Karlsruher Institut für Technologie

Betriebssysteme (Operating Systems)

Prof. Dr. Frank Bellosa
Peter Maucher, M.Sc.
Daniel Habicht, M.Sc.
Dipl.-Inform. Thorsten Gröninger
Lukas Werling, M.Sc.

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

Nachklausur 05.09.2025

- Bitte tragen Sie zuerst auf dem Deckblatt Ihren Nachnamen, Ihren Vornamen und Ihre Matrikelnummer ein. Tragen Sie dann auf allen 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 26 Blättern: Einem Deckblatt, 19 Aufgabenblättern mit insgesamt 3 Aufgaben sowie 6 Seiten Man-Pages.

  The examination consists of 26 pages: One cover sheet, 19 sheets containing 3 assignments, and 6 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 also 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 according to the lecture.

Die folgende Tabelle wird von uns ausgefüllt!

The following table is completed by us!

Aufgabe	1	2	3	Total
Max. Punkte	20	20	20	60
Erreichte Punkte				

Aufgabe 1: Virtualisierung (20 P)

Assignment 1: Virtualization (20 P)

In Ihrem Betriebssystem entwickeln Sie an der Unterstützung von Prozessen und Threads. Statt dazu zwei Systemaufrufe zu entwickeln, lassen Sie sich von Linux und dessen `clone`-Systemaufruf inspirieren, der für beide Zwecke genutzt werden kann. In dieser Aufgabe entwerfen und benutzen Sie diesen Systemaufruf mit folgender Signatur:

In your operating system you are developing the support for processes and threads. Instead of developing two system calls you take inspiration from Linux and its `clone` system call, which can be used for both purposes. In this assignment you design and use this system call with the following signature:

```
enum clone_flags {
  DUP_DATA = 1,  // duplicates all remaining modifyable properties into the child
  DUP_STACK = 2, // creates a new stack for the child
  DUP_REGS = 4,  // creates a new set of register descriptors for the child
};
int clone(enum clone_flags flags);
```

a) Begründen Sie für die folgenden 3 Kategorien, wie diese beim Erstellen eines neuen Threads kopiert und geändert werden müssen oder beibehalten werden können. **3 P**

Explain for the following 3 categories, how they must be copied and modified or kept identical on thread creation.

Registers:

Stack:

Heap:

b) Nennen Sie zwei weitere Prozesseigenschaften, die beim `fork()`-Aufruf modifiziert oder kopiert werden müssen, und beschreiben Sie kurz warum die Operation notwendig ist. **2 P**

Name two additional process properties which must be modified or copied using `fork()` and briefly describe why the operation is necessary.

c) Begründen Sie, warum `clone()` im Kernel implementiert sein muss. **1 P**

Give reasons why `clone()` must be implemented in the kernel.

d) **`pthread` Creation**

In dieser Teilaufgabe nutzen Sie den oben beschriebenen `clone()`-Systemaufruf, um darauf basierend das Starten eines Threads und das Einsammeln des Ergebnisses zu implementieren. Sie implementieren dazu die zwei Funktionen `pthread_create` und `pthread_join`. Beachten Sie dabei:

- Sie dürfen annehmen, dass es zwei weitere Systemaufrufe gibt:
  - **`void`** *`exit_thread`*`(`**`void`**`);` das den aufrufenden Thread, aber nicht den ganzen Prozess beendet und
  - **`void`** *`wait_tid`*`(`**`int`** *`thread_id`*`);` das blockiert, bis der angegeben Thread beendet ist. Sie müssen dazu im Parameter `thread_id` die Thread-ID übergeben.
- Sie müssen den *Output-Parameter* `thread` in `pthread_create` mit Werten füllen, um diese Werte in `pthread_join` nutzen zu können. Die Definition von **`struct`** `pthread_t` ist durch die Aufgabenstellung vorgegeben.
- Sie können für diese Aufgabe davon ausgehen, dass keine Fehler auftreten.
- Denken Sie daran, dass Sie alle allokierten Ressourcen wieder freigeben.
- *Hinweis:* Beim Erstellen eines Threads können Änderungen im Adressraum passieren. Stellen Sie sicher, dass Pointer valide bleiben.

In this subtask you use the previously described `clone()` system call to implement the starting of a thread and collection of its result. For this, you implement two functions `pthread_create` and `pthread_join`. Keep in mind:

- *You can assume that there a two additional system calls:*
  - ***`void`*** `exit_thread`*`(`**`void`**`);` *which terminates the calling thread, but not the whole process and*
  - ***`void`*** `wait_tid`*`(`**`int`** `thread_id`*`);` *which blocks until the named thread terminates. You have to pass the thread-id in parameter `thread_id`.*
- *You have to fill the* output parameter *`thread` in `pthread_create` with values to use said values in `pthread_join`. The definition of **`struct`** `pthread_t` is given by the exercise.*
- *In this subtask you can assume that no errors occur.*
- *Keep in mind that you must free all allocated resources.*
- Hint: *During thread creation, changes to the address space can happen. Ensure that pointers remain valid.*

```
typedef struct pthread_t {
    void **return_value; // hint: double pointer
    int thread_id;
} pthread_t;
```

i) Der `clone()`-Systemaufruf soll einen **`int`**-Wert zurückgeben. Diskutieren Sie, was in diesen Wert kodiert werden muss. **1 P**

Hinweis: Denken Sie zum Beispiel daran, wie in POSIX neue Prozesse erstellt werden.

