Moduldokumentation

Modul Systemprogrammierung (syspr)

Simon Wächter

2017

# Setcard

## Speicheroperationen

* **void \*malloc(int n x sizeof(item)**: Alloziert n Mal die Grösse von item an Bytes und gibt diesen zurück
* **void free(void \*ptr)**: Löscht den allozierten Speicher
* **void \*memcpy(void \*desptr, void \*srcptr, size\_t length)**: Kopiert den Eingangsbuffer in den Ausgangsbuffer, sofern die Länge nicht überschritten wird. Zurückgeben wird der Ausgangsbuffer

## Stringoperationen

* **char \*strcpy(char \*destptr, char \*srcptr)**: Kopiert den Eingangsstring in den Ausgangsstring. Zurückgegeben wird der Ausgangsstring
* **size\_t strlen(char \*ptr)**: Gibt die Länge eines Strings zurück
* **int sprintf(char \*destptr, char \*specifiers, va\_list arguments)**: Formatiert einen String und schreibt ihn in den Ausgangsbuffer

## Terminaloperationen

* **int scanf(char \*specifiers, va\_list arguments)**: Liest Werte von der Standardeingabe
* **int printf(char \*specifiers, va\_list arguments)**: Schreibt Werte auf die Standardausgabe

## Dateioperationen

* **FILE \*fopen(char \*path, char \*mode)**: Öffnet eine Datei im Modus (r=Read, w+=Read) und gibt ein Handle zurück
* **int fclose(FILE \*handle)**: Schliesst einen Handle
* **int fgetc(FILE \*handle)**: Liest einen Chrakter als Integer aus der Datei. Als Ende wird EOF zurückgegeben
* **int fputc(FILE \*handle, char characer)**: Schreibt einen Charakter in die Datei
* **int rename(char \*oldpath, char \*newpath)**: Benennt eine Datei um
* **FILE \*dup(FILE \*handle)**: Dupliziert einen Handle

## Verzeichnisoperationen

* **DIR \*opendir(char \*path)**: Öffnet ein Verzeichnis und gibt ein Handle zurück
* **dirent \*readdir(DIR \*handle)**: Gibt als Iterator alle Verzeichniseinträge als dirent oder aber als NULL zurück
* **int closedir(DIR \*handle)**: Schliesst einen handle
* **int mkdir(char \*path, mode\_t mode)**: Erstellt ein Verzeichnis mit dem Mode (z.B. 0777)
* **int rmdir(char \*path)**: Löscht ein Verzeichnis
* **struct dirent**: Verzeichnisstruktur mit folgenden Elementen:
  + d\_name: Name des Verzeichnisses oder der Datei
  + d\_type: Typ des Verzeichnisses (z.B. DT\_DIR)

## Prozessoperationen

* **void exit(int status)**: Terminiert den Prozess mit dem angebenen Status
* **pid\_t fork()**: Forkt einen Prozess und gibt für den Parent die PID des Childs und für das Child selber 0 zurück
* **int execl(char \*path, char\*arg, char \*env[])**: Führt einen neuen Prozess in der aktuellen prozesshülle aus
* **pid\_t getpid()**: Gibt die aktuelle Prozessnummer zurück
* **pid\_t getppid()**: Gibt die aktuelle Prozessnummer des Parent zurück
* **int kill(pid\_t pid)**: Terminiert einen Prozess
* **unsigned int sleep(int seconds)**: Lässt einen Prozess schlafen
* **pid\_t wait(0)**: Wartet auf den nächsten Prozess
* **pid\_t waitpid(pid\_t pid)**: Wartet auf einen bestimmten Prozess

## Pipeoperationen (Unnamed)

* **int pipe(int[2] pipe)**: Öffnet eine Pipe (0=Reader, 1=Writer)
* **int close(int pipeentry)**: Schliesst eine Ende
* **ssize\_t read(int pipeentry, char \*buffer, size\_t buffersize)**: Liest von der Pipe
* **ssize\_t write(int pipeentry, char \*buffer, size\_t buffersize)**: Schreibt auf die Pipe

