

Gedächtnisprotokoll - Systemprogrammierung

Klausur vom 30. Juli 2026 · 29 Seiten, insgesamt ca. 100 Punkte (*aus dem Gedächtnis rekonstruiert, kein Anspruch auf Vollständigkeit oder exakten Wortlaut*)

Aufgabe 1: Multiple-Choice

Zehn Kurzfragen (a - j), je 1 Punkt, meist “genau eine Antwort korrekt”, eine Frage mit mehreren möglichen Antworten.

- Grundformen der Prozessinteraktion (Konkurrenz vs. Kooperation), Mehrfachauswahl möglich
 - Definition “Kooperation von Prozessen” (Informationsaustausch + Kenntnis der Existenz anderer Prozesse)
 - Was ist eine Semaphore? (Zählsperre war die richtige Antwort, glaube ich)
 - Definition Prozesssynchronisation
 - Welche Ressource wird **nicht** von Threads eines Prozesses geteilt? (Stack, denke ich)
 - Eigenschaft, die ein Child-Prozess bei fork/join **nicht** hat (Prozessnummer)
 - Aussage über Threads, die zutrifft (Thread fest einem Prozess zugeordnet)
 - Falsche Aussage über Caches (etwas mit Cache-Hit/Cache-Miss)
 - Aussage, die auf unprivilegierten CPU-Zustand **nicht** zutrifft
 - Komponente, die **nicht** zur Definition eines Prozesses gehört (vermutlich “Typ des Betriebssystems”)
-

Aufgabe 2: Prozesse

(a)

- Prozesszustandsdiagramm vervollständigen
- Vorgegeben waren die Zustände “Beendet” und “Blockiert”, dazwischen zwei leere Ellipsen
- Pfeil “Add” führte in einen der leeren Zustände (vermutlich “Bereit”/“Neu”)
- Pfeil “Block” führte von einem Zustand nach “Blockiert”
- Man sollte die fehlenden Zustände (wahrscheinlich Neu/Bereit/Laufend) und Übergänge ergänzen und jeden Zustand mit einem Satz beschreiben

(b)

- Abhängigkeitsgraph mit fork/join in Pseudocode umsetzen
- Graph mit Prozessen P1 - P9, P1 verzweigt zu P2 und P3, dann weitere Verzweigungen/Zusammenführungen über P4 - P7 bis alles in P9 mündet

- Aufgabe: den Graphen mit `fork/join`-Befehlen aus der Abbildung nachbauen
-

Aufgabe 3: Scheduling größte Aufgabe der Klausur

(a)

- Multilevel-Feedback-Scheduling (Round Robin, Quantum $\tau = 2^i$)
- Tabelle mit 5 Prozessen A-E, jeweils Laufzeit und Ankunftszeit (ungefähr: $A=6/1$, $B=3/3$, $C=8/4$, $D=2/7$, $E=7/12$)
- Schedule über 30 Zeiteinheiten in eine Tabelle mit CPU-Zeile und Level-Zeilen (LVL 0-3) eintragen
- anschließend Warte- und Antwortzeiten pro Prozess berechnen

(b)

- Scheduling-Implementierungen zuordnen
- Drei Code-Implementierungen mit Fantasienamen (Kiesel, Wendekreis, Morgenrot) mussten je einem Scheduling-Verfahren aus der Vorlesung zugeordnet werden, mit Begründung anhand eines Code-Merkmals
- Code selbst stand im separaten Begleitheft, nicht in der Hauptklausur

(c)

- Fehlerhafte Implementierung “Nebelschleier” debuggen
- Teilaufgaben: Fehlermeldung/Signal verstehen, Codezeile finden, die den Absturz auslöst, und die Funktion `nebelschleier_tick` korrigieren
- Auch hier: Code im Begleitheft (B-2(c))

(d)

- Lückentext zu einer MLF-Scheduling-Implementierung, drei Lücken
 - **Lücke (1)**
 - Quantum abgelaufen/Prozesswechsel: 5 Codeblock-Optionen (A-E), genau einer korrekt; ging um Zurückstufung auf nächstes Level (max. Level 3, `min(current_level+1, 3)`) und Nicht-Wiedereinordnen fertiger Prozesse
 - **Lücke (2)**
 - nächsten Prozess aus den vier Levels wählen (Level 3 = LCFS, sonst FCFS): fünf Codeblöcke A-E, jeweils korrekt/falsch ankreuzen; ging um Reihenfolge `queue_poll` vs. `queue_poll_tail` je nach Level
 - **Lücke (3)**
 - Neuankömmling einordnen: 5 Optionen, genau eine korrekt (vermutlich Einordnung auf Level 0, also `mlf_add_queue(..., 0)`)
-

