


```

pthread_attr_t *attr; pthread_attr_init(&sa); // initialize attributes
sched_param p;
pthread_attr_setschedparam(&sa, &p); // read parameter
pthread_attr_setschedparam(&sa, &p); // write modified parameters
pthread_create(&id, &attr, thread_function, argument);
pthread_attr_destroy(&sa); // destroy attributes

```

7. MUTEX AND SEMAPHORE

Jeder Thread hat seinen eigenen Instruction-Pointer und Stack-Pointer. Wenn Ergebnisse von der **Ausführungsreihenfolge** einzelner Instruktionen abhängen, spricht man von einer **Race Condition**. Threads müssen **synchronisiert** werden, damit keine **Race Condition** entsteht.

Critical Section: Code-Bereich, in dem Daten mit anderen Threads **geteilt** werden. Muss unbedingt synchronisiert werden.

Atomare Instruktionen: Eine atomare Instruktion kann vom Prozessor **unterbrechungsfrei** ausgeführt werden. (Achtung: Selbst einige Assembly-Instruktionen nicht immer atomic!)

Anforderungen an Synchronisations-Mechanismen: **Gegenseitiger Ausschluss** (Entscheidend, wer in der Critical Section darf), muss in **edeller Zeit** getroffen werden, **Begründetes Warten** (Thread wird nur über Reboots hinweg, bevor er in die Critical Section darf).

Implementierung: Nur **MWU-Hilfestellung** möglich. Es gibt zwei atomare Instruktionen:

Test-&-Set: `test_and_set(&int var, &int value);` und **Compare-&-Swap** (`int compare(&int var, &int target);`) mit einem **lock** mit einer **target**, wenn dieser dem erwarteten Wert entspricht: `compare_and_swap(&int var, &int expected, &int new);`

7.1. SEMAPHORE

Enthält Zähler $Z \geq 0$. Wird nur über **Post** (zum J) und **Wait** (zugegriffen). (Wenn $Z = 0$, verzögert es um 1 und fehlt Wert. Wenn $Z = 0$, setzt den Thread in waiting, bis anderer Thread z erhöht).

sem_sewait(&sem, tsem, tsem_shared, unsigned int value); Initialisiert den Semaphore, typischerweise als **globale Variable**, phaserd: $tsem = 1$: Verwendung über mehrere Prozesse.

sem_t sem; int main (int argc, char ** argv) { sem_init (&sem, 0, 0); ... }; oder als Parameter für den Thread (speicher auf Stack oder Heap): `struct T { sem_t *sem; ...};`

int sem_sepost(&sem, tsem); Implementieren **Post** und **Wait**.

int sem_trywait(&sem, tsem); **sem_t sem; const struct timespec *abs_timeout; int sem_timedwait(&sem, tsem, abs_timeout);**

Sind wie **sem_sewait**, aber **abrechen**, kann falls Dekrementierung **nicht** durchgeführt werden kann. **sem_sewait** reicht sofort ab, **sem_timedwait** nach der angegebenen Zeitdauer.

sem_destroy(&sem); Entfernt Speicher, den das OS mit **assoziiert** hat.

semaphore free = 0; semaphore used = 0;

// Producer // Consumer while(1) { // Warte, falls Customer zu langsam // Warte, falls Producer zu langsam WAIT(); // Hat es Platz in Queue? (usleep()); // Es Element in Queue consume (buf[buf_r]); POST(); // free; // 1 Element weniger in Q r = (r+1) % BUFFER_SIZE; }

7.2. MUTEX

Ein Mutex hat einen **bindenden Zustand**, der nur durch zwei Funktionen verändert werden kann: **Aquire** (Wenn $v = 0$, setze $1 + fahrt$ Wert. Wenn $v = 1 + b$, blockiere den Thread, bis $v = 0$), **Release** (Setzt $v = 0$). Auch als non-blocking-Funktion: `int pthread_mutex_trylock(pthread_mutex_t *mutex);`

// Beispiel Initialisierung // Beispiel Verwendung in Threads

pthread_mutex_t mutex; // global variable void * thread_function(void * args) { while(1) { ... } } (runnig); ...

int main() { pthread_mutex_init(&mutex, NULL);

// Enter critical section:

// Run threads until it's time to finish pthread_join(thread, &status); // Leave critical section pthread_mutex_unlock(&mutex); } ...}

Priority Inversion: Ein **hoch-priorisierte Thread C** wartet auf eine Ressource, die von einem **niedrig-priorisierten Thread A** gehalten wird. Ein Thread mit Priorität zwischen diesen beiden Threads erhält den Prozessor. Kann mit **Priortiy Inheritance** gelöst werden: Die Priorität von A wird temporär auf die Priorität von C gesetzt, damit der Mutex schnell wieder freigegeben wird.

