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* ____________

Hauptklausur

03.03.2021

- 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 31 Blättern: Einem Deckblatt und 25 Aufgabenblättern mit insgesamt 5 Theorieaufgaben, 3 Programmieraufgaben und 5 Seiten Man-Pages.

  The examination consists of 31 pages: One cover sheet and 25 sheets containing 5 theory assignments, 3 programming assignments and 5 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									

Aufgabe T1: Grundlagen
Assignment T1: Basics

a) Nennen Sie je ein Beispiel für einen Mechanismus und für eine Policy eines Betriebssystems. **1 pt**

Name one example for a mechanism as well as for a policy of an operating system.

Mechanism:

Policy:

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

Which advantage does the separation between mechanism and policy provide?

b) Erklären Sie den Unterschied zwischen Nebenläufigkeit und Parallelität. **1 pt**

Explain the difference between concurrency and parallelism.

In welchen Situationen erhöhen Nebenläufigkeit bzw. Parallelität jeweils die Performance des Systems? **1 pt**

In which situations do concurrency and parallelism each increase the performance of the system?

c) Java wird allgemein als einfachere Programmiersprache als C angesehen. Erläutern Sie einen Grund, weshalb es verglichen mit C jedoch schwieriger ist, einen Betriebssystemkernel in Java zu schreiben. **1 pt**

Java is commonly viewed as an easier programming language than C. Explain one reason why it is, however, more difficult to write an operating system kernel in Java compared to C.

d) Erklären Sie, was man unter dem Begriff *Limited Direct Execution* versteht. **1 pt**

Explain what is meant by the term limited direct execution.

Microkernel-Betriebssysteme implementieren Gerätetreiber und Komponenten des Betriebssystems als separate Prozesse im User-Space, anstatt sie im gemeinsamen Kernel-Space zu platzieren. Nennen Sie einen Vorteil und einen Nachteil dieser Architektur. Begründen Sie jeweils Ihre Antworten.

3 pt

Microkernel operating systems implement device drivers and operating system components as separate processes in user space instead of placing them in the shared kernel space. Name one advantage and one disadvantage of this architecture. Justify your answers.

Vorteil / *Advantage:*

Nachteil / *Disadvantage:*

Total: 9.0pt

Aufgabe T2: Prozesse und Threads

Assignment T2: Processes and Threads

a) Wo befinden sich die Prozesskontrollblöcke (PCB) im Adressraum? **0.5 pt**

Where are the process control blocks (PCB) located in the address space?

Kann ein Debugger in einem Mehrprozessorsystem die Ausführung eines gleichzeitig laufenden Prozesses beobachten, indem er dessen Zustand aus PCB und TCB liest? Erklären Sie. **1 pt**

Is it possible for a debugger in a multiprocessor system to observe the execution of another process running in parallel by reading its PCB and TCB? Explain.

b) Geben Sie zwei freiwillige Ereignisse an, die einen Threadwechsel im One-to-One-Modell auslösen können. **1 pt**

Give two voluntary events that might trigger a thread switch in the One-to-One model.

c) Was ist die Aufgabe eines Langzeitschedulers? **0.5 pt**

What does a long-term scheduler do?

d) Welches Problem tritt häufig beim Einsatz von striktem Prioritätsscheduling auf? Beschreiben Sie das Problem kurz. **1 pt**

Which issue commonly occurs in systems with strict priority scheduling? Describe the issue shortly.

e) Gegeben seien drei Prozesse auf einem Einprozessorsystem mit einem Multi-Level Feedback Queue Scheduler. **5 pt**

- Der Scheduler verwendet die drei untenstehenden Warteschlangen.
- Jeder Prozess startet in der Warteschlange 2 und blockiert einmal, nachdem er eine gewisse Dauer gelaufen ist.
- Ein Prozess, der seine Zeitscheibe komplett ausnutzt, wird in die nächste Warteschlange mit niedrigerer Priorität verschoben. Ansonsten wird er in die Warteschlange mit nächsthöherer Priorität verschoben.
- Der Scheduler wird nach jeder halben Zeiteinheit ausgeführt.
- Das Ablaufdiagramm gibt an, welcher Prozess wann ausgeführt wird (Zeile P) und in welche Warteschlange er als nächstes eingefügt wird (Zeile Q).

Füllen Sie die fehlenden Informationen in den beiden untenstehenden Tabellen aus. Markieren Sie zusätzlich zu Ihrer Lösung eine Zelle mit (?), falls mehrere Lösungen möglich sind.

Consider three processes on a uniprocessor system with a Multi-Level Feedback Queue scheduler.

- *The scheduler uses the three queues given below.*
- *Each process starts in queue 2 and blocks once after running for a certain time.*
- *A process that uses all of its timeslice is inserted into the next queue with lower priority. Otherwise, it is inserted into the next queue with higher priority.*
- *The scheduler is executed after each half unit of time.*
- *The scheduling plan shows which process runs in each time slot (row P), as well as the queue the process will be inserted into next (row Q).*

Fill in the missing information in the two tables below. In addition to your solution, mark a cell with (?) if multiple solutions are possible.