*The `clone()` system call should return an **`int`** value. Discuss what must be encoded in this value.*

Hint: *For example, think about how new processes are created in POSIX.*

ii) Implementieren Sie den `pthread_create`-Funktionsaufruf mithilfe des `clone()`-Systemaufrufs. **3.5 P**

Sie können hier den Parameter `attr` ignorieren.

Implement the `pthread_create` function call by using the `clone()` system call.

Here, you can ignore the parameter `attr`.

```
enum clone_flags { DUP_DATA = 1, DUP_STACK = 2, DUP_REGS = 4, };
int clone(enum clone_flags flags);
void exit_thread(void);

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
    void *(*start_routine)(void *), void *argument) {

}
```

iii) Implementieren Sie den `pthread_join`-Funktionsaufruf, sodass Sie das Ergebnis des Funktionsaufrufs im Kindsthread erhalten. **2.5 P**

Implement the `pthread_join` function call so that you get the result of the function call in the child thread.

```
void wait_tid(int thread_id);

int pthread_join(pthread_t thread, void **retval) {

----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
}
```

e) Ihr `clone()`-Systemaufruf muss bei manchen Verwendungen große Mengen Speicher kopieren. Nennen Sie die Optimierung, die dieses Kopieren in manchen Fällen vermeiden kann, und beschreiben Sie das Verfahren. **2 P**

Your `clone()` system call sometimes has to copy large amounts of memory. Name the optimization which in some cases can avoid copying, and describe the technique.

f) Führen Sie die folgenden Scheduling-Algorithmen aus und berechnen Sie die durchschnittliche Turnaround-Time des Systems. Zu jedem Zeitschritt kann dabei eine Scheduling-Entscheidung getroffen werden. Neue Prozesse werden vorne in Warteschlangen eingefügt. Sie haben die folgenden Prozesse gegeben: **5 P**

Execute the following scheduling algorithms and calculate the average turnaround time of the system. At each time step a new scheduling decision can be made. New processes are added at the front of waiting queues. The following processes are given:

ID	Arrival Time	Runtime
P1	$t = 0.5$	3
P2	$t = 1.5$	4
P3	$t = 3.5$	2
P4	$t = 4.5$	1

$t =$	1	2	3	4	5	6	7	8	9	10
FCFS										
RR										

Average Turnaround Times

FCFS: _____

_____

RR: _____

_____

Platz für Skizzen / *Sketch space*

1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	9	10

Total: 20 P

Aufgabe 2: Nebenläufigkeit (20 P)

Assignment 2: Concurrency (20 P)

a) Sie haben bereits einige Funktionen für das Threading implementiert. Im Folgenden arbeiten Sie an einer Kompatibilitätsbibliothek, die erlaubt, alten Code auf unterschiedlichen Plattformen auszuführen.

Diese Platformen haben:
- Mehr als einen Prozessorkern
- Unterschiedliche Prozessorfunktionalität
- Unterschiedliche POSIX-kompatibel Betriebssysteme

You already implemented some functions for threading. In the following you are working on a compatibility library which runs old code on various platforms.

These platforms have:
- *More than one processor core*
- *Different processor functionality*
- *Different POSIX-compatible operating systems*

i) Man kann Nebenläufigkeit mit verschiedenen Ansätzen erreichen. Beschreiben Sie kurz, welche Funktionalität ein Betriebssystem nicht bzw. nur eingeschränkt bereitstellt, wenn man auf User Level Threads (ULT) zurückgreifen muss. Welche Threading-Modelle werden von so einem Betriebssystem nicht unterstützt? Geben Sie zusätzlich sowohl einen Vorteil, als auch einen Nachteil oder Problem für ULT Implementierungen an. **2.5 P**

You can achieve concurrency with various means. Briefly describe which functionality of the operating system is missing or limited, so User Level Threads (ULT) must be used. Which threading models are not supported by such operating system? Additionally, give both one advantage and also one disadvantage or problem for ULT implementations.

ULT:

Missing Threading Models:

Pro:

Con:

ii) ULTs können sowohl kooperativ als auch nicht-kooperativ implementiert werden. Beide Implementierungen können sich auf Locks auswirken, die einen kritischen Abschnitt schützen. **2 P**

Wie erfolgt der Wechsel zu einem anderen Thread bei einer kooperativen Implementierung? Welche Funktion muss von der Laufzeitumgebung bereitgestellt werden und was muss diese Funktion machen?

Wie könnten Locks im kooperativen Fall realisiert werden? Sind Locks notwendig? Begründen Sie ihre Antwort!

ULTs can be implemented either as cooperative or non-cooperative. Both implementations can affect how locks guarding a critical section work.

How does the switch to another thread work in a cooperative implementation? What function must be provided by the runtime environment and what must this function do?

How could locks be realized in the cooperative case? Are locks necessary? Justify your answer!

Cooperative: _____

_____

_____

Realization of Locks: _____

_____

_____

iii) Nennen und beschreiben Sie kurz einen Mechanismus, welchen das Betriebssystem oder der Prozessor bereitstellen muss, um nicht-kooperative ULTs zu ermöglichen. **2 P**

Wie könnten Locks (für kritische Abschnitte) im nicht-kooperativen Fall realisiert werden? Sind Locks notwendig? Begründen Sie ihre Antwort!

Name and briefly describe a mechanism the operating system or processor has to provide to enable non-cooperative ULTs.

How could locks (for critical sections) be realized in the non-cooperative case? Are locks necessary? Justify your answer!

