Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Hans P. Reiser
Mathias Gottschlag, M.Sc.
Dipl.-Inform. Marc Rittinghaus

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

Scheinklausur

22.03.2018

- Bitte tragen Sie zuerst auf dem Deckblatt Ihren Namen, Ihren Vornamen und Ihre Matrikelnummer ein. Tragen Sie dann auf den anderen Blättern (auch auf Konzeptblättern) Ihre Matrikelnummer ein.
  Please fill in your last name, your first name, and your matriculation number on this page and fill in your matriculation number on all other pages (including draft pages).
- Die Prüfung besteht aus 23 Blättern: 1 Deckblatt, 17 Aufgabenblättern mit insgesamt 3 Aufgaben und 5 Blättern Man-Pages.
  The examination consists of 23 pages: 1 cover sheet, 17 sheets containing 3 assignments, and 5 sheets for man pages.
- Es sind keinerlei Hilfsmittel erlaubt!
  No additional material is allowed.
- Die Prüfung gilt als nicht bestanden, wenn Sie versuchen, aktiv oder passiv zu betrügen.
  You fail the examination if you try to cheat actively or passively.
- 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	20	20	20	60
Erreichte Punkte				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Betrachten Sie die folgende Funktion `read_input()`. Nehmen Sie an, dass die Datei, auf die der Filedeskriptor `fd` verweist, die folgende Folge von Zeichen enthält: a, b, c, d.

Consider the following function `read_input()`. Assume that the file which the descriptor `fd` refers to contains the following sequence of characters: a, b, c, d.

```
void read_input(int fd, char **buf) {
  *buf = malloc(1024);
  memset(*buf, 0, 1024);
  lseek(fd, 3, SEEK_SET);
  read(fd, *buf, 1023);
}
```

Welchen String enthält der Buffer `buf`, nachdem der `read()`-Aufruf zurückgekehrt ist? Nehmen Sie an, dass keine Fehler auftreten. **1 pt**

Which string does the buffer `buf` contain after the `read()` call returns? Assume that no errors occur.

Was ist der Zweck des `memset()`-Aufrufs? **2 pt**

What is the purpose of the `memset()` call?

Nehmen Sie an, mehrere Threads rufen gleichzeitig `read_input()` auf. Dabei übergeben alle Threads denselben Wert für `fd`, aber nicht für `buf`. Welches Problem kann in diesem Fall auftreten? **2 pt**

Assume that multiple threads simultaneously call `read_input()`, passing the same value for `fd` but not for `buf`. Which problem can occur in that case?

b) Ein Programm verwendet die folgende Funktion, um zu verhindern, dass geheime Daten aus freigegebenen Puffern im Speicher zurückbleiben. Obwohl nach dem `free()`-Aufruf noch einmal auf den Puffer `buf` zugegriffen wird, funktioniert die Funktion zunächst wie gewünscht. Erklären Sie, warum. **1 pt**

A program uses the following function to ensure that secret data from freed buffers does not remain in memory. Even though the buffer `buf` is accessed after the `free()` call, the function works as intended. Explain why.

```
void secure_free(void *buf, size_t size) {
  free(buf);
  memset(buf, 0, size);
}
```

Nachdem das Programm um mehrere Threads erweitert wird, treten gelegentlich Abstürze auf. Welches Problem verursacht wahrscheinlich diese Abstürze? **1 pt**

After the program is modified to use multiple threads, the program starts to crash occasionally. Which problem is probably causing these crashes?

c) Wieviele neue Prozesse startet die folgende Funktion? **1 pt**

How many new processes does the following function start?

```
void start_processes() {
  fork();
  fork();
  fork();
  return;
}
```

d) `A` sei ein Array. Was bedeutet der folgende Ausdruck? **1 pt**

Let `A` be an array. What is the meaning of the following statement?

```
&A[42]
```

e) Ist es in C problemlos möglich, einen Zeiger nach `int` zu casten? Begründen Sie Ihre Antwort. **1.5 pt**

In C, is it possible to cast a pointer to `int` without causing problems? Justify your answer.

f) In dieser Aufgabe sollen Sie eine einfache Hashtabelle implementieren.

