Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Frank Bellosa
Peter Maucher, M.Sc.
Lukas Werling, M.Sc.
Yussuf Khalil, M.Sc.

Musterlösung Hauptklausur

29.02.2024

Alle Punkteangaben ohne Gewähr!

- Bitte tragen Sie zuerst auf dem Deckblatt Ihren Namen, Ihren Vornamen und Ihre Matrikelnummer ein. Tragen Sie dann auf den anderen Blättern (auch auf Konzeptblättern) Ihre Matrikelnummer ein.

  Please fill in your last name, your first name, and your matriculation number on this page and fill in your matriculation number on all other (including draft) pages.

- Die Prüfung besteht aus 17 Blättern: Einem Deckblatt, 16 Aufgabenblättern mit insgesamt 3 Aufgaben sowie 0 Seiten Man-Pages.

  The examination consists of 17 pages: One cover sheet, 16 sheets containing 3 assignments, and 0 sheets with man pages.

- Es sind keinerlei Hilfsmittel erlaubt!

  No additional material is allowed!

- Die Prüfung ist nicht bestanden, wenn Sie versuchen, aktiv oder passiv zu betrügen.

  You fail the examination if you try to cheat actively or passively.

- Sie können auch die Rückseite der Aufgabenblätter für Ihre Antworten verwenden. Wenn Sie zusätzliches Konzeptpapier benötigen, verständigen Sie bitte die Klausuraufsicht.

  You can use the back side of the assignment sheets for your answers. If you need additional draft paper, please notify one of the supervisors.

- Bitte machen Sie eindeutig klar, was Ihre endgültige Lösung zu den jeweiligen Teilaufgaben ist. Teilaufgaben mit mehreren widersprüchlichen Lösungen werden mit 0 Punkten bewertet.

  Make sure to clearly mark your final solution to each question. Questions with multiple, contradicting answers are void (0 points).

- Programmieraufgaben sind gemäß der Vorlesung in C zu lösen.

  Programming assignments have to be solved in C.

Die folgende Tabelle wird von uns ausgefüllt!
The following table is completed by us!

Aufgabe	1	2	3	Total
Max. Punkte	20	20	20	60
Erreichte Punkte				

Aufgabe 1: Virtualisierung

Assignment 1: Virtualization

a) Erklären Sie Programm, Prozess und Adressraum. Nennen Sie insgesamt vier Dinge, die nur entweder im Programm oder im Prozess vorkommen, aber nicht in beidem. **3.5 pt**

Explain program, process, and address space. Name a total of four things that are only part of either program or process, but not both.

Solution:

A program is a static entity that can be started and loaded into a dynamic runtime process **(0.5 P)**. *The address space is part of the runtime process and denotes how memory is mapped* **(0.5 P)**, *and the layout of the address space to be created is described in the program* **(0.5 P)**.

Only Program:

- *segment descriptors* **(0.5 P)**
- *relocation tables* **(0.5 P)**
- *debug symbols* **(0.5 P)**
- *description how to launch the process (`#!`, description in ELF, ...)* **(0.5 P)**

Common mistake: *The path / filename is also stored in the process as it is passed to the process in* `argv[0]`.

Only Process:

- *register state (of halted threads)* **(0.5 P)**
- *program counter* **(0.5 P)**
- *threads* **(0.5 P)**
- *open file table* **(0.5 P)**
- *runtime segments (stack, heap)* **(0.5 P)**
- *dynamiccally loaded libraries / shared objects* **(0.5 P)**
- *process control block (PCB)* **(0.5 P)**
- *signals*
- *children*

Note: *The answer "process state" was not accepted as it is too general, basically everything has state.*

b) Was ist ein Thread und woraus besteht er? Nennen Sie zwei Vorteile von Multithreading gegenüber Multiprogramming, und zwei Vorteile von Multiprogramming. **3 pt**

What is a thread and what is it made of? Give two advantages of multithreading over multiprogramming, and two for multiprogramming.