Non-cooperative: _____

_____

_____

Realization of Locks: _____

_____

_____

b) Atomare Instruktionen werden gewöhnlich für lock-freie Algorithmen und Spinlocks eingesetzt.

Im Folgenden schreiben Sie weiter an einer Kompatibilitätsbibliothek, die auf verschiedenen Plattformen (unterschiedliche Prozessorarchitekturen und Betriebssystemen) läuft.

Atomic instructions are commonly used to implement lock-free algorithms and spinlocks.

In the following you continue writing a compatibility library that runs on various platforms (different processor architectures and operating systems).

i) Implementieren Sie `AtomicFetchAndAdd()` für eine Plattform, die diese Funktionalität nicht bereitstellt. Sie können die `acquire_lock()` und `release_lock()` verwenden, welche typische Lock-Semantik haben. Minimieren Sie die vom Lock geschützte Region. **3.5 P**

Implement `AtomicFetchAndAdd()` for a platform that does not provide this functionality. You can use the `acquire_lock()` and `release_lock()` which have common lock semantics. Minimize the region protected by the lock.

```
void acquire_lock(lock_t* lock);
void release_lock(lock_t* lock);
// Used for function emulation already initialized
lock_t glLock;

/// Returns the initial value of the variable pointed to by Addend
long AtomicFetchAndAdd(volatile long *Addend, long Value) {
  #ifdef _X86
    // Use LOCK XADD
  #else

-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
  #endif
}
```

ii) Begründen Sie warum ihre Lösung minimal ist. Wieso kann die Region nicht weiter verkleinert werden? **1 P**

Give reasons why your solution is minimal! Why can the region not be reduced?

iii) Nennen Sie zwei weitere atomare Instruktionen, die Lese- und Schreibinstruktionen verbinden. **1 P**

Name two additional atomic instructions which combine load and store instructions.

c) Spinlocks werden häufig zur Realisierung von Locks auf Multiprozessorsystemen eingesetzt. Sie haben bei mobilen Anwendungen einen Nachteil und verhindern das effektive Power-Management. Wie können Spinlocks verbessert werden, damit sich im Idealfall nicht so viel Energie verbrauchen? Nennen Sie eine Verbesserung und beschreiben Sie diese kurz. Welches bekannte Lock kann stattdessen verwendet werden? **2 P**

Spinlocks are commonly used to realize locks on multiprocessor systems. They have a disadvantage for mobile applications and hinder effective power management. How can spinlocks be improved so they do not waste that much energy in the ideal case? Name one optimization and briefly describe it. What known lock could be used instead?

Name of Improved Lock:

d) Nennen Sie zwei Methoden zur Behandlung von Verklemmungssituationen (Deadlocks), und beschreiben Sie kurz ihre Tauglichkeit für den Alltag. Welche dieser Methoden ist am besten für neugeschriebene Programme geeignet und kann proaktiv genutzt werden? Benennen Sie hierfür auch zwei Deadlockbedingungen, die mit dieser Methode gelöst werden können! **2 P**

Name two methods to handle deadlocks and briefly describe their practical suitability. Which method is best suited for newly written programs and can be proactively used? Also name two deadlock conditions which can be solved with this method!

2 Conditions:

e) Implementieren Sie basierend auf `AtomicFetchAndAdd()` oder kurz `AFAA()` einen einfachen Ringpuffer fester Größe für das Logging in ihrer Bibliothek. Ist am Ende des Ringpuffers nicht ausreichend Platz um eine beliebig große Allokation zu erfüllen, muss die Allokation wiederholt werden. Der bereits reservierte Bereich bleibt ungenutzt. **2 P**

Hinweis: `WRAP_BUFFER()` stellt sicher, dass die Eingabe innerhalb der definierten Puffergröße bleibt. Betrachten Sie z.B. den Index 0x10002, welcher die Puffergröße um 3 überschreitet. Nach Anwendung von `WRAP_BUFFER` wird dieser auf 0x00002 reduziert.

Implement based on `AtomicFetchAndAdd()` or short `AFAA()` a simple ring buffer of fixed size for the logging in your library. If there is not enough space at the end of the ring buffer to fullfill the variable-sized allocation, the allocation must be retired. The already reserved region remains unused.

Hint: *`WRAP_BUFFER()` ensures that the input stays within the defined buffer size. Note, e.g., the index 0x10002 which exceeds buffer size by 3. After applying `WRAP_BUFFER` it is reduced to 0x00002.*

```
#define AFAA AtomicFetchAndAdd
#define LOG_BUF_SIZE 0x10000
#define WRAP_BUFFER(x) (((unsigned long)x) % LOG_BUF_SIZE)

char datalog[LOG_BUF_SIZE];
/// Assume long will correctly wrap around on supported platforms
volatile long curpos = 0;

/// This function must always return a valid pointer with enough space for size
/// Uses AtomicFetchAndAdd/AFAA to increment curpos by size
char* reserveLogEntry(uint8_t size) {
  long checkSize, retIdx;

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
  do {

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------


     checkSize = size + WRAP_BUFFER( ----------------------------------------);
  }
  while (checkSize > LOG_BUF_SIZE);


  retIdx = WRAP_BUFFER( ------------------------------------------------------);
  return &datalog[retIdx];
}
```

f) Der Ringpuffer verwendet eine atomare Operation und bedarf, je nach Verwendung, einer Schleife. Wie könnte das Reservieren alternativ gelöst werden, damit die Allokation nicht wiederholt werden muss? **1 P**

Begründen Sie ihre Antwort.

