Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Musterlösung Scheinklausur

27.03.2017

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 pages (including draft pages).

- Die Prüfung besteht aus 17 Blättern: 1 Deckblatt, 16 Aufgabenblättern mit insgesamt 3 Aufgaben und 0 Blättern Man-Pages.
  The examination consists of 17 pages: 1 cover sheet, 16 sheets containing 3 assignments, and 0 sheets for man pages.

- Es sind keinerlei Hilfsmittel erlaubt!
  No additional material is allowed.

- Die Prüfung gilt als nicht bestanden, wenn Sie versuchen, aktiv oder passiv zu betrügen.
  You fail the examination if you try to cheat actively or passively.

- Wenn Sie zusätzliches Konzeptpapier benötigen, verständigen Sie bitte die Klausuraufsicht.
  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 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				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Eine zweidimensionale Matrix mit 64 Spalten wird auf ein Array abgebildet, indem die Zeilen der Matrix wie folgt aneinandergehängt in das Array geschrieben werden:

A two-dimensional matrix with 64 columns is to be mapped into an array. To that end, the rows of the matrix are concatenated and written into the array as follows:

row 0	row 1	row 2	row 3	···

Schreiben Sie eine Funktion, die aus den Zeilen- (`row`) und Spaltenkoordinaten (`col`) eines Matrixelements den zugehörigen Arrayindex berechnet und diesen zurückgibt. **1 pt**

Write a function that returns the corresponding array index from the row (`row`) and column coordinates (`col`) of a matrix element.

Lösung:

```
int get_array_index(int row, int col) {
    return row * 64 + col;
}
```

Computing the right index **(0.5 P)**, *Returning the result* **(0.5 P)**

Schreiben Sie eine Funktion, die aus dem Arrayindex `index` die Koordinaten des zugehörigen Matrixelements berechnet und diese in den Variablen `row` und `col` ablegt. **1.5 pt**

Write a function that calculates the coordinates of the corresponding matrix element from the array index `index` and puts these coordinates into the variables `row` and `col`.

Lösung:

```
void get_matrix_coordinates(int index, int *row, int *col) {
    *row = index / 64;
    *col = index % 64;
}
```

Computing the correct row index **(0.5 P)**, *Computing the correct col index* **(0.5 P)**, *Assigning via dereferencing* **(0.5 P)**

b) Woran lässt sich in einem 32-Bit-System typischerweise erkennen, dass eine Adresse auf eine Datenstruktur im Kernel verweist? **1 pt**

On a 32 bit system, how can you typically tell if a given address references a kernel data structure?

Lösung:

Typically, the uppermost one or two gigabytes of virtual address space are reserved for the kernel **(0.5 P)**. *If the address points into that area (e.g., the most significant bit is set), it references a kernel data structure* **(0.5 P)**.

c) Welchen Wert hat die Variable `i` nach der Ausführung? Begründen Sie Ihre Antwort. **2 pt**

What value does the variable `i` have after execution? Explain your answer.

```
int i = 1 - (uint8_t)(-1);
```

Lösung:

The correct value is -254 **(1 P)**. *Casting -1 to `uint8_t` leads to an overflow, which yields the maximum value allowed in an `uint8_t` (255)* **(0.5 P)**. *Subtracting 255 from 1 gives -254. Since the result of the subtraction is implicitly an int (and so is `i`), there are no further overflows* **(0.5 P)**.

d) Was berechnet die folgende Funktion? Begründen Sie Ihre Antwort. **2 pt**

What does the following function calculate? Explain your answer.

```
unsigned int calc(unsigned int a) {
    int b = ~((1 << 12) - 1);
    return a & b;
}
```

Lösung:

The function rounds down `a` to a multiple of 4096 **(1 P)**.

To this end, the function calculates the bit mask `b`, where the 12 lowermost bits are zeros and all other bits are ones **(0.5 P)**. *Performing the bitwise `and` between `a` and the mask, resets `a`'s 12 lowermost bits and leaves the rest unchanged* **(0.5 P)**.

e) Erklären Sie den Begriff *Speicherleck* ("Memory Leak"). Nennen Sie darüber hinaus eine weitere Betriebssystemressource, auf die diese Problemklasse häufig zutrifft. **1.5 pt**

Explain the term "memory leak". In addition, give another operating system resource that is often leaked.

