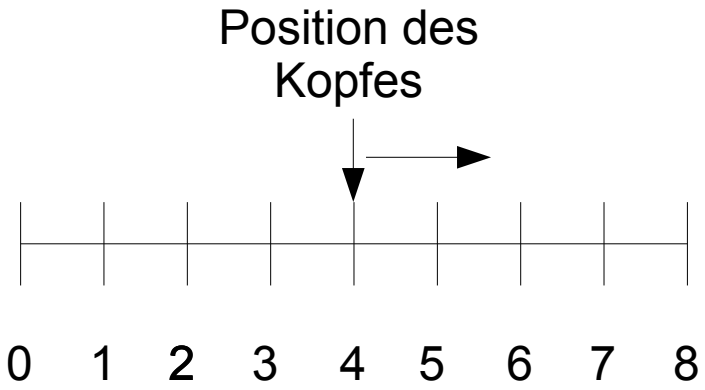
T = 2 I/O-Anfragen:
4, 7



1	2							

4a) IO-Scheduling (4 Punkte)

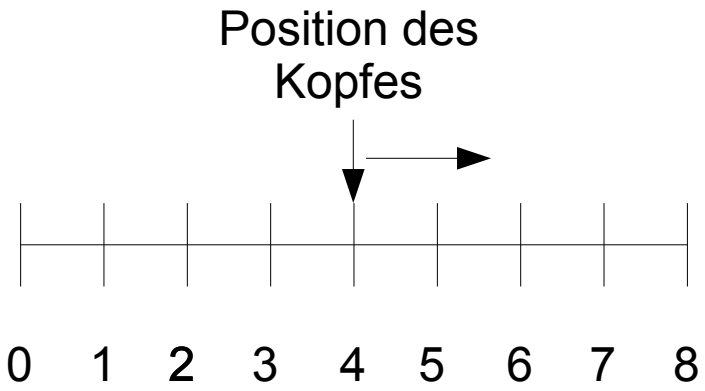
T = 3 I/O-Anfragen:
7



1	2	4						

4a) IO-Scheduling (4 Punkte)

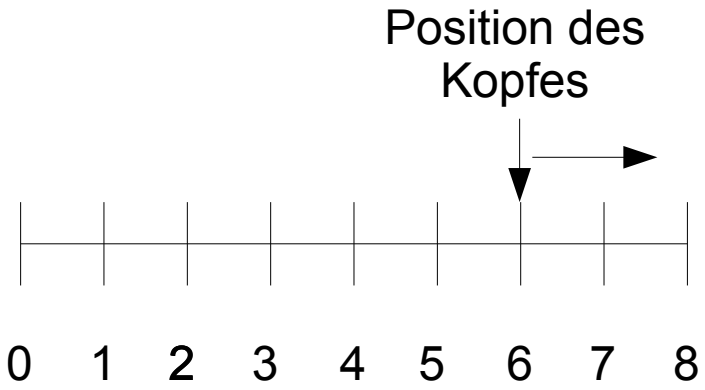
T = 3 I/O-Anfragen:
7, **3, 6, 0**



1	2	4						

4a) IO-Scheduling (4 Punkte)

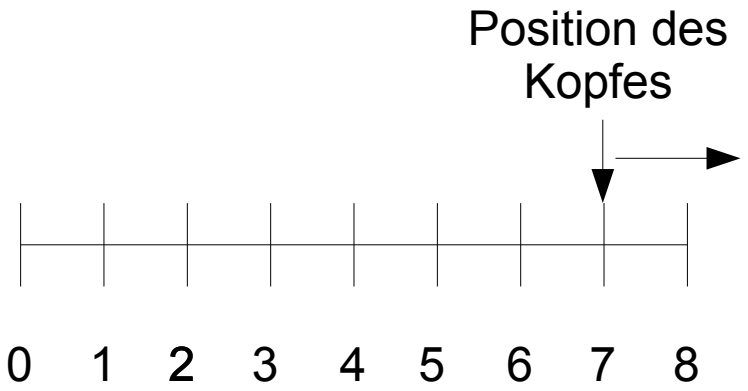
T = 4 I/O-Anfragen:
7, 3, 0



1	2	4	6					

4a) IO-Scheduling (4 Punkte)