- Die Tabelle verwendet den Datentyp `int` sowohl für Schlüssel (`key`) als auch für Werte (`value`).
- Die Position jedes Elements im Array entspricht dem Schlüssel des Elements modulo der Anzahl der Felder des Arrays (`LIST_SIZE`).
- Für den Fall, dass sich beim Einfügen eines Elements bereits ein anderes Element an der Zielposition befindet, verwendet die Tabelle *linear probing*: Beginnend mit der Zielposition werden alle Einträge des Arrays linear durchsucht, bis ein freier Eintrag gefunden wird.

In this problem, you are to implement a simple hash table.

- *The table uses the data type `int` for both keys and values.*
- *The position of each element in the array is the element's key modulo the number of slots in the array (`LIST_SIZE`).*
- *If an element is to be inserted into a slot that already contains a valid element, the table uses* linear probing: *Starting with the calculated slot, all slots in the array are linearly searched until a free slot is found.*

Definieren Sie zunächst eine Datenstruktur, die ein einzelnes Element der Tabelle repräsentiert. **1.5 pt**

First, define a data structure representing a single table element.

```
struct element {




};
```

Vervollständigen Sie die Funktion `insert()`, die ein Element in die Tabelle einfügt. **3.5 pt**

- Nehmen Sie an, dass freie Einträge gemäß Ihrer Definition von `struct element` korrekt initialisiert sind, dass noch kein Eintrag mit dem selben Schlüssel existiert und dass mindestens ein freier Eintrag existiert.

Complete the function `insert()` which inserts an element into the table.

- *Assume that unused entries are initialized correctly according to your definition of `struct element`, that no entry with the same key exists and that at least one unused entry exists.*

```
#define TABLE_SIZE 1024
struct element table[TABLE_SIZE];

void insert(int key, int value) {
















}
```

Vervollständigen Sie die Funktion `find()`, die ein Element mit einem bestimmten Schlüssel aus der Tabelle liest. **4.5 pt**

- Der Rückgabewert der Funktion soll 1 sein, wenn ein entsprechendes Element gefunden wurde, und 0 falls nicht.
- Falls ein Element gefunden wurde, soll die Funktion seinen Wert im Parameter `value` ablegen.
- Ihre Implementierung soll so wenige Arrayelemente wie möglich durchsuchen.

Complete the function `find()` which reads an element with a given key from the table.

- *The function shall return 1 if an element with the given key was found and 0 if not.*
- *If an element was found, the function shall store the element's value into the parameter `value`.*
- *Your implementation shall search as few array elements as possible.*

```
int find(int key, int *value) {

}
```

Total: 20.0pt

Aufgabe 2: Dateisystemgröße

Assignment 2: File System Size

Schreiben Sie ein Programm, das rekursiv den Platzverbrauch aller Dateien in einem Verzeichnis und seinen Unterordnern auf der Festplatte bestimmt.

- Das Programm soll Hardlinks anhand der I-Node-Nummer erkennen, sodass der Platzverbrauch von Dateien nur einmal pro I-Node gezählt wird.
- Das Programm wird mithilfe einer festen Anzahl von Threads parallelisiert, indem Unterverzeichnisse in eine zentrale Warteschlange eingetragen werden, die von den Threads abgearbeitet wird. Verwenden Sie, wo notwendig, Mutexe, um gleichzeitigen Zugriff auf Daten zu synchronisieren.
- Geben Sie alle vom Betriebssystem angeforderte Ressourcen (z. B. Speicher) explizit zurück.
- Binden Sie die in den Teilaufgaben notwendigen C-Header in dem gekennzeichneten Bereich ein.
- Soweit nicht anderweitig bestimmt, gehen Sie davon aus, dass bei Systemaufrufen und Funktionen der Standardbibliothek keine Fehler auftreten.

Write a program which recursively determines the disk space consumption of all files in a directory and all its subdirectories.

- *The program shall recognize hard links by their i-node number, so that the space consumption of files is only counted once per i-node.*
- *The program is parallelized with a fixed number of threads, and subdirectories are inserted into a central queue from which they are processed by the threads. Use mutexes where necessary to synchronize concurrent data accesses.*
- *Explicitly return all allocated operating system resources (e. g., memory).*
- *Include necessary C headers in the marked area.*
- *Unless stated otherwise, assume that system calls and standard library functions do not fail.*

```
/* global variables */
pthread_mutex_t lock;
unsigned long long total = 0; /* total file size */

/* include statements for the required C headers */
```

....