The ring buffer uses one atomic operation and needs, depending on the use case, a loop. How could the reservation be solved alternatively, so the allocation must not be retried?

Give reasons for your answer.

g) Welchen Mechanismus könnten Sie verwenden, um andere Prozesse über einen vollen Ringpuffer zu informieren? **1 P**

Nennen und beschreiben Sie kurz einen Mechanismus.

What mechanism could you use to inform other processes about a full ring buffer?

Name and briefly describe a mechanism.

Total: 20 P

Aufgabe 3: Persistenz (20 P)

Assignment 3: Persistence (20 P)

a) Write-Ahead-Logging ist eine Journaling-Technik, bei der Änderungen an einer Datenbank vor ihrer Ausführung in einen sogenannten Write-Ahead-Log (WAL) geschrieben und auf einen Hintergrundspeicher persistiert werden. Für jede Änderung wird ein weiterer Eintrag hinten am WAL angehängt. In dieser Aufgabe implementieren Sie diese Technik in einem einfachen Key-Value-Store mit nur einem Thread. Nehmen Sie für die folgenden Aufgaben an, dass alle notwendigen Header bereits inkludiert sind und dass Sie keine Fehlerbehandlung implementieren müssen. Der Zustand des Key-Value-Stores wird durch folgende Datenstruktur beschrieben:

Write-ahead logging is a journaling technique that writes updates of a database to a so-called write-ahead log (WAL) before executing them and persists them on a storage device. For each update, an additional entry is appended to the WAL. In this exercise, you will implement this technique in a basic key-value store with only a single thread. Assume for this exercise that all necessary headers are already included and that error handling does not need to be implemented. The state of the key-value store is described using the following data structure:

```
struct kv_ctx {
      int wal_fd;
      // ... in-memory kv representation
};
```

i) Implementieren Sie die Funktion `open_wal`, welche die Datei `wal` im aktuellen Verzeichnis öffnet und den Dateideskriptor in `wal_fd` (Teil von `struct kv_ctx`) speichert. Die geöffnete Datei soll folgende Anforderungen erfüllen: **1.5 P**

- ermöglicht lesenden und schreibenden Zugriff
- schreibende Zugriffe sollen immer am Dateiende angehängt werden (*append*)
- falls die Datei nicht existiert, soll sie angelegt werden (übergeben Sie `0660` als `mode`-Argument)

Implement the function `open_wal`, that opens the file `wal` in the current directory and stores the file descriptor in `wal_fd` (part of `struct kv_ctx`). The opened file should adhere to the following requirements:

- *allows read and write access*
- *writes should always be appended to the end of the file*
- *if the file does not exist, it shall be created (pass `0660` as `mode` argument)*

```
void open_wal(struct kv_ctx *ctx) {

---------------------------------------------------------------------

---------------------------------------------------------------------

---------------------------------------------------------------------

---------------------------------------------------------------------

---------------------------------------------------------------------
}
```

ii) Vervollständigen Sie die Funktion `write_wal_cmd`, welche mittels `dprintf` die durch `cmd` beschriebene Operation in den WAL schreibt. Nutzen Sie die vordefinierten Format-Strings (`PR_UPDATE_FMT` und `PR_DEL_FMT`) und stellen Sie durch geeignete Systemaufrufe sicher, dass in den WAL geschriebene Operationen vor dem Verlassen der Funktion garantiert im Hintergrundspeicher persistiert sind. **1.5 P**

Complete the function `write_wal_cmd`, that uses `dprintf` for writing the operation described by `cmd` to the WAL. Use the predefined format strings (`PR_UPDATE_FMT` and `PR_DEL_FMT`) and ensure by using suitable system calls that operations written to the WAL get persisted to the backing storage before leaving the function.

```
struct cmd {
  enum cmd_op op;
  char key[16]; // alphanumeric C-String
  char value[16]; // alphanumeric C-String
};
```

```
enum cmd_op {
  OP_GET,
  OP_UPDATE,
  OP_DEL,
};
```

```
#define PR_UPDATE_FMT "UPDATE %s %s\n" // UPDATE [KEY] [VALUE]
#define PR_DEL_FMT "DEL %s\n" // DEL [KEY]

void write_wal_cmd(int wal_fd, struct cmd *cmd) {

------------------------------------------------------------------------

------------------------------------------------------------------------
  switch(cmd->op) {
    case OP_UPDATE:
      dprintf(

          ------------------------------------------------------------
          ------------------------------------------------------------
          ------------------------------------------------------------
      );
      break;
    case OP_DEL:
      dprintf(

          ------------------------------------------------------------
          ------------------------------------------------------------
          ------------------------------------------------------------
      );
      break;
    default: break; // only called for writes, i.e., UPDATE and DEL
  }

------------------------------------------------------------------------

------------------------------------------------------------------------
}
```

iii) Eignet sich die *contiguous allocation* Dateiallokationsstrategie für einen WAL auf einem stark fragmentierten Datenträger? Begründen Sie Ihre Antwort. **1 P**

Hinweis: bedenken Sie, dass für jede Änderung am Datenbestand ein neuer Eintrag am WAL angehängt wird.

Is the contiguous file allocation strategy suitable for a WAL on a heavily fragmented storage device? Explain your answer.