Queue	Scheduler	Timeslice	Priority
1	Round Robin		High
2	Round Robin		Mid
3	Round Robin	2	Low

Process	Arrival Time	Job Length	Blocks after …	Blocks for …
1	0	2		
2	0.1			
3	1			

Ablaufplan/*scheduling plan*:

	0–0.5	0.5–1	1–1.5	1.5–2	2–2.5	2.5–3	3–3.5	3.5–4	4–4.5	4.5–5	5–5.5	5.5–6	6–6.5	6.5–7	7–7.5	7.5–8
P	1	2	2	2	3	2	1	1	3	3	2	1	3	3	2	3
Q	1			3	1	2		2		2	1	X		1	X	X

0 1 2 3 4 5 6 7 8

Total: 9.0pt

Aufgabe T3: Speicher
Assignment T3: Memory

a) Gehen Sie von einem System mit 16-Bit virtuellen Adressen aus, welches vier Segmente unterstützt und die Segmentnummer in den höchstwertigsten Bits (MSB) der virtuellen Adresse kodiert. Verwenden Sie die gegebene Segmenttabelle, um die Lücken in den Übersetzungen zu füllen. **3 pt**

Assume a system with 16-bit virtual addresses that supports four segments and encodes the segment number in the most significant bits (MSB) of the virtual address. Use the provided segment table to fill the gaps in the translations.

Segment Nr.	Base	Limit
0	0x1000	0x3000
1	0x4000	0x0A00
2	0x8400	0x1000
3	0xEE00	0x0200

Virtual Address	Segment Nr.	Offset	Physical Address
0x8450			
	1	0x00A0	
			0xEFDE

b) Nennen Sie zwei Nachteile von segmentbasierter Speicherverwaltung gegenüber seitenbasierter Speicherverwaltung. **1 pt**

Give two disadvantages of segment-based memory management compared to page-based memory management.

c) Wie groß ist der virtuelle Adressraum für eine 5-stufige Seitentabelle mit 64-Bit Einträgen und 4 KiB Seiten? Machen Sie Ihren Rechenweg deutlich. **1 pt**

What size is the virtual address space for a 5-level page table with 64-bit entries and 4 KiB pages? Explain your calculation.

d) Eine bereits ausgewählte Heap-Seite soll aus dem Adressraum von Prozess A ausgelagert und dem Prozess B als freie Seite zur Verfügung gestellt werden. Erläutern Sie, welche grundlegenden Schritte für diesen Vorgang notwendig sind. Gehen Sie davon aus, dass der Kernel über ein eigenes Mapping der physischen Seite verfügt. **2.5 pt**

An already selected heap page should be swapped out from the address space of process A and made available to process B as a free page. Explain which basic steps are necessary for this operation. Assume that the kernel has its own mapping of the physical page.

e) Was versteht man unter dem Begriff *Thrashing*? **0.5 pt**

What is meant by the term thrashing*?*

Weshalb kann die Berechnung von Working Sets in dieser Situation nützlich sein? Erläutern Sie Ihre Antwort. **1 pt**

Why can the calculation of working sets be useful in this situation? Justify your answer.

Total: 9.0pt

Aufgabe T4: Koordination und Kommunikation von Prozessen
Assignment T4: Process Coordination and Communication

a) Eingehende Nachrichten sollen von einer Gruppe von Prozessen bearbeiten werden können. Welche Art des Message Passing müssen Sie verwenden? **0.5 pt**

Incoming messages shall be processed by a group of processes. Which type of message passing do you have to use?

b) Nennen Sie die drei Klassen von Gegenmaßnahmen gegen Deadlocks. **1.5 pt**

Name the three classes of countermeasures against deadlocks.

c) Erklären Sie, weshalb ein Spinlock nicht dafür geeignet ist, Zugriffe von zwei Threads zu synchronisieren, die auf dem gleichen CPU-Kern ausgeführt werden. **1.5 pt**

Explain why a spinlock is not suitable for synchronization of accesses by two threads which execute on the same CPU core.

Kernel verwenden häufig Spinlocks, Anwendungen dagegen nicht. Weshalb ist ein sinnvoller Einsatz im Kernel einfacher als in Anwendungen? **1 pt**

Kernels often use spinlocks, whereas applications do not. Why is sensible use easier in the kernel than in applications?

Erklären Sie, wie ein *Two-Phase-Lock* funktioniert. **1 pt**

Explain how a two-phase lock *works.*

d) Beschreiben Sie die Auswirkung einer `fence`-Instruktion auf das Verhalten des Prozessors. Weshalb ist dieses Verhalten für den Code zum Betreten bzw. Verlassen eines kritischen Abschnittes notwendig? **2 pt**

Describe the impact of a `fence` instruction on the behavior of the processor. Why is this behavior necessary for the code to enter and exit a critical section?

e) Skizzieren Sie eine Methode, mit der das Betriebssystem die Kommunikation zwischen zwei beliebigen mittels Shared Memory kommunizierenden Prozessen aufzeichnen kann. **1.5 pt**

Sketch a method with which the operating system can record communication between two arbitrary processes which communicate via shared memory.