Lösung:

A memory leak is a situation where an application loses access to a piece of memory **(0.5 P)** *without having explicitly freed that memory* **(0.5 P)**.

Other resource: file descriptors **(0.5 P)**.

f) Betrachten Sie die folgenden zwei alternativen Funktionssignaturen:

Consider the following two alternative function signatures:

① `void f(struct mystruct_t s);`

② `void f(struct mystruct_t *s);`

Erläutern Sie, worin sich die Signaturen semantisch unterscheiden. Gehen Sie dabei kurz auf Vor- und Nachteile ein. **2 pt**

Explain how the signatures differ semantically and briefly discuss pros and cons.

Lösung:

The first version copies the source argument on the stack (call-by-value) **(0.5 P)**, *whereas the second version only passes a pointer to the source structure (call-by-reference)* **(0.5 P)**.

Depending on the size of `mystruct_t`, *passing a pointer to a structure is usually faster, because the caller does not need to copy the structure* **(0.5 P)**. *However, this does allow the function to modify the source structure (can be an advantage, too)* **(0.5 P)**.

Ergänzen Sie einen Aufruf der zweiten Variante und übergeben Sie `s`. **0.5 pt**

Add a call to the second version, supplying `s`.

Lösung:

```
struct mystruct_t s = {0};
f(&s);
```

Nennen Sie einen typischen Anwendungsfall für folgende Variante: **1 pt**

Give a typical use-case for the following version:

```
void f(struct mystruct_t **s);
```

Lösung:

- `f` *allocates a new instance of* `mystruct_t` *and wants to return the pointer to the caller.*
- `f` *expects an array of pointers to* `mystruct_t` *as its argument.*
- `f` *expects a two-dimensional array of* `mystruct_t` *as its argument.*

(1 P) *if any one of the above is given.*

Answers declaring `**s` *an array of* `mystruct_t` *(as opposed to an array of pointers) are* ***not*** *acceptable.*

g) Können Linux-Prozesse jedes Signal, das an sie gesendet wird, selbst behandeln? Begründen Sie Ihre Antwort. **1 pt**

Are Linux processes able to handle every signal that might be sent to them? Justify your answer.

Lösung:

No **(0.5 P)**.

Handling a signal implicitly allows the process to ignore that signal. In Linux, however, some signals are meant to bring misbehaving processes back under control – e.g., `SIGKILL` *is meant to terminate a process that refuses to do so voluntarily. Allowing processes to ignore such signals would defeat the whole purpose of the signal* **(0.5 P)**.

h) Kreuzen Sie für jede der folgenden Funktionen in Linux an, ob diese immer, manchmal oder nie einen Systemaufruf ausführt. **2.5 pt**

For each of the following functions in Linux, mark whether it always, sometimes, or never executes a system call.

Immer/ *Always*	Manchmal/ *Sometimes*	Nie/ *Never*	
■	☐	☐	`close()`
☐	■	☐	`pthread_mutex_lock()`
☐	☐	■	`memset()`
☐	■	☐	`malloc()`
■	☐	☐	`mq_unlink()`

i) Warum ist es für Entwickler wichtig, zu wissen, welche Funktionen Systemaufrufe ausführen und welche nicht? Geben Sie ein Beispiel an. **2 pt**

Why do developers need to know which functions cause system calls and which do not? Give an example.

Lösung:

System calls inevitably cause overhead for switching to kernel mode and validating parameters. Therefore, developers should pay attention to the number of system calls their program does **(1 P)**.

For example, reading a bunch of integers individually from a file will result in poor performance, because each read is a system call. Reading large blocks at once and then processing each block in the application avoids most of the overhead **(1 P)**.

j) Welche Systemaufrufe benötigt eine Shell, um ein externes Programm im Hintergrund zu starten? Wofür werden diese Funktionen jeweils benötigt? **2 pt**

Which system calls does a shell need to start an external program in the background? What is each function needed for?

