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
03.03.2022

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

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

- Die Prüfung besteht aus 28 Blättern: Einem Deckblatt, 24 Aufgabenblättern mit insgesamt 5 Theorieaufgaben und 3 Programmieraufgaben sowie 3 Seiten Man-Pages.

  The examination consists of 28 pages: One cover sheet, 24 sheets containing 5 theory assignments as well as 3 programming assignments, and 3 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) Warum ergibt es Sinn, auch dann ein Betriebssystem zu verwenden, wenn auf einem System nur eine Anwendung ausgeführt werden soll? **1 pt**

Why does it make sense to use an operating system even if only one application is to be executed on a system?

b) Bei einem Systemaufruf können Parameter über den Stack oder Register übergeben werden. Erläutern Sie eine zusätzliche Maßnahme, die bei der Übergabe per Stack ergriffen werden muss, um die Sicherheit des Systems zu gewährleisten. **1 pt**

In a system call, parameters can be passed via the stack or registers. Explain one additional measure that must be taken when passing parameters by stack to ensure the security of the system.

Nennen Sie je einen Vorteil und einen Nachteil ohne Sicherheitsbezug für die Parameterübergabe mittels Registern. **1 pt**

Without taking security into account, name one advantage and one disadvantage of passing parameters via registers.

c) Ein `try`/`catch`-Block behandelt zwei Ausnahmen: *ArithmeticException* und *IllegalArgumentException*. Welche der Ausnahmen könnte ursprünglich von der CPU ausgelöst werden? Geben Sie ein Beispiel für eine solche Situation an. **1 pt**

A `try`/`catch` *block handles two exceptions:* ArithmeticException *and* IllegalArgumentException*. Which of the exceptions might originally be raised by the CPU? Give an example for such a situation.*

d) Für die Synchronisation eines kritischen Abschnitts im Kernel wird ein Spinlock verwendet. Welche zusätzliche Maßnahme kann nötig sein, um eine korrekte Synchronisation zu erreichen und wann ist diese notwendig? **1.5 pt**

To synchronize a critical section in the kernel, a spinlock is used. What additional action may be necessary to ensure correct synchronization and when is it necessary?

e) Die Tabelle zeigt die Sektionen einer ELF-Datei. Die zweite Spalte gibt die Position der Sektion in der Datei an. Die dritte Spalte spezifiziert, an welche Adresse die Sektion geladen werden soll. **2 pt**

Das Betriebssystem soll die in der letzten Spalte eingetragenen Zugriffsberechtigungen auf den virtuellen Speicher umsetzen. Zu welchem Problem kommt es bei seitenbasierter Speicherverwaltung mit 4 KiB Seiten? Passen Sie die Tabelle so an, dass die Rechte mit minimalem Ressourcenverbrauch umgesetzt werden können.

The table shows the sections of an ELF file. The second column is the section's position in the file. The third column specifies at which address it should be loaded.

The operating system shall apply permissions to the virtual memory as given in the last column. What problem might occur with page-based memory management and 4 KiB pages? Adjust the table so that the permissions can be applied with minimal resource consumption.

Name	File Offset	Virtual Address	Permissions[1]
`.text`	`0x01000`	`0x01000`	`R-X`
`.data`	`0x03c10`	`0x03c10`	`RW-`
`.rodata`	`0x04f60`	`0x04f60`	`R--`

[1] R: read | W: write | X: execute

Passen Sie die Tabelle so an, dass zwischen `.data` und `.rodata` in der Datei eine 10 KiB `.bss` Sektion liegt, die mit so geringen Berechtigung wie möglich an Adresse `0x10000` geladen wird. **1.5 pt**

Adjust the table so there is a 10 KiB `.bss` section between `.data` and `.rodata` in the file, which is loaded at address `0x10000` with as few permissions as possible.

Total: 9.0pt

Aufgabe T2: Prozesse und Threads

Assignment T2: Processes and Threads

a) Geben Sie einen anderen Namen für das hybride Threadmodell an. **0.5 pt**

Give an alternate name for the hybrid threading model.

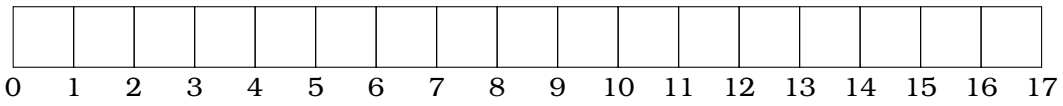
---

b) Gegeben seien vier Prozesse auf einem Einprozessorsystem mit den angegebenen Ankunftszeiten (0 = Start) und Burst-Zeiten. Vervollständigen Sie die untenstehenden Scheduling-Pläne für die Strategie PSJF sowie die Strategie RR für die Zeitscheibenlänge 2 Zeiteinheiten. Bei der RR-Strategie hängt der Scheduler zuerst neue Prozesse (falls vorhanden) ans Ende der Ready-Queue und fügt dann den vorherigen Prozess ans Ende ein (wenn er noch lauffähig ist). Ein Kasten im Zeitplan stellt eine Zeiteinheit dar. Der Scheduler wird nach jeder Zeiteinheit ausgeführt. **4 pt**