Hint*: keep in mind that for each update of the dataset, a new entry is appended to the WAL.*

iv) Um den Datenbestand aus dem WAL wiederherzustellen, wird dieser Zeile für Zeile eingelesen. Vervollständigen Sie dafür die Implementation von `read_line`, welche die nächste Zeile aus `fd` liest und in den übergebenen Puffer `line` kopiert. Ihre Implementation soll dabei wie folgt vorgehen: **3.5 P**

- Lesen Sie die nächsten 64 Bytes aus `fd` in `line` ein. Sie dürfen annehmen, dass `read()` nur weniger Bytes als angefordert liest, wenn das Ende der Datei (*EOF*) erreicht ist, und dass jede Zeile im WAL aus weniger als 64 Zeichen besteht.
- Lokalisieren Sie das Ende der ersten Zeile im Puffer `line` mittels `memchr`.
- Stellen Sie sicher, dass auf **die erste Zeile** in `line` direkt ein terminierendes NULL-byte folgt.
- Setzen Sie abschließend die aktuelle Leseposition mittels `lseek` auf das erste Zeichen, welches nicht zur gelesenen Zeile gehört. Sollte `memchr` keinen Zeilenumbruch in `line` finden, setzen Sie die Leseposition auf das Offset vor dem `read`-Aufruf.

To recover the dataset from the WAL, it is read line by line. For this, complete the implementation of `read_line` that reads the next line from `fd` and copies it into the buffer `line`. Your implementation should work as follows:

- *Read the next 64 bytes from `fd` into `line`. You may assume that `read()` only returns fewer bytes than requested when the end of the file (EOF) is reached, and that each line in the WAL consists of less than 64 characters.*
- *Locate the end of the first line in the buffer `line` using `memchr`.*
- *Ensure that **the first line** in `line` is directly followed by a terminating NULL byte.*
- *Finally, set the current read position using `lseek` to the first character, that does not belong to the line read. If `memchr` does not find a line break in `line`, set the read position to the offset before the call to `read`.*

```c
// maximum length of line in WAL (including line break)
#define WAL_LINE_LEN 64

/* read next complete line (i.e., '\n'-terminated) from fd into supplied buffer
 * including the line break. fd is assumed to support seeking and line must be
 * able to fit at least 65 byte (WAL line + NULL byte).
 *
 * On success, returns NULL-terminated string read into buffer.
 * When reaching EOF before next '\n', returns NULL. */
char *read_line(int fd, char line[WAL_LINE_LEN + 1]) {

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ssize_t bytes = read( ------------------------------------------------ );

  if (bytes ------------------------------------------------------------ )
    return NULL; // reached EOF, no line read

  char *end = memchr( -------------------------------------------------- );
  if (end != NULL) {
    // length of line including new line
    ssize_t len = (end - line) + 1;

    ----------------------------------------------------------------------

    ----------------------------------------------------------------------

    ----------------------------------------------------------------------

    lseek( -------------------------------------------------------------- );
    return line; // return first read line as NULL-terminated string
  }

  // no line break found -> partial line

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  ----------------------------------------------------------------------

  lseek( -------------------------------------------------------------- );
  return NULL; // don't return partial line
}
```

b) Bei einem Stromausfall gehen alle Änderung am Dateisystem, welche in volatilem Speicher gepuffert sind, verloren. Wenn nicht vorsichtig vorgegangen wird, könnte dabei das Dateisystem in einem inkonsistenten Zustand hinterlassen werden. Geben Sie ein Beispiel für einen inkonsistenten Zustand an, welcher beim Verschieben einer Datei auftreten kann, und beschreiben Sie, was passieren muss, dass sich dieser inkonsistente Zustand manifestiert. Nennen Sie zudem den Namen des Werkzeugs, das typischerweise für die Erkennung von inkonsistenten Zuständen verwendet wird, nachdem sie bereits aufgetreten sind. **2 P**

On power loss, all file system updates buffered in volatile memory are lost. When no proper care is taken, this might leave the file system in an inconsistent state. Give one example of an inconsistent state that might occur while moving a file and describe what needs to happen for this inconsistent state to manifest itself. Further, name the tool that is typically used for detecting inconsistent states after they have already occurred.

Inkonsistenz/*inconsistency*:

Erkennung/*detection*:

c) Erklären Sie, warum sequentielles Lesen auf Festplatten erheblich schneller als wahlfreies Lesen ist. Beschreiben Sie weiter, wie sich das ein Betriebssystem zu Nutze machen kann, um den I/O-Durchsatz zu erhöhen. **1.5 P**

Explain why reading form hard drives sequentially is significantly faster than random reads. Further, describe how an operating system can make use of this to increase the I/O throughput.

d) Nehmen Sie an, dass das folgende Shell-Script in einem leeren Verzeichnis ausgeführt wird. Beschreiben Sie die Dateiinhalte nach der Ausführung des Scripts, indem Sie Tabelle 2 ausfüllen. Wenn nicht auf eine Datei zugegriffen werden kann, schreiben sie X in den entsprechenden Tabelleneintrag. Tabelle 1 dient als Beispiel und zeigt die Dateiinhalte, nachdem das Script bis zu Zeile 5 ausgeführt wurde. **2.5 P**

Hinweis: der Befehl `rm` löscht die angegebene Datei ohne dabei Links zu folgen.

Assume that the following shell script is executed in an empty directory. Describe the file contents after running the script by filling out table 2. If a file cannot be accessed, write X into the corresponding table entry. Table 1 serves as an example and shows the file contents after executing the script up until line 5.