Lösung:

The shell first needs `fork()` **(0.5 P)** *to create a new process* **(0.5 P)**. *Then, the new process executes* `execve()` *(or other variants)* **(0.5 P)** *to replace its own process image with the external program to be started* **(0.5 P)**.

Total: 20.0pt

Aufgabe 2: Dateikompression

Assignment 2: 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.*

```c
/* include statements for the required C headers */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h> /* not requested */

/* global variables */
int parallel_processes = 0;
```

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.

Lösung:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
size_t get_file_size(const char *path) {
    struct stat info;
    stat(path, &info);
    return info.st_size;
}
```

Adding necessary includes **(0.5 P)**, *Calling `stat` on the file* **(1 P)**, *Returning `st_size`* **(0.5 P)**.

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.

- 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.

6.5 pt

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.*

Lösung:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
size_t read_file(const char *path, char *buf, size_t buf_len) {
    int fd = open(path, O_RDONLY);
    if (fd == -1)
        return 0;

    size_t n = 0;
    while (n < buf_len) {
        ssize_t count = read(fd, buf, buf_len - n);
```

```
        if (count == 0) {
            break; /* End of file */
        } else if (count == -1) {
            if (errno == EINTR)
                continue; /* Signal interrupt */

            break; /* Other error */
        }

        buf += count;
        n   += count;
    }

    close(fd);

    return n;
}
```

Adding necessary includes **(1 P)**, *Opening the file* **(1 P)**, *Aborting on* `open()` *error* **(0.5 P)**, *Reading the file* **(1 P)**, *Handling signals* **(0.5 P)**, *Handling errors* **(0.5 P)**, *Stopping on EOF* **(0.5 P)**, *Stopping on buffer full* **(0.5 P)**, *Closing the file (no leak)* **(0.5 P)**, *Returning the number of bytes read* **(0.5 P)**

c) Der Systemaufruf `mmap()` könnte ebenfalls für den Zugriff auf die Dateiinhalte genutzt werden. Nennen Sie einen Vor- und einen Nachteil dieses Ansatzes. Begründen Sie Ihre Antwort. **2 pt**

The system call `mmap()` *could also be used to access the file contents. Give one advantage and one disadvantage of this approach. Justify your answer.*

Lösung:

Possible answers (at most one advantage **(1 P)** *and one disadvantage* **(1 P)***):*

Pro *Easier to use - no buffer management needed .*

Pro *Improved performance - I/O and computation can be overlapped.*

Contra *Slower - Each first access to a mapped file page causes a page fault* **(1 P)**.

Contra *Address space usage - especially on 32-bit systems large files quickly fill up the available address space.*

Answers which assume the whole file to be loaded into memory at once during the call to `mmap()` *will result in at most* **(0.5 P)** *per advantage/disadvantage.*

d) Vervollständigen Sie die Funktion `compress_file()`, die die Eingabedatei an Pfad `in` vollständig einliest, die Daten komprimiert und in die Ausgabedatei an Pfad `out` schreibt.

- Verwenden Sie `get_file_size()` und `read_file()`.
- Verwenden Sie die Funktion `write_compressed()`, die einen Puffer komprimiert und an den gegebenen Ausgabepfad schreibt. Existierende Inhalte der Zieldatei werden bei einem Aufruf gelöscht.
- Die Eingabedatei wird während des kompletten Vorgangs nicht modifiziert.

3 pt

Complete the function `compress_file()`*, which reads the entire input file at the path* `in` *and writes the compressed data to the output paths* `out`*.*

- *Use* `get_file_size()` *and* `read_file()`*.*
- *Use the function* `write_compressed()`*, which compresses a buffer and writes it to the given output path. A call will delete existing contents in the output file.*
- *The input file is not modified during the entire operation.*

Lösung:

```
#include <stdlib.h>
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);
}
```

Retrieving file size **(0.5 P)**, *Allocating a buffer* **(1 P)**, *Reading the file* **(0.5 P)**, *Writing the compressed file* **(0.5 P)**, *Freeing allocated memory* **(0.5 P)**.

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

- 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 2 reservierten Platz einfügen.