Solution:

A thread is an abstraction of an execution state inside of the processes address space **(0.5 P)**, *which encapsulates register state* **(0.5 P)**, *an instruction pointer, and a stack.*

(+) Multithreading:

- *single address space allows for easy shared memory communication* **(0.5 P)**
- *thread switches are cheap (no AS change needed)* **(0.5 P)**
- *thread creation usually cheaper than process creation* **(0.5 P)**
- *widely available interface with a lot of language support* **(0.5 P)**
- *less resource usage in kernel* **(0.5 P)**

(+) Multiprogramming:

- *address separation: only shared what is needed ⇐ more security* **(0.5 P)**
- *different processes can run on different machines (e.g., MPI)* **(0.5 P)**
- *communication via inter-process-communication interfaces might be easier than shared memory* **(0.5 P)**
- *different privileges per process* **(0.5 P)**
- *long term scheduling* **(0.5 P)**

Common mistakes:

(a) Multiprogramming is not inherently race-condition free or deadlock free, as IPC mechanisms can be used. This also means that (if actively configured) shared memory can be allocated in multiple processes to break a lot of separation.

(b) On the other hand, multiprogramming does allow parallel execution (e.g., on different CPU cores), and, using the IPC mechanisms mentioned above, can also be used to work one singular exercise between processes.

c) Der Betriebssystemkernel läuft nicht dauerhaft. Wie kann er trotzdem Hintergrundaufgaben und Aufräumaufgaben abarbeiten? Nennen Sie zwei Möglichkeiten. **1 pt**

The operating system kernel is not running constantly. How can it still work on bookkeeping and background tasks? Name two possibilities.

Solution:

The OS has three options: do bookkeeping when it is called anyway via a system call **(0.5 P)**, *exception* **(0.5 P)** *or interrupt* **(0.5 P)**, *use a kernel-mode thread* **(0.5 P)**, *or have a deamon with root privileges run that enters the kernel via syscalls* **(0.5 P)**.

d) Erklären Sie das Konzept Calling Convention. **1 pt**

Explain the concept of a calling convention.

Solution:

When a function in one compilation unit calls another function **(0.5 P)** *in a different compilation unit, the two functions need to agree on the placement of arguments and return values* **(0.5 P)**, *as well as other factors like alignment* **(0.5 P)** *or even function names* **(0.5 P)**. *All these are specified in the calling convention.*
Common mistake: *The calling convention does not govern the structure of the system call interface, and in fact the system call interface can diverge quite heavily (e.g., if you run Wine on Linux) from the calling convention of the process. Usually, the kernel also uses a similar but slightly different calling convention internally as compared to userspace.*

e) Was ist der Unterschied zwischen Base- und Limitregistern und Segmentation? **1 pt**

What is the difference between base and limit registers and segmentation?

Solution:

The idea of base+limit registers is that physical addresses are used in the process, but the process can only access those physical addresses between the (lower) base and the upper bound base+limit **(0.5 P)**. *Segmentation uses multiple base+limit pairs which are selected in the process* **(0.5 P)**, *usually with a bit prefix in the virtual address.*

f) Erklären Sie die Funktionsweise des Buddy-Allokators. Nennen Sie einen Nachteil dieser Technik, und eine weitere Allokationsstrategie die diesen Nachteil nicht hat. **3 pt**

Explain how the buddy allocator works. Name one disadvantage of this technique, and another allocation strategy which does not have that disadvantage.

Solution:

The idea behind a buddy allocator is to break the address space into a block whose length is a power of two **(0.5 P)**. *Whenever memory is requested, round it up to the nearest power of two, check whether a block of this size is currently available in the buddy* **(0.5 P)**. *When no such block is available, recursively try the next larger block, and split it into two blocks* **(0.5 P)**. *When a block and its neighbour are both free again, merge them recursively* **(0.5 P)**. *Disadvantage: buddy allocators suffer from major internal fragmentation (up to a factor of* $2 - \epsilon$*)* **(0.5 P)**, *which could be solved by, e.g., linked list allocation* **(0.5 P)** *or slab allocators of a fitting size* **(0.5 P)**. *Disadvantage: if two adjacent blocks are free, but not a child of the same parent, they cannot be merged* **(0.5 P)**. *Again, linked list allocation* **(0.5 P)** *does not suffer from these problems.*
Common mistakes: *Both segmentation and paging are not allocation strategies, but memory translation mechanisms, and while they may help in designing an allocation strategy, they themselves are unrelated and therefore not accepted.*

g) Was ist der Vorteil eines read-only Mappings einer Datei, wenn man swappen muss? **1 pt**

What is the advantage of a file mapped read-only when you need to swap?