Aufgabe 4: Koordination

(a)

- Parkhaus-Beispiel mit 8 Stellplätzen
- Pseudocode mit `lock/unlock` und aktivem Warten (Spinlock/Busy-Waiting) vervollständigen
- Globale Variable `belegt`, Funktionen `init()`, `proc_auto()` mit `enter_Parkhaus()/leave_Parkhaus()` sollten ergänzt werden, sodass nie mehr als 8 Autos gleichzeitig im Parkhaus sind

(b)

- Vier Anforderungen an kritische Abschnitte bei der Prozesssynchronisation nennen (vermutlich: gegenseitiger Ausschluss, kein Deadlock, kein Verhungern/Fairness, keine Annahmen über relative Geschwindigkeit der Prozesse)
-

Aufgabe 5: Betriebsmittelverwaltung

(a)

- Graph mit Prozessen P1-P4 und Betriebsmitteln BM1-BM4 (je mit mehreren Instanzen, dargestellt als Kästchen)
- Aus dem Graphen sollten Belegungsmatrix **B**, Restanforderungsmatrix **R**, Gesamtanforderungsmatrix **G**, Vektor der freien Betriebsmittel **f** und Vektor der vorhandenen Betriebsmittel **v** abgelesen werden
- Zwei identische Tabellensets waren vorgegeben (man musste nur eins ausfüllen, das andere war Reserve bei Fehlern)

(b)

- Bankier-Algorithmus auf die Situation aus (a) anwenden
- Schritte inkl. Freiheitsvektoren angeben und begründen, ob der Zustand sicher ist

(c)

- Drei Klassen von Betriebsmitteln nennen, mit je einem Beispiel (vermutlich: entziehbare vs. nicht-entziehbare, bzw. Hardware-/Software-Betriebsmittel; genaue Klassifikation aus der Vorlesung war gefragt)
-

Aufgabe 6: Speicherverwaltung

(a)

- Unterschied statische vs. dynamische Speicherallokation beschreiben

(b)

- Drei Kontroll-/Statusbits (Flags) in einem Seitentabelleneintrag nennen (z. B. Present/Valid-Bit, Dirty-Bit, Referenced/Accessed-Bit ungefähr in diese Richtung)
-

Aufgabe 7: Dateisystem - zweitgrößte Aufgabe

(a)

- Löschen einer Datei in drei Schritten (Verzeichniseintrag entfernen -> Inode freigeben -> Datenblöcke in Freiliste markieren)
- Frage: Wie heißt der Inode-Zustand, wenn das System zwischen Schritt 1 und 2 abstürzt, und was bedeutet das konkret?
- (Stichwort: verwaister/“orphaned” Inode ist noch als belegt markiert, aber ohne Verzeichniseintrag erreichbar -> Speicherplatzverlust bis zur Konsistenzprüfung, z. B. fsck)

(b)

- Lückentext `fs_mkdir`-Funktion (Verzeichnis anlegen), 4 Lücken, je genau eine C-Zeile
- Struktur: freien Inode suchen -> Inode initialisieren (Name setzen, Typ, Größe) -> im Elternverzeichnis eintragen
- Lücken betrafen u. a. das Setzen von `new_inode_idx = i`, Initialisieren von Feldern wie `n_type/parent`, und den Eintrag in `parent->direct_blocks[j]`

(c)

- Lückentext `find_child_inode`-Funktion (Kind-Eintrag in Verzeichnis suchen), 4 Lücken
- Schleife über `DIRECT_BLOCKS_COUNT`, Vergleich von `search_name` mit Namen der Kind-Inodes, Rückgabe der Inode-Nummer oder -1

(d)