3.5 pt

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 2.*

Lösung:

```
int parallel_processes = 0;

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void spawn_process(const char *in, const char *out) {
    /* wait until only 3 or less compression processes remain */
    if (parallel_processes >= 4) {
```

```
        wait(NULL);
    } else {
        parallel_processes++;
    }

    /* start a new process */
    if (fork() == 0)
        compress_file(in, out);
}
```

Adding necessary includes **(0.5 P)**, *Defining global process counter* **(0.5 P)** *Maintaining the correct process count* **(0.5 P)**, *Using* `wait` *to wait for a child to exit* **(1 P)**, *Using* `fork` *to create a new process* **(0.5 P)**, *Calling* `compress_file` *in the child* **(0.5 P)**.

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

- 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.

3 pt

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.*

Lösung:

```
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char **argv) {
    /* read the arguments and spawn a process for each input file */
    int i;
    for (i = 1; i + 1 < argc; i += 2)
        spawn_process(argv[i], argv[i + 1]);

    /* wait until all spawned processes have finished */
    while (parallel_processes > 0) {
        wait(NULL);
        parallel_processes--;
    }

    return 0;
}
```

Adding necessary includes **(0.5 P)**, *Looping through argument pairs* **(0.5 P)**, *Correctly handling an odd number of arguments* **(0.5 P)**, *Calling* `spawn_process` *on each pair* **(0.5 P)**, *Calling* `wait` *for all remaining processes* **(1 P)**.

Total: 20.0pt

Aufgabe 3: Buddy-Speicherallokation

Assignment 3: Buddy Memory Allocation

Für die Verwaltung des physischen Speicherbereichs $[0; 1\,\text{GiB})$ kommt in einem System ein Buddy-Allokator zum Einsatz. Der Allokator ist als Binärbaum implementiert, dessen Knoten mit der `node`-Struktur beschrieben werden. Jeder Knoten repräsentiert einen Speicherbereich der Länge $2^{\texttt{order}}$, der an der Adresse `base` beginnt. `free` ist für freie Blätter 1, ansonsten 0. Kindknoten werden mit den Zeigern `l` (links) und `r` (rechts) referenziert. `p` zeigt auf den Elternknoten. `root` ist die Wurzel des Baums.

Abbildung 3.1 zeigt den Binärbaum (a) vor der ersten Allokation, (b) nach einer Allokation von 256 MiB und (c) nach einer weiteren Allokation von 512 MiB und einer anschließenden Freigabe der ersten 256 MiB. Die Blätter der Größe 2^{28} wurden wieder vereint.

A system uses a buddy allocator to manage the physical memory area $[0; 1\,\text{GiB})$. The allocator is implemented as a binary tree, whose nodes are described by the `node` structure. Each node represents a memory area of size $2^{\texttt{order}}$, which starts at address `base`. `free` is 1 for free leaves, 0 otherwise. Child nodes are referenced via `l` (left) and `r` (right). `p` points to the parent node. `root` is the root of the tree.

Figure 3.1 depicts the binary tree (a) before the first allocation, (b) after an allocation of 256 MiB, and (c) after another allocation of 512 MiB and a following free of the first 256 MiB. The leaves of size 2^{28} have been merged again.