Solution:

If the file is mapped read-only, the process cannot have modified it. For swapping, we therefore do not need to explicitly swap the contents out **(0.5 P)**, *but can just remove the pages from memory* **(0.5 P)** *and load them from disk later* **(0.5 P)** *via the file system.*
Note: *No points were awarded for a discussion of either swapping or read-only mappings if they did not also talk about the other half of the question.*

h) Schreiben Sie den `C`-Aufruf des `mmap()`-Systemaufrufs, um 2 MiB an les- und schreibbarem, privatem Speicher in ihren Prozess zu mappen. **2 pt**

Write down the `C` call to the `mmap()` system call to map 2 MiB *of readable and writable, private memory into your process.*

Solution:

```
mmap(NULL, (1<<21), PROT_READ | PROT_WRITE,
     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

(0.5 P) *for the correct size* if stated in C, **(0.5 P)** *for protection,* **(0.5 P)** *for the flags,* **(0.5 P)** *for the rest.*

i) In dieser Aufgabe betrachten Sie ein System mit 32 bit Addressbreite, 4 KiB Seiten und einer zweistufigen Seitentabelle. Beschreiben Sie zunächst, wie die Adressübersetzung funktioniert, und geben Sie die Anzahl der Bits für die erste und zweite Stufe der Seitentabelle sowie den Offset innerhalb der Seite an. **1.5 pt**

In this exercise you will consider a system with 32 bit *addresses,* 4 KiB *pages and a two-level page table. First, describe how the address translation works, and give the number of bits for the first and second level of the page table as well as the offset into the page.*

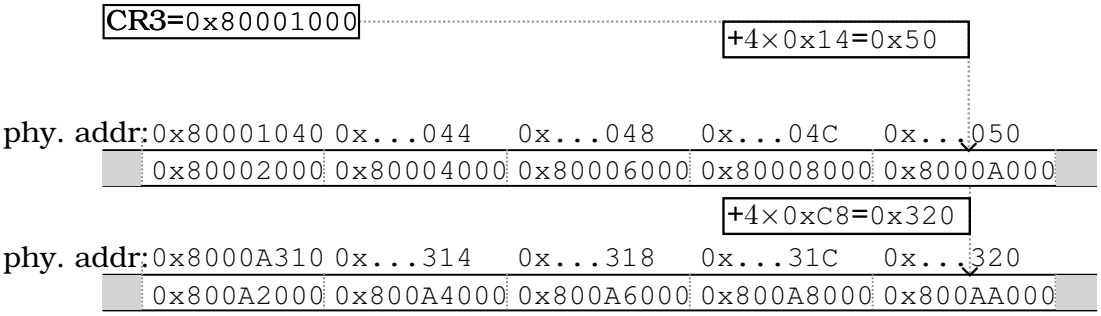
Solution:

The virtual address is divided into three parts, the first and second level page table index as well as tthe offset. Whenever an address is accessed, we first index into the page given by the CR3 register with the first level index, then the page given by that with the second level index **(0.5 P)**. *We then add the offset to the address we just loaded, which is the physical address* **(0.5 P)**.

Bits First Level: 10 **Bits Second Level:** 10 **Bits Offset:** 12

Zu dem oben beschriebenen System mit 32 bit Adressen und 4 KiB Seitengröße finden Sie folgendes Speicherabbild inlusive CR3-Register mit dem Einstiegspunkt in die Seitentabelle. Füllen Sie anhand dessen die Tabelle aus, und zeichnen Sie Pfeile zwischen den Speicheradressen, anhand derer Sie ihre Speicherübersetzung durchführen. **3 pt**