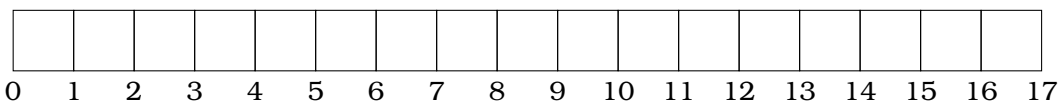
Consider four processes on a uniprocessor system with given arrival times (0 = start) and burst times. Complete the scheduling plans given below for the policy PSJF and the policy RR for a timeslice length of 2 units of time. For RR, the scheduler first adds new processes (if any) to the tail of the ready queue and then inserts the previous process to the tail (if it is still runnable). A box in the scheduling plan represents one unit of time. The scheduler is executed after each unit of time.

Process	Arrival Time	Burst Time
1	1.5	5
2	2.5	2
3	0	6
4	3.5	4

PSJF



RR (timeslice length 2)



Berechnen Sie für den obigen *PSJF-Scheduling-Plan* die Wartezeit der angegebenen Prozesse. **1 pt**

For the above PSJF scheduling plan, calculate the waiting time of each given process.

Process	Waiting time
1	
3	

Berechnen Sie für den obigen *RR-Scheduling-Plan* die Tournaround-Zeit der angegebenen Prozesse. **1 pt**

For the above RR scheduling plan, calculate the tournaround time of each given process.

Process	Turnaround time
2	
4	

c) Füllen Sie die Lücken in dem untenstehenden Text aus. **2.5 pt**

Fill in the gaps in the text below.

Ein Prozess kann zu einem bestimmten Zeitpunkt verschiedene **Zustände** haben. Im _____-Zustand führt ein Prozess Instruktionen auf einem Prozessor aus. Wenn ein Prozess _____ initiiert, wird er eine Zeit lang _____, sodass er nicht unnütz Rechenzeit verschwendet. Nachdem ein Prozess schließlich `exit` aufruft, bleibt er im _____ _____-Zustand erhalten, bis sein Elternprozess seinen _____ _____ ausliest.

*A process can be in different **states** at a given time. In the _____ state, a process is executing instructions on a processor. When a process initiates _____, it becomes _____ for a while so that it does not unnecessarily waste processing time. Finally, after a process calls* `exit`*, it sticks around in the _____ state until its parent process reads its _____.*

Total: 9.0pt

Aufgabe T3: Speicher

Assignment T3: Memory

a) Erläutern Sie zwei Nachteile von einfachen Basis- und Limitregistern gegenüber Segmenttabellen. **2 pt**

Explain two disadvantages of simple base and limit registers compared to segment tables.

b) Weshalb kann der Heap konzeptionell als Stack fungieren, jedoch nicht umgekehrt? **1 pt**

Why can the heap conceptually act as a stack, but not vice versa?

c) Nennen Sie zwei Gründe, warum es Sinn ergibt, in einem Speicherallokator CPU-lokale Listen für kürzlich freigegebene Speicherbereiche zu führen. **1 pt**

Give two reasons why it makes sense to maintain CPU-local lists for recently freed memory areas in a memory allocator.

d) Gegeben sei ein System mit 38-bit virtuellen Adressen. Die Seitengröße beträgt 2 KiB, jeder Seitentabelleneintrag umfasst 32 Bit. Vergleichen Sie rechnerisch den Speicherbedarf einer linearen mit dem einer hierarchischen Seitentabelle. Gehen Sie davon aus, dass der gesamte virtuelle Adressraum belegt ist. **2 pt**

Given a system with 38-bit virtual addresses. The page size is 2 KiB, each page table entry comprises 32 bits. Compare the memory requirements of a linear page table with those of a hierarchical page table via calculation. Assume that the whole virtual address space is in use.

Wieviel virtueller Speicher darf bei der hierarchischen Seitentabelle nicht eingeblendet werden, damit beide Seitentabellen in dem genannten Szenario den gleichen Speicherverbrauch haben? **2 pt**

How much virtual memory must not be mapped with the hierarchical page table so that both page tables consume the same amount of memory in the given scenario?

e) Erläutern Sie, welchen Vorteil Tagging beim TLB hat. **1 pt**

Explain the advantage of tagging in a TLB.

Total: 9.0pt

Aufgabe T4: Koordination und Kommunikation von Prozessen

Assignment T4: Process Coordination and Communication

a) Erklären Sie die Bedingung "No preemption" für Deadlocks. **1 pt**

Explain the condition "No preemption" for deadlocks.

