Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

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

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

Nachklausur
Programmieren
29.09.2020

- 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 19 Blättern: 1 Deckblatt, 14 Aufgabenblättern mit insgesamt 3 Aufgaben und 4 Blättern Man-Pages.

  The examination consists of 19 pages: 1 cover sheet, 14 sheets containing 3 assignments, and 4 sheets for man pages.

- Es sind keinerlei Hilfsmittel erlaubt!

  No additional material is allowed.

- Die Prüfung ist nicht bestanden, wenn Sie aktiv oder passiv 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 task 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 widersprüchlichen Lösungen werden mit 0 Punkten bewertet.

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

- Programmieraufgaben sind gemäß der Vorlesung in C zu lösen.

  Programming assignments have to be solved in C.

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

Aufgabe	1	2	3	Total
Max. Punkte	15	15	15	45
Erreichte Punkte				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Was gibt der unten stehende Code bei der Ausführung der Funktion `print_test()` aus? **1 pt**

What does the code below print when running the function `print_test()`?

```
struct test {
    int a, b, c;
};

void print_test(void) {
    struct test t = {.b = 2};
    printf("%d/%d/%d", 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. **3 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 {`	—	—
`  uint32_t a;`		
`  char b;`		
`  void *c;`		
`  int16_t d;`		
`};`	—	—

c) Definieren Sie ein C-Makro `MIN`, das das kleinere von zwei übergebenen Argumenten zurückgibt. **1.5 pt**

Define a C macro `MIN` that returns the smaller of the supplied arguments.

Examples:

```
assert(MIN(3, 5) == 3);
assert(MIN(3, 0) == 0);
assert(MIN(1 | 2, 13 & 5) == 3);
```

Welches Problem mit dem Makro `MIN` könnte im Codeausschnitt unten auftreten? **1 pt**

Which issue with the macro `MIN` could occur in the code snippet below?

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

int main() {
    /* upper bound 10 */
    int x = MIN(10, f());
    /* ... */
}
```

d) Geben Sie ein Beispiel eines *Include-Guards* an. Machen Sie deutlich, wo jedes Stück Code stehen würde. Nutzen Sie keine Spracherweiterungen. **1.5 pt**

Give an example of an include guard. *Clearly indicate where each piece of code would be placed. Do not use any language extensions.*

Wann and warum werden Include-Guards in C benötigt? **1 pt**

When and why are include guards required in C?

e) Schreiben Sie eine Funktion `concat()`, die zwei Strings aneinanderhängt und als neuen String zurückgibt. **3 pt**

- Allozieren Sie genau so viel Heap-Speicher für den neuen String, wie nötig.
- Von den C-Stringfunktionen dürfen Sie lediglich `strlen()` verwenden.

Implement the function `concat()` that concatenates two strings and returns the result as a new string.

- *Allocate exactly as much heap memory for the new string as necessary.*
- *Of the C string functions you may only use `strlen()`.*

```
char *concat(const char *s1, const char *s2) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
.....
```

Warum kann `concat()` bei gleichbleibender Signatur nicht stattdessen Speicher auf dem Stack allozieren? **0.5 pt**

Why is it not possible for `concat()` to allocate memory on the stack instead? Assume you cannot change the function signature.

f) C `enum`s werden häufig verwendet, um Bitflags deskriptive Namen zu geben. Füllen Sie die Deklaration von `enum permission` mit drei unterschiedlichen Bitflags `READ`, `WRITE` und `EXECUTE` aus. **0.5 pt**

C `enum`s are often used to give bitflags descriptive names. Fill in the declaration of `enum permission` with three different bitflags `READ`, `WRITE`, and `EXECUTE`.

```
enum permissions {
.....
.....
.....
.....
.....
};
.....
```

Schreiben Sie eine Funktion `set_rw()`, die die `READ`- und `WRITE`-Flags aus `enum permissions` im übergebenen Wert setzt und diesen anschließend zurückgibt. **1 pt**

Write a function `set_rw()` that sets the `READ` and `WRITE` flags of `enum permissions` in the passed value and then returns this value.

```
unsigned set_rw(unsigned flags) {
.....
.....
.....
.....
}
.....
```

g) Nach welcher Aufrufkonvention werden Funktionsparameter in C ausschließlich über den Stack übergeben? **0.5 pt**

According to which calling convention are function parameters in C passed exclusively via the stack?