For the system described above with 32 bit *addresses and* 4 KiB *page size you find the following memory dump including the CR3 register which holds the entry point into the page table. Use it to fill out the table below, and draw arrows along the memory addresses which you use for memory translation.*



Virtual Address	First Level	Second Level	Offset	Physical Address
0x050C8FEE	0x014	0xC8	0xFEE	*0x800AAFEE*

Solution:

(1 P) *each for the correct virtual and physical addresses, and* **(1 P)** *for the arrows.* **Total: 20.0pt**

Aufgabe 2: Nebenläufigkeit

Assignment 2: Concurrency

Apple-Betriebssysteme (bspw. macOS) verfügen über ein Parallelisierungsframework namens *Grand Central Dispatch* (GCD). Mittels GCD können Anwendungsentwickler nebenläufige Aufgaben definieren und diese parallel ausführen lassen, ohne sich Gedanken über die Verwaltung von Threads machen zu müssen. Die zentrale Idee von GCD ist das Konzept von *dispatch queues*. Ein:e Entwickler:in fügt Aufgaben zu einer *dispatch queue* hinzu, welche dann von GCD in FIFO-Reihenfolge (first-in, first-out) abgearbeitet werden. Wir betrachten zunächst den einfachsten Fall, sogenannte *serial dispatch queues*. Diese führen jeweils nur eine Aufgabe gleichzeitig aus.

Die Funktion
`void dispatch(queue *q, void (*func)(void *arg), void *arg)`
fügt eine Aufgabe `func` mit dem Argument `arg` in die Warteschlange `q` ein. Diese wird dann asynchron von GCD abgearbeitet. `dispatch()` ist garantiert frei von *race conditions*.

`account_q` sei eine gültige, definierte *serial dispatch queue*. `my_account` sei eine gültige Instanz von `account`. Betrachten Sie folgende Codeabschnitte:

Apple operating systems (e.g., macOS) feature a parallelization framework called Grand Central Dispatch *(GCD). Using GCD, application developers can define concurrent tasks and execute them in parallel, without having to worry about managing threads. The central idea of GCD is the concept of* dispatch queues*. A developer adds tasks to a* dispatch queue*, which then get processed by GCD in FIFO order (first-in, first-out). We first consider the simplest case, the so-called* serial dispatch queues*. This type of queue executes only one task at a time.*

The function
`void dispatch(queue *q, void (*func)(void *arg), void *arg)`
adds a task `func` *with the argument* `arg` *to the queue* `q`*. This task is then processed asynchronously by GCD.* `dispatch()` *is guaranteed to be free of* race conditions*.*

Let `account_q` *be a valid, defined* serial dispatch queue*. Let* `my_account` *be a valid instance of* `account`*. Consider the following code snippets:*