Wie nennt man Techniken, die eine solche Voraussetzung für Deadlocks negieren? **0.5 pt**

How are techniques called that negate such a precondition for deadlocks?

b) Erklären Sie den Unterschied zwischen direktem und indirektem Message Passing. **1 pt**

Explain the difference between direct and indirect message passing.

Wann und weshalb kann asynchrones Empfangen komplett ohne Puffer implementiert werden? **1.5 pt**

When and why can asynchronous receiving be implemented completely without a buffer?

c) Weshalb ist die x86-Instruktion `add memory, 1`, die den Wert an der Speicheradresse `memory` inkrementiert, nicht atomar? **1 pt**

Why is the x86 instruction `add memory, 1` which increments the value at the address `memory` not atomic?

d) Eine Anwendung schützt sämtliche von mehreren Threads genutzten Daten mit einem einzigen Mutex. Nennen Sie einen Vor- und einen Nachteil dieses Ansatzes gegenüber vielen Mutexes, die jeweils kleine Teilmengen der Daten schützen. **1 pt**

An application protects all data that is used by multiple threads with a single mutex. Name an advantage and a disadvantage of this approach compared to many mutexes which each protect a small subset of the data.

Gegeben sei eine Variable `x`, die durch einen Mutex `mutex` geschützt wird. Der folgende Code versucht, Berechnungen auf `x` außerhalb des kritischen Abschnittes durchzuführen. Diskutieren Sie die Lösung bezüglich des Auftretens von Race Conditions. **1.5 pt**

Assume a variable `x` that is protected by a mutex `mutex`. The following code tries to execute calculations on `x` outside of the critical section. Discuss the solution regarding the occurrance of race conditions.

```
while (1) {
    int old_value = atomic_load(&x);
    /* calculation outside of the CS
     * (no side effects, ONLY uses old_value) */
    int new_value = very_expensive_calculation(old_value);

    mutex_lock(&mutex);
    if (old_value == x) {
        /* only write the result if the variable did not change */
        x = new_value;
        mutex_unlock(&mutex);
        break;
    } else {
        /* try again */
        mutex_unlock(&mutex);
    }
}
```

Wie wirkt sich der Ansatz auf die Performance des Programms aus, wenn `lock` außer `x` keine weiteren Variablen schützt? Begründen Sie Ihre Antwort. **1.5 pt**

How does the approach affect the performance of the program if `lock` *does not protect any other variables besides* `x`*? Justify your answer.*

Total: 9.0pt

Aufgabe T5: I/O, Hintergrundspeicher und Dateisysteme

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

a) Nehmen Sie an, der Pfadname `/a/b/c/orig.txt` verweist auf eine normale Datei. Auf wie viele Ordner greift das Betriebssystem beim Öffnen dieser Datei zu? **0.5 pt**

Assume you have a regular file that is referred to by the pathname `/a/b/c/orig.txt`. How many directories will the operating system access when opening this file?

Gehen Sie nun davon aus, dass wir einen Hardlink sowie einen Symlink auf diese Datei wie unten angegeben anlegen. Auf wie viele Ordner greift das Betriebssystem beim Öffnen von `/hard.txt` bzw. von `/a/b/c/soft.txt` zu? **1 pt**

Now assume we create a hard link and a symlink to this file, as given below. How many directories will the operating system access when opening `/hard.txt` or `/a/b/c/soft.txt`?

```
$ ln /a/b/c/orig.txt /hard.txt
```

```
$ ln -s /a/b/c/orig.txt /a/b/c/soft.txt
```

`orig.txt` wird gelöscht. Können wir weiterhin auf `hard.txt` zugreifen? Erklären Sie. **1 pt**

`orig.txt` is deleted. Can we still access `hard.txt`? Explain.

b) Geben Sie zwei Beispiele von Dateitypen an, die üblicherweise in der Inode kodiert sind. **1 pt**

Give two examples for file types that are commonly encoded in the inode.

Wie erkennt ein Unix-System ein Shellskript in einer ausführbaren Datei? **1 pt**

How does a Unix system detect a shell script in an executable file?

c) Füllen Sie die Lücken in dem untenstehenden Text aus. **2.5 pt**

Fill in the gaps in the text below.

_____ ist eine **Dateiallokationsstrategie** bei der jeder Datenträgerblock einen Pointer zu dem nächsten Block der Datei enthält. Sie erreicht gute Performance für _____ Zugriffe, ist aber sehr langsam bei _____ Zugriffen. _____ ist eine verbesserte Strategie, die alle Pointer in _____ lädt.

