Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Frank Bellosa
Lukas Werling, M.Sc.
Dr. Marc Rittinghaus

Nachname/ *Last name*	Vorname/ *First name*	Matrikelnr./ *Matriculation no*

Nachklausur
08.09.2022

- 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 28 Blättern: Einem Deckblatt, 23 Aufgabenblättern mit insgesamt 5 Theorieaufgaben und 3 Programmieraufgaben sowie 4 Seiten Man-Pages.

  The examination consists of 28 pages: One cover sheet, 23 sheets containing 5 theory assignments as well as 3 programming assignments, and 4 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	T1	T2	T3	T4	T5	P1	P2	P3	Total
Max. Punkte	9	9	9	9	9	15	15	15	90
Erreichte Punkte									

Matrikelnummer/*Matriculation number*

Aufgabe T1: Grundlagen

Assignment T1: Basics

a) Welchen Vorteil bietet die Trennung zwischen Mechanismus und Policy? **1 pt**

Which advantage does the separation between mechanism and policy provide?

b) Erläutern Sie die Begriffe *Interrupt Vector* und *Interrupt Service Routine.* **2 pt**

Describe the terms interrupt vector *and* interrupt service routine.

c) Erläutern Sie einen Vorteil und einen Nachteil von statischem gegenüber dynamischem Linking von Bibliotheken. **1 pt**

Explain an advantage and a disadvantage of static versus dynamic linking of libraries.

d) Geben Sie ein Segment einer ELF-Datei an, das zwischen Prozessen geteilt werden kann, und ein anderes Segment, das nicht geteilt werden kann. Begründen Sie, warum das jeweilige Segment geteilt bzw. nicht geteilt werden kann. **1.5 pt**

Name a segment of an ELF file that can be shared between processes, and another segment that cannot be shared. Give reasons why each segment can be shared or cannot be shared.

e) Nennen Sie zwei Möglichkeiten, die Parameter eines Systemaufrufs an den Kernel zu übergeben. Welche dieser Möglichkeiten würden Sie wählen? Begründen Sie Ihre Antwort. **2 pt**

Name two ways of passing the parameters of a system call to the kernel. Which of these ways would you prefer? Explain your answer.

f) Wie bemerkt das Betriebssystem, dass ein nicht privilegierter Prozess versucht hat, eine privilegierte Instruktion auszuführen? **0.5 pt**

How does the operating system know that a non-privileged process has tried to execute a privileged instruction?

g) Wofür stehen die folgenden Abkürzungen im Kontext der Vorlesung? **1 pt**

What do the following abbreviations stand for in the context of the lecture?

TCB:

IPC:

Total: 9.0pt

Aufgabe T2: Prozesse und Threads

Assignment T2: Processes and Threads

a) Nennen Sie vier in der Vorlesung besprochene Bestandteile eines Prozesskontrollblocks. **2 pt**

Name four elements of a Process Control Block discussed in the lecture.

b) Gegeben seien ein Lottery-Scheduler auf einem Uniprozessorsystem mit der Prozessliste $\{P_1, P_2, P_3\}$. P_1 habe 2 Tickets, P_2 habe 5 Tickets, P_3 habe 1 Ticket. Alle Prozesse sind permanent lauffähig. **3 pt**

Vervollständigen Sie den untenstehenden Plan (10 Kreuze). Die erste Zeile (TIME) gibt die Zeit an. Eingeplant wird jeweils beim Wechsel der Zeiteinheit. Um die Lotterie deterministisch zu gestalten, enthält die zweite Zeile (LOTTERY) das vorgegebene Lotterie-Ergebnis zum jeweiligen Anfang der Zeiteinheit.

Consider a lottery scheduler on a uniprocessor system with process list $\{P_1, P_2, P_3\}$. P_1 has 2 lottery tickets, P_2 has 5 lottery tickets, and P_3 has 1 ticket.

Complete the scheduling plan given below (10 crosses). The first row (TIME) states the time. Scheduling occurs at time unit changes. In order to ensure deterministic results, the second row (LOTTERY) contains a pre-defined lottery result for each time unit start.