## Pipeoperationen (Named)

* **int mknod(char \*path, mode\_t mode, rdev\_t device\_id)**: Erstellt eine normale Datei oder Pipe
* **int mkfifo(char \*path, mode\_t mode)**: Erstellt eine FIFO Pipe mit dem Mode (z.B. 0777)
* **int open(char \*path, int oflag)**: Öffnet eine Pipe in Mode (z.B. O\_RDONLY oder O\_WRONLY)
* **int close(int pipeentry)**: Schliesst eine Ende
* **ssize\_t read(int pipeentry, char \*buffer, size\_t buffersize)**: Liest von der Pipe
* **ssize\_t write(int pipeentry, char \*buffer, size\_t buffersize)**: Schreibt auf die Pipe

## Signaloperationen

* **void (\*signal(int sig, void (\*func)(int)))(int)**: Installiert eine neuen Signalhandler und gibt den alten zurück. Gängige Signale sind SIGABRT, SIGFPE, SIGINT oder SIGALARM
* **int raise(int signal)**: generiert ein Signal und schickt es dem Betriebssystem
* **unsigned int alarm(unsigned int seconds)**: Generiert nach der angegebenen Zeit ein Alarmsignal

## Sprungoperationen

* **int setjmp(jmp\_buf environment)**: Setzt den Buffer auf das lokale Environment und initialisiert dieses für einen Sprung. Wird die Funktion direkt aufgerufen, wird 0 zurückgegeben. Wird die Funktion nach einem Jump via longjmp aufgerufen, wird ein Wert ungleich 0 zurückgegeben
* **longjmp(jmp\_buf environment, int value)**: Führt einen Sprung basierend auf dem Environment aus und übergibt den angegeben Wert an die setjmp Funktion (z.B. Errorcode)
* **struct jmp\_buf**: Liefert ein Environment für die Jumps

## Fehlerbehandlung

* **void perror(char \*message)**: Gibt die angegebene nachricht mit dem letzten Fehler (erno) auf der Standardausgabe aus

## Lokales IPC Schlüsselgenerierung

* **key\_t ftok(char \*path, int projid)**: Generiert aus dem Pfad und den letzten 8 Bits von projid einen eindeutigen Schlüssel oder gibt -1 zurück
* **IPC\_CREAT** erstellt ein Objekt falls der Schlüssel noch unbelegt ist
* **IPC\_CREAT | IPC\_EXCL** erstellt ein Objekt falls der Schlüssel noch unbelegt ist oder bricht allenfalls bei Existenz ab
* **IPC\_PRIVATE** erstellt in jedem Fall ein neues Objekt

## Lokales IPC via Shared Memory

* **int shmget(key\_t key, int memlength, int ormode)**: Erstellt Shared Memory oder liefert den existierenden Shared Memory als Schlüssel oder -1 bei einem Fehler zurück
  + ormode: IPC\_CREAT | int chmod: Erstellt ein neues Segment
  + ormode: int chmod: Zugriff auf ein existierendes Segment
* **void \*shmat(int shmid, void \*address = 0, int flags = 0)**: Hängt das Shared Memory an den Prozess an und gibt den Speicherbereich oder -1 bei einem Fehler zurück
* **int shmdt(void \*shmaddress)**: Hängt das Shared Memory aus dem Addressraum des Prozesses aus und gibt 0 oder -1 bei einem Fehler zurück
* **int shmctl(int shmid, IPC\_RMID, 0)**: Löscht das Shared Memory und gibt 0 oder -1 bei einem Fehler zurück

## Lokales IPC via Message Queues

* **struct msg\_buffer { long type; char text[length]; } msg\_buffer\_t;**
* **int msgget(key\_t key, int ormode)**: Erstellt eine Message Queue oder liefert eine existierende Message Queue als Schlüssel oder -1 bei einem Fehler zurück
  + ormode: IP\_CREAT | int chmod: Erstellt eine neue Message Queue
  + ormode: int chmod: Zugriff auf eine existierende Message Queue
* **int msgsnd(int msgid, void \*message, size\_t messagetext, int flags = 0)**: Sendet eine Message und gibt 0 oder -1 bei einem Fehler zurück
* **ssize\_t msgrcv(int msgid, void \*message, size\_t messagetext, long type = 0, int flags = 0)**: Liest eine Message und gibt die Anzahl gelesener Bytes oder -1 bei einem Fehler zurück
* **int msgctl(int msgid, IPC\_RMID, 0)**: Löscht die Message Queue und gibt 0 oder -1 bei einem Fehler zurück