- Lückentext zu `create_directory`, drei Lücken (größte Teilaufgabe in Aufgabe 7)
- **Lücke (1)**
- freien Inode finden: 5 Codeblöcke A-E, jeweils korrekt/falsch; ging um Schleife über `fs->inodes[i].n_type == free_block` vs. Alternativen mit `free_list` o. Ä.
- **Lücke (2)**
- Inode initialisieren: 5 Optionen A-E, genau eine korrekt (richtige Reihenfolge: `n_type = directory`, `parent = parent_inode_idx`, Name kopieren, `size = 0`)
- **Lücke (3)**

- Eintrag im Elternverzeichnis: 5 Optionen A-E, genau eine korrekt (Schleife über `parent->direct_blocks[j]`, freien Slot mit `new_inode_idx` befüllen)
-

Aufgabe 8: Virtualisierung

(a)

- Zwei Diagramme (Prozessvirtualisierung vs. Systemvirtualisierung) mit den Schichten *Anwendung*, *Virtualisierungssoftware* und *Betriebssystem* korrekt beschriften, jeweils oberhalb von “Hardware”
- Prozessvirtualisierung: vermutlich Virtualisierungssoftware sitzt zwischen (Gast-)Anwendung und Betriebssystem/Hardware, mit einer Schicht, die nur einen Teilbereich abdeckt
- Systemvirtualisierung: Virtualisierungssoftware (Hypervisor) direkt über Hardware, darüber Betriebssystem(e) und Anwendung(en)

Begleitheft zur Klausur Systemprogrammierung

Codeauszüge und Dokumentation

Zur Klausur am 30. Juli 2026

Dieses Begleitheft enthält ausschließlich Material zur Referenz während der Bearbeitung. Es sollen hier keine Lösungen eingetragen werden. Die Inhalte sind jeweils auf konkrete Aufgaben im Klausurheft referenziert, damit bei Bedarf nur die benötigten Abschnitte ausgedruckt werden müssen.

Inhaltsverzeichnis

B-2 Scheduling: gemeinsame Referenz	1
B-2(b) Scheduling-Implementierungen zuordnen	3
B-2(c) Scheduling: Segmentation Fault	6
B-2(d) Scheduling: Lückentext (Multiple Choice)	8
B-6 Dateisystem: gemeinsame Referenz	10
B-6(b) Dateisystem: Lückentext (Multiple Choice)	11

B-2 Scheduling: gemeinsame Referenz

Diese Referenz gehört zu den **Aufgaben 2(b), 2(c) und 2(d)** im Klausurheft. Sie enthält Ausschnitte aus den Header-Dateien der Scheduling-Praxisaufgabe.

Prozessdarstellung (**process.h**)

Listing 1: B-2-Ref-A: process.h (Ausschnitt)

```
1 typedef struct _process {  
2     unsigned int start_time;  
3     unsigned int time_left;  
4     unsigned int priority;  
5     char id;  
6 } process;
```

Listing 4: B-2(b)-B: Implementierung Wendekreis

```

1 static queue_object *wendekreis_queue;
2
3 process *wendekreis_tick(process *running_process)
4 {
5     if (running_process == NULL || running_process->time_left == 0) {
6         running_process = queue_poll(wendekreis_queue);
7     }
8
9     if (running_process != NULL) {
10         running_process->time_left--;
11     }
12
13     return running_process;
14 }
15
16 process *wendekreis_new_arrival(process *arriving_process,
17                                 process *running_process)
18 {
19     if (arriving_process != NULL) {
20         if (running_process != NULL && running_process->time_left > 0) {
21             queue_add(running_process, wendekreis_queue);
22         }
23         return arriving_process;
24     }
25
26     return running_process;
27 }

```

B-2(b) Scheduling-Implementierungen zuordnen

Dieser Abschnitt gehört zu **Aufgabe 2(b)** im Klausurheft. Die folgenden drei Implementierungen basieren auf Scheduling-Algorithmen aus der Scheduling-Praxishausaufgabe, wurden aber absichtlich mit irreführenden Namen versehen. Nutzen Sie bei Bedarf auch die gemeinsame Referenz **B-2**.