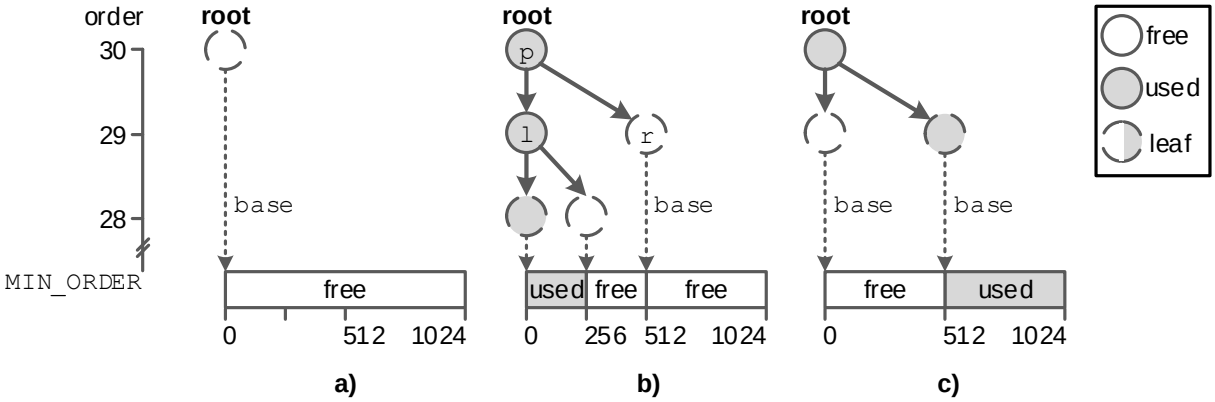

Abbildung 3.1 / *Figure 3.1*

```
typedef struct node {
    int order;          // Order of node's represented area
    void* base;         // Base address of represented area
    int free;           // = 1 for free leaves, = 0 otherwise

    struct node *p;     // Pointer to parent node, NULL for root
    struct node *l;     // Pointer to left child, NULL for leaves
    struct node *r;     // Pointer to right child, NULL for leaves
} node;

// Minimum valid allocation size is one frame (4 KiB)
#define MIN_ORDER 12

// BitScanReverse (BSR) - Returns the position of the first most
// significant bit (MSB) set (e.g., 0b00001010 -> 3). This equals log2(s).
int bsr(size_t s);
```

```
node root = {
    .order = 30,          // 2^30 = 1 GiB
    .base  = (void*)0x0, // Starts at address 0x0
    .free  = 1,           // Node is a free leaf

    .p     = NULL,        // Root has no parent
    .l     = NULL,        // No children, yet.
    .r     = NULL
};
```

a) Vervollständigen Sie die Funktion `_order()`, die eine positive Ganzzahl auf die nächste Zweierpotenz aufrundet und den Wert des Exponenten zurückgibt.
Beispiel: $6523 \rightarrow 8192 = 2^{13}$, Rückgabe: 13.

- Ist der Exponent kleiner als `MIN_ORDER`, wird `MIN_ORDER` zurückgegeben.
- Ist die Eingabe bereits eine Zweierpotenz, findet keine Aufrundung statt.
- Sie können die Funktion `bsr()` verwenden.
- Gehen Sie davon aus, dass `size_t` in jedem Fall ausreichend ist.

2.5 pt

Complete the function `_order()`, which rounds a positive integer up to the next power of two and returns the exponent. Example: $6523 \rightarrow 8192 = 2^{13}$, *Return:* 13.

- *If the exponent is less than `MIN_ORDER`, the function returns `MIN_ORDER`.*
- *If the input is already a power of two, the value is not rounded up.*
- *You can use the function `bsr()`.*
- *Assume `size_t` to be sufficient in any case.*

Lösung:

```
int _order(size_t s) {
    if (s <= (1ULL << MIN_ORDER))
        return MIN_ORDER;

    return bsr(s - 1) + 1;
}
```

Ensuring `MIN_ORDER` **(0.5 P)**, *Correctly using `bsr()` to compute* $log_2(s)$ **(0.5 P)**, *Correctly rounding up to the next power of two* **(1 P)**, *Returning the exponent* **(0.5 P)**

Alternative solution (less efficient):

```
int _order(size_t s) {
    size_t v = 1ULL << MIN_ORDER;
    int exp  = MIN_ORDER;

    while (v < s) {
        v <<= 1;
        ++exp;
    }

    return exp;
}
```

b) Vervollständigen Sie die Funktionen `_nalloc()` und `_nfree()`, die eine neue `node`-Struktur allozieren, initialisieren und zurückgeben bzw. wieder freigeben. Gehen Sie davon aus, dass dynamische Speicherallokationen nicht fehlschlagen, der Speicher jedoch nicht vorinitialisiert wurde. **2.5 pt**

Complete the functions `_nalloc()` and `_nfree()`, which allocate, initialize, and return a new `node` structure, or release it, respectively. Assume dynamic memory allocations to not fail. However, the memory is not pre-initialized.