Total: 9.0pt

Aufgabe T5: I/O, Hintergrundspeicher und Dateisysteme
Assignment T5: I/O, Secondary Storage, and File Systems

a) Nennen Sie eine Blockallokationsstrategie und bewerten Sie ihre Eignung für wahlfreien Dateizugriff. Erklären Sie, welche Schritte für den Dateizugriff bei einem bestimmten Offset notwendig sind. **2.5 pt**

Name a block allocation policy and rate its suitability for random file access. Explain the necessary steps for accessing file contents at a certain offset.

b) Sie schreiben einen vollständigen Block auf einem RAID 4-System mit vier Festplatten für Daten und einer Festplatte für Parität. Wie viele Schreib- bzw. Lesezugriffe in Blockgröße werden auf welchen Festplatten durchgeführt? **1 pt**

You are writing a full block to a RAID 4 system with four disks for data and one disk for parity. How many block-sized read and write accesses are performed on which of the disks?

Welches Problem hat RAID 4? Wie wird dieses Problem von RAID 5 gelöst? **1 pt**

What issue does RAID 4 have? How does RAID 5 solve this issue?

c) Das untenstehende Shellskript wird auf einem Linux-System ausgeführt. Welchen Dateiinhalt haben die einzelnen Dateien danach? Füllen Sie die Tabelle aus. Schreiben Sie „X“, falls auf eine Datei nicht zugegriffen werden kann. **2.5 pt**

The shell script below is run on a Linux system. What content does each file have afterwards? Fill in the table. Write “X” if a file cannot be accessed.

```
# -n: no newline
echo -n A > a
echo -n B > b
# ln target link_name
ln a c
# -s: symlink
ln -s c d
ln -s b e
# >>: append to file
echo -n 1 >> c
echo -n 2 >> d
# mv source dest
mv b c
echo -n 3 >> d
```

File	Contents (`cat File`)
a	
b	
c	
d	
e	

d) Welche Betriebssystemkomponente erlaubt das Einhängen von mehreren Dateisystemen in einem gemeinsamen Ordnerbaum? **0.5 pt**

Which operating system component allows mounting multiple file systems into a shared directory tree?

e) Welcher Systemaufruf sorgt dafür, dass alle veränderten Blöcke im Dateisystemcache auf das Speichergerät geschrieben werden? **0.5 pt**

Which system call flushes all dirty blocks in the file system cache to the disk?

f) Beschreiben Sie einen Vorteil von Log-Structured-Dateisystemen (bzw. Copy-on-Write-Dateisystemen) gegenüber Journaling-Dateisystemen. **1 pt**

Describe an advantage of log-structured file systems (or copy-on-write file systems) over journaling file systems.

Total: 9.0pt

Aufgabe P1: C Grundlagen

Assignment P1: C Basics

a) Was gibt der unten stehende Code bei der Ausführung der Funktion `print_test()` aus? Nehmen Sie an, dass keine andere Funktion in `t` schreibt. **1 pt**

What does the code below print when running the function `print_test()`? Assume that no other function is writing to `t`.

```
struct test {
    int a;
    unsigned b;
    unsigned char c;
};

static struct test t;

void print_test(void) {
    t.c = 65;
    printf("%d/%x/%u\n", t.a, t.b, t.c);
}
```

b) Geben Sie für alle Felder des unten stehenden `struct mystruct` die Größe des Feldes und die Größe des Paddings *nach* dem Feld in Bytes an. Schreiben Sie „0“, falls kein Padding eingefügt wird. Gehen Sie von einem 64-Bit-System aus. **1.5 pt**

For each field of the `struct mystruct` below, give the field's size and the size of the padding after *the field in bytes. Write “0” if the compiler does not insert any padding. Assume a 64-bit system.*

Code	Field size [Bytes]	Padding size [Bytes]
`struct mystruct {`	—	—
`int *a;`		
`uint16_t b;`		
`};`	—	—

Warum fügt der Compiler Padding nach den Feldern ein? **1 pt**

Why does the compiler adds padding after struct fields?

c) Nehmen Sie an, dass Sie eine Bibliothek entwickeln. Sie haben die beiden unten stehenden Headerdateien `types.h` und `funcs.h` geschrieben. Ein Kunde versucht, Ihre Bibliothek zu verwenden (`main.c`), aber der Code kompiliert nicht. Warum? Gehen Sie davon aus, dass die Includes korrekt aufgelöst werden. **1 pt**

Imagine you are developing a library. You have written the two header files `types.h` and `funcs.h` below. A customer is trying to use your library (`main.c`), but the code does not compile. Why? Assume that the includes resolve correctly.

types.h

```
typedef struct {
    int x, y;
} point;
```

funcs.h

```
#include "types.h"

int distance(point a,
             point b);
```

main.c

```
#include "types.h"
#include "funcs.h"

int main() { /* ... */ }
```

Was müssen Sie zur Datei `types.h` hinzufügen, um das Problem zu lösen? **1.5 pt**

What do you need to add to `types.h` to resolve the issue?

```
typedef struct {
    int x, y;
} point;
```

d) In dieser Teilaufgabe sollen Sie ein wachsendes Array (`ga`) implementieren. Es besteht aus einer Länge (`len`), einer Kapazität (`cap`) und einem Pointer zu den separat allozierten Daten (`data`), welche `int`-Werte beinhalten sollen.