Listing 3: B-2(b)-A: Implementierung Kiesel

```
1 static queue_object *kiesel_queue;
2 static int kiesel_quantum;
3 static int kiesel_quantum_left;
4
5 process *kiesel_tick(process *running_process)
6 {
7     if (running_process == NULL || running_process->time_left == 0
8         || kiesel_quantum_left == 0) {
9         if (running_process != NULL && kiesel_quantum_left == 0
10            && running_process->time_left != 0) {
11             queue_add(running_process, kiesel_queue);
12         }
13         running_process = NULL;
14     }
15
16     if (running_process == NULL) {
17         running_process = queue_poll(kiesel_queue);
18         if (running_process != NULL) {
19             kiesel_quantum_left = kiesel_quantum;
20         }
21     }
22
23     if (running_process != NULL) {
24         running_process->time_left--;
25         kiesel_quantum_left--;
26     }
27
28     return running_process;
29 }
30
31 process *kiesel_new_arrival(process *arriving_process,
32                             process *running_process)
33 {
34     if (arriving_process != NULL) {
35         queue_add(arriving_process, kiesel_queue);
36     }
37     return running_process;
38 }
```


Listing 4: B-2(b)-B: Implementierung Wendekreis

```

1 static queue_object *wendekreis_queue;
2
3 process *wendekreis_tick(process *running_process)
4 {
5     if (running_process == NULL || running_process->time_left == 0) {
6         running_process = queue_poll(wendekreis_queue);
7     }
8
9     if (running_process != NULL) {
10        running_process->time_left--;
11    }
12
13    return running_process;
14 }
15
16 process *wendekreis_new_arrival(process *arriving_process,
17                                 process *running_process)
18 {
19     if (arriving_process != NULL) {
20         if (running_process != NULL && running_process->time_left > 0) {
21             queue_add(running_process, wendekreis_queue);
22         }
23         return arriving_process;
24     }
25
26     return running_process;
27 }

```

Listing 5: B-2(b)-C: Implementierung Morgenrot

```

1 static queue_object *morgenrot_queue;
2
3 process *morgenrot_tick(process *running_process)
4 {
5     if (running_process == NULL || running_process->time_left == 0) {
6         running_process = queue_poll(morgenrot_queue);
7     }
8     if (running_process != NULL) {
9         running_process->time_left--;
10    }
11    return running_process;
12 }
13
14 process *morgenrot_new_arrival(process *arriving_process,
15                                process *running_process)
16 {
17     process *queued;
18
19     if (arriving_process == NULL) {
20         return running_process;
21     }
22     if (running_process == NULL) {
23         queue_add(arriving_process, morgenrot_queue);
24         return running_process;
25     }
26
27     if (arriving_process->time_left < running_process->time_left) {
28         queued = running_process;
29     } else {
30         queued = arriving_process;
31     }
32
33     /* queued wird nach Restlaufzeit in die Queue einsortiert */
34     queue_object *old = morgenrot_queue;
35     morgenrot_queue = new_queue();
36     int inserted = 0;
37     while (queue_peek(old) != NULL) {
38         process *polled = queue_poll(old);
39         if (!inserted && polled->time_left > queued->time_left) {
40             queue_add(queued, morgenrot_queue);
41             inserted = 1;
42         }
43         queue_add(polled, morgenrot_queue);
44     }
45     if (!inserted) {
46         queue_add(queued, morgenrot_queue);
47     }
48     free_queue(old);
49
50     if (queued == running_process) {
51         return arriving_process;
52     }
53     return running_process;
54 }

```


B-2(c) Scheduling: Segmentation Fault

Dieser Abschnitt gehört zu **Aufgabe 2(c)** im Klausurheft. Gegeben ist eine (absichtlich fehlerhafte) Scheduling-Implementierung, die an FCFS aus der Scheduling-Praxisausaufgabe angelehnt ist, sowie die beim Ausführen beobachtete Fehlermeldung. Nutzen Sie bei Bedarf auch die gemeinsame Referenz B-2.