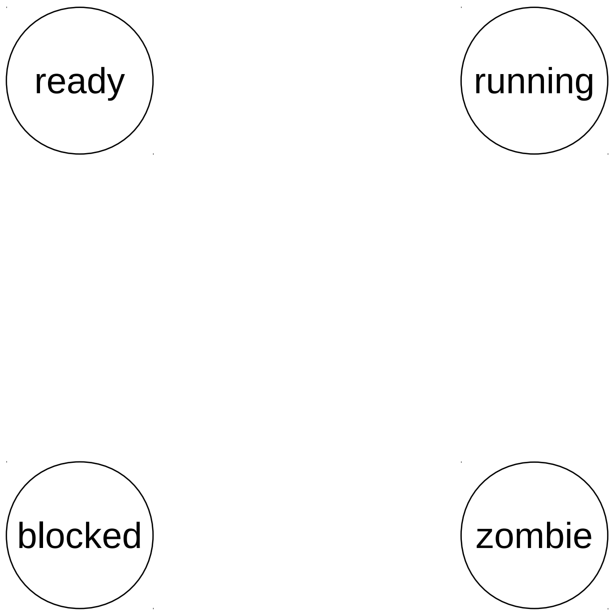
TIME	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
LOTTERY	*4*	*5*	*0*	*0*	*1*	*3*	*2*	*4*	*1*	*6*
P_1										
P_2										
P_3										

c) Erklären Sie, warum ein *FCFS-Scheduler* eine schlechte Wahl für Desktopsysteme ist. **1 pt**

Explain why a FCFS-scheduler is a bad choice for desktop systems.

d) Nehmen Sie an, dass ein Betriebssystem die vier Prozesszustände „bereit" („ready"), „rechnend" („running"), „blockiert" („blocked") und „zombie" unterstützt. Stellen Sie grafisch dar, zwischen welchen Zuständen Übergänge möglich sind und beschriften Sie jede Kante mit einem Ereignis, das den jeweiligen Übergang auslöst. **2.5 pt**

Consider an operating system, which supports the four process states "ready", "running", "blocked", and "zombie". Depict the possible state transitions and label each edge with an event causing the transition.



Durch welche Aktion verlässt ein Prozess den „Zombie"-Zustand? **0.5 pt**

Which action causes a process to leave the "zombie" state?

Total: 9.0pt

Aufgabe T3: Speicher
Assignment T3: Memory

a) Erläutern Sie, für welche Szenarien die folgenden Seitentabellentypen geeignet sind. Begründen Sie Ihre Antwort. **2 pt**

Explain for which scenarios the following page table types are suitable. Justify your answer.

Lineare Seitentabelle/*Linear page table*:

Invertierte Seitentabelle/*Inverted page table*:

b) Ein Prozess ruft `fork()` auf. Beschreiben Sie für den Eltern- und den Kindprozess, wie der jeweilige Adressraum konfiguriert werden muss, um Copy-On-Write (COW) zu ermöglichen. Begründen Sie Ihre Antwort. **1.5 pt**

A process calls `fork()`. Describe for the parent and the child process, how each address space has to be configured to allow copy-on-write (COW). Explain your answer.

c) Erläutern Sie knapp den Begriff *ASID*. **0.5 pt**

Briefly explain the term ASID.

d) Gegeben sei ein System, das virtuelle Adressen mit einer vierstufigen Seitentabelle übersetzt. Die Tabelle nutzt 32 von 1024 Einträgen in der obersten Stufe und alle 1024 Einträge in den folgenden Stufen. Die Seitengröße beträgt 4 KiB. Jeder Tabelleneintrag ist 4 Bytes groß. **2 pt**

Bei gleichem Speicherverbrauch für die Seitentabelle, wieviel größer wäre der maximal adressierbare virtuelle Speicher in Bytes bei einer linearen Seitentabelle, wenn alle Einträge genutzt werden?

Consider a system which translates virtual addresses with a four-level page table. The table uses 32 of 1024 entries in the top-most level and all 1024 entries in each following level. The page size is 4 KiB. Each table entry is 4 bytes in size.

With the same memory consumption for the page table, how much larger would the maximum addressable virtual memory in bytes be for a linear page table, if all entries are used?

e) Worin unterscheidet sich ein software- von einem hardwaregesteuerten TLB? **1 pt**

What is the difference between a software- and a hardware-managed TLB?

f) Erklären Sie die Begriffe Demand Paging und Pre-Paging. Bei welchem Verfahren ist mit mehr Speichernutzung zu rechnen? Warum kann dieses Verfahren dennoch von Vorteil sein? **2 pt**

Explain the terms demand paging and pre-paging. Which method is expected to use more memory? Why can this method nevertheless be advantaguous?