_____ *is a* ***file allocation strategy*** *where each disk block contains a pointer to the next block in the file. It provides good performace for* _____ _____ *accesses, but is very slow for* _____ *accesses.* _____ *is an improved strategy that loads all the pointers in* _____.

d) Gegeben seien ein RAID-0-System mit 4 Festplatten sowie ein RAID-4-System mit 5 Festplatten. Welchen Nachteil hat das RAID-4-System? Erklären Sie. **1 pt**

Assume you have a RAID-0 system with 4 disks and a RAID-4 system with 5 disks. What disadvantage does the RAID-4 system have? Explain.

Ein Programm schreibt auf 12 zufällige Blöcke. Wie lange muss es jeweils auf dem RAID-0 bzw. RAID-4-System insgesamt auf Festplattenzugriffe warten? Gehen Sie davon aus, dass die Schreibvorgänge soweit wie möglich gleichmäßig auf den Festplatten verteit sind. Jeder Lese- oder Schreibzugriff auf eine Festplatte benötigt D Zeiteinheiten. **1 pt**

A program writes to 12 random blocks. How long in total does it need to wait for disk accesses on the RAID-0 and RAID-4 system, respectively? Assume that these random writes are spread out evenly as much as possible across the disks. Every read or write access to a disk takes D time units.

Total: 9.0pt

Aufgabe P1: C Grundlagen

Assignment P1: C Basics

a) In dem untenstehenden Code haben sich 7 Fehler eingeschlichen. Markieren Sie die fehlerhaften Zeilen mit einem X und korrigieren Sie den Code. Gehen Sie von einem 64-Bit-System aus. **7 pt**

There are 7 errors in the code below. Mark the incorrect lines with an X and correct the code. Assume a 64-bit system.

```
#define CHUNK_SIZE 14
struct chunk {
    int32_t data[CHUNK_SIZE];
    struct chunk *prev;
};
```

```
struct head {
    size_t len;
    struct chunk *tail;
};
```

```
☐ int push(struct head *head, int32_t data) {
☐     struct chunk *tail = head->tail;
☐     if (head->len % CHUNK_SIZE == 0) {
☐         tail = malloc(sizeof(chunk));
☐         if (tail == NULL) return -1;
☐         tail.prev = head->tail;
☐         head->tail = tail;
☐     }
☐     tail->data[head->len++ % CHUNK_SIZE] = data;
☐     return 0;
☐ }

☐ int pop(struct head *head, int32_t *data) {
☐     struct chunk *tail = head->tail;
☐     if (head->len == 0) return -1;
☐     *data = tail->data[head->len-- % CHUNK_SIZE];
☐     if (head->len % CHUNK_SIZE == 0) {
☐         free(tail);
☐         head->tail = tail->prev;
☐     }
☐     return 0;
☐ }

☐ int swap(struct head *head) {
☐     if (head->len < 2) return -1;
☐     struct chunk *prev = head->tail;
☐     size_t i = (head->len - 1) % CHUNK_SIZE;
☐     size_t j = (head->len - 2) % CHUNK_SIZE;
☐     if (i == 1) prev = prev->prev;
☐     int16_t tmp = head->tail->data[i];
☐     head->tail->data[j] = prev->data[i];
☐     prev->data[j] = tmp;
☐     return 0;
☐ }
```

```
/* example usage on next page */
```

Welche Datenstruktur implementiert der Code? **0.5 pt**

Which data structure does the code implement?

Was gibt das Programm (nach Korrektur aller Fehler) bei Ausführung der untenstehenden main-Funktion aus? **1 pt**

What does the program (after correcting all errors) print when executing the main function below?

```
int main() {
    struct head head = {0};
    int32_t i, x;
    for (i = 0; i < 5; i++) {
        push(&head, i);
        swap(&head);
    }
    while (pop(&head, &x) != -1)
        printf("%" PRId32 "␣", x);
    printf("\n");
}
```

Warum ist die `CHUNK_SIZE` (= 14) sinnvoll gewählt? **1 pt**

Why is the `CHUNK_SIZE` (= 14) sensibly chosen?

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

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

```
uint32_t a[] =
  { 0x00112233, 0x44556677,
    0x8899aabb, 0xccddeeff,
    0xff00ff00, 0x11221100 };

uint16_t b = a[0] + a[1];
uint32_t *p0 = a;
uint32_t *p1 = 2 + p0;
// <-- Evaluate expression here
```

Expression	Value
`a[3] & a[4]`	
`~a[5] >> 16`	
`b | 10`	
`*p1`	

c) Ergänzen Sie die Abbruchbedingung der Schleife unten, sodass die Funktion die Länge des Strings `str` berechnet. Lassen Sie den übrigen Code unverändert. **1 pt**

Complete the condition of the loop below so that the function calculates the length of the string `str`. Do not modify other parts of the code.

```
size_t my_strlen(char *str) {
  size_t len = 0;

  while (                                                    )
.....................................................................................
    len++;
  return len;
}
```

d) Die Funktion `g()` des untenstehenden C-Programms wird zu der Assembly rechts kompiliert. Ergänzen Sie die `push`-Instruktionen, sodass die Parameter gemäß cdecl übergeben werden. **0.5 pt**