```
1  typedef struct account {
2      int id;
3      int balance;
4  } account;
5  typedef struct task {
6      account *acc;
7      int amount;
8  } task;
9
10 void upd_balance(void *arg) {
11     task *t = (task *)arg;
12     t->acc->balance += t->amount;
13 }
```

Thread 1

```
20 task *t = malloc(sizeof(task));
21 t->account = my_account;
22 t->amount = 100;
23 dispatch(account_q, upd_balance, t);
```

Thread 2

```
30 task *t = malloc(sizeof(task));
31 t->account = my_account;
32 t->amount = -200;
33 dispatch(account_q, upd_balance, t);
```

a) In dem Codeabschnitt ist ein kritischer Abschnitt zu sehen. Geben Sie die Zeilennummer(n) an, in denen sich der kritische Abschnitt befindet. **1 pt**

The code snippet contains a critical section. Give the line number(s) where the critical section is located.

Solution: *Line 12.*

b) Die beiden Threads werden parallel ausgeführt. Beurteilen Sie für jedes der drei Kriterien (ohne Performance) zur Lösung des Problems kritischer Abschnitte, ob es erfüllt ist und begründen Sie Ihre Antworten. **6 pt**

The two threads are executed in parallel. Evaluate for each of the three criteria (ignoring performance) for solving the critical section problem whether it is fulfilled and justify your answers.

Kriterium / *Criterion*: **Mutual Exclusion** Erfüllt / *Fulfilled*: ■ Ja □ Nein
Begründung / *Justification*:
Serial queues execute only one task at a time, so mutual exclusion is fulfilled.

Kriterium / *Criterion*: **Bounded Waiting** Erfüllt / *Fulfilled*: ■ Ja □ Nein
Begründung / *Justification*:
Bounded waiting is fulfilled, as the tasks are executed in FIFO order.

Kriterium / *Criterion*: **Progress** Erfüllt / *Fulfilled*: ■ Ja □ Nein
Begründung / *Justification*:
As control lies entirely with GCD, which executes tasks asynchronously in FIFO order, no completed task can stop another task from being executed.

(0.5 P) per criterion name, **(0.5 P)** for fulfillment, **(1 P)** for reasoning (each). Fulfillment and justification give **(0 P)** if the criterion was not named properly.

c) Angenommen, es gibt insgesamt 1000 Accounts (Instanzen von `account`). Wie viele `upd_balance`-Aufgaben könnten theoretisch insgesamt parallel ausgeführt werden? Wie müssten dazu die *dispatch queues* erstellt werden? **2 pt**

Assume there are a total of 1000 accounts (instances of `account`). How many `upd_balance` tasks could theoretically be executed in parallel? How would the dispatch queues *have to be created for this?*

Solution:

Theoretically, 1000 tasks could be executed in parallel **(1 P)**. *To achieve this, one* serial dispatch queue *is required per account* **(1 P)**.

d) Thread 1 und 2 allokieren jeweils Instanzen von `task`. Geben Sie **präzise** (mit Position und Code des/der Funktionsaufruf(s/e)) an, wie der allokierte Speicher freigegeben werden sollte. **1.5 pt**

Threads 1 and 2 each allocate instances of `task`. Give a ***precise*** *(with position and code of the function call(s)) description of how the allocated memory should be deallocated.*

Solution:

After line 12: `free(t);` **(0.5 P)** *if* `free` *is called after lines 23 and 33, causing a potential use-after-free bug. Dispatching a call to* `free` *was also accepted if done correctly and at the right position.* **(0 P)** *for anything else.*

Im Folgenden betrachten wir *concurrent dispatch queues*. Diese führen mehrere Aufgaben gleichzeitig aus, allerdings werden die Aufgaben weiterhin in FIFO-Reihenfolge ausgewählt. **Nehmen Sie für die folgenden Teilaufgaben an, dass System- und Bibliotheksaufrufe nicht fehlschagen und übergebene Funktionsparameter valide sind. Inkludieren Sie notwendige C-Header im gekennzeichneten Bereich.**

Next, we consider concurrent dispatch queues. *These execute multiple tasks simultaneously, but tasks are still selected in FIFO order.* ***For all following subquestions, assume that system and library calls do not fail and all passed function parameters are valid. Include necessary C headers in the marked area.***

e) GCD entscheidet dynamisch, wie viele Threads für die Ausführung einer *concurrent dispatch queue* verwendet werden. Implementieren Sie die Funktion `update_threads()`, die periodisch aufgerufen wird und entscheidet, ob ein neuer Thread gestartet wird. **3.5 pt**

- Starten Sie einen Thread, wenn die Anzahl wartender Aufgaben in der *dispatch queue* mind. doppelt so hoch ist wie beim letzten Threadstart.
- Aktualisieren Sie `last_task_count` und `thread_count` entsprechend.
- Neu gestartete Threads führen `dispatch_thread()` aus und erhalten die *dispatch queue* als Argument.
- Nutzen Sie `ringbuf_count()`, um die Anzahl wartender Aufgaben in der *dispatch queue* zu ermitteln.

GCD dynamically decides how many threads are used for executing a concurrent dispatch queue. *Implement the function* `update_threads()`, *which is called periodically and decides whether a new thread should be started.*

- *Start a new thread if the number of pending tasks in the* dispatch queue *is at least twice as high as when the last thread was started.*
- *Update* `last_task_count` *and* `thread_count` *accordingly.*
- *Newly started threads execute* `dispatch_thread()` *and receive the* dispatch queue *as an argument.*
- *Use* `ringbuf_count()` *to get the number of pending tasks in the* dispatch queue.

```c
/* include statements for the required C headers */
#include "ringbuf.h"
#include <pthread.h>
#include <sched.h>

typedef struct queue {
    ringbuf tasks;
    int last_task_count; // initialized to 0
    int thread_count;
} queue;

int ringbuf_count(ringbuf *rb);
void *dispatch_thread(void *q);
void update_threads(queue *q) {
    int task_count = ringbuf_count(&q->tasks);

    if (task_count >= 2 * q->last_task_count) {
        pthread_t t;