In this task, you will implement a growing array (`ga`). It consists of a length (`len`), a capacity (`cap`), and a pointer to separately-allocated data (`data`) which shall contain `int` values.

```
typedef struct _ga {
    size_t len, cap; /* number of elements */
    int *data;
} ga;
```

Warum ist es bei einer solchen Datenstruktur sinnvoll, Länge und Kapazität unabhängig voneinander verändern zu können? **1 pt**

Why does it make sense with such a data structure to be able to change length and capacity independently of each other?

Implementieren Sie die Funktion `ga_init()`, die ein leeres wachsendes Array mit der gegebenen Kapazität alloziert und initialisiert. **3 pt**

- Geben Sie `NULL` zurück, falls ein Fehler auftritt.
- Stellen Sie sicher, dass kein Speicherleck auftritt.
- Der Datenbereich muss nicht initialisiert werden.

Implement the function `ga_init()` that allocates and initializes an empty growing array with the given capacity.

- *Return `NULL` if an error occurs.*
- *Make sure you do not leak any memory.*
- *You do not need to initialize the data buffer.*

```
ga *ga_init(size_t cap) {

}
```

Implementieren Sie die Funktion `ga_push()`, die ein Element an das wachsende Array anhängt. **5 pt**

- Verdoppeln Sie die Kapazität des Arrays wenn nötig.
- Stellen Sie sicher, dass kein Speicherleck auftritt.
- Geben Sie einen Pointer auf das neu angefügte Element zurück, oder `NULL`, wenn ein Fehler auftritt. Verändern Sie `*a` in dem Fall nicht.

Implement the function `ga_push()` that appends an element to the growing array.

- *Double the capacity of the array if necessary.*
- *Make sure you do not leak any memory.*
- *Return a pointer to the inserted item, or `NULL` if an error occured. Leave `*a` unchanged in that case.*

```
int *ga_push(ga *a, int d) {





















}
```

Total: 15.0pt

Matrikelnummer/*Matriculation number*

Aufgabe P2: Ausgabeumleitung

Assignment P2: Output Redirection

Sie sollen ein Programm `log_output` schreiben, das ein anderes Programm startet und dessen Ausgabe sowohl auf die Standardausgabe als auch in eine Logdatei umleitet. Wird das Programm zum Beispiel als `log_output ls -l` aufgerufen, so startet es `ls -l` und gibt die Ausgabe sowohl in einer Datei "log.txt" im aktuellen Arbeitsverzeichnis sowie auf der Standardausgabe aus.

- Sie müssen keine C-Header inkludieren.
- Sie müssen keine Fehlerbehandlung implementieren.
- Geben Sie jegliche im Code angeforderten Ressourcen sofern möglich wieder frei. Dies schließt die Freigabe mit `fork()` erstellter Prozesse ein.

You shall write a program `log_output` which starts another program and redirects its output to the standard output as well as into a log file. If, for example, the program is executed via `log_output ls -l`, it starts `ls -l` and writes its output into a file "log.txt" in the current working directory as well as to the standard output.

- *You do not need to include any C headers.*
- *You do not have to implement any error handling.*
- *Free all resources allocated in the code whenever possible. This includes freeing resources in processes created via `fork()`.*

a) Weshalb benötigt man in der Praxis meistens nach dem `exec()`-Systemaufruf noch einen Aufruf von `exit()`? **1 pt**

In practice, why do most calls to the `exec()` system call have to be followed by a call to `exit()`?

Vervollständigen Sie die Funktion `launch_child()`, die ein Programm startet und einen Dateideskriptor einer Pipe zurückgibt, aus der die Ausgabe des Programms gelesen werden kann. **6 pt**

- Das Programm sowie die Parameter werden in `program` übergeben – wie bei den Parametern für `main()` ist `program[0]` das zu startende Programm.
- `program` ist der Einfachheit halber ein nullterminiertes Array – bei N Parametern (inkl. Programm) ist `program[N]` immer `NULL`.
- Leiten Sie sowohl Standardausgabe (Dateideskriptornummer `STDOUT_FILENO`) als auch Fehlerausgabe (Dateideskriptornummer `STDERR_FILENO`) um.
- Sie können `dup2()` verwenden, um einen Dateideskriptor durch eine Kopie eines anderen zu ersetzen und dadurch Zugriffe auf den Deskriptor umzuleiten.

Complete the function `launch_child()` which starts a program and returns a file descriptor of a pipe from which the output of the program can be read.

- *The program as well as the parameters are passed in `program` – as in the case of the parameters for `main()`, `program[0]` is the program to start.*
- *For reasons of simplicity, `program` is a null-terminated array – for N parameters (including the program), `program[N]` is always `NULL`.*
- *Redirect standard output (file descriptor number `STDOUT_FILENO`) as well as error output (file descriptor number `STDERR_FILENO`).*
- *You can use `dup2()` to replace a file descriptor with a copy of another to redirect accesses to the file descriptor.*

```
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

int launch_child(char **program) {
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
}
```

b) Vervollständigen Sie die Funktion `redirect_output()`, die sämtliche Daten aus dem übergebenen Dateideskriptor liest und sowohl in eine Datei “log.txt” im aktuellen Arbeitsverzeichnis als auch auf die Standardausgabe (`STDOUT_FILENO`) schreibt. **5 pt**

- Die Datei soll geleert werden, wenn sie bereits existiert. Falls sie noch nicht existiert, soll sie angelegt werden. In diesem Fall soll sie nur für den aktuellen Benutzer les- und schreibbar sein, andere Benutzer sollen keinen Zugriff auf die Datei haben.