....

....

....

....

....

....

....

....

....

a) Vervollständigen Sie die Funktion `listDir()`, die die Einträge eines Verzeichnisses auflistet, für jeden Eintrag die Funktion `processEntry()` aufruft und ihr jeweils den Verzeichnispfad (`path`-Parameter von `listDir()`) und den Verzeichniseintrag übergibt. **5.5 pt**

- Behandeln Sie Fehler sämtlicher Systemaufrufe, indem Sie mit `exit(-1)` das Programm beenden.

Complete the function `listDir()` which lists the entries of a directory and which calls `processEntry()` for each entry, each time passing the directory path (`path` parameter of `listDir()`) and the directory entry to it.

- *Handle errors from all system calls by terminating the program with `exit(-1)`.*

```
void processEntry(const char *dir, struct dirent *e);

void listDir(const char *path) {





















}
```

b) Vervollständigen Sie die Funktion `makePath()`, die einen Pfad aus einem Ordnerpfad und einem Dateinamen zusammensetzt. So soll `makePath("/home/student", "exam.txt")` zum Beispiel den String `"/home/student/exam.txt"` zurückgeben. **3.5 pt**

Complete the function `makePath()` which joins a directory path and a file name to get a path to the file. For example, `makePath("/home/student", "exam.txt")` shall return the string `"/home/student/exam.txt"`.

```
char *makePath(const char *dir, const char *name) {












}
```

c) Vervollständigen Sie die Funktion `processEntry()`, die für den übergebenen Verzeichniseintrag je nach Typ die passende Funktion zum Verarbeiten des Eintrags aufruft. **3 pt**

- Für Dateien soll `processFile()` aufgerufen werden, um die Dateigröße zu bestimmen.
- Für Verzeichnisse soll `addDirectory()` aufgerufen werden. Die Funktion löst eine rekursive Auflistung des Unterverzeichnisses aus.
- Beide Funktionen erhalten als Argument den vollständigen Pfad des Verzeichniseintrags.
- Ignorieren Sie die Verzeichniseinträge "." und "..".

Complete the function `processEntry()` which, for the directory entry passed to it, calls the appropriate function to process the entry depending on the type.

- *For files, `processFile()` shall be called to determine the file size.*
- *For directories, `addDirectory()` shall be called to trigger a recursive listing of the subdirectory.*
- *Both functions are passed the full path to the directory entry.*
- *Ignore the directory entries "." und "..".*

```
/* triggers recursive listing of the specified subdirectory */
void addDirectory(const char *dirpath);
/* determines the size of the specified file */
void processFile(const char *path);
/* creates a full path of a directory entry;
 * returns a dynamically allocated buffer containing the path */
char *makePath(const char *dirpath, const char *name);

void processEntry(const char *dirpath, struct dirent *e) {
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
}
...............................................................................
```

d) Vervollständigen Sie die Funktion `processFile()`, die die Größe der Datei an dem übergebenen Pfad bestimmt und auf die Gesamtgröße der betrachteten Dateien aufsummiert. **4.5 pt**

- Die Funktion wird parallel aus verschiedenen Threads des Thread-Pools aufgerufen. Achten Sie auf korrekte Synchronisierung.
- Ignorieren Sie die Datei, falls ein anderer Hardlink auf den selben I-Node bereits betrachtet wurde. Verwenden Sie hierfür die gegebene globale Hashtabelle, um eine Liste bereits betrachteter I-Nodes zu halten.
- Verwenden Sie die Funktion `contains()`, die in dieser Tabelle prüft, ob ein bestimmter I-Node enthalten ist. Ist der I-Node enthalten, gibt die Funktion 1 und ansonsten 0 zurück.
- Verwenden Sie die Funktion `insert()`, um I-Nodes in die Tabelle einzutragen.
- Beide Funktionen führen keine interne Synchronisierung von Zugriffen auf die Tabelle durch.

Complete the function `processFile()` which determines the size of the file at the specified path and adds it to the total size of all processed files.