Hint*: the `rm` command deletes the given file without following links.*

```
1   # >>: append to file
2   # -n: no newline
3   echo -n A >> a
4   echo -n B >> b
5   # ln [TARGET] [LINK_NAME]
6   # -s: symbolic link
7   ln -s a c
8   ln -s b d
9   ln b e
10  echo -n 1 >> d
11  rm c
12  rm b
13  echo -n 2 >> e
14  echo -n 3 >> c
```

a	**b**	**c**	**d**	**e**
A	B	X	X	X

Table 1: Zustand nach Zeile 4 / *state after line 4*

a	**b**	**c**	**d**	**e**

Table 2: Zustand nach Skript / *state after script*

e) Beschreiben Sie den Unterschied zwischen *mandatory file locks* (verpflichtend) und *advisory file locks* (empfehlend/beratend). **1 P**

Describe the difference between mandatory file locks and advisory file locks.

f) **UNIX Access Rights**

i) Welche Zugriffsrechte werden durch `rwxr-xr--` im klassischen UNIX-Zugriffsmodell beschreiben? Vervollständigen Sie die folgende Tabelle für jedes Subjekt (ja/✓ für gewährte Berechtigungen, nein/✗ für fehlende Berechtigungen). **1.5 P**

Which access rights are described by `rwxr-xr--` in the traditional UNIX access model? Complete the following table for each subject (yes/✓ for granted permissions, no/✗ for missing permissions).

Subjekt/*Subject*	**Read**	**Write**	**Execute**

ii) Welchen Zweck erfüllt die Execute-Berechtigung auf Verzeichnissen? **0.5 P**

What purpose does the execute permission serve on directories?

iii) Klassische UNIX-Zugriffsrechte bieten aufgrund der begrenzten Anzahl an Subjekte nicht immer die benötigte Flexibilität. Nennen Sie ein anderes UNIX-Konzept, was hier Abhilfe schaffen kann, und beschreiben Sie kurz, wie es funktioniert. **1 P**

Traditional UNIX access rights sometimes fail to offer enough flexibility due to the limited number of subjects. Name a different UNIX concept that can help here, and describe briefly how it works.

g) Nehmen Sie an, Sie verwenden einen Festplattenverbund aus fünf Festplatten. Wie viele Festplatten können in den folgenden RAID-Leveln ausfallen, **bevor** Daten unwiederbringlich verloren gehen? Begründen Sie Ihre Antwort für jedes RAID-Level. **2.5 P**

Geben Sie zudem an, wofür das Akronym RAID steht.

*Assume that you use a hard drive array consisting of five hard drives. How many hard drives can fail **before** data is lost irrecoverably? Justify your answer for each RAID level.*

Further, state what the acronym RAID stands for.

Level	**#Ausfälle/#*Failures***	**Erklärung/*Explanation***
RAID 0		
RAID 5		

Akronym RAID / *Acronym RAID* : _____

Total: 20 P

NAME

malloc, free – allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

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

DESCRIPTION

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

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

RETURN VALUE

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

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

ERRORS

malloc() can fail with the following error:

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

NOTES

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

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

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

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

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

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

NAME

errno – number of last error

SYNOPSIS

#include <errno.h>

DESCRIPTION

The *<errno.h>* header file defines the integer variable *errno*, which is set by system calls and some library functions in the event of an error to indicate what went wrong. Its value is significant only when the return value of the call indicated an error (i.e., −1 from most system calls; −1 or NULL from most library functions); a function that succeeds *is* allowed to change *errno*.

Valid error numbers are all nonzero; *errno* is never set to zero by any system call or library function.

For some system calls and library functions (e.g., **getpriority**(2)), −1 is a valid return on success. In such cases, a successful return can be distinguished from an error return by setting *errno* to zero before the call, and then, if the call returns a status that indicates that an error may have occurred, checking to see if *errno* has a nonzero value.

errno is defined by the ISO C standard to be a modifiable lvalue of type *int*, and must not be explicitly declared; *errno* may be a macro. *errno* is thread-local; setting it in one thread does not affect its value in any other thread.

All the error names specified by POSIX.1 must have distinct values, with the exception of **EAGAIN** and **EWOULDBLOCK**, which may be the same.

Below is a list of the symbolic error names that are defined on Linux. Some of these are marked *POSIX.1*, indicating that the name is defined by POSIX.1-2001, or *C99*, indicating that the name is defined by C99.

EAGAIN	Resource temporarily unavailable (POSIX.1).
EPERM	Operation not permitted (POSIX.1).
EINVAL	Invalid argument (POSIX.1).
ENOMEM	Not enough space (POSIX.1).
ENOENT	No such file or directory (POSIX.1).
ENOTDIR	Not a directory (POSIX.1).
EDEADLK	Resource deadlock avoided (POSIX.1).
ESRCH	No such process (POSIX.1).

SEE ALSO

err(3), **error**(3), **perror**(3), **strerror**(3)

COLOPHON

This page is part of release 3.54 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man−pages/.

NAME

pthread_create – create a new thread

SYNOPSIS

#include <pthread.h>

int pthread_create(pthread_t ***thread*, const pthread_attr_t** ***attr*,**
void ***(***start_routine*) **(void** ***), void** ***arg*);**

DESCRIPTION

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

The new thread terminates in one of the following ways:

* It returns from *start_routine*(). The return value of *start_routine*(), also called the exit status, can then be queried in **pthread_join**(3).

* Any of the threads in the process calls **exit**(3), or the main thread performs a return from *main*(). This causes the termination of all threads in the process.