Complete the function `redirect_output()` which reads all data from the specified file descriptor and writes it into a file “log.txt” in the current working directory as well as to the standard output (`STDOUT_FILENO`).

- *The file shall be emptied if it already exists. If it does not exist yet, it shall be created. In this case, only the current user shall be able to read and write the file, other users shall not be able to access the file.*

```
void redirect_output(int pipe_fd) {
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
}
```

c) Vervollständigen Sie die Hauptfunktion `main()` des Programms. **3 pt**

- Die Funktion soll das in `argv[1]` spezifizierte Programm mit den Parametern in `argv[2]` bis `argv[argc-1]` starten und die Ausgabe mittels `redirect_output()` umleiten.
- Wenn das gestartete Programm beendet wurde, soll die Funktion den Rückgabewert des Programms ermitteln und selbst zurückgeben.
- Der C-Standard schreibt vor, dass `argv[argc]` immer `NULL` ist – auch dieses Array ist also bereits nullterminiert.

Complete the main function of the program.

- *The function shall start the program specified in `argv[1]` with the parameters in `argv[2]` to `argv[argc-1]` and shall redirect its output using `redirect_output()`.*
- *When the program exits, the function shall determine the return value of the program and shall return it.*
- *The C standard stipulates that `argv[argc]` is always `NULL` – this array is therefore already null-terminated.*

```
int launch_child(char **program);
void redirect_output(int pipe_fd);

int main(int argc, char **argv) {
........................................................................................
........................................................................................
........................................................................................
........................................................................................
........................................................................................
........................................................................................
........................................................................................
........................................................................................
}
```

Total: 15.0pt

Aufgabe P3: Dynamic Loader

Assignment P3: Dynamic Loader

Um ein Programm zu starten, lädt der Dynamic Loader zuvor nötige Sektionen des Programms und abhängiger Bibliotheken in den Adressraum. Da die Adressen importierter Funktionen (z. B. `print()`) erst nach dem Laden feststehen, verfügt jede Binärdatei über eine Global Offset Table (GOT), welche beim Laden mit den Adressen importierter Symbole initialisiert wird.

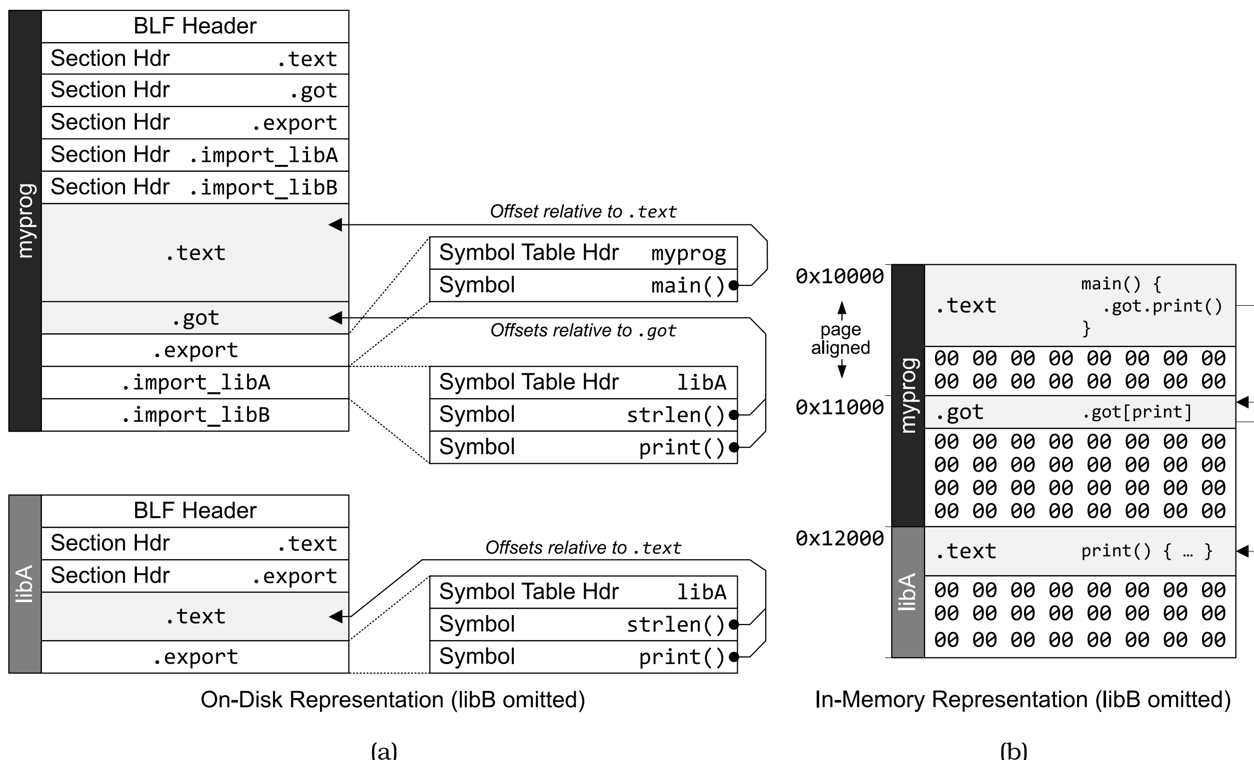
Implementieren Sie einen Dynamic Loader für das Binary Linking Format (BLF):