Die Funktion `and()` wird vom C-Compiler wie folgt nach Intel x86 übersetzt. In welchen Registern werden die Parameter übergeben? Eine exakte Zuordnung zu `a` und `b` ist nicht notwendig. **0.5 pt**

The function `and()` is translated by the C compiler to Intel x86 as follows. In which registers are the parameters passed? An exact mapping to `a` and `b` is not required.

```
int and(int a, int b) {
    return a & b;
}
```

```
and:
    mov     eax, edi
    and     eax, esi
    ret
```

Total: 15.0pt

Aufgabe 2: Volltextsuche

Assignment 2: Full-Text Search

Sie sollen ein Programm schreiben, das die Vorkommen eines Schlüsselworts in einer Datei sucht. Die Suche soll innerhalb der Datei abschnittsweise mittels mehrerer Prozesse parallelisiert werden. Wird das Schlüsselwort gefunden, so soll auf der Kommandozeile die Position in der Datei in Bytes ausgegeben werden. So soll, wenn das Schlüsselwort „test“ in einer Datei mit dem Inhalt „**testest**asdf**test**“ gesucht wird, das Programm die Zahlen 0, 3 und 11 ausgeben.

- Sie können davon ausgehen, dass es sich um eine Textdatei ohne Null-Bytes handelt.
- Sie müssen keine C-Header inkludieren.
- Sie müssen in dieser Aufgabe keine Fehlerbehandlung implementieren.
- Geben Sie jegliche im Code angeforderten Ressourcen wieder frei.

*You shall write a program which searches a file for a keyword. Inside the file, the search shall be parallelized section-wise using multiple processes. When the keyword is found, the position in the file in bytes shall be printed to the command line. For example, if the keyword “test” is searched in a file containing “**testest**asdf**test**”, the program shall output the numbers 0, 3, and 11.*

- *You may assume that the file is a text file without null bytes.*
- *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.*

a) Vervollständigen Sie die Funktion `print_u64()`, die den übergebenen 64-Bit-Integer und einen Zeilenumbruch ausgibt. **1 pt**

Complete the function `print_u64()` which prints the specified 64-bit integer and a line break.

```
void print_u64(uint64_t x) {
.....

.....

.....
}
```

b) Vervollständigen Sie die Funktion `search_str()`, die alle Vorkommnisse des Schlüsselwortes `word` innerhalb des Strings `buffer` mittels `print_u64()` ausgibt. **4 pt**

- `buffer` enthält die Inhalte der Datei beginnend an der Position `offset`. Die Funktion soll also vor der Ausgabe `offset` auf die jeweilige Position des Schlüsselworts addieren.
- Sie können `strstr()` zum Suchen des Schlüsselworts verwenden.

Complete the function `search_str()`, which prints all occurrences of the keyword `word` in the string `buffer` using `print_u64()`.

- *`buffer` contains the file contents starting at the position `offset`. The function shall therefore add `offset` to the position of each occurrence before printing.*
- *You may use `strstr()` to search for the keyword.*

```
void print_u64(uint64_t x);

void search_str(uint64_t offset, const char *buffer, const char *word) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

c) Vervollständigen Sie die Funktion `search_fd()`, die einen Abschnitt der gegebenen Datei beginnend an der Position `offset` mit der Länge `len` in den Speicher lädt und dann mittels `search_str()` nach dem Schlüsselwort `word` durchsucht. **3.5 pt**

- Beachten Sie, dass `search_str()` nullterminierte Strings erwartet.

Complete the function `search_fd()` which reads a section of the specified file starting at the position `offset` with the length `len` into memory and then uses `search_str()` to search for the keyword `word`.

- *Note that `search_str()` expects null-terminated strings.*

```
void search_str(uint64_t offset, const char *buffer, const char *word);

void search_fd(int fd, uint64_t offset, size_t len, const char *word) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

d) Vervollständigen Sie die Funktion `spawn_search()`, die einen neuen Prozess startet, der `search_fd()` mit den gegebenen Parametern ausführt. **2 pt**

Complete the function `spawn_search()` which starts a new process that executes `search_fd()` with the given parameters.

```
void search_fd(int fd, uint64_t offset, size_t len, const char *word);

void spawn_search(int fd, uint64_t offset, size_t len, const char *word) {
.....
.....
.....
.....
.....
.....
.....
.....
}
```

Die Datei soll stückweise mittels `spawn_search()` durchsucht werden. Wie sehr müssen sich die von `offset` und `len` bestimmten Blöcke überlappen, damit sämtliche Vorkommnisse des Wortes gefunden werden? **1 pt**

The file shall be searched piece-wise by `spawn_search()`. How much do the blocks specified by `offset` and `len` need to overlap so that all occurrences of the word are found?

e) Vervollständigen Sie die Funktion `main()`, die die zu durchsuchende Datei öffnet und dann eine blockweise parallele Suche mittels `spawn_search()` durchführt. **3.5 pt**

- Das Programm soll auf sämtliche Kindprozesse warten, bevor es sich beendet.