Total: 9.0pt

Matrikelnummer/*Matriculation number*

Aufgabe T4: Koordination und Kommunikation von Prozessen

Assignment T4: Process Coordination and Communication

a) Erläutern Sie, was man unter einer *Race Condition* versteht. **1 pt**

Explain what is meant by a race condition.

b) Unter welchen Voraussetzungen funktioniert das Deaktivieren von Interrupts als Synchronisationsmechanismus? Begründen Sie Ihre Antwort. **1 pt**

In which circumstances may disabling interrupts work as a synchronization mechanism? Explain why.

c) Welche der Bedingungen für einen Deadlock wird nicht erfüllt, wenn sämtliche Ressourcen sortiert und ausschließlich in dieser Reihenfolge alloziert werden? Begründen Sie Ihre Antwort. **1 pt**

Which of the requirements for a deadlock is negated by ordering all resources and always acquiring them in this order? Justify your answer.

d) Warum muss beim Reader-Writer-Problem der Eintritt von schreibenden Threads in den kritischen Abschnitt gegenüber lesenden Threads priorisiert werden, wenn Bounded Waiting erfüllt werden soll? **1 pt**

Why does the reader-writer problem require the entry of writer threads into the critical section to be prioritized over the entry of reader threads if bounded waiting shall be fulfilled?

e) Welche zwei weiteren Bedingungen neben Bounded Waiting sind für die gültige Lösung des Problems kritischer Abschnitte notwendig? Erläutern Sie diese kurz. **2 pt**

Besides bounded waiting, what two other requirements must be met for a valid solution to the critical section problem? Explain them briefly.

f) Erklaren Sie den Begriff *Mailbox* im Kontext von Message Passing. **1 pt**

Explain the term mailbox *in the context of message passing.*

g) Erklären Sie, wie ein *Two-Phase-Lock* funktioniert. Ist die Verwendung auch im Kernel sinnvoll? **2 pt**

Explain how a two-phase lock *works. Does it make sense to use it also in the kernel?*

Total: 9.0pt

Aufgabe T5: I/O, Hintergrundspeicher und Dateisysteme

Assignment T5: I/O, Secondary Storage, and File Systems

a) Wie ist es möglich, einem einzelnen Benutzer außerhalb der eigenen Benutzergruppe Zugriff auf ein Verzeichnis zu geben, ohne die Gruppe des Benutzers zu ändern? **1 pt**

How is it possible to give a single user outside your user group access to a directory without changing the user's group?

b) Wenn in einem Unix Dateisystem ein symbolischer Link gelöscht wird, dann muss die zu der verlinkten Datei gehörige inode nicht modifiziert werden. Bei harten Links ist dies allerdings der Fall. Warum? **2 pt**

When a symbolic link is deleted in a Unix file system, the inode of the linked file does not need to be modified. This is not true for hard links. Why is this the case?

c) Ein Verbund aus vier Festplatten kann für unterschiedliche RAID Level konfiguriert werden. Wie viele Festplatten können in der jeweiligen Konfiguration ausfallen, bevor Daten endgültig verloren gehen? Begründen Sie jeweils Ihre Antwort. **4 pt**

An array of four hard disks can be configured for different RAID levels. How many disks can fail in each configuration before data is permanently lost? Explain your answer.

RAID 0:

RAID 10:

RAID 4:

RAID 5:

d) Mit dem Programm `fsck` können unter Unix Dateisystemfehler gefunden und behoben werden. Geben Sie für die folgenden Fehlermeldungen an, was `fsck` im inkonsistenten Dateisystem gesehen haben könnte um auf den Fehler zu schließen. **2 pt**

Unix file systems can be checked and fixed with the `fsck` program. What could `fsck` have seen to generate the following error messages in an inconsistent file system.

Der Inode Linkzähler ist 1, er sollte 2 sein.
The inode link count is 1, it should be 2.

Der Frei-Bitmap Eintrag für Block 1234 ist 0 (frei), er sollte 1 sein (zugewiesen).
The free-bitmap entry for block 1234 is 0 (free), it should be 1 (allocated).

Total: 9.0pt

Aufgabe P1: C Grundlagen
Assignment P1: C Basics

a) In dem untenstehenden Code haben sich 6 Fehler eingeschlichen. Markieren Sie die fehlerhaften Zeilen mit einem X und korrigieren Sie den Code. Gehen Sie von einem 64-Bit-System aus. **6 pt**

There are 6 errors in the code below. Mark the incorrect lines with an X and correct the code. Assume a 64-bit system.

```
typedef struct {
    int valid;
    char *key;
    int value;
} element;
```

```
typedef struct {
    element *elements;
    size_t capacity;
} hashtable;
```

```
☐ uint64_t hashString(char *c) {
☐     uint64_t hash = 5381;
☐     while (c)
☐         hash = hash * 33 + *c++;
☐     return hash;
☐ }

☐ int insert(hashtable *table, char *key, int value) {
☐     size_t start = hashString(key) % table->capacity;
☐     size_t pos = start;
☐     while (table->elements[pos].valid) {
☐         if (strcmp(*key, table->elements[pos].key) == 0)
☐             continue;
☐         pos = (pos + 1) % table->capacity;
☐         if (pos == start)
☐             return 0;
☐     }
☐     table->elements[pos] = (element) {key, value, 1};
☐     return 1;
☐ }

☐ int find(hashtable *table, char *key, int *value) {
☐     size_t start = hashString(key) % table->capacity;
☐     size_t pos = start;
☐     while (table->elements[pos].valid) {
☐         if (strcmp(key, table->elements[pos].key)) {
☐             value = table->elements[pos].value;
☐             return 1;
☐         }
☐         pos = (pos + 1) % table->capacity;
☐         if (pos == start)
☐             break;
☐     }
☐     return 0;
☐ }
```

Wie löst die Hashtabelle Kollisionen auf? **0.5 pt**

How does the hash table resolve collisions?