Lösung:

```
node* _nalloc(int order) {
    node* n  = (node*)malloc(sizeof(node));
    n->order = order;
    n->free  = 1;
    n->base  = NULL;
    n->p     = NULL;
    n->l     = NULL;
    n->r     = NULL;

    return n;
}

void _nfree(node *n) {
    free(n);
}
```

Allocating new node with `malloc()` **(0.5 P)**, *Initializing `n->order`, `n->free`, and `n->base`* **(0.5 P)**, *Initializing the node links* **(0.5 P)**, *Returning the newly allocated node* **(0.5 P)**, *Freeing the node with `free()`* **(0.5 P)**

c) Vervollständigen Sie die Funktion `_isLeaf()`, die überprüft, ob ein Knoten ein Blatt ist. In dem Fall wird 1 zurückgegeben, ansonsten 0. **1 pt**

Complete the function `_isLeaf()`, which tests if a node is a leaf. In that case, it returns 1, 0 otherwise.

Lösung:

```
int _isLeaf(node *n) {
    return (n->l == NULL) && (n->r == NULL);
}
```

Checking at least one child **(1 P)**.

d) Vervollständigen Sie die Funktion `_nsplit()`, die den Knoten `n` im Binärbaum in zwei gleichgroße Kindknoten aufteilt (siehe Abbildung 3.1).

- Verwenden Sie `_nalloc()` zur Allokation der Kindknoten.
- Gehen Sie davon aus, dass `MIN_ORDER` nicht unterschritten wird.
- Achten Sie darauf, dass alle Knoten korrekt verbunden und konfiguriert sind. **2.5 pt**

Complete the function `_nsplit()`, which splits the node `n` in the binary tree into two equally-sized child nodes (see Figure 3.1).

- *Use `_nalloc()` for allocating the child nodes.*
- *Assume that the order won't fall below `MIN_ORDER`.*
- *Be sure to correctly connect and configure all nodes.*

Lösung:

```
void _nsplit(node *n) {
    n->l       = _nalloc(n->order - 1);
    n->l->p    = n;
    n->l->base = n->base;

    n->r       = _nalloc(n->order - 1);
    n->r->p    = n;
    n->r->base = n->base + (1ULL << n->r->order);

    n->free    = 0;
}
```

Using `_nalloc()` *with correct order* **(0.5 P)**, *Setting up node linking* **(0.5 P)**, *Setting correct base for left child* **(0.5 P)**, *Setting correct base for right child* **(0.5 P)**, *Updating* `n->free` **(0.5 P)**

e) Vervollständigen Sie die Funktion `_nmerge()`, die einen Knoten `n` im Binärbaum mit seinem Buddy (falls vorhanden), zusammenführt, sofern beide Knoten frei sind.

- Verwenden Sie `_nfree()` zur Freigabe von Knoten.
- Achten Sie darauf, dass gemäß dem Buddy-System die Operation rekursiv fortzusetzen ist, falls infolge weitere Zusammenführungen möglich sind.

3.5 pt

Complete the function `_nmerge()`, *which merges a node* `n` *in the binary tree with its buddy (if existing), if both nodes are free.*

- *Use* `_nfree()` *for freeing nodes.*
- *Be sure to continue merging according to the buddy system, if, as a result of the operation, further merges are possible.*

Lösung:

```
void _nmerge(node *n) {
    node *p = n->p;
    if (p == NULL)
        return;

    if (p->l->free && p->r->free) {
        _nfree(p->l);
        _nfree(p->r);

        p->l    = NULL;
        p->r    = NULL;

        p->free = 1;

        _nmerge(p);
    }
}
```

Checking if a parent exists/node is root **(0.5 P)**, *Checking if both nodes are free* **(0.5 P)**, *Using* `_nfree()` *to free child nodes* **(0.5 P)**, *Resetting node links* **(0.5 P)**, *Updating* `n->free` **(0.5 P)**, *Recursively calling* `_nmerge()` **(1 P)**

f) Vervollständigen Sie die Funktion `_balloc()`, die ausgehend von einem Knoten `n` mittels Tiefensuche im Binärbaum eine Allokation der Länge $2^{\texttt{order}}$ gemäß des Buddy-Systems durchführt. Im Erfolgsfall wird die Basisadresse `base` des allozierten Knotens zurückgegeben, `NULL` falls kein freies Blatt ausreichender Größe gefunden wird.