Implementierung Nebelschleier (fehlerhaft)

Listing 6: B-2(c)-A: fehlerhafte Implementierung Nebelschleier

```
1 static queue_object *nebelschleier_queue;
2
3 process *nebelschleier_tick(process *running_process)
4 {
5     if (running_process == NULL || running_process->time_left == 0) {
6         running_process = queue_poll(nebelschleier_queue);
7     }
8     running_process->time_left--;
9     return running_process;
10 }
11
12 int nebelschleier_startup()
13 {
14     nebelschleier_queue = new_queue();
15     if (nebelschleier_queue == NULL) {
16         return 1;
17     }
18     return 0;
19 }
20
21 process *nebelschleier_new_arrival(process *arriving_process,
22                                     process *running_process)
23 {
24     if (arriving_process != NULL) {
25         queue_add(arriving_process, nebelschleier_queue);
26     }
27     return running_process;
28 }
29
30 void nebelschleier_finish()
31 {
32     free_queue(nebelschleier_queue);
33 }
```

Fehlermeldung

Die Implementierung wird mit dem folgenden Testfall ausgeführt:

Listing 7: B-2(c)-B: verwendeter Testfall

```
1 +-----+
2 | ID | Start Time | Duration | Priority |
3 +-----+
4 | A |          0 |          3 |          1 |
5 | B |          4 |          2 |          1 |
6 +-----+
```

Dabei tritt folgende Fehlermeldung auf:

Listing 8: B-2(c)-C: beobachtete Fehlermeldung

```
1 $ ./nebelschleier
2 Segmentation fault (core dumped)
3
4 $ gdb -q ./nebelschleier
5 (gdb) run
6 Program received signal SIGSEGV, Segmentation fault.
7 0x0000555555555485 in nebelschleier_tick (running_process=0x0) at nebelschleier.c:8
8      running_process->time_left--;
```

B-2(d) Scheduling: Lückentext (Multiple Choice)

Dieser Abschnitt gehört zu **Aufgabe 2(d)** im Klausurheft. Gegeben ist eine lückenhafte MLF-Implementierung in Anlehnung an die Scheduling-Praxisaufgabe (4 Level, $\tau = (i + 1)!$, Level 3 mit LCFS). Die nummerierten Lücken (1)–(3) sind im Klausurheft als Multiple-Choice-Fragen zu beantworten. Nutzen Sie bei Bedarf auch die gemeinsame Referenz **B-2**.

Hilfsfunktionen (gegeben)

Listing 9: B-2(d)-A: gegebene Hilfsfunktionen

```
1 static unsigned int min(unsigned int a, unsigned int b);
2
3 /* Tau = (level + 1)! */
4 static unsigned int calc_MLF_quantum(unsigned int level);
5
6 /* Fuegt arriving_process in MLF_queues[level] ein. */
7 int mlf_add_queue(process *arriving_process,
8                  process *running_process,
9                  int level);
10
11 /**
12  * Entfernt und liefert das juengste Element (LCFS / vom Queue-Ende).
13  * Fuer Level 0..2 wird queue_poll (FCFS) verwendet.
14  */
15 void *queue_poll_tail(queue_object *queue);
```


Lückenhafte Implementierung MLF

Listing 10: B-2(d)-B: lückenhafte MLF-Implementierung

```
1 static queue_object **MLF_queues; /* 4 Queues: Level 0..3 */
2 static unsigned int time_left_quantum;
3 static unsigned int current_level;
4
5 process *MLF_tick(process *running_process)
6 {
7     /* ===== Luecke (1) ===== */
8     /*                                     */
9     /*                                     */
10    /*                                     */
11    /*                                     */
12    /*                                     */
13    /* ===== */
14
15    process *new_running_process = running_process;
16    if (new_running_process == NULL) {
17        /* ===== Luecke (2) ===== */
18        /*                                     */
19        /*                                     */
20        /*                                     */
21        /*                                     */
22        /*                                     */
23        /* ===== */
24    }
25
26    if (new_running_process != NULL) {
27        new_running_process->time_left--;
28        time_left_quantum--;
29    }
30    return new_running_process;
31 }
32
33 process *MLF_new_arrival(process *arriving_process,
34                          process *running_process)
35 {
36     if (arriving_process != NULL) {
37         /* ===== Luecke (3) ===== */
38         /*                                     */
39         /* ===== */
40     }
41     return running_process;
42 }
```

Hinweis: MLF_startup allokiert MLF_queues[0..3] und setzt current_level = 0. Das muss hier nicht ergänzt werden.

B-6 Dateisystem: gemeinsame Referenz

Dieser Abschnitt gehört zu den **Aufgaben 6(a) und 6(b)** im Klausurheft. Er enthält Ausschnitte aus der Dateisystem-Praxisaufgabe (filesystem.h) sowie kurze Hinweise zum Datenmodell.