Schreiben Sie eine Funktion `allocHashtable()`, die eine leere, auf dem Heap allozierte Hashtabelle mit der gegebenen Kapazität zurückgibt. Bei einem Fehler soll die Funktion `NULL` zurückgeben. **3 pt**

Write a function `allocHashtable()` which shall return an empty hash table allocated on the heap with the given capacity. In case of an error, the function shall return `NULL`.

```
hashtable *allocHashtable(size_t cap) {

}
```

Wie kann durch Anpassung von `element` Speicher eingespart werden? Wie viel Speicher kann pro Eintrag eingespart werden? Erklären Sie. **1.5 pt**

How can we save memory by changing `element`? How much memory can we save per entry? Explain.

b) Was ist der Wert der Ausdrücke nach Ausführung des Programmschnipsels? **2 pt**

What is the value of the expressions after execution of the given code snippet?

```
uint32_t a[] =
  { 0x00112233, 0x44556677,
    0x8899aabb, 0xccddeeff,
    0xff00ff00, 0x0123abcd };

uint16_t b = a[0] + a[2];
uint32_t c = b << 8;
// <-- Evaluate expression here
```

Expression	Value
`a[4] | a[1]`	
`~a[5] >> 16`	
`b ^ ~b`	
`c`	

c) `A` sei ein Array. Was bedeutet der folgende Ausdruck? **1 pt**

Let `A` be an array. What is the meaning of the following expression?

`&A[42]`

Schreiben Sie einen äquvalenten Ausdruck, der ohne den `&`-Operator auskommt. **0.5 pt**

Write an equivalent expression without the `&` operator.

d) Wie werden in der cdecl-Aufrufkonvention Funktionsargumente übergeben? **0.5 pt**

In the cdecl calling convention, how are arguments passed to functions?

Total: 15.0pt

Aufgabe P2: Dateikompression

Assignment P2: File Compression

Schreiben Sie ein Programm zur Kompression von Dateien. Die Pfade der Eingabedateien sowie die entsprechenden Ausgabepfade für die komprimierten Daten sollen als Paare über Kommandozeilenargumente übergeben werden. Ein Aufruf des Programms könnte wie folgt aussehen:

```
compress a.txt a.txt.out b.txt b.txt.out
```

Das Programm lädt dabei die Inhalte der Dateien `a.txt` und `b.txt` in den Hauptspeicher, komprimiert diese und schreibt sie in die jeweiligen Ausgabedateien (z. B. `a.txt` → `a.txt.out`).

- Geben Sie vom Betriebssystem angeforderte Ressourcen (z. B. Speicher) explizit zurück.
- Binden Sie die in den Teilaufgaben notwendigen C-Header in dem gekennzeichneten Bereich ein.
- Sofern nicht anderweitig bestimmt, gehen Sie davon aus, dass (1) bei Systemaufrufen und Speicherallokationen keine Fehler auftreten, (2) Systemaufrufe nicht durch Signale unterbrochen werden und (3) Funktionsparameter valide sind.

Write a program that compresses files. The paths to input files as well as the corresponding output paths for the compressed data are passed to the program as pairs of command line arguments. As an example, consider the following command line:

```
compress a.txt a.txt.out b.txt b.txt.out
```

The program loads the contents of the files `a.txt` and `b.txt` into memory, compresses them, and writes them into the corresponding output files (e.g., `a.txt` → `a.txt.out`).

- *Explicitly return allocated operating system resources (e.g., memory).*
- *Include necessary C headers in the marked area.*
- *Unless stated otherwise, assume that (1) system calls and memory allocations do not fail, (2) system calls are not interrupted by signals, and (3) function arguments are valid.*

```
/* include statements for the required C headers */
```

```
/* global variables */
```

a) Vervollständigen Sie die Funktion `get_file_size()`, die die Größe der Datei an dem angegebenen Pfad zurückgibt. **2 pt**