- Verwenden Sie `_isLeaf()` und `_nsplit()`.
- Nutzen Sie Rekursion, um die Tiefensuche zu implementieren.
- Es gilt: `order` $\in$ [`MIN_ORDER`; `root.order`].

4.5 pt

Complete the function `_balloc()`, which, starting at node `n`, uses a depth-first search in the binary tree to perform an allocation of length $2^{\texttt{order}}$ according to the buddy scheme. The function returns the base address `base` of the allocated node, or `NULL`, if no free leaf of sufficient size can be found.

- *Use `_isLeaf()` and `_nsplit()`.*
- *Employ recursion for implementing the depth-first search.*
- *Assume:* `order` $\in$ [`MIN_ORDER`; `root.order`].

Lösung:

```
void* _balloc(node *n, int order) {
    /* If this node is not a leaf do depth-first search
     * by descending the left node. */
    if (!_isLeaf(n)) {
        void* res = _balloc(n->l, order);
        if (res != NULL)
            return res;

        return _balloc(n->r, order);
    }

    /* This is a leaf. Ensure that its free and of right
     * size. If its too large, split it. */
    if (!n->free) {
        return NULL;
    }

    if (n->order > order) {
        _nsplit(n);

        return _balloc(n->l, order);
    }

    n->free = 0;
    return n->base;
}
```

Correct check for leaf node **(0.5 P)**, *Descending left if* `n` *is not a leaf* **(0.5 P)**, *Descending right if left node was not successful* **(0.5 P)**, *Checking if node is free before any modification* **(0.5 P)**, *Returning* `NULL` *on error* **(0.5 P)**, *Splitting the node if its to large* **(0.5 P)**, *Retrying the descend after splitting* **(0.5 P)**, *Allocating the node if possible (setting* `n->free`*)* **(0.5 P)**, *Returning the base address on success* **(0.5 P)**

g) Vervollständigen Sie die Funktion `_bfree()`, die ausgehend von einem Knoten `n` im Binärbaum das passende Blatt zur Adresse `a` findet (d. h. `base == a`) und freigibt.

- Verwenden Sie `_isLeaf()` und `_nmerge()`.
- Nutzen Sie Rekursion, um die Suche im Baum zu implementieren.
- Gehen Sie davon aus, dass `a` eine gültige, allozierte Basisadresse ist und ein passenden Blatt existiert.

2.5 pt

Complete the function `_bfree()`, which, starting at node `n`, searches in the binary tree for the leaf that represents address `a` (i.e., `base == a`) and frees it.

- *Use `_isLeaf()` and `_nmerge()`.*
- *Employ recursion for implementing the search in the tree.*
- *Assume that `a` is a valid and allocated base address and that the corresponding leaf exists.*

Lösung:

```
void _bfree(node *n, void *a) {
    if (!_isLeaf(n)) {
        if (a < n->r->base) {
            _bfree(n->l, a);
        } else {
            _bfree(n->r, a);
        }
    } else {
        n->free = 1;

        _nmerge(n);
    }
}
```

Correct check for leaf node **(0.5 P)**, *Descending left if `a` is smaller* **(0.5 P)**, *Descending right otherwise* **(0.5 P)**, *Marking the right leaf as free* **(0.5 P)**, *Trying to merge the leaf* **(0.5 P)**

Note: Accessing `n` after calling `_bfree()` for any child is a bug, because `n` might have already been freed due to recursive merging! However, we did not incorporate that error into the rating.

h) Nennen Sie einen Nachteil, der sich durch eine rekursive Implementierung der Baumfunktionen gegenüber einer iterativen Implementierung ergibt. Begründen Sie Ihre Antwort. **1 pt**

Give a disadvantage of a recursive implementation of the tree functions compared to an iterative implementation. Justify your answer.

Lösung:

Stack Size *Each step in the recursion appends a new stack frame to the thread's stack. The stack thus has to grow accordingly to accommodate the required space. Much space on the stack is wasted by calling overhead (arguments, return addresses etc.).*

Execution Overhead *Setting up and freeing stack frames wastes CPU cycles.*

Total: 20.0pt