- *The function is called in parallel from multiple threads of the thread pool. Ensure proper synchronization.*
- *Ignore the file if another hard link to the same i-node has already been processed. Use the given global hash table to maintain a list of i-nodes which have already processed.*
- *Use the function `contains()` to check whether this table contains the specified i-node. If the i-node is present, the function returns 1, else it returns 0.*
- *Use the function `insert()` to insert i-nodes into the table.*
- *Both functions do not internally synchronize accesses to the table.*

```
/* inserts the inode into the table of processed files */
void insert(ino_t inode);
/* checks whether the inode has already been processed */
int contains(ino_t inode);

void processFile(const char *path) {















}
```

e) Vervollständigen Sie die `main()`-Funktion des Programmes. **3.5 pt**

- Initialisieren Sie globale Variablen und starten Sie 10 Threads, die als Teil des oben genannten Thread-Pools jeweils die Funktion `doWork()` ausführen.
- `addDirectory()` übergibt einen Pfad zur rekursiven Gesamtgrößenberechnung an den Thread-Pool und kehrt dann sofort zurück. Die Threads des Thread-Pools beenden sich, sobald die Berechnung abgeschlossen ist.
- Geben Sie am Funktionsende verbliebene Resourcen frei.

Complete the `main()` function of the program.

- *Initialize global variables and start 10 threads which all execute the function `doWork()` as part of the thread pool mentioned above.*
- *`addDirectory()` passes a path to the thread pool to recursively determine its total size. The function then returns immediately. The threads of the thread pool terminate once the computation of the size is complete.*
- *At the end of the function, release any remaining resources.*

```
void *doWork(void *param);
int main(int argc, char **argv) {
    pthread_t threads[10];
    /* initialization */









    /* start scanning the directory */
    addDirectory(argv[1]);









    /* print the result */
    printf("Size:␣%lld␣bytes\n", total);
}
```

Total: 20.0pt

Aufgabe 3: Speicherdeduplikation

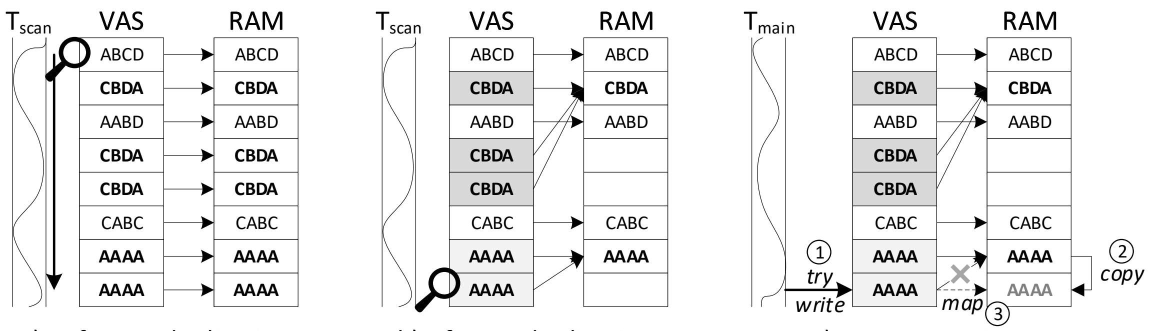
Assignment 3: Memory Deduplication

Um physischen Speicher einzusparen, sollen in einem virtuellen Adressraum (VAS) die Seitentabelleneinträge (PTEs) derart verändert werden, dass Seiten mit identischem Inhalt auf die gleiche physische Seite (PFN) zeigen. Ein (nicht optimales) Verfahren identifiziert Duplikate, indem die Seiteninhalte gehasht und über eine Hashtabelle abgeglichen werden. Das Verfahren verwendet Copy-On-Write (CoW), um bei schreibendem Zugriff deduplizierte Seiten wieder aufzubrechen.

- Die Seitengröße beträgt 4 KiB.
- Adressräume enthalten keine schreibgeschützten Bereiche wie `.rodata`.
- Das OS gibt nicht referenzierte physische Seiten automatisch frei.
- Die Suche läuft parallel zum Hauptthread T_{main} in einem eigenen Thread T_{scan}.

To save physical memory the page table entries (PTEs) in a virtual address space (VAS) should be adjusted so that pages with identical contents point to the same physical page (PFN). A best-effort approach identifies duplicates by hashing the pages' contents and matching them with a hash table. The approach uses copy-on-write (CoW) for breaking deduplicated pages on write access.