        pthread_create(&t, NULL, dispatch_thread, q);
        q->thread_count++;
        q->last_task_count = task_count;
    }
}
```

- **(1 P)** for correctly using `ringbuf_count()` to determine the number of pending tasks.
- **(0.5 P)** for correctly checking whether the number of pending tasks has doubled.
- **(1 P)** for correctly starting a new thread.
- **(0.5 P)** for correctly updating `thread_count`.
- **(0.5 P)** for correctly updating `last_task_count`.
- **(-0.5 P)** for each missing or incorrect header include.

f) In dieser vereinfachten Version unterstützt GCD drei verschiedene Prioritätsstufen für *dispatch queues*. Diese sollen durch entsprechende Anpassung der Threadprioritäten reflektiert werden. Implementieren Sie die Funktion `update_priority()`. **6 pt**

- Ermitteln Sie den gültigen numerischen Bereich für die Prioritäten.
- `LOW` soll auf die niedrigste numerische Priorität abgebildet werden, `HIGH` auf die höchste und `MEDIUM` auf die Mitte zwischen den numerischen Prioritätsgrenzen.
- Aktualisieren Sie die Priorität des übergebenen Threads.
- Behalten Sie die derzeit für den Thread gesetzte Scheduling-Policy bei.

Hinweis: Überlegen Sie sich, welche der Funktionen aus den angehängten Man-Pages hier nützlich sein könnten.

In this simplified version, GCD supports three different priority levels for dispatch queues*. These should be reflected by adjusting the thread priorities accordingly. Implement the function* `update_priority()`*.*

- *Determine the valid numerical range for priorities.*
- `LOW` *should be mapped to the lowest numerical priority,* `HIGH` *to the highest, and* `MEDIUM` *to the middle between the numerical priority limits.*
- *Update the priority of the passed thread.*
- *Do not change the scheduling policy currently set for the thread.*

***Note**: Consider which of the functions from the attached man pages could be useful for this task.*

```c
typedef enum prio {
    LOW,
    MEDIUM,
    HIGH
} prio;

void update_priority(pthread_t thread, prio p) {
    struct sched_param sp;
    int policy;

    pthread_getschedparam(thread, &policy, &sp);

    int min = sched_get_priority_min(policy);
    int max = sched_get_priority_max(policy);

    switch (p) {
        case LOW:
            sp.sched_priority = min;
            break;
        case MEDIUM:
            sp.sched_priority = (min + max) / 2;
            break;
        case HIGH:
            sp.sched_priority = max;
            break;
    }

    pthread_setschedparam(thread, policy, &sp);
}
```

- **(1 P)** for correctly reading the current scheduling parameters.
- **(1 P)** for correctly determining the valid numerical range for priorities.
- **(1 P)** each for correctly mapping `LOW`, `MEDIUM`, and `HIGH` to their numerical counterparts.
- **(1 P)** for correctly updating the priority.
- **(-0.5 P)** for each missing or incorrect header include.

Total: 20.0pt

Aufgabe 3: Persistenz

Assignment 3: Persistence

Ringpuffer werden häufig zur Kommunikation mit Geräten verwendet. Der folgende Code definiert einen solchen Ringpuffer.

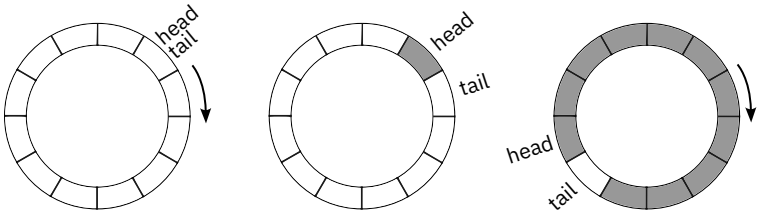
- `head` und `tail` sind Indices in `entries`, die am Ende des Arrays wieder zum Anfang springen.
- Elemente werden am `tail` eingefügt und vom `head` entfernt.
- Der Ringpuffer ist leer, falls `head` und `tail` gleich sind.
- Der Ringpuffer ist voll, falls `tail` genau ein Eintrag hinter `head` ist.

Ring buffers are often used for communication with devices. The following code defines such a ring buffer.

- *`head` and `tail` are indexes in `entries` that wrap around to the beginning at the end of the array.*
- *Entries are inserted at the `tail` and removed from the `head`.*
- *The ring buffer is empty if `head` equals `tail`.*
- *The ring buffer is full if `tail` is exactly one entry behind `head`.*