Complete the function `get_file_size()`, which returns the size of the file at the specified path.

```
size_t get_file_size(const char *path) {






}
```

b) Vervollständigen Sie die Funktion `read_file()`, die die Datei an dem gegebenen Pfad `path` öffnet und die ersten `buf_len` Bytes der Datei mithilfe des `read()` Systemaufrufs in den Puffer `buf` liest. Die Funktion gibt in jedem Fall die Anzahl tatsächlich gelesener Bytes zurück. **6.5 pt**

- Schreiben Sie nicht über das Ende des Puffers hinaus (`buf_len`).
- Implementieren Sie Fehlerbehandlung für sämtliche Systemaufrufe. Die Funktion soll auch dann `buf_len` Bytes einlesen, wenn Signale während der Ausführung der Funktion auftreten. Sonstige Fehler (inkl. Erreichen des Dateiendes) sollen zum Abbruch des Lesevorgangs führen.

Complete the function `read_file()`, which opens the file at the specified path `path` and reads the first `buf_len` bytes of the file into the buffer `buf` using the `read()` system call. In any case, the function returns the actual number of bytes read.

- *Do not write beyond the end of the buffer (`buf_len`).*
- *Implement error handling for all system calls. The function shall read `buf_len` bytes, even if signals occur during the execution of the function. Other errors (incl. reaching the end of file) shall abort the read operation.*

```
size_t read_file(const char *path, char *buf, size_t buf_len) {
}
```

c) Vervollständigen Sie die Funktion `spawn_process()`, die einen separaten Prozess erzeugt, in dem die gegebene Eingabedatei asynchron komprimiert wird. **3.5 pt**

- Verwenden Sie `compress_file()` zur Kompression.
- Die Funktion soll sicherstellen, dass nie mehr als 4 zusätzliche Prozesse gleichzeitig ausgeführt werden.
- Sie können globale Variablen in dem zu Beginn der Aufgabe P2 reservierten Platz einfügen.

Complete the function `spawn_process()`, which creates a separate process that asynchronously compresses the specified input file.

- *Use `compress_file()` for compression.*
- *The function shall ensure that there are never more than 4 additional processes running at the same time.*
- *You may add global variables to the reserved space at the beginning of Assignment P2.*

```
void write_compressed(const char *path, const char *buf, size_t buf_len);
void compress_file(const char *in, const char *out) {
    size_t size = get_file_size(in);
    char *buf = malloc(size);
    read_file(in, buf, size);
    write_compressed(out, buf, size);
    free(buf);

    exit(0);
}

void spawn_process(const char *in, const char *out) {

















}
```

d) Vervollständigen Sie die `main()`-Funktion des Programms, die für je zwei aufeinanderfolgende Kommandozeilenargumente `spawn_process()` aufruft. **3 pt**

- Das Programm soll sich erst beenden, nachdem alle zusätzlich gestarteten Prozesse terminiert sind.
- Gehen Sie von einer beliebigen Anzahl von Kommandozeilenargumenten aus. Bei ungerader Anzahl kann das letzte Argument ignoriert werden.

Complete the program's `main()` function, which calls `spawn_process()` for each pair of two consecutive command line arguments.

- *The program shall terminate only after all additionally spawned processes have terminated.*
- *Assume an arbitrary number of command line arguments. In case of an odd number of arguments, the last one can be ignored.*

```
void spawn_process(const char *in, const char *out);

int main(int argc, char **argv) {
















}
```

Total: 15.0pt

Aufgabe P3: Speicherverwaltung
Assignment P3: Memory Management

Gegeben sei ein System mit 32-Bit Addressraum, zweistufiger Seitentabelle, 4 KiB Seiten und Software-verwaltetem TLB.

Ein virtueller Adressraum (VAS) wird im Betriebssystem mit der `vas_t`-Struktur beschrieben. Die `vma_t`-Struktur stellt einen gültigen Speicherbereich (VMA) innerhalb eines Adressraums dar. Alle VMAs eines Adressraums sind in einer unsortierten, einfach-verketteten Liste abgelegt, wobei ein `vma_t`-Zeiger von `NULL` das Listenende bzw. eine leere Liste markiert.

Assume a system with 32-bit address space, two-level page table, 4 KiB pages, and software-managed TLB.