Complete the function `main()` which opens the file to be searched and then executes a block-wise parallel search using `spawn_search()`.

- *The program shall wait for all child processes before terminating.*

```
uint64_t file_size(int fd);
void spawn_search(int fd, uint64_t offset, size_t len, const char *word);

int main(int argc, char **argv) {
    const char *file = argv[1];
    const char *word = argv[2]; /* assume that the string is non-empty */

    int fd = open(file, O_RDONLY);
    uint64_t size = file_size(fd); /* length of the file */
    const size_t BLOCK = 4096; /* block size - add the overlap! */
    uint64_t num_blocks = (size + BLOCK - 1) / BLOCK; /* block count */
.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

    close(fd);
    return 0;
}
```

Total: 15.0pt

Aufgabe 3: Dateisystem-Cache

Assignment 3: File System Cache

Der Dateisystem-Cache puffert zur Beschleunigung von Dateizugriffen den Dateiinhalt im Hauptspeicher. Für jede offene Datei muss das Betriebssystem eine Datenstruktur speichern, die einen Datei-Offset in eine Adresse im Cache übersetzt.

Im Folgenden soll zu diesem Zweck ein Radix-Baum implementiert werden. Ein Radix-Baum ist eine dynamisch wachsende Struktur, die einen Ganzzahlindex auf einen Pointer übersetzt.

Ähnlich der Adressübersetzung bei seitenbasierter Speicherverwaltung teilt der Radix-Baum den Index in mehrere gleich große Teile auf. Jeder Teil bestimmt die nächste Ebene im Baum. Die Höhe des Baums wird dabei durch den größten gespeicherten Index festgelegt.

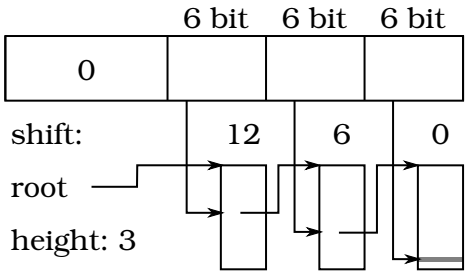
- Ein Baum der Höhe 0 kann genau einen Eintrag bei Index 0 speichern.
- Die `void*`-Pointer sind in der letzten Ebene des Baums Nutzerdaten und in allen anderen Ebenen Pointer auf `rt_node`.

The file system cache buffers file contents in main memory to speed up file access. For every open file, the operating system needs to keep a data structure that translates file offsets to a memory address in the cache.

In the following, you will implement a radix tree for this purpose. A radix tree is as a dynamically growing structure that translates an integer index to a pointer.

Similar to address translation with paging, the radix tree partitions the index into equally-sized parts. Every part determines the next level of the tree. The largest index determines the height of the tree.

- *A tree with height 0 can store exactly one element at index 0.*
- *The `void*` pointers at the last level of the tree are user data. At all other levels they are pointers to `rt_node`.*




```
#define RT_SHIFT 6              /* number of index bits per level */
#define RT_SIZE (1 << 6)        /* number of entries per level */
#define RT_MASK (RT_SIZE - 1)

typedef struct _rt_node {
    uint8_t shift;             /* index shift for this level */
    void *entries[RT_SIZE];
} rt_node;
```

```
typedef struct _rt_tree {
  uint8_t height;              /* number of levels */
  void *root;                  /* root entry */
} rt_tree;
```

a) Vervollständigen Sie die Funktion `entry_idx()`, die für einen Index `idx` den Offset in dem `entries`-Array einer `rt_node`-Struktur mit dem gegebenen `shift` berechnet. **1 pt**

Complete the function `entry_idx()`, which takes an index `idx` and calculates the offset in the `entries` array of an `rt_node` structure with the given `shift`.

```
int entry_idx(uint64_t idx, uint8_t shift) {
.....

.....
}
.....
```

b) Das Einfügen in den Radix-Baum besteht aus zwei Schritten: Dem Hinzufügen von Ebenen in den Baum, sodass der Index passt, und dem Durchlaufen des Baumes bis zum passenden Blattknoten.