- *The page size is 4 KiB.*
- *Address spaces do not contain read-only areas such as `.rodata`.*
- *The OS frees not referenced physical pages automatically.*
- *The search runs in parallel to the main thread T_{main} in a dedicated thread T_{scan}.*



a) Before Deduplication   b) After Deduplication   c) Copy-On-Write

```c
/* PTE: Bits[12:31] = PFN, Bits[4:11] = unused, Bits[0:3] = flags      */
#define PTE_P_MASK   0x1 /* Present - if set, PTE is valid mapping     */
#define PTE_RO_MASK  0x2 /* Read-only - if set, page fault on write    */
#define PTE_COW_MASK 0x4 /* CoW - if set, CoW enabled (ignored by MMU) */

/* Returns or sets PTE for a given virtual address */
uint32_t getPte(uint32_t va);
void setPte(uint32_t va, uint32_t pte);

/* Locks or unlocks the virtual address space.
 * Calling lockVas() will block a thread if the VAS is already locked */
void lockVas(void);
void unlockVas(void);

#define INV_PFN ((uint32_t)-1) /* Invalid physical frame number */

/* Returns a pointer to the contents of a physical page, NULL for INV_PFN */
uint32_t *getPage(uint32_t pfn);
```

a) Vervollständigen Sie die Funktion `hash()`, die für eine gegebene physische Seite `pfn` den 32-bit djb2-Hash $h(i) = h(i-1) * 33 + p[i]$ zurückgibt, wobei $p[i]$ das i-te 32-bit Wort der Seite ist und $h(1023)$ den Hashwert der Seite enthält. **3.0 pt**

- Wählen Sie $h(-1) = 5381$.
- Verwenden Sie zur Implementierung der Hashfunktion keine Multiplikation.
- `pfn` ist eine valide physische Seite (`pfn` $\neq$ `INV_PFN`).

Complete the function `hash()`, which returns for a given physical page `pfn` the 32-bit djb2-hash $h(i) = h(i-1) * 33 + p[i]$, where $p[i]$ is the i-th 32-bit word in the page and $h(1023)$ is the hash of the page.

- *Choose $h(-1) = 5381$.*
- *Do not use multiplication for implementing the hash function.*
- *`pfn` is a valid physical page (`pfn` $\neq$ `INV_PFN`).*

```
uint32_t *getPage(uint32_t pfn);

uint32_t hash(uint32_t pfn) {











}
```

b) Wo sehen Sie potentiell Probleme bei der Verwendung eines Hashwertes zur Identifikation gleicher Seiteninhalte? **1 pt**

Where do you see potential problems in the use of a hash function for identifying equal page contents?

c) Vervollständigen Sie die Funktion `updatePte()`, die den Seitentabelleneintrag für die virtuelle Seite mit der Adresse `va` aktualisiert. **3.0 pt**

- Nutzen Sie das im Kommentar gegebene PTE-Format und die Bitmasken.
- Bei `pfn` = `INV_PFN` darf das Present-Bit nicht gesetzt werden.
- `ro` und `cow` bestimmen, ob die Seite als Read-Only oder Copy-On-Write markiert werden soll.

Complete the function `updatePte()`, which updates the page table entry for the virtual page with the address `va`.

- *Use the PTE format and the bit masks described in the comments.*
- *For `pfn` = `INV_PFN` the present bit must not be set.*
- *`ro` and `cow` determine, if the page should be marked read-only or copy-on-write.*

```
/* PTE: Bits[12:31] = PFN, Bits[4:11] = unused, Bits[0:3] = flags   */
#define PTE_P_MASK   0x1 /* Present - if set, PTE is valid mapping  */
#define PTE_RO_MASK  0x2 /* Read-only - if set, page fault on write */
#define PTE_COW_MASK 0x4 /* CoW - if set, copy-on-write enabled     */
                         /* (CoW flag ignored by MMU)               */

void setPte(uint32_t va, uint32_t pte);

void updatePte(uint32_t va, uint32_t pfn, int ro, int cow) {

}
```

d) Vervollständigen Sie die Funktion `getPfn()`, die die physischen Seitennummer (PFN) aus dem PTE der virtuellen Seite mit der Adresse `va` zurückgibt. **1.5 pt**

- Geben Sie `INV_PFN` zurück, falls das Present-Bit nicht gesetzt ist.