The operating system describes a virtual address space (VAS) with the `vas_t` structure. The `vma_t` structure represents a valid virtual memory area (VMA) within an address space. All VMAs of an address space are stored in an unsorted, singly-linked list, where a `vma_t` pointer of `NULL` indicates the list's end, or an empty list.

```c
typedef struct pte {        // Page table entry (PTE)
    uint32_t pfn;           // Page frame number (PFN)
} pte_t;

typedef struct pt {         // Page table (PT)
    pte_t ptes[1024];
} pt_t;

typedef struct pd {         // Page directory (PD)
    pt_t *tables[1024];
} pd_t;

typedef struct vma {        // Virtual memory area (VMA)
    struct vma *next;       // Pointer to next VMA in list, or NULL
    uint32_t start;         // First virtual page number in VMA
    uint32_t end;           // Last virtual page number in VMA
} vma_t;

typedef struct vas {        // Virtual address space (VAS)
    vma_t *vmas;            // Unsorted list of VMAs in address space
    pd_t pdir;              // Page directory
} vas_t;

#define INV_PFN ((uint32_t)(-1)) // Invalid page frame number (PFN)

// Allocates a number of bytes from kernel virtual memory
// (not zero-initialized) on success, NULL otherwise
void* kmalloc(size_t bytes);

// Frees a physical page frame with the given PFN.
// Calling with INV_PFN is undefined behavior!
void freeFrame(uint32_t pfn);

// Flushes the TLB, thereby invalidating all TLB entries
void flushTlb(void);
```

a) Vervollständigen Sie die Funktion `allocVas()`, die einen neuen Adressraum (`vas_t`) aus Kernelspeicher alloziert, initialisiert und zurückgibt. **3 pt**

- Der Adressraum soll keine gültigen Speicherbereiche enthalten.
- Der durch die Seitentabellenhierarchie belegte Platz soll möglichst klein sein.
- Die Funktion gibt im Fehlerfall `NULL` zurück.

Complete the function `allocVas()`, which allocates, initializes, and returns a new address space (`vas_t`) from kernel memory.

- *The address space shall not contain any valid virtual memory areas.*
- *The memory required by the page table hierarchy shall be minimal.*
- *The function returns `NULL` on error.*

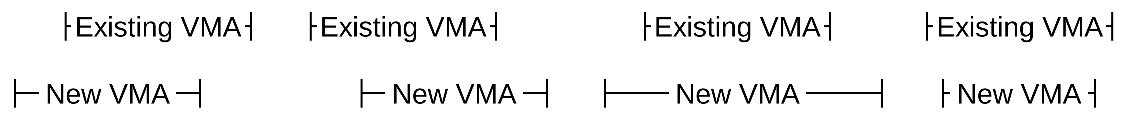
```
void* kmalloc(size_t bytes);

vas_t* allocVas(void) {
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
}
.....................................................................................
```

b) Vervollständigen Sie die Funktion `addVma()`, die den gegebenen virtuellen Speicherbereich `vma` in die Liste der gültigen Speicherbereiche des Adressraums `vas` einfügt. **3.5 pt**

- Die Operation schlägt fehl, wenn der neue Speicherbereich sich mit einer bereits eingetragenen Region überschneidet (siehe Abbildung).
- Die Funktion gibt im Fehlerfall `-1` zurück, ansonsten `0`.

Complete the function `addVma()`, which adds the specified virtual memory area (`vma_t`) to the list of valid VMAs in the address space `vas`.

- *The operation fails if the new area overlaps with an existing area (see figure).*
- *The function returns `-1` on error, `0` otherwise.*



```
int addVma(vas_t *vas, vma_t *vma) {



}
```

c) Vervollständigen Sie die Funktion `getPte()`, die einen Pointer auf den Seitentabelleneintrag (PTE) zu der virtuellen Adresse `vaddr` liefert. **4.5 pt**

- Die Funktion allokiert eine neue Seitentabelle falls nötig und initialisiert deren PTEs, sodass diese auf keine gültige physische Seite zeigen.
- Sie müssen keine Fehlerbehandlung durchführen.

Complete the function `getPte()`, which returns a pointer to the page table entry (PTE) for the virtual address `vaddr`.

- *The function allocates a new page table if necessary and initializes the PTEs to point to no valid physical frame.*
- *You do not need to handle errors.*

```
void* kmalloc(size_t bytes);

pte_t* getPte(vas_t *vas, uint32_t vaddr) {
















}
```

d) Vervollständigen Sie die Funktion `invalidateVma()`, die alle Abbildungen von virtuellen auf physische Seiten des Speicherbereichs `vma` aufhebt und betroffene physische Seiten freigibt. **4.0 pt**

- Sie können die Funktion `getPte()` verwenden.
- Allozierte Seitentabellen müssen nicht freigegeben werden.
- Beachten Sie, dass nicht alle PTEs gültige Abbildungen auf physische Seiten enthalten müssen.

Complete the function `invalidateVma()`, which invalidates all mappings of virtual to physical pages for the memory area `vma`, and frees the physical frames.