## Lokales IPC via Semaphoren

* **struct sembuf { int sem\_op = -1 | 1; int sem\_num = 0; int sem\_flg = 0; };**
* **int semget(key\_t key, int counter, int ormode)**: Erstellt ein Semaphore oder liefert ein existierendes Semaphore als Schlüssel oder -1 bei einem Fehler zurück
  + counter = n + ormode: IPC\_CREAT | PERM | int chmod: Erstellt ein neues Semaphore mit dem Counter n
  + counter = 0 + ormode: int chmod: Zugriff auf ein existierendes Semaphore mit dem Counter 0
* **int semctl(int semid, 0, SETVAL, n = 1)**: Initialisiert das Semaphore mit dem Wert n und gibt einen nicht negativen Wert oder -1 bei einem Fehler zurück
* **int semctl(int semid, 0, IPC\_RMID, 0)**: Analog, löscht aber das Semaphore
* **int semop(int semid, sembuf &semaphore, 1)**: Setzt den Wert des Semaphore und gibt 0 oder -1 bei einem Fehler zurück
  + semaphore.sem\_op = -1 (Lock) oder 1 (Unlock)

## Socket Struktur sockadd\_in

* **struct sockaddr\_in saddr;**
* **saddr.sin\_family = AF\_INET;**
* **saddr.sin\_port = htons(port);**
* **saddr.sin\_addr.s\_addr = inet\_addr(hostname);**
* **socklen\_t saddr\_length = sizeof(saddr);**

## Server Socket

* **int socket(AF\_INET, <SOCK\_STREAM | SOCK\_DGRAM>, 0)**: Erstellt einen neuen Socket und gibt die ID des Sockets oder -1 bei einem Fehler zurück
* **int bind(int socketid, (struct sockaddr \*) &saddr, socklen\_t saddr\_length)**: Bindet den Socket auf den Port und gibt 0 oder -1 bei einem Fehler zurück
* **int listen(int socketid, int backlog)**: Öffnet den Port für eingehende Verbindungen und gibt 0 oder -1 bei einem Fehler zurück
* **int accept(int socketid, (struct sockaddr \*) &caddr, socklen\_t &caddr\_length)**: Wartet blockierend auf einen eingehende Verbindung und erstellt einen Port mit seinem lokalen Port
* **int close(int socketid)**: Schliesst einen Socket und gibt 0 oder -1 bei einem Fehler zurück
* **int recv(int socketid, char \*buffer, size\_t length, 0)**: Wartet blockierend auf einen TCP Request und gibt die gelesenen Bytes oder -1 bei einem Fehler zurück
* **int send(int socketid, char \*buffer, size\_t length, 0)**: Schreibt eine TCP Response und gibt die geschriebenen Bytes oder -1 bei einem Fehler zurück
* **int recvfrom(int socketid, char \*buffer, size\_t length, 0, &caddr, &caddr\_length)**: Wartet blockierend auf einen UDP Request und gibt die gelesenen Bytes oder -1 bei einem Fehler zurück
* **int sendto(int socketid, char \*buffer, size\_t length, 0, &caddr, &caddr\_length)**: Schreibt eine UDP Response und gibt die geschriebenen Bytes oder -1 bei einem Fehler zurück

## Server Socket Operationen für den Client

* **int getsockname(int socketid, (struct sockaddr \*) &saddr, &saddr\_length)**: Initialisiert die Adresse für den gegebenen Socket und gibt 0 oder -1 bei einem Fehler zurück
* **int inet\_ntop(AF\_INET, &saddr.sin\_addr, buffer, length)**: Liefert einen gültigen Pointer oder Null bei einem Fehler zurück

## Client Socket

* **int socket(AF\_INET, <SOCK\_STREAM | SOCK\_DGRAM>, 0)**: Erstellt einen neuen Socket und gibt die ID des Sockets oder -1 bei einem Fehler zurück
* **int connect(int socketid, (struct sockaddr \*) &saddr, socklen\_t saddr\_length)**: Verbindet den Socket mit dem Server und gibt 0 oder -1 bei eine Fehler zurück