Complete the function `getPfn()`, which returns the physical frame number (PFN) from the PTE of the virtual page with the address `va`.

- *Return `INV_PFN` if the present bit is not set.*

```
uint32_t getPte(uint32_t va);
uint32_t getPfn(uint32_t va) {







}
```

e) Vervollständigen Sie die Funktion `tryMerge()`, die versucht die virtuelle Seite mit der Adresse `va` zu deduplizieren. **6.5 pt**

- T_{scan} sucht in Richtung aufsteigender Adressen und ruft `tryMerge()` für jede gültige (Present-Bit gesetzt) Seite einmal auf.
- Verwenden Sie `lockVas()`/`unlockVas()` sowie Schreibschutz wo nötig.
- Verwenden Sie das CoW-Flag nur für redundante Seiten.
- Bei einem Seitenfehler wird eine schreibgeschützte Seite unter dem VAS-Lock aus der Hashtabelle entfernt.
- Übereinstimmende Hashes sind zur Deduplikation ausreichend.

Complete the function `tryMerge()`, which tries to deduplicate the virtual page with the address `va`.

- *T_{scan} scans in the direction of increasing addresses and calls `tryMerge()` once per valid (present bit set) page.*
- *Use `lockVas()`/`unlockVas()` and write protection where necessary.*
- *Use the CoW-flag for redundant pages only.*
- *On a page fault a write-protected page is removed from the hash table under the VAS lock.*
- *Matching hashes are sufficient for deduplication.*

```
/* Inserts a new <hash, va>-pair into the hash table.
 * Every hash value can occur only once in the table. */
void insert(uint32_t hash, uint32_t va);

/* Returns the virtual address for the given hash, or
 * 0 if the hash cannot be found in the hash table */
uint32_t find(uint32_t hash);
```

```
uint32_t getPfn(uint32_t va);
void updatePte(uint32_t va, uint32_t pfn, int ro, int cow);
uint32_t hash(uint32_t pfn);
void tryMerge(uint32_t va) {

}
```

f) Vervollständigen Sie die Funktion `writeFault()`, die bei schreibendem Zugriff auf eine gültige, schreibgeschützte virtuelle Seite mit der Adresse `va` aufgerufen wird. **5.0 pt**

- Verwenden Sie `lockVas()`/`unlockVas()` wo nötig.
- Die PTEs von anderen Seiten außer `va` müssen nicht angepasst werden.
- Nutzen Sie `memcpy()`, um Seiteninhalte zu kopieren. Kopieren Sie Seiten nur, wenn sie dedupliziert wurden.

Complete the function `writeFault()`, which is called on write access to a valid, write-protected virtual page with the address `va`.

- *Use `lockVas()`/`unlockVas()` where necessary.*
- *The PTEs of pages other than `va` do not need to be updated.*
- *Use `memcpy()` for copying page contents. Only copy pages which have been deduplicated.*

```c
/* Computes the hash of the virtual address and removes it from the table.
 * If the hash does not exist, does nothing */
void remove(uint32_t va);

/* Allocates a free physical page and returns the PFN. Never fails! */
uint32_t allocPfn(void);

uint32_t *getPage(uint32_t pfn);
void updatePte(uint32_t va, uint32_t pfn, int ro, int cow);
void writeFault(uint32_t va) {


















}
```

Total: 20.0pt

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

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

memset − fill memory with a constant byte

SYNOPSIS

#include <string.h>

void *memset(void ***_s_, int** _c_**, size_t** _n_**);**

DESCRIPTION

The **memset**() function fills the first _n_ bytes of the memory area pointed to by _s_ with the constant byte _c_.

RETURN VALUE

The **memset**() function returns a pointer to the memory area _s_.

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

opendir, closedir − open/close a directory
readdir − read a directory

SYNOPSIS

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

DIR *opendir(const char ***_name_);**
struct dirent *readdir(DIR ***_dirp_);**
int closedir(DIR ***_dirp_);**

DESCRIPTION

The **opendir**() function opens a directory stream corresponding to the directory _name_, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The **closedir**() function closes the directory stream associated with _dirp_. A successful call to **closedir**() also closes the underlying file descriptor associated with _dirp_. The directory stream descriptor _dirp_ is not available after this call.