The function `g()` in the C program below compiles to the assembly on the right. Complete the `push` instructions so that the code passes the parameters according to cdecl.

main.c

```
int f(int, int);

int g() {
  return f(3, 5);
}
```

main.S

```
g:
        push
.....................................................
        push
.....................................................
        call    f
        add     esp, 8
        ret
```

Auf welchem Weg wird das `int`-Ergebnis zurückgegeben? **0.5 pt**

How is the `int` result returned?

e) 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 32-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 32-bit system.*

Code	Field size [Bytes]	Padding size [Bytes]
`struct mystruct {`	—	—
`  int64_t a;`		
`  char b;`		
`};`	—	—

Total: 15.0pt

Aufgabe P2: Build-System

Assignment P2: Build System

Sie sollen eine stark vereinfachte Version des `make`-Tools zum Übersetzen mehrerer Quellcodedateien schreiben. Dieses Programm nimmt als ersten Kommandozeilenparameter eine Datei, die eine Liste von Pfaden wie die folgende enthält:

```
file1.c
file1.o
file2.c
file2.o
```

Das Programm soll für je zwei aufeinanderfolgende Zeilen prüfen, ob die erste Datei neuer ist als die zweite. Falls ja, oder falls die zweite Datei nicht existiert, soll es `gcc` so ausführen, dass die erste Datei in eine Objektdatei übersetzt und an dem zweiten Pfad gespeichert wird:

```
gcc -c file1.c -o file1.o    # Falls file1.c neuer als file1.o
gcc -c file2.c -o file2.o    # Falls file2.c neuer als file2.o
```

- Sie müssen keine C-Header inkludieren.
- Sie müssen nur Fehlerbehandlung implementieren, wenn es explizit gefordert wird.
- Geben Sie im Hauptprozess jegliche angeforderten Ressourcen wieder frei. In Kindprozessen müssen Sie keine Ressourcen freigeben.

You shall write a vastly simplified version of the `make` tool which compiles multiple source code files. As its first command line parameter, this program takes a file which contains a list of paths such as the following:

```
file1.c
file1.o
file2.c
file2.o
```

For each pair of files, the program shall check whether the first file is newer than the second. If so, or if the second file does not exist, the program shall execute `gcc` to compile the first file and to store the resulting object file at the second path:

```
gcc -c file1.c -o file1.o    # If file1.c is newer than file1.o
gcc -c file2.c -o file2.o    # If file2.c is newer than file2.o
```

- *You do not need to include any C headers.*
- *You only have to implement error handling if doing so is explicitly requested.*
- *Free all resources allocated in the main process. You do not need to free resources in child processes.*

```c
/* no lines are longer than the following constant (including '\n'): */
#define PATH_MAX 4096 /* maximum path length (including '\0') on Linux */
```

a) Vervollständigen Sie die Funktion `need_rebuild()`, die prüft, ob die zweite Datei (`path_b`) neu aus der ersten (`path_a`) erstellt werden soll. **3 pt**

- Vergleichen Sie die Zeiten der letzten Änderung. Ist die letzte Änderung von `path_a` jünger als die von `path_b`, so soll die Funktion 1 zurückgeben, ansonsten 0.
- Falls die Datei `path_b` nicht existiert oder die Änderungszeit von `path_b` aus einem anderen Grund nicht ermittelt werden kann, soll die Funktion ebenfalls 1 zurückgeben.

Complete the function `need_rebuild()` which checks whether the second file (`path_b`) shall be rebuilt from the first file (`path_a`).

- *Compare the time of the last modification. If the last modification to `path_a` is more recent than that of `path_b`, the function shall return 1, else it shall return 0.*
- *If the file `path_b` does not exist or the modification time cannot be determined for some other reason, the function shall also return 1.*

```
int need_rebuild(const char *path_a, const char *path_b) {

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}
```

b) Vervollständigen Sie die Funktion `run_gcc()`, die den Befehl `gcc -c <input> -o <output>` ausführt, wobei `<input>` und `<output>` durch die jeweilgen Funktionsparameter ersetzt werden. Die Funktion soll erst zurückkehren, wenn der Kindprozess beendet wurde. **4 pt**

- Verwenden Sie `fork()` und einen der `exec`-Systemaufrufe.

Complete the function `run_gcc()` which executes the command `gcc -c <input> -o <output>`, where `<input>` and `<output>` are replaced by the respective function parameters. The function shall only return once the child process has completed.