The *attr* argument points to a *pthread_attr_t* structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using **pthread_attr_init**(3) and related functions. If *attr* is NULL, then the thread is created with default attributes.

Before returning, a successful call to **pthread_create**() stores the ID of the new thread in the buffer pointed to by *thread*; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

RETURN VALUE

On success, **pthread_create**() returns 0; on error, it returns an error number, and the contents of **thread* are undefined.

ERRORS

EAGAIN
Insufficient resources to create another thread.

EINVAL
Invalid settings in *attr*.

EPERM
No permission to set the scheduling policy and parameters specified in *attr*.

NAME

pthread_join – join with a terminated thread

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t *thread*, **void** ***retval*);**

Compile and link with *–pthread*.

DESCRIPTION

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately.

If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread returned) into the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined.

RETURN VALUE

On success, **pthread_join**() returns 0; on error, it returns an error number.

ERRORS

EDEADLK
A deadlock was detected. (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL
Another thread is already waiting to join with this thread.

ESRCH
No thread *thread* could be found.

NOTES

After a successful call to **pthread_join**(), the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

NAME

open, creat – open and possibly create a file

SYNOPSIS

#include <fcntl.h>

int open(const char ****path*, int** *flags***, ...**
    /* mode_t *mode* ***/);**

int creat(const char ****path*, mode_t** *mode***);**

DESCRIPTION

The **open**() system call opens the file specified by *path*. 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 an index to an entry in the process's table of open file descriptors. The file descriptor 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.

By default, the new file descriptor is set to remain open across an **execve**(2). The file offset is set to the beginning of the file (see **lseek**(2)).

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 *path* is subsequently removed or modified to refer to a different file.

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 ORed in *flags*. The *file creation flags* are **O_CREAT**, **O_EXCL**, 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.

The full 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_CREAT
: If *path* 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 group ownership (group ID) of the new file is set to the effective group ID of the process (System V semantics).

  The *mode* argument specifies the file mode bits to be applied when a new file is created. If neither **O_CREAT** nor **O_TMPFILE** is specified in *flags*, then *mode* is ignored (and can thus be specified as 0, or simply omitted). The *mode* argument **must** be supplied if **O_CREAT** or **O_TMPFILE** is specified in *flags*; if it is not supplied, some arbitrary bytes from the stack will be applied as the file mode.

  Note that *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.

O_DIRECT (since Linux 2.4.10)
: Try to minimize cache effects of the I/O to and from this file. In general this will degrade performance, but it is useful in special situations, such as when applications do their own caching. File I/O is done directly to/from user-space buffers. The **O_DIRECT** flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the **O_SYNC** flag that data and necessary metadata are transferred. To guarantee synchronous I/O, **O_SYNC** must be used in addition to **O_DIRECT**.

O_DSYNC
: Write operations on the file will complete according to the requirements of synchronized I/O *data* integrity completion.

  By the time **write**(2) (and similar) return, the output data has been transferred to the underlying hardware, along with any file metadata that would be required to retrieve that data (i.e., as though each **write**(2) was followed by a call to **fdatasync**(2)). See VERSIONS.

O_EXCL
: Ensure that this call creates the file: if this flag is specified in conjunction with **O_CREAT**, and *path* already exists, then **open**() fails with the error **EEXIST**.

O_SYNC
: Write operations on the file will complete according to the requirements of synchronized I/O *file* integrity completion (by contrast with the synchronized I/O *data* integrity completion provided by **O_DSYNC**.)

  By the time **write**(2) (or similar) returns, the output data and associated file metadata have been transferred to the underlying hardware (i.e., as though each **write**(2) was followed by a call to **fsync**(2)).

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.

creat()
: A call to **creat**() is equivalent to calling **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

RETURN VALUE

On success, **open**(), **openat**(), and **creat**() return the new file descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

open() and **creat**() can fail with the following errors:

EEXIST
: *path* already exists and **O_CREAT** and **O_EXCL** were used.

VERSIONS

Synchronized I/O

O_SYNC provides synchronized I/O *file* integrity completion, meaning write operations will flush data and all associated metadata to the underlying hardware. **O_DSYNC** provides synchronized I/O *data* integrity completion, meaning write operations will flush data to the underlying hardware, but will only flush metadata updates that are required to allow a subsequent read operation to complete successfully. Data integrity completion can reduce the number of disk operations that are required for applications that don't need the guarantees of file integrity completion.

To understand the difference between the two types of completion, consider two pieces of file metadata: the file last modification timestamp (*st_mtime*) and the file length. All write operations will update the last file modification timestamp, but only writes that add data to the end of the file will change the file length. The last modification timestamp is not needed to ensure that a read completes successfully, but the file length is. Thus, **O_DSYNC** would only guarantee to flush updates to the file length metadata (whereas **O_SYNC** would also always flush the last modification timestamp metadata).

SEE ALSO

close(2), **lseek**(2), **read**(2), **write**(2)

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

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

lseek — move the read/write file offset

SYNOPSIS

#include <unistd.h>

off_t lseek(int *fildes*, off_t *offset*, int *whence*);

DESCRIPTION

The *lseek*() function shall set the file offset for the open file description associated with the file descriptor *fildes,* as follows:

* If *whence* is SEEK_SET, the file offset shall be set to *offset* bytes.
* If *whence* is SEEK_CUR, the file offset shall be set to its current location plus *offset*.
* If *whence* is SEEK_END, the file offset shall be set to the size of the file plus *offset*.

The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <*unistd.h*>.