The **readdir**() function returns a pointer to a _dirent_ structure representing the next directory entry in the directory stream pointed to by _dirp_. It returns NULL on reaching the end of the directory stream or if an error occurred.

On Linux, the _dirent_ structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    unsigned char  d_type;      /* type of file */
    char           d_name[256]; /* filename */
};
```

glibc defines the following macro constants for the value returned in _d_type_:

DT_DIR This is a directory.

DT_FIFO This is a named pipe (FIFO).

DT_REG This is a regular file.

The data returned by **readdir**() may be overwritten by subsequent calls to **readdir**() for the same directory stream.

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and _errno_ is set appropriately.

On success, **readdir**() returns a pointer to a _dirent_ structure. (This structure may be statically allocated; do not attempt to **free**(3) it.) If the end of the directory stream is reached, NULL is returned and _errno_ is not changed. If an error occurs, NULL is returned and _errno_ is set appropriately.

ERRORS

ENOENT Directory does not exist, or _name_ is an empty string.

ENOTDIR _name_ is not a directory.

EBADF Invalid directory stream descriptor.

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 calls **pthread_exit**(3), specifying an exit status value that is available to another thread in the same process that calls **pthread_join**(3).
* It returns from _start_routine_(). This is equivalent to calling **pthread_exit**(3) with the value supplied in the _return_ statement.
* It is canceled (see **pthread_cancel**(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_.

NOTES

A thread may either be _joinable_ or _detached_. If a thread is joinable, then another thread can call **pthread_join**(3) to wait for the thread to terminate and fetch its exit status. Only when a terminated joinable thread has been joined are the last of its resources released back to the system. When a detached thread terminates, its resources are automatically released back to the system: it is not possible to join with the thread in order to obtain its exit status. Making a thread detached is useful for some types of daemon threads whose exit status the application does not need to care about. By default, a new thread is created in a joinable state, unless _attr_ was set to create the thread in a detached state (using **pthread_attr_setdetachstate**(3)).

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. The thread specified by _thread_ must be joinable.

If _retval_ is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to **pthread_exit**(3)) into the location pointed to by _retval_. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by _retval_.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

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
thread is not a joinable thread.

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

ESRCH
No thread with the ID _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.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
```

DESCRIPTION

The *pthread_mutex_destroy*() function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*() to set the object referenced by *mutex* to an invalid value.

A destroyed mutex object can be reinitialized using *pthread_mutex_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

The *pthread_mutex_init*() function shall initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Only *mutex* itself may be used for performing synchronization. The result of referring to copies of *mutex* in calls to *pthread_mutex_lock*(), *pthread_mutex_trylock*(), *pthread_mutex_unlock*(), and *pthread_mutex_destroy*() is undefined.

RETURN VALUE

If successful, the *pthread_mutex_destroy*() and *pthread_mutex_init*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_init*() function shall fail if:

ENOMEM
Insufficient memory exists to initialize the mutex.

NAME

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock − lock and unlock a mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

The *pthread_mutex_trylock*() function shall be equivalent to *pthread_mutex_lock*(), except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately.

The *pthread_mutex_unlock*() function shall release the mutex object referenced by *mutex*. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

RETURN VALUE

If successful, the *pthread_mutex_lock*() and *pthread_mutex_unlock*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_mutex_trylock*() function shall return zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

COPYRIGHT

NAME

stat, fstat − get file status

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

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

The *st_dev* field describes the device on which this file resides. (The **major**(3) and **minor**(3) macros may be useful to decompose the device ID in this field.)

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

The *st_blksize* field gives the "preferred" blocksize for efficient filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

RETURN VALUE

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

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

strcpy, strncpy − copy a string

SYNOPSIS

#include <string.h>

char *strcpy(char ****dest*, const char ****src***);**
char *strncpy(char ****dest*, const char ****src***, size_t** *n***);**

DESCRIPTION

The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!*

The **strncpy**() function is similar, except that at most *n* bytes of *src* are copied. **Warning**: If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.

If the length of *src* is less than *n*, **strncpy**() writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

RETURN VALUE

The **strcpy**() and **strncpy**() functions return a pointer to the destination string *dest*.

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*.