- *Use `fork()` and one of the `exec` system calls.*

```
void run_gcc(const char *input, const char *output) {

.....

.....

.....

.....

.....

.....

.....

.....

.....
}
```

c) Die Dateipfade sollen aus einer Datei gelesen werden. Vervollständigen Sie die Funktion `read_line()`, die so lange Daten aus dem Dateideskriptor `file` in den Puffer `buffer` liest, bis sie auf einen Zeilenumbruch trifft. **4.5 pt**

- Der Zeilenumbruch soll nicht nach `buffer` kopiert werden.
- `buffer` soll durch die Funktion immer nullterminiert werden.
- Der Puffer hat die Länge `buf_len`. Passt die Zeile nicht in den Puffer, so soll die Funktion eine Fehlermeldung ausgeben und das Programm unmittelbar mit `exit()` beenden.
- Wenn die Funktion auf das Ende der Datei trifft, soll sie 0 zurückgeben. Ansonsten soll sie 1 zurückgeben, um gültige Daten in `buffer` zu signalisieren. Sie dürfen davon ausgehen, dass die Datei mit einem Zeilenumbruch endet, wie es vom POSIX-Standard vorgeschrieben wird.

The file paths shall be read from a file. Complete the function `read_line()` which reads data from the file descriptor `file` into the buffer `buffer` until it hits a line break.

- *The line break shall not be copied to `buffer`.*
- *`buffer` shall always be null-terminated by the function.*
- *The buffer has the length `buf_len`. If the line does not fit into the buffer, the function shall print an error message and shall immediately quit the program via `exit()`.*
- *If the function meets the end of the file, it shall return 0. Else, it shall return 1 to signal that `buffer` holds valid data. You may assume that the file ends with a newline character as stipulated by the POSIX standard.*

```
int read_line(int file, char *buffer, size_t buf_len) {
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
}
```

d) Vervollständigen Sie die `main()`-Funktion des Programms, die die von `fd` referenzierte Datei zeilenweise einliest. **3.5 pt**

- Nachdem je zwei Zeilen eingelesen wurden, soll die Funktion die durch die zweite Zeile bestimmte Datei aus der durch die erste Zeile bestimmte Datei neu erstellen, falls dies wie oben beschrieben notwendig ist.
- Unter Linux bezeichnet `PATH_MAX` die maximale Länge von Pfaden in Byte, inklusive abschließendem Nullbyte. Ihr Programm soll Zeilen dieser Länge einlesen können.

Complete the `main()` function of the program which reads the file specified by `fd` line by line.

- *Whenever two lines have been read, the function shall rebuild the file specified by the second line from the file specified by the first line if doing so is necessary as described above.*
- *On Linux, `PATH_MAX` specifies the maximum length of paths in bytes, including the terminating null byte. Your program shall be able to read lines of this length.*

```c
int need_rebuild(const char *path_a, const char *path_b);
void run_gcc(const char *input, const char *output);
int read_line(int file, char *buffer, size_t buf_len);

#define PATH_MAX 4096 /* maximum path length (including null character) */

int main(int argc, char **argv) {
    const char *list_file = argv[1];
    int fd = open(list_file, O_RDONLY);

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

    close(fd);
}
```

Total: 15.0pt

Aufgabe P3: Speicherallokator

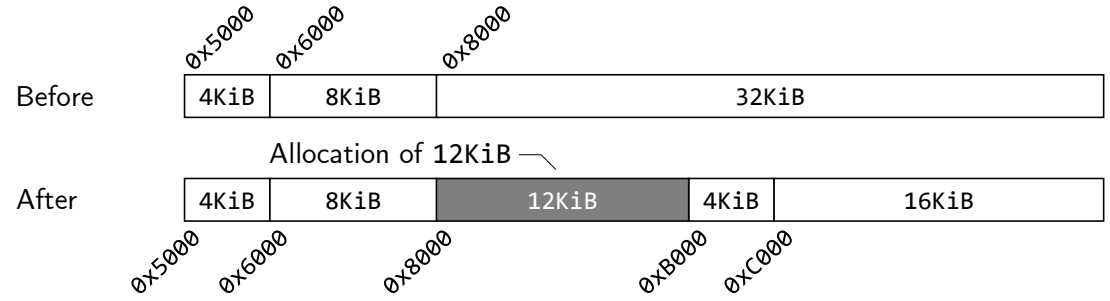
Assignment P3: Memory Allocator

Im Folgenden entwickeln Sie einen Speicherallokator. Der Allokator hält Listen freier Speicherblöcke vor (`free_list[]`), wobei alle Blöcke der Liste mit Index l die Größe 2^{l+12} haben und ihre Startadresse an ihrer Größe ausgerichtet ist. Die minimale Allokationsgröße entspricht demnach 4 KiB.