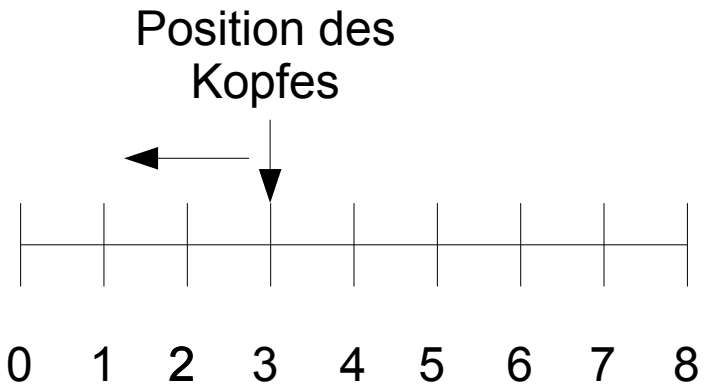
T = 5 I/O-Anfragen:
3, 0



1	2	4	6	7				

4a) IO-Scheduling (4 Punkte)

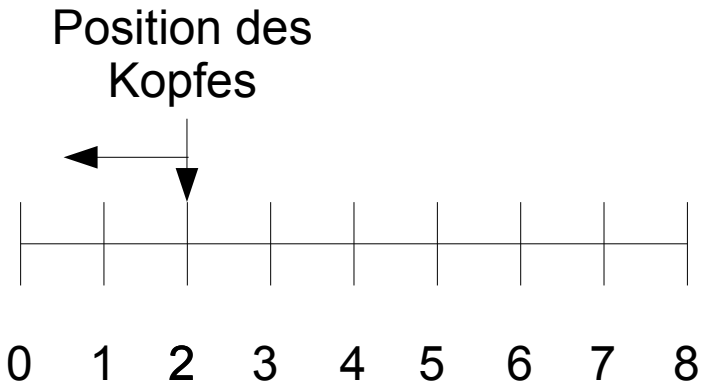
T = 6 I/O-Anfragen:
0, **5, 2**



1	2	4	6	7	3			

4a) IO-Scheduling (4 Punkte)

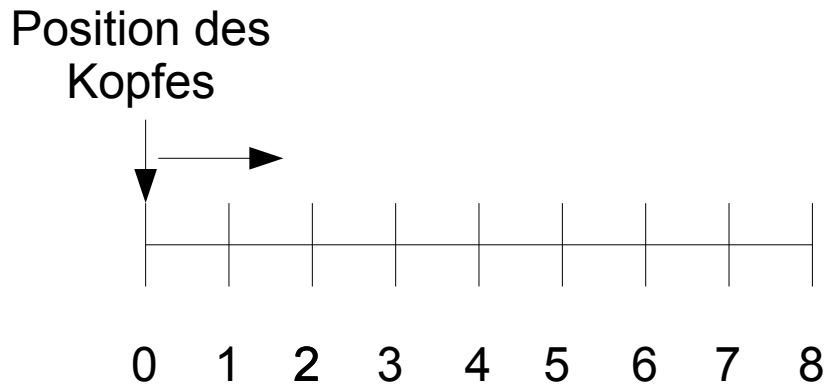
T = 7 I/O-Anfragen:
0, 5



1	2	4	6	7	3	2		

4a) IO-Scheduling (4 Punkte)

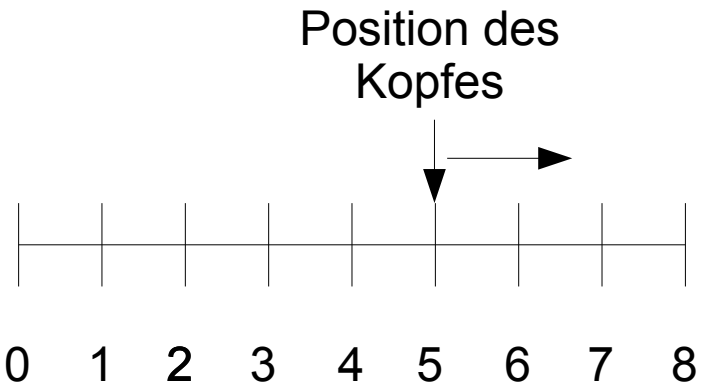
T = 8 I/O-Anfragen:
5



1	2	4	6	7	3	2	0	

4a) IO-Scheduling (4 Punkte)