- Jede Sektion verfügt über einen Header (`Sec`) zu Beginn der BLF-Datei, welcher die Position und Länge der Sektionsdaten angibt.
- Symbole werden in Tabellen (`SymTab`) aufgelistet, welche in den `.export` und `.import` Sektionen gespeichert sind. Für jede importierte Bibliothek existiert eine eigene `.import` Sektion.

To start a program, the dynamic loader first loads necessary sections of the program and dependent libraries into the address space. Since the addresses of imported functions (e.g., `print()`) are not known until after loading, each binary has a Global Offset Table (GOT), which is initialized with the addresses of imported symbols during loading.

Implement a dynamic loader for the Binary Linking Format (BLF):

- *Each section has a header (`Sec`) at the beginning of the BLF file, which specifies the position and length of the section's data.*
- *Symbols are listed in tables (`SymTab`), which are stored in the `.export` and `.import` sections. For each imported library there is a separate `.import` section.*



On-Disk Representation (libB omitted)

In-Memory Representation (libB omitted)

(a) (b)

```
typedef struct {        // BLF header ----
    char magic[4];      // Magic
    uint8_t nsec;       // Number of sections
} Blf;

const char BLF_MAGIC[] = {'\x7f', 'B', 'L', 'F'};

typedef struct {        // Section header ----
    uint8_t type;       // Section type (-> SEC_*)
    uint32_t foff;      // Offset of section data in file (bytes)
    uint32_t size;      // Size of section data in file (bytes)
} Sec;

#define SEC_TEXT   0    // .text section
#define SEC_GOT    1    // .got section (contains dummy values on disk)
#define SEC_EXPORT 2    // Export symbol table (-> SymTab)
#define SEC_IMPORT 3    // Import symbol table (-> SymTab)

typedef struct {        // Symbol table ----
    char name[16];      // Own name (export) or imported BLF file (import)
                        // - both null-terminated
    uint8_t nsym;       // Number of symbols
} SymTab;

typedef struct {        // Symbol ----
    char name[16];      // Null-terminated symbol name
    uint32_t soff;      // Offset in .text (export) or .got (import)
} Sym;
```

a) Vervollständigen Sie die Funktion `check_magic_blf()`, welche verifiziert, dass das Magic-Feld des übergebenen BLF-Handles mit `BLF_MAGIC` übereinstimmt. **1 pt**

- Geben Sie 0 zurück, wenn keine Übereinstimmung vorliegt, ansonsten einen Wert ungleich 0.

Complete the function `check_magic_blf()` which verifies that the magic field matches `BLF_MAGIC` for the passed BLF handle.

- *Return 0 if there is no match, a value not equal 0 otherwise.*

```
int check_magic_blf(Blf *blf) {
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
}
```

Gehen Sie nachfolgend von validen BLF-Dateien und keinen Fehlern aus.
Assume valid BLF files and no errors in the following.

b) Vervollständigen Sie die Funktion `loadsecblf()`, welche die gegebene Sektion `s` des BLF-Programms verarbeitet. Die `.text` und `.got` Sektionen sollen gemäß Abbildung (b) in den Adressraum geladen werden. Für die Symboltabellen soll stattdessen die jeweils passende Funktion aufgerufen werden. **6 pt**

- Nutzen Sie `mmapblf()`, um einen Teil der BLF-Datei in den Adressraum zu laden. `mmapblf()` rundet die Länge auf das nächste Vielfache der Seitengröße (4 KiB) auf und füllt zusätzlichen Raum mit 0-Bytes.
- Laden Sie Sektionen beginnend an `base`. Aktualisieren Sie `base` nach dem Aufruf von `mmapblf()`. Achten Sie darauf, dass `base` stets an Seitengrenzen ausgerichtet ist.
- Speichern Sie die Basisadressen der Sektionen in `sec_base` wie vorgegeben.
- Die Reihenfolge der Sektionen ist durch ihren Typ (`SEC_*`) vorgegeben.
- Sie müssen keine Abhängigkeiten laden.

Complete the function `loadsecblf()` which processes the given section `s` of the BLF program. The `.text` and `.got` sections shall be loaded into the address space according to Figure (b). For the symbol tables, the appropriate function shall be called instead.