Bei einer Speicheranfrage sucht der Allokator nach einem möglichst kleinen freien Block, der die Anfrage erfüllt. Ist der gefundene Block größer als die angefragte Speichermenge, wird der überschüssige Platz als freier Speicher den entsprechenden Listen zugewiesen. Die Abbildung zeigt den Zustand vor und nach einer Allokation von 12 KiB. Der Allokator nimmt 12 KiB vom Anfang des 32 KiB-Blocks. Der restliche Speicher des 32 KiB-Blocks wird gemäß der Größen- und Ausrichtungskriterien aufgeteilt und an die Listen freier Blöcke zurückgegeben.

In the following, you develop a memory allocator. The allocator holds lists of free memory blocks (`free_list[]`), where all blocks in the list with index l have the size 2^{l+12} and their starting address is aligned with their size. Thus, the minimum allocation size is equal to 4 KiB.

Given a memory request, the allocator searches for the smallest possible free block that satisfies the request. If the block found is larger than the requested amount of memory, the excess space is assigned as free memory to the corresponding lists. The figure shows the state before and after an allocation of 12 KiB. The allocator takes 12 KiB from the beginning of the 32 KiB block. The remaining space of the 32 KiB block is split according to the size and alignment constraints and returned to the free lists.




```c
// Metadata for free block.
// Stored in the free memory itself at the beginning of each block.
typedef struct block {
    struct block *next;   // Next free block of same size. NULL for last.
    size_t size;          // Size of block in bytes.
} block;

#define MIN_ORDER 12                    // OTS(MIN_ORDER) = 4 KiB
#define MAX_ORDER 30                    // OTS(MAX_ORDER) = 1 GiB

#define OTL(order) (order - MIN_ORDER)  // Order to index into free list.
#define OTS(order) (1ULL << order)      // Order to size in bytes.

block *free_list[OTL(MAX_ORDER)]; // Lists of free blocks. NULL if empty.
```

a) Vervollständigen Sie die Funktion `largest_order()`, welche die größte Zweierpotenz berechnet, an der die Adresse `base` ausgerichtet ist und die kleiner oder gleich der Länge `len` ist. Die Funktion gibt den Wert des Exponenten (Order) zurück. **3.5 pt**

- Gehen Sie davon aus, dass für alle erwartbaren Eingabewerte gilt, dass der Exponent im Intervall [`MIN_ORDER`, `MAX_ORDER`) liegt.

Complete the function `largest_order()` which calculates the largest power of two to which the address `base` is aligned and which is smaller or equal the length `len`. The function returns the exponent (order).

- *Assume that for every input value to be expected the exponent is in the interval [`MIN_ORDER`, `MAX_ORDER`).*

Beispiele/*examples*:

```
largest_order(0x10000, 0x100000) = 16
largest_order(0x10000, 0x1000) = 12
```

```
unsigned int largest_order(uintptr_t base, size_t len) {
.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}
```

b) Vervollständigen Sie die Funktion `link()`, welche den freien Speicherbereich an der Adresse `base` mit der Länge $2^{\texttt{order}}$ an den Anfang der entsprechende Liste freier Blöcke (`free_list[]`) einhängt. **2.5 pt**

- Gehen Sie davon aus, dass `order` im Intervall $[\texttt{MIN\_ORDER}, \texttt{MAX\_ORDER})$ liegt.
- Legen Sie die Struktur `block` an der Adresse `base` ab und initialisieren Sie alle Felder.

Complete the function `link()` which appends the free memory area starting at address `base` and with length $2^{\texttt{order}}$ to the beginning of the corresponding free list (`free_list[]`).

- *Assume that `order` is in the interval $[\texttt{MIN\_ORDER}, \texttt{MAX\_ORDER})$.*
- *Store the structure `block` at the address `base` and initialize all fields.*

```
void link(uintptr_t base, unsigned int order) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

c) Vervollständigen Sie die Funktion `free()`, welche den freien Speicherbereich an der Adresse `base` mit der Länge `len` derart aufteilt, dass alle entstehenden Blöcke an ihrer Länge ausgerichtet sind und die Anzahl der Blöcke minimal ist - also möglichst große Blöcke entstehen (siehe Abbildung auf Seite 21). Die entstehenden Blöcke sollen der passenden Liste freier Blöcke hinzugefügt werden. **2.5 pt**

- Gehen Sie davon aus, dass `len` ein Vielfaches von 4 KiB ist und `base` mindestens an 4 KiB ausgerichtet ist.
- Nutzen Sie die Funktionen `largest_order()` und `link()`.

Complete the function `free()` which splits the free memory area starting at address `base` and with length `len` so that all resulting blocks are aligned to their size and the number of blocks is minimal - that is the blocks are as large as possible (see the figure on page 21). The resulting blocks shall be added to the corresponding free lists.

- *Assume that `len` is a multiple of 4 KiB and that `base` is at least aligned to 4 KiB.*
- *Use the functions `largest_order()` and `link()`.*