Inserting into the tree requires two steps: adding levels to the tree until the index fits, and then descending the tree to the correct leaf node.

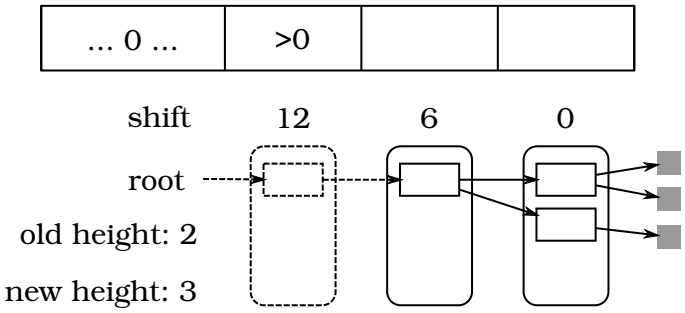
```
void add_levels(rt_tree *rt, uint64_t idx);
void **descend(rt_tree *rt, uint64_t idx);

// Insert an element (ptr) into the radix tree (rt) at index (idx).
void rt_insert(rt_tree *rt, uint64_t idx, void *ptr) {
    add_levels(rt, idx);

    void **entry = descend(rt, idx);
    *entry = ptr;
}
```

Vervollständigen Sie die Funktion `add_levels`, die dem Baum so lange Ebenen hinzufügt, bis der Index `idx` abgebildet werden kann (`idx` < 2^{shift}). **4 pt**

- Die Funktion `alloc_node()` gibt einen neuen Knoten zurück, dessen Felder alle auf 0 initialisiert sind.
- Falls der Baum nicht leer ist, wird der bisherige Wurzelknoten zum 0. Eintrag im neuen Wurzelknoten.
- Für einen leeren Baum sollen Sie keine neuen Knoten allozieren, passen Sie `rt->height` aber in jedem Fall entsprechend an.

Complete the function `add_levels()`, which adds levels to the tree until the index `idx` can be mapped (`idx` < 2^{shift}).

- *The function `alloc_node()` returns a new node with all fields initialized to 0.*
- *If the tree is not empty, the old root node is the 0th entry in the new root node.*
- *For empty trees, you shall not allocate new nodes, but always adjust `rt->height` accordingly.*



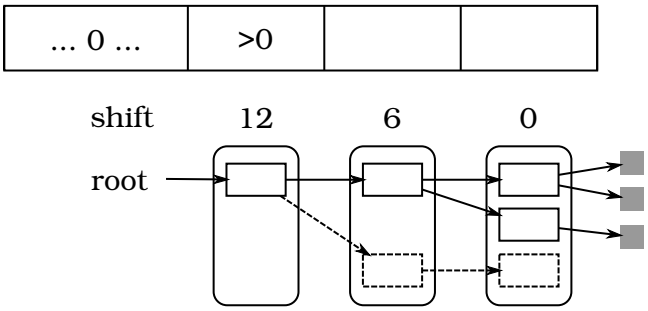

```
rt_node *alloc_node(); /* does not fail */
void add_levels(rt_tree *rt, uint64_t idx) {
    uint8_t shift = rt->height * RT_SHIFT; /* current shift at root */
}
```

c) Vervollständigen Sie die Funktion `descend()`, die den Baum von der Wurzel bis zu dem indizierten Element durchläuft und einen Pointer auf dieses zurückgibt. **4 pt**

- Fehlende Zwischenknoten müssen mit `alloc_node()` hinzugefügt werden.

Complete the function `descend()`, which walks the tree from the root to the indexed element and returns a pointer to it.

- *Add missing intermediate nodes using `alloc_node()`.*




```
rt_node *alloc_node(); /* does not fail */
void **descend(rt_tree *rt, uint64_t idx) {
    uint8_t shift = rt->height * RT_SHIFT;
    void **entry = &rt->root;
    while (shift > 0) {
    }
    return entry;
}
```

d) Vervollständigen Sie die Funktion `count()`, die die Anzahl gültiger (nicht `NULL`) Einträge in einem Knoten zurückgibt. **2 pt**

Complete the function `count()`, which returns the number of valid (not `NULL`) entries in a node.

```
int count(rt_node *n) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

e) Vervollständigen Sie die Funktion `shrink()`, die unnötige Ebenen des Baums entfernt. **4 pt**

- Unnötige Ebenen besitzen lediglich einen gültigen Eintrag an Index 0 oder überhaupt keinen gültigen Eintrag.
- Nutzen Sie die Funktion `free_node()`, um Knoten freizugeben.
- Passen Sie `rt->height` entsprechend an.

Complete the function `shrink()`, which removes unnecessary levels from the tree.

- *An unnecessary level has only one valid entry at index 0 or no valid entry at all.*
- *Call the function `free_node()` to free nodes.*
- *Adjust `rt->height` accordingly.*