```
typedef char entry[512];
#define RB_SIZE 64

typedef struct ringbuf {
   entry entries[RB_SIZE];
   int head, tail;
} ringbuf;
```



empty ring buffer | ring buffer with one entry | full ring buffer

a) Geben Sie ein Beispiel für ein Gerät, das üblicherweise über solche Ringpuffer kommuniziert. **0.5 pt**

Give an example of a device that usually communicates with such ring buffers.

Solution:

GPU, network adapters, audio interfaces, HDDs, (NVMe) SSDs

Note*: The design of the ringbuffer here is inspired by NVMe command queues.*

b) Nehmen Sie an, das Gerät nutzt DMA, um auf den Ringpuffer zuzugreifen. Wie kann das Betriebssystem signalisieren, dass neue Einträge vorhanden sind? **1 pt**

Assume that the device uses DMA to access the ring buffer. How can the operating system signal that there are new entries?

Solution:

The operating system can signal that there are new entries in the ring buffer by writing to a device register **(0.5 P)** *via memory-mapped I/O or an I/O port* **(0.5 P)**.

c) Nennen Sie zwei Möglichkeiten, wie das Betriebssystem erfahren kann, dass ein Gerät neue Einträge in den Ringpuffer geschrieben hat. Erklären Sie, welche Möglichkeit besser ist für ein Gerät mit konstant hohem Durchsatz. **2 pt**

Name two ways for the operating system to learn about new entries from a device in the ring buffer. Explain which way is better for a device with constant high throughput.

Solution:

Polling / programmed I/O **(0.5 P)** *and interrupts* **(0.5 P)**. *Polling is better for a device with high throughput since it avoids overhead from handling high frequency interrupts.* **(1 P)**

d) Implementieren Sie die Funktionen `ringbuf_empty()` und `ringbuf_full()`, die zurückgeben, ob der Ringpuffer leer bzw. voll ist. **1.5 pt**

Implement the functions `ringbuf_empty()` and `ringbuf_full()` that return whether the ring buffer is empty or full.

Solution:

```
bool ringbuf_empty(ringbuf *rb) {
        return rb->head == rb->tail;
}

bool ringbuf_full(ringbuf *rb) {
        return rb->head == (rb->tail + 1) % RB_SIZE;
}
```

empty **(0.5 P)**, *full condition* **(0.5 P)**, *full wraparound* **(0.5 P)**

Note*: Be careful with `%` and negative numbers.*
`rb->tail == (rb->head - 1) % RB_SIZE` does not work since `-1 % 64 == -1`

e) Implementieren Sie die Funktion `ringbuf_push()`, die einen Eintrag in den Ringpuffer einfügt. Dazu kopiert sie zunächst den Eintrag und passt dann `tail` an. Die Funktion gibt -1 zurück, falls der Eintrag nicht eingefügt werden konnte, sonst 0. **2.5 pt**

Implement the function `ringbuf_push()` that inserts an entry into the ring buffer. It copies the entry first and then adjusts `tail`. The function returns -1 if the entry could not be inserted, otherwise 0.

Solution:

```
int ringbuf_push(ringbuf *rb, entry *e) {
        if (ringbuf_full(rb)) return -1;
        memcpy(&rb->entries[rb->tail], e, sizeof(entry));
        rb->tail = (rb->tail + 1) % RB_SIZE;
        return 0;
}
```

full check and return **(0.5 P)**, *memcpy* **(1 P)**, *update tail* **(0.5 P)**, *return* **(0.5 P)**

Note*: C does not allow assigning to arrays, so `memcpy` must be used to copy the `entry`.*

f) Implementieren Sie die Funktion `ringbuf_pop()`, die einen Eintrag aus dem Ringpuffer entfernt. Allokieren Sie Speicher für den Eintrag, den Sie schließlich zurückgeben. Aktualisieren Sie nach dem Kopieren des Eintrags `head`. Falls ein Fehler auftritt, geben Sie `NULL` zurück. **3.5 pt**

Implement the function `ringbuf_pop()` that removes an entry from the ring buffer. Allocate memory for the entry that will be returned. After copying the entry, update `head`. Return `NULL` if an error occurs.

Solution:

```
entry *ringbuf_pop(ringbuf *rb) {
        if (ringbuf_empty(rb)) return NULL;
        entry *e = malloc(sizeof(entry));
        if (e == NULL) return NULL;
        memcpy(e, &rb->entries[rb->head], sizeof(entry));
        rb->head = (rb->head + 1) % RB_SIZE;
        return e;
}
```

empty check and return **(0.5 P)**, *malloc* **(0.5 P)**, *malloc error check* **(0.5 P)**, *memcpy* **(1 P)**, *update head* **(0.5 P)**, *return* **(0.5 P)**

g) Nennen Sie drei aus der Vorlesung bekannte Verfahren zur Dateiallokation. Ordnen Sie die Verfahren in aufsteigender Reihenfolge bzgl. der Performance von wahlfreien Zugriffen. **1.5 pt**

Name three strategies for file allocation introduced in the lecture. Order them by increasing performance of random accesses.

Solution:

(a) chained allocation

(b) FAT

(c) indexed allocation

(d) contiguous allocation

(0.5 P) *per strategy,* **(-0.5 P)** *if the order is incorrect*

h) Gegeben sei das folgende Programm. Gehen Sie davon aus, dass alle Systemaufrufe erfolgreich sind.

Have a look at the following program. Assume that all system calls are successful.