```
unsigned int largest_order(uintptr_t base, size_t len);
void link(uintptr_t base, unsigned int order);

void free(uintptr_t base, size_t len) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

d) Vervollständigen Sie die Funktion `malloc()`, welche einen Speicherbereich der Länge `len` alloziert und einen Zeiger darauf zurückgibt oder `NULL`, wenn die Anfrage nicht erfüllt werden kann. **6.5 pt**

- Gehen Sie davon aus, dass `len` ein Vielfaches von 4 KiB ist und `len` $< 2^{\texttt{MAX\_ORDER}}$.
- Die Funktion wählt den ersten freien Block aus der nicht leeren Liste freier Blöcke mit der kleinstmöglichen Größe, um die Anfrage zu erfüllen.
- Nutzen Sie zum Aushängen die Funktion `try_unlink()`, welche `NULL` zurückgibt, wenn die Liste leer ist.
- Die Blockstruktur liegt direkt am Beginn innerhalb des ausgewählten freien Speicherbereichs selbst.
- Rufen Sie die Funktion `free()` auf, um überschüssigen Platz am Ende des Blocks wieder freizugeben.

Complete the function `malloc()` which allocates a memory area of length `len` and returns a pointer to it or `NULL` if the request cannot be satisfied.

- *Assume that `len` is a multiple of 4 KiB and `len` $< 2^{\texttt{MAX\_ORDER}}$.*
- *The function selects the first free block from the non-empty free list with the smallest block size to statisfy the request.*
- *To unlink a block use `try_unlink()` which returns `NULL` if the list is empty.*
- *The block structure is stored directly at the beginning within the selected free memory area itself.*
- *Call `free()` to release excess space at the end of the block.*

```
block *try_unlink(unsigned int order);
void free(uintptr_t base, size_t len);

void *malloc(size_t len) {
```

```
}
```

Total: 15.0pt

NAME

execl, execlp, execv, execvp – execute a file

SYNOPSIS

```
#include <unistd.h>

int execl(const char *pathname, const char *arg, ...
          /*, (char *) NULL */);
int execlp(const char *file, const char *arg, ...
          /*, (char *) NULL */);
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.

l - execl(), execlp()

The *const char *arg* and subsequent ellipses can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast *(char *) NULL*.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

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 - execlp(), execvp()

These functions duplicate 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 this variable isn't defined, the path list defaults to a list that includes the directories returned by *confstr(_CS_PATH)* (which typically returns the value "/bin:/usr/bin") and possibly also the current working directory; see NOTES for further details.

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

In addition, certain errors are treated specially.

If permission is denied for a file (the attempted **execve**(2) failed with the error **EACCES**), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with *errno* set to **EACCES**.

If the header of a file isn't recognized (the attempted **execve**(2) failed with the error **ENOEXEC**), these functions will execute the shell (*/bin/sh*) with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

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

exit – cause normal process termination

SYNOPSIS

```
#include <stdlib.h>

noreturn void exit(int status);
```

DESCRIPTION

The **exit**() function causes normal process termination and the least significant byte of *status* (i.e., *status & 0xFF*) is returned to the parent (see **wait**(2)).

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

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

- If the parent has set **SA_NOCLDWAIT**, or has set the **SIGCHLD** handler to **SIG_IGN**, the status is discarded and the child dies immediately.
- 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

system_data_types – overview of system data types

DESCRIPTION

blkcnt_t

Include: <*sys/types.h*>. Alternatively, <*sys/stat.h*>.

Used for file block counts. According to POSIX, it shall be a signed integer type.

See also: **stat**(2)

blksize_t

Include: <*sys/types.h*>. Alternatively, <*sys/stat.h*>.

Used for file block sizes. According to POSIX, it shall be a signed integer type.

See also: **stat**(2)

off_t

Include: <*sys/types.h*>. Alternatively, <*aio.h*>, <*fcntl.h*>, <*stdio.h*>, <*sys/mman.h*>, <*sys/stat.h.h*>, or <*unistd.h*>.

Used for file sizes. According to POSIX, this shall be a signed integer type.

time_t

Include: <*time.h*> or <*sys/types.h*>. Alternatively, <*sched.h*>, <*sys/msg.h*>, <*sys/select.h*>, <*sys/sem.h*>, <*sys/shm.h*>, <*sys/stat.h*>, <*sys/time.h*>, or <*utime.h*>.

Used for time in seconds. According to POSIX, it shall be an integer type.

See also: **stime**(2), **time**(2), **ctime**(3), **difftime**(3)

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

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

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*.

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

stat, fstat − get file status

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

NAME

wait, waitpid − wait for process to change state

SYNOPSIS

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

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

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

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

```
waitpid(-1, &wstatus, 0);
```

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

−1 meaning wait for any child process.

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

> 0 meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG
return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus*)
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*wstatus*)
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

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

RETURN VALUE

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

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On failure, −1 is returned.