```
int count(rt_node *n);
void free_node(rt_node *n);
void shrink(rt_tree *rt) {
.....
.....
.....
.....
.....
.....
.....
.....
```

}

Total: 15.0pt

NAME

close – close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int *fd*);

DESCRIPTION

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

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

RETURN VALUE

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

NAME

exit – cause normal process termination

SYNOPSIS

#include <stdlib.h>

void exit(int *status*);

DESCRIPTION

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

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

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

RETURN VALUE

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

NOTES

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

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

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

NAME

fork – create a child process

SYNOPSIS

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

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

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

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

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

Note the following further points:

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

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, −1 is returned in the parent, no child process is created, and *errno* is set appropriately.

NOTES

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

NAME

inttypes.h — fixed size integer types

SYNOPSIS

#include <inttypes.h>

DESCRIPTION

The *<inttypes.h>* header shall define the following macros. Each expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the *format* argument of a formatted input/output function when converting the corresponding integer type. These macros have the general form of PRI (character string literals for the *fprintf*() and *fwprintf*() family of functions) , followed by the conversion specifier, followed by a name corresponding to a similar type name in *<stdint.h>*. In these names, *N* represents the width of the type as described in *<stdint.h>*. For example, *PRIdFAST32* can be used in a format string to print the value of an integer of type **int_fast32_t**.

The *fprintf*() macros for signed integers are:

PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR

The *fprintf*() macros for unsigned integers are:

PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR

For each type that the implementation provides in *<stdint.h>*, the corresponding *fprintf*() and *fwprintf*() macros shall be defined and the corresponding *fscanf*() and *fwscanf*() macros shall be defined unless the implementation does not have a suitable modifier for the type.

NAME

pread, pwrite − read from or write to a file descriptor at a given offset

SYNOPSIS

#include <unistd.h>

ssize_t pread(int *fd*, **void ***buf*, **size_t** *count*, **off_t** *offset*);**

ssize_t pwrite(int *fd*, **const void ***buf*, **size_t** *count*, **off_t** *offset*);**

DESCRIPTION

pread() reads up to *count* bytes from file descriptor *fd* at offset *offset* (from the start of the file) into the buffer starting at *buf*. The file offset is not changed.

pwrite() writes up to *count* bytes from the buffer starting at *buf* to the file descriptor *fd* at offset *offset*. The file offset is not changed.

The file referenced by *fd* must be capable of seeking.

RETURN VALUE

On success, **pread**() returns the number of bytes read (a return of zero indicates end of file) and **pwrite**() returns the number of bytes written.

Note that it is not an error for a successful call to transfer fewer bytes than requested (see **read**(2) and **write**(2)).

On error, −1 is returned and *errno* is set to indicate the cause of the error.

NOTES

The **pread**() and **pwrite**() system calls are especially useful in multithreaded applications. They allow multiple threads to perform I/O on the same file descriptor without being affected by changes to the file offset by other threads.

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The function **printf**() writes output to *stdout*, the standard output stream; **fprintf**() writes output to the given output *stream*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

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

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*.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given).

The conversion specifier

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 The *int* argument is converted to an *unsigned char*, and the resulting character is written.

s 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').

NAME

strlen – calculate the length of a string

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
```

DESCRIPTION

The **strlen**() function calculates the length of the string pointed to by *s*, excluding the terminating null byte ('\0').

RETURN VALUE

The **strlen**() function returns the number of characters in the string pointed to by *s*.

NAME

strcmp, strncmp – compare two strings

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

The **strcmp**() function compares the two strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

The **strncmp**() function is similar, except it compares only the first (at most) *n* bytes of *s1* and *s2*.

RETURN VALUE

The **strcmp**() and **strncmp**() functions return an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

NAME

strstr – locate a substring

SYNOPSIS

```
#include <string.h>

char *strstr(const char *haystack, const char *needle);
```

DESCRIPTION

The **strstr**() function finds the first occurrence of the substring *needle* in the string *haystack*. The terminating null bytes ('\0') are not compared.

The **strcasestr**() function is like **strstr**(), but ignores the case of both arguments.

RETURN VALUE

These functions return a pointer to the beginning of the located substring, or NULL if the substring is not found.

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.