- *Use `mmapblf()` to load a part of the BLF file into the address space. `mmapblf()` rounds the length up to the next multiple of the page size (4 KiB) and fills additional space with 0-bytes.*
- *Start loading the sections at `base`. Update `base` after the call to `mmapblf()`. Keep `base` page-aligned.*
- *Store the base addresses of the sections in `sec_base` as specified.*
- *The order of the sections is given by their type (`SEC_*`).*
- *You do not need to load dependencies.*

```
void mmapblf(void *addr, size_t length, Blf *blf, off_t offset);

void *base = (void*)0x10000; // mmap virtual base address

#define ADD_ADDR(a, b) ((void*)((uintptr_t)(a) + (uintptr_t)(b)))
#define NEXT_PAGE(a) (((uintptr_t)(a) + 0x1000 - 1) & ~0xFFFul)

void export_symbols(Blf *blf, off_t symtab, void *text_base);
void import_symbols(Blf *blf, off_t symtab, void *got_base);

void loadsecblf(Blf *blf, Sec *s, void *sec_base[2]) {
    // void *sec_base[2] = {/*.text*/, /*.got*/};
```

...

}

c) Vervollständigen Sie die Funktion `export_symbols()`, welche für jedes Symbol der Exporttabelle an Offset `symtab` mittels eines Aufrufs von `register_symbol()` die absolute Adresse registriert. **4 pt**

- Nutzen Sie `preadblf()`, um den Symboltabellen-Header zu lesen.
- Nutzen Sie `read_symbol()`, um das Symbol mit dem Index `idx` (beginnt bei 0) aus der Tabelle zu lesen.
- Berechnen Sie die absolute Adresse des Symbols im Adressraum, indem Sie die Basisadresse des Textsegments addieren.

Complete the function `export_symbols()` which registers the absolute address of each symbol in the export table at offset `symtab` via a call to `register_symbol()`.

- *Use `preadblf()` to read the symbol table header.*
- *Use `read_symbol()` to read the symbol at index `idx` (begins at 0) of the table.*
- *Calculate the absolute address of the symbol in the address space by adding the base address of the text segment.*

```
#define ADD_ADDR(a, b) ((void*)((uintptr_t)(a) + (uintptr_t)(b)))

void preadblf(Blf *blf, void *buf, size_t count, off_t offset);
void read_symbol(Blf *blf, off_t symtab, uint8_t idx, Sym *sym);
void register_symbol(Blf *blf, const char *name, void *addr);
```

```
void export_symbols(Blf *blf, off_t symtab, void *text_base) {
    SymTab t;
    Sym s;
















}
```

d) Vervollständigen Sie die Funktion `import_symbols()`, welche für jedes Symbol der Importtabelle an Offset `symtab` mittels `SET_GOT()` die absolute Symboladresse in der angegebenen GOT ablegt. **4 pt**

- Nutzen Sie `preadblf()`, um den Symboltabellen-Header zu lesen.
- Nutzen Sie `loadblf()`, um die referenzierte BLF-Bibliothek zu laden. Bereits geladene Bibliotheken werden nicht mehrfach geladen.
- Nutzen Sie `read_symbol()`, um das Symbol mit dem Index `idx` (beginnt bei 0) aus der Tabelle zu lesen.
- `lookup_symbol()` liefert die absolute Adresse eines zuvor registrierten Symbols.

Complete the function `import_symbols()` which saves the absolute address of each symbol in the import table at offset `symtab` via `SET_GOT()` in the specified GOT.

- *Use `preadblf()` to read the symbol table header.*
- *Use `loadblf()` to load the referenced BLF library. Libraries that have already been loaded will not be loaded multiple times.*
- *Use `read_symbol()` to read the symbol at index `idx` (begins at 0) of the table.*
- *`lookup_symbol()` returns the absolute address of a registered symbol.*

```
#define SET_GOT(got_base, soff, addr) \
    (*((void**)ADD_ADDR(got_base, soff)) = (addr))

Blf* loadblf(const char *path);
void preadblf(Blf *blf, void *buf, size_t count, off_t offset);

void read_symbol(Blf *blf, off_t symtab, uint8_t idx, Sym *sym);
void* lookup_symbol(Blf *blf, const char *name);
```

```
void import_symbols(Blf *blf, off_t symtab, void *got_base) {
    SymTab t;
    Sym s;















}
```

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

dup, dup2 − duplicate a file descriptor

SYNOPSIS

#include <unistd.h>

int dup(int *oldfd***);**
int dup2(int *oldfd***, int** *newfd***);**

DESCRIPTION

The **dup**() system call creates a copy of the file descriptor *oldfd*, using the lowest-numbered unused file descriptor for the new descriptor.

After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see **open**(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using **lseek**(2) on one of the file descriptors, the offset is also changed for the other.

The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (**FD_CLOEXEC**; see **fcntl**(2)) for the duplicate descriptor is off.

dup2()

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in *newfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused.

RETURN VALUE

On success, these system calls return the new file descriptor. On error, −1 is returned, and *errno* is set appropriately.

NAME

stdin, stdout, stderr − standard I/O streams

SYNOPSIS

#include <stdio.h>

extern FILE *__stdin__***;**
extern FILE *__stdout__***;**
extern FILE *__stderr__***;**

DESCRIPTION

Under normal circumstances every UNIX program has three streams opened for it when it starts up, one for input, one for output, and one for printing diagnostic or error messages. These are typically attached to the user's terminal (see **tty**(4)) but might instead refer to files or other devices, depending on what the parent process chose to set up. (See also the "Redirection" section of **sh**(1).)