The behavior of *lseek*() on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The *lseek*() function shall allow the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The *lseek*() function shall not, by itself, extend the size of a file.

If *fildes* refers to a shared memory object, the result of the *lseek*() function is unspecified.

If *fildes* refers to a typed memory object, the result of the *lseek*() function is unspecified.

RETURN VALUE

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and the file offset shall remain unchanged.

ERRORS

The *lseek*() function shall fail if:

EBADF
The *fildes* argument is not an open file descriptor.

EINVAL
The *whence* argument is not a proper value, or the resulting file offset would be negative for a regular file, block special file, or directory.

EOVERFLOW
The resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.

ESPIPE
The *fildes* argument is associated with a pipe, FIFO, or socket.

NAME

fsync, fdatasync − synchronize a file's in-core state with storage device

sync, syncfs − commit filesystem caches to disk

SYNOPSIS

#include <unistd.h>

int fsync(int *fd***);**

int fdatasync(int *fd***);**

void sync(void);

int syncfs(int *fd***);**

DESCRIPTION

fsync() transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor *fd* to the disk device (or other permanent storage device) so that all changed information can be retrieved even if the system crashes or is rebooted. This includes writing through or flushing a disk cache if present. The call blocks until the device reports that the transfer has completed.

As well as flushing the file data, **fsync**() also flushes the metadata information associated with the file.

Calling **fsync**() does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit **fsync**() on a file descriptor for the directory is also needed.

fdatasync() is similar to **fsync**(), but does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled. For example, changes to *st_atime* or *st_mtime* (respectively, time of last access and time of last modification) do not require flushing because they are not necessary for a subsequent data read to be handled correctly. On the other hand, a change to the file size (*st_size*, as made by say **ftruncate**(2)), would require a metadata flush.

The aim of **fdatasync**() is to reduce disk activity for applications that do not require all metadata to be synchronized with the disk.

sync() causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystems.

syncfs() is like **sync**(), but synchronizes just the filesystem containing file referred to by the open file descriptor *fd*.

RETURN VALUE

On success, these system calls return zero. On error, −1 is returned, and *errno* is set to indicate the error. **sync**() is always successful.

ERRORS

EBADF
fd is not a valid open file descriptor.

EIO An error occurred during synchronization. This error may relate to data written to some other file descriptor on the same file. Since Linux 4.13, errors from write-back will be reported to all file descriptors that might have written the data which triggered the error. Some filesystems (e.g., NFS) keep close track of which data came through which file descriptor, and give more precise reporting. Other filesystems (e.g., most local filesystems) will report errors to all file descriptors that were open on the file when the error was recorded.

ENOSPC
Disk space was exhausted while synchronizing.

EROFS
EINVAL
fd is bound to a special file (e.g., a pipe, FIFO, or socket) which does not support synchronization.

SEE ALSO

open(2)

NAME

printf, dprintf – formatted output conversion

SYNOPSIS

#include <stdio.h>

int printf(const char * *format*, **...);**
int fprintf(FILE * *stream*, **const char *** *format*, **...);**
int sprintf(char * *str*, **const char *** *format*, **...);**
int dprintf(int *fd*, **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*; and **sprintf**() write to the character string *str*.

The function **dprintf**() is the same as **fprintf**() except that it outputs to a file descriptor, *fd*, instead of to a **stdio**(3) stream.

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

C99 and POSIX.1-2001 specify that the results are undefined if a call to **sprintf**() would cause copying to take place between objects that overlap (e.g., if the target string array and one of the supplied input arguments refer to the same buffer).

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*.

The overall syntax of a conversion specification is:

%[flags][width][.precision][length modifier]conversion

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. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

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'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

p The *void* * pointer argument is printed in hexadecimal (as if by **%#x** or **%#lx**).

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

RETURN VALUE

Upon successful return, these functions return the number of bytes printed (excluding the null byte used to end output to strings).

On error, a negative value is returned, and *errno* is set to indicate the error.

NAME

memchr, memrchr – scan memory for a character

SYNOPSIS

#include <string.h>

void *memchr(size_t n;
const void *s*[*n*], **int** *c*, **size_t** *n*);
void *memrchr(size_t n;
const void *s*[*n*], **int** *c*, **size_t** *n*);

DESCRIPTION

The **memchr**() function scans the initial *n* bytes of the memory area pointed to by *s* for the first instance of *c*. Both *c* and the bytes of the memory area pointed to by *s* are interpreted as *unsigned char*.

The **memrchr**() function is like the **memchr**() function, except that it searches backward from the end of the *n* bytes pointed to by *s* instead of forward from the beginning.

RETURN VALUE

The **memchr**() and **memrchr**() functions return a pointer to the matching byte or NULL if the character does not occur in the given memory area.

NAME

memcpy – copy memory area

SYNOPSIS

#include <string.h>

void *memcpy(void * *dest*, **const void *** *src*, **size_t** *n*);

DESCRIPTION

The **memcpy**() function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas must not overlap. Use **memmove**(3) if the memory areas do overlap.

RETURN VALUE

The **memcpy**() function returns a pointer to *dest*.

NOTES

Failure to observe the requirement that the memory areas do not overlap has been the source of significant bugs. (POSIX and the C standards are explicit that employing **memcpy**() with overlapping areas produces undefined behavior.) Most notably, in glibc 2.13 a performance optimization of **memcpy**() on some platforms (including x86-64) included changing the order in which bytes were copied from *src* to *dest*.

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*.