```
int main(int argc, char **argv) {
   int i, n, fd;
   char buf[100];
   for (i = 1; i < argc; i++) {
      fd = open(argv[i], O_RDONLY);
      while ((n = read(fd, buf, sizeof(buf))))
         write(STDOUT_FILENO, buf, n);
      close(fd);
   }
   return 0;
}
```

Welche Funktion hat das Programm? **1 pt**

What function does the program perform?

Solution:

The program is a simple implementation of `cat`*. It prints the contents of all files given as command line arguments to the standard output.*

Note*: Make sure to explain the meaning of* `argv` *(command line arguments) and* `STDOUT_FILENO` *(standard output, console, etc.).*

Das Programm wird mit einer 150 Byte großen Datei als Argument aufgerufen. Wie viele `read`-Systemaufrufe führt das Programm aus? Füllen Sie in der Tabelle unten den Rückgabewert sowie den Offset in der Datei nach jedem `read`-Aufruf aus. Gehen Sie davon aus, dass jeder Aufruf so viel wie möglich liest. **2 pt**

The program is called with a file of size 150 bytes as argument. How many `read` *system calls does the program perform? In the table below, fill in the return value and the offset in the file after each call to* `read`*. Assume that every call reads as much as possible.*

Anzahl `read`-Aufrufe:
Number of `read` *calls:*

Solution:
3 **(0.5 P)**

syscall	return value	offset
`open`	3	0
`read`	100	100
	50	150
	0	150

(0.5 P) per row

Wo wird der Offset durch das Betriebssystem gespeichert? **0.5 pt**

Where does the operating system store the offset?

Solution:

The operating system stores the offset in the global open file table.

i) Die Berechtigung `x` hat unterschiedliche Bedeutungen abhängig von dem in der Inode kodierten Dateityp. Nennen Sie zwei Dateitypen und erklären Sie jeweils die zugehörige Bedeutung von `x`. **2 pt**

The access right `x` has different meanings depending on the file type coded in the inode. Name two file types and explain the respective meaning of `x`.

Solution:

regular file *whether the file is executable*

directory *"search" permission = permission to access files in the directory*

symbolic link *no meaning*

Note*: Opening a directory and reading its directory entries is controlled by the read permission (r) and is possible without search permission (x).*

j) Gegeben sei ein Dateisystem, das Dateien mittels Inodes umsetzt. Nehmen Sie eine Blockgröße von 4 KiB und 8 Byte große Zeiger auf Blöcke an. Eine Inode beinhaltet 10 Zeiger auf direkte Blöcke, und je einen Zeiger zu einem einfach, doppelt, sowie dreifach indirekten Block. Berechnen Sie die Menge an Dateispeicherplatz, der mit den jeweiligen Komponenten adressiert werden kann. Vereinfachen Sie Ihre Ergebnisse zu den gegebenen Einheiten. **2 pt**

Consider a file system that uses inodes to represent files. Assume that disk blocks are 4 KiB in size and a pointer to a disk block is 8 bytes long. An inode contains 10 pointers to direct blocks, and one pointer to a single, double, and triple indirect block, respectively. Calculate the amount of file data that is addressable with each of these components. Simplify your results to the given units.

Component	Amount of file data
10 direct	$10 \times 4\,\text{KiB} = 40\,\text{KiB}$
1 single indirect	$1 \times \frac{4\,\text{KiB}}{8\,\text{B}} \times 4\,\text{KiB} = 2^{12-3+12}\text{B} = 2^{21}\text{B} = 2\,\text{MiB}$
1 double indirect	$1 \times (\frac{4\,\text{KiB}}{8\,\text{B}})^2 \times 4\,\text{KiB} = 2^{(12-3)\times 2+12}\text{B} = 2^{30}\text{B} = 1\,\text{GiB}$
1 triple indirect	$1 \times (\frac{4\,\text{KiB}}{8\,\text{B}})^3 \times 4\,\text{KiB} = 2^{(12-3)\times 3+12}\text{B} = 2^{39}\text{B} = 512\,\text{GiB}$

Total: 20.0pt