Each of the corresponding symbols is a **stdio**(3) macro of type pointer to *FILE*, and can be used with functions like **fprintf**(3) or **fread**(3).

Since *FILE*s are a buffering wrapper around UNIX file descriptors, the same underlying files may also be accessed using the raw UNIX file interface, that is, the functions like **read**(2) and **lseek**(2).

On program startup, the integer file descriptors associated with the streams *stdin*, *stdout*, and *stderr* are 0, 1, and 2, respectively. The preprocessor symbols **STDIN_FILENO**, **STDOUT_FILENO**, and **STDERR_FILENO** are defined with these values in *<unistd.h>*. (Applying **freopen**(3) to one of these streams can change the file descriptor number associated with the stream.)

NAME

execv, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execv(const char * *pathname*, **char *const** *argv*[]);
int execvp(const char * *file*, **char *const** *argv*[]);

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are layered on top of **execve**(2). (See the manual page for **execve**(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

v - execv(), execvp()

The *char *const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

p - execvp()

This function duplicates the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the **PATH** environment variable.

If the specified filename includes a slash character, then **PATH** is ignored, and the file at the specified pathname is executed.

All other **exec**() functions (which do not include 'p' in the suffix) take as their first argument a (relative or absolute) pathname that identifies the program to be executed.

RETURN VALUE

The **exec**() functions return only if an error has occurred. The return value is −1, and *errno* is set to indicate the error.

NAME

pipe – create pipe

SYNOPSIS

#include <unistd.h>

int pipe(int *pipefd***[2]);**

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see **pipe**(7).

RETURN VALUE

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

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

malloc, free, realloc – allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

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

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**(), or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), or **realloc**(). If the area pointed to was moved, a *free(ptr)* is done.

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.

The **realloc**() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type, or NULL if the request failed. The returned pointer may be the same as *ptr* if the allocation was not moved (e.g., there was room to expand the allocation in-place), or different from *ptr* if the allocation was moved to a new address. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails, the original block is left untouched; it is not freed or moved.

ERRORS

malloc() and **realloc**() 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).

NAME

printf – formatted output conversion

SYNOPSIS

#include <stdio.h>

int printf(const char * *format***, ...);**

DESCRIPTION

printf() writes output to the standard output stream according to a *format* as described below.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

Length modifier

Here, "integer conversion" stands for **d**, **i**, **o**, **u**, **x**, or **X** conversion.

hh A following integer conversion corresponds to a *signed char* or *unsigned char* argument, or a following **n** conversion corresponds to a pointer to a *signed char* argument.

h A following integer conversion corresponds to a *short int* or *unsigned short int* argument, or a following **n** conversion corresponds to a pointer to a *short int* argument.

l (ell) A following integer conversion corresponds to a *long int* or *unsigned long int* argument, or a following **n** conversion corresponds to a pointer to a *long int* argument, or a following **c** conversion corresponds to a *wint_t* argument, or a following **s** conversion corresponds to a pointer to *wchar_t* argument.

ll (ell-ell). A following integer conversion corresponds to a *long long int* or *unsigned long long int* argument, or a following **n** conversion corresponds to a pointer to a *long long int* argument.

Conversion specifiers

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, **i** The *int* argument is converted to signed decimal notation.

o, **u**, **x**, **X**
The *unsigned int* argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation.

c If no **l** modifier is present, the *int* argument is converted to an *unsigned char*, and the resulting character is written. If an **l** modifier is present, the *wint_t* (wide character) argument is converted to a multibyte sequence by a call to the **wcrtomb**(3) function, with a conversion state starting in the initial state, and the resulting multibyte string is written.

s If no **l** modifier is present: the *const char ** argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0').

RETURN VALUE

Upon successful return, the function returns the number of characters printed.

If an output error is encountered, a negative value is returned.

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 current file offset, and the file offset is incremented by the number of bytes read. If the current 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. On error, −1 is returned, and *errno* is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

NAME

write − write to a file descriptor

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

write() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see **setrlimit**(2)), or the call was interrupted by a signal handler after having written less than *count* bytes.

For a seekable file (i.e., one to which **lseek**(2) may be applied, for example, a regular file) writing takes place at the current file offset, and the file offset is incremented by the number of bytes actually written. If the file was **open**(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). On error, −1 is returned, and *errno* is set appropriately.

If *count* is zero and *fd* refers to a regular file, then **write**() may return a failure status if an error is detected. If no errors are detected, 0 will be returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

NAME

wait, waitpid, waitid − wait for process to change state

SYNOPSIS

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int ****wstatus*);**

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.

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 (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of **sigaction**(2)). 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*.

If *wstatus* is not NULL, **wait**() stores 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**()!):

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.

WTERMSIG(*wstatus***)**
: returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

RETURN VALUE

On success, returns the process ID of the terminated child; on error, −1 is returned.

A call sets *errno* to an appropriate value in the case of an error.

ERRORS

ECHILD
: The calling process does not have any unwaited-for children.

EINTR
: **WNOHANG** was not set and an unblocked signal or a **SIGCHLD** was caught; see **signal**(7).

EINVAL
: The *options* argument was invalid.