T = 9 I/O-Anfragen:



1	2	4	6	7	3	2	0	5

4b) Geräteklassen (3 Punkte)

- Nennen Sie die aus der Vorlesung bekannten Geräteklassen von E/A-Geräten und erläutern Sie kurz den Unterschied zwischen denen.
  - Zeichenorientierte Geräte (0.5 Punkte):

    In der Regel sequentieller Zugriff, nicht wahlfrei. (1 Punkt)
  - Blockorientierte Geräte (0.5 Punkte):

    Wahlfreier (random-access) Zugriff. (1 Punkt)

5) Programmieraufgabe (15 Punkte)

- Implementieren Sie das Programm *countfiles*, welches mit einer beliebigen Anzahl an Verzeichnissen, mindestens jedoch mindestens einem, aufgerufen wird und Summe der Anzahl an *Dateien* (nicht *Verzeichnisse*) ausgibt.

5a) main-Funktion (5 Punkte)

```
int main(int argc, const char *argv[]) {
    int sum = 0;

    if (argc < 2) {  // Prüft die Argumentenliste
        printf("Insufficient amount of arguments\n");
        return EXIT_FAILURE;
    }

   // Ruft für jedes Argument count_files auf
   // und summiert die Rückgaben auf
    for (int i = 1; i < argc; i++) {
        sum += count_files(argv[i]);
    }

   // Gebt die Summe aus
    printf("sum of all files: %d\n", sum);

    return EXIT_SUCCESS;
}
```

5b) counf_files-Funktion (10 Punkte)

```
static int count_files(const char *dirName) {
    int sum = 0;
    DIR *dir = opendir(dirName);  // Verzeichnis öffnen
    if (!dir) {/* Fehlerbehandlung */}

    struct dirent *entry;
    do {
        errno = 0;
        entry = readdir(dir); // Einträge lesen
        if (!entry) {
             if (errno != 0) {/* Fehlerbehandlung */}
            break;
        }
        if (is_dot_dir(entry->d_name)) {continue;}

        char *path = concat_paths(dirName, entry→d_name);
       // Typ bestimmen
        if (dir_entry_type(path) == 1) {sum++;}
        free(path);
    } while (entry);
    // Verzeichnis schließen und Anzahl zurückgeben
    if (closedir(dir) == -1){/* Fehlerbehandlung */}
    return sum;
}
```

Auswertung

- Bitte schnell einmal die Punkte zusammenzählen …
- Notenspiegel:

Punkte	Note
38,5–45	1
33,5–38	2
28–33	3
22,5–27,5	4
0–22	5

Weitere Hinweise zur Vorbereitung

- Inhalt der Folien lernen
  - Klassifizieren: Was muss ich **lernen**? Was muss ich **begreifen**?
- Übungsaufgaben verstehen, C und UNIX „können“
  - AsSESS-System bleibt mindestens bis zur Klausur offen
    - bei Fragen zur Korrektur melden
  - Am besten die Aufgaben noch einmal lösen
  - Optionale Zusatzaufgaben bearbeiten
- Literatur zur Lehrveranstaltung durchlesen
- Matrix-Raum nutzen

Empfohlene Literatur

[1] A. Silberschatz et al. *Operating System Concepts*. Wiley, 2004. ISBN 978-0471694663

[2] A. Tanenbaum: Modern Operating Systems (2nd ed.). Prentice Hall, 2001. ISBN 0-13-031358-0

[3] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988. ISBN 0-13-110362-8 (paperback) 0-13-110370-9 (hardback)

[4] R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 2005. ISBN 978-0201433074

Viel Erfolg bei der Klausur!