Konstanten und Knotentypen

Listing 11: B-6-Ref-A: Konstanten und enum node_type

```
1 #define BLOCK_SIZE 1024
2 #define NAME_MAX_LENGTH 32
3 #define DIRECT_BLOCKS_COUNT 12
4
5 enum node_type {
6     reg_file    = 1,
7     directory   = 2,
8     free_block  = 3
9 };
```

Inode und Dateisystem

Listing 12: B-6-Ref-B: inode und file_system

```
1 typedef struct _inode {
2     enum node_type n_type;
3     uint16_t size;
4     char name[NAME_MAX_LENGTH];
5     /* Verzeichnis: Indizes der Kind-Inodes;
6      * Datei: Indizes der Datenblöcke.
7      * Ungenutzte Einträge haben den Wert -1. */
8     int direct_blocks[DIRECT_BLOCKS_COUNT];
9     int parent; /* Inode-Nummer des Elternknotens, -1 falls keiner */
10 } inode;
11
12 typedef struct _superblock {
13     uint32_t num_blocks;
14     uint32_t free_blocks;
15 } superblock;
16
17 typedef struct _fs {
18     superblock *s_block;
19     uint8_t *free_list; /* Datenblock frei == 1 */
20     inode *inodes;
21     data_block *data_blocks;
22     int root_node;
23 } file_system;
```

Hinweise zum Datenmodell

- Ein freier Inode erkennt man an `n_type == free_block`.
- Beim Anlegen eines Verzeichnisses wird ein freier Inode belegt (`n_type = directory`), der Name und der Parent gesetzt und der neue Inode-Index im `direct_blocks`-Array des Elternverzeichnisses eingetragen.
- Die `free_list` verwaltet **Datenblöcke**, nicht die Freiheit von Inodes.
- Zum Namensvergleich können Sie `strncmp(..., NAME_MAX_LENGTH)` verwenden.

- Zum Kopieren eines Namens in `inode.name` können Sie `strncpy(..., NAME_MAX_LENGTH - 1)` verwenden (danach das letzte Byte manuell auf `'\0'` setzen).

B-6(b) Dateisystem: Lückentext (Multiple Choice)

Dieser Abschnitt gehört zu **Aufgabe 6(b)** im Klausurheft. Gegeben ist eine lückenhafte Hilfsfunktion zum Anlegen eines Verzeichnisses in Anlehnung an die Dateisystem-Praxisaufgabe (`create_in_dir` / `fs_mkdir`). Die nummerierten Lücken (1)–(3) sind im Klausurheft als Multiple-Choice-Fragen zu beantworten. Nutzen Sie bei Bedarf auch die gemeinsame Referenz **B-6**.

Lückenhafte Funktion `create_directory`

Listing 13: B-6(b)-A: lückenhafte `create_directory`

```

1  /* parent_inode_idx: Inode-Nummer des Elternverzeichnisses
2  * dir_name:         Name des neuen Verzeichnisses
3  * Rueckgabe: 0 bei Erfolg, -1 bei Fehler
4  * (vereinfacht: kein Namenskonflikt-Check, Parent existiert) */
5  int create_directory(file_system *fs, int parent_inode_idx,
6                      char *dir_name)
7  {
8      int new_inode_idx = -1;
9
10     /* ===== Luecke (1) ===== */
11     /* Freien Inode suchen und Index in new_inode_idx speichern.
12        Falls keiner frei ist: return -1; */
13     /*
14     /*
15     /*
16     /*
17     /* ===== */
18
19     inode *new_dir = &fs->inodes[new_inode_idx];
20     inode *parent  = &fs->inodes[parent_inode_idx];
21
22     /* ===== Luecke (2) ===== */
23     /* Neuen Inode als Verzeichnis initialisieren
24        (Typ, Parent, Name, size). */
25     /*
26     /*
27     /*
28     /*
29     /* ===== */
30
31     /* ===== Luecke (3) ===== */
32     /* Neuen Inode-Index im Elternverzeichnis eintragen
33        (ersten freien direct_blocks-Slot mit -1 belegen). */
34     /*
35     /*
36     /*
37     /*
38     /* ===== */
39
40     return 0;
41 }

```