- *You can use the function `getPte()`.*
- *Allocated page tables do not need to be released.*
- *Keep in mind that not all PTEs may contain valid mappings to physical frames.*

```
pte_t* getPte(vas_t *vas, uint32_t vaddr);
void freeFrame(uint32_t pfn);
void flushTlb(void);

void invalidateVma(vas_t *vas, vma_t *vma) {














}
```

Total: 15.0pt

NAME

open – open and possibly create a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

DESCRIPTION

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise-*or*'d in *flags*. The *file creation flags* are **O_CLOEXEC**, **O_CREAT**, **O_DIRECTORY**, **O_EXCL**, **O_NOCTTY**, **O_NOFOLLOW**, **O_TMPFILE**, and **O_TRUNC**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The file status flags can be retrieved and (in some cases) modified; see **fcntl**(2) for details.

The abridged list of file creation flags and file status flags is as follows:

O_APPEND

The file is opened in append mode. Before each **write**(2), the file offset is positioned at the end of the file, as if with **lseek**(2). The modification of the file offset and the write operation are performed as a single atomic step.

O_APPEND may lead to corrupted files on NFS filesystems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

O_CLOEXEC (since Linux 2.6.23)

Enable the close-on-exec flag for the new file descriptor. Specifying this flag permits a program to avoid additional **fcntl**(2) **F_SETFD** operations to set the **FD_CLOEXEC** flag.

Note that the use of this flag is essential in some multithreaded programs, because using a separate **fcntl**(2) **F_SETFD** operation to set the **FD_CLOEXEC** flag does not suffice to avoid race conditions where one thread opens a file descriptor and attempts to set its close-on-exec flag using **fcntl**(2) at the same time as another thread does a **fork**(2) plus **execve**(2). Depending on the order of execution, the race may lead to the file descriptor returned by **open**() being unintentionally leaked to the program executed by the child process created by **fork**(2). (This kind of race is in principle possible for any system call that creates a file descriptor whose close-on-exec flag should be set, and various other Linux system calls provide an equivalent of the **O_CLOEXEC** flag to deal with this problem.)

O_CREAT

If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective user ID of the process.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O_CREAT** or **O_TMPFILE** is specified in *flags*; if neither **O_CREAT** nor **O_TMPFILE** is specified, then *mode* is ignored. The effective mode is

modified by the process's *umask* in the usual way: in the absence of a default ACL, the mode of the created file is *(mode & ˜umask)*. Note that this mode applies only to future accesses of the newly created file; the **open**() call that creates a read-only file may well return a read/write file descriptor.

The following symbolic constants are provided for *mode*:

S_IRWXU
00700 user (file owner) has read, write, and execute permission

S_IRUSR
00400 user has read permission

S_IWUSR
00200 user has write permission

S_IXUSR
00100 user has execute permission

S_IRWXG
00070 group has read, write, and execute permission

S_IRGRP
00040 group has read permission

S_IWGRP
00020 group has write permission

S_IXGRP
00010 group has execute permission

S_IRWXO
00007 others have read, write, and execute permission

S_IROTH
00004 others have read permission

S_IWOTH
00002 others have write permission

S_IXOTH
00001 others have execute permission

According to POSIX, the effect when other bits are set in *mode* is unspecified.

O_DIRECTORY

If *pathname* is not a directory, cause the open to fail. This flag was added in kernel version 2.1.126, to avoid denial-of-service problems if **opendir**(3) is called on a FIFO or tape device.

O_TRUNC

If the file already exists and is a regular file and the access mode allows writing (i.e., is **O_RDWR** or **O_WRONLY**) it will be truncated to length 0. If the file is a FIFO or terminal device file, the **O_TRUNC** flag is ignored. Otherwise, the effect of **O_TRUNC** is unspecified.

RETURN VALUE

open() returns the new file descriptor, or −1 if an error occurred (in which case, *errno* is set appropriately).

NAME
close – close a file descriptor

SYNOPSIS
#include <unistd.h>

int close(int *fd*);

DESCRIPTION
close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see **fcntl**(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description (see **open**(2)), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using **unlink**(2), the file is deleted.

RETURN VALUE
close() returns zero on success. On error, −1 is returned, and *errno* is set appropriately.

NAME
exit – cause normal process termination

SYNOPSIS
#include <stdlib.h>

void exit(int *status*);

DESCRIPTION
The **exit**() function causes normal process termination and the value of *status & 0377* is returned to the parent (see **wait**(2)).

All open **stdio**(3) streams are flushed and closed. Files created by **tmpfile**(3) are removed.

The C standard specifies two constants, **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to **exit**() to indicate successful or unsuccessful termination, respectively.

RETURN VALUE
The **exit**() function does not return.

NOTES
The use of **EXIT_SUCCESS** and **EXIT_FAILURE** is slightly more portable (to non-UNIX environments) than the use of 0 and some nonzero value like 1 or −1. In particular, VMS uses a different convention.

After **exit**(), the exit status must be transmitted to the parent process. There are two cases:

- If the parent was waiting on the child, it is notified of the exit status and the child dies immediately.
- Otherwise, the child becomes a "zombie" process: most of the process resources are recycled, but a slot containing minimal information about the child process (termination status, resource usage statistics) is retained in process table. This allows the parent to subsequently use **waitpid**(2) (or similar) to learn the termination status of the child; at that point the zombie process slot is released.

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 and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (**mmap**(2)), and unmappings (**munmap**(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)) or session.
* The child's parent process ID is the same as the parent's process ID.

Note the following further points:

* The child process is created with a single thread—the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of **pthread_atfork**(3) may be helpful for dealing with problems that this can cause.
* The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see **open**(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in **fcntl**(2)).

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.

NOTES
Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

NAME
memset – fill memory with a constant byte

SYNOPSIS
#include <string.h>

void *memset(void **s*, **int** *c*, **size_t** *n*);

DESCRIPTION
The **memset**() function fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*.

RETURN VALUE
The **memset**() function returns a pointer to the memory area *s*.

NAME

malloc, free − allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t *size***);**
void free(void ****ptr*);**

DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

RETURN VALUE

The **malloc**() function return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero.

The **free**() function returns no value.

ERRORS

malloc() can fail with the following error:

ENOMEM
Out of memory. Possibly, the application hit the **RLIMIT_AS** or **RLIMIT_DATA** limit described in **getrlimit**(2).

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when **malloc**() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of */proc/sys/vm/overcommit_memory* and */proc/sys/vm/oom_adj* in **proc**(5), and the Linux kernel source file *Documentation/vm/overcommit-accounting*.

Normally, **malloc**() allocates memory from the heap, and adjusts the size of the heap as required, using **sbrk**(2). When allocating blocks of memory larger than **MMAP_THRESHOLD** bytes, the glibc **malloc**() implementation allocates the memory as a private anonymous mapping using **mmap**(2). **MMAP_THRESHOLD** is 128 kB by default, but is adjustable using **mallopt**(3). Prior to Linux 4.7 allocations performed using **mmap**(2) were unaffected by the **RLIMIT_DATA** resource limit; since Linux 4.7, this limit is also enforced for allocations performed using **mmap**(2).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using **brk**(2) or **mmap**(2)), and managed with its own mutexes.

SUSv2 requires **malloc**() to set *errno* to **ENOMEM** upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private malloc implementation that does not set *errno*, then certain library routines may fail without having a reason in *errno*.

Crashes in **malloc**() or **free**() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The **malloc**() implementation is tunable via environment variables; see **mallopt**(3) for details.

NAME

read − read from a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t read(int *fd***, void ****buf***, size_t** *count***);**

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. See also NOTES.

On error, −1 is returned, and *errno* is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

ERRORS

EBADF
fd is not a valid file descriptor or is not open for reading.

EFAULT
buf is outside your accessible address space.

EINTR
The call was interrupted by a signal before any data was read; see **signal**(7).

EINVAL
fd is attached to an object which is unsuitable for reading; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling terminal, and either it is ignoring or blocking **SIGTTIN** or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape.

EISDIR
fd refers to a directory.

Other errors may occur, depending on the object connected to *fd*. POSIX allows a **read**() that is interrupted after reading some data to return −1 (with *errno* set to **EINTR**) or to return the number of bytes already read.

NOTES

The types *size_t* and *ssize_t* are, respectively, unsigned and signed integer data types specified by POSIX.1.

NAME

stat, fstat – get file status

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char ****pathname*, struct stat ****buf***);**
int fstat(int *fd***, struct stat ****buf***);**

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *buf*. No permissions are required on the file itself, but—in the case of **stat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

stat() retrieves information about the file pointed to by *pathname*.

fstat() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* inode number */
    mode_t    st_mode;        /* file type and mode */
    nlink_t   st_nlink;       /* number of hard links */
    uid_t     st_uid;         /* user ID of owner */
    gid_t     st_gid;         /* group ID of owner */
    dev_t     st_rdev;        /* device ID (if special file) */
    off_t     st_size;        /* total size, in bytes */
    blksize_t st_blksize;     /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;      /* number of 512B blocks allocated */

    time_t st_atime;  /* Time of last access */
    time_t st_mtime;  /* Time of last modification */
    time_t st_ctime;  /* Time of last status change */
};
```

The field *st_atime* is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

NAME

wait, waitpid – wait for process to change state

SYNOPSIS

#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. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. 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).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call. In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

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 any child process whose process group ID is equal to that of the calling process at the time of the call to **waitpid**().

> 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.

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.

WIFSIGNALED(*wstatus***)**
returns true if the child process was terminated by a signal.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on failure, −1 is returned.

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 failure, −1 is returned.