8. SIGNALS, PIPES AND SOCKETS

8.1. SIGNALS

Signale ermöglichen es, einen Prozess von aussen zu unterbrechen, wie ein **Interrupt**. Unterbrechen des gerade laufenden Prozesses/Treads, Auswahl und Ausführen der **Signal-Handler-Funktionen**.

Fortsätzen des Prozesses. Werden über ungültige Instruktionen oder Abbruch auf Seitens Benutzer ausgelöst. Jeder Prozess hat pro Signal einen Handler.

Handler: Ignore-Handler (ignoriert das Signal), Terminate-Handler (beendet das Programm), Abnormal-Terminate-Handler (beendet das Programm und erzeugt Core-Dump), Fast alle Signale ausser SIGKILL und SIGSTOP können überwischen werden.

Programmfehler-Signale: SIGFPE (Fehler in arithmetischen Operationen), SIGILL (ungültige Instruktion), SIGSEGV (Unqualifizierter Spezialisierungsfehler), SIGSYS (Systemfehler)

Prozesse abbrechen: SIGTERM (Normaler Anfrage auf den Prozess, sich zu beenden), SIGINT (Nachdrückliche Aufforderung an den Prozess, sich zu beenden), SIGQUIT (Wie SIGTERM, aber anomale Terminierung), SIGABRT (Wie SIGQUIT, aber vom Prozess an sich selber), SIGKILL (Prozess wird abgekämpft, kann nicht verhindert werden)

Stop and Continue: SIGSTP (Versetzt den Prozess in den Zustand stopped, wiehrlig wating), SIGSTOP (Wie SIGSTP, aber kann nicht ignoriert oder abgefangen werden), SIGCONT (Setzt den Prozess fort)

Signale der Shell senden: kill 1234 5678 sendet SIGTERM an Prozesse 1234 und 5678

int sigaction(SIG_int, struct sigaction *new, struct sigaction *old); Definiert Signal-Handler für signal, wenn new ≠ 0. (Eigen-Handler definiert via sigaction struc. **sig_handler:** Zu **call function**, **seh:** Blockiert die Signale während Ausführung, bearbeitet nur durch sigset(SIG_BLOCK, handler); **functions:** sigset(SIG_BLOCK, handler); sigset(SIG_BLOCK, handler); sigset(SIG_BLOCK, handler);)

8.2. PIPES

Eine geöffnete Pipe dient einspielt im **File-Descriptor-Tabelle (FDT)** im Prozess. Zugriff über `File API` (open, close, read, write...). Das OS speichert je **Eintrag der Prozess-DTD** einen Verweis auf die **globale FDT**. Bei `fork()` wird die FDT auch kopiert.

int dup (int source_fd, int dup_to); **int dup2 (int source_fd, int destination_fd);** Duplizieren den File-Descriptor `source_fd`, dup alloziert einen neuen FD, dup2 überschreibt `destination_fd`.

8.2.1. Umleiten des Ausbares

int fd[2]; // 0 = read, 1 = write

fd[0] = open("log.txt", ...);

fd[1] = fork();

if (fd[1]) { // child process

dup2(fd[0], 1); // duplicate fd for log.txt as standard output

// e.g. log new image with exec, fd's remain

} else { // parent process

close(fd[1]); // Child thread

close(fd[1]); // don't use write end

char buffer[BSIZE];

int n = read(fd[0], buffer, BSIZE);

else { // Parent thread

close(fd[1]); // don't use read end

char *text = "1<>segfault";

write(fd[1], text, strlen(text) + 1);

Pipe lebt nur so lange, wie mind. ein Ende geöffnet ist. Alle Read-Ends geschlossen

SIGPIPE an End-Writer. Mehrere Writes können zusammengefassen werden. Lesen mehrere Prozesse diese Pipe, ist unklar, wer die Daten erhält.

int mkfifo (const char *path, mode_t mode); Erzeugt eine Pipe mit **Namen** und **Pfad** im Dateisystem. Hat via mode Permission Bits wie normale Datei. Lebt unabhängig vom erzeugenden Prozess, nur je System auch über Reboots hinweg. Muss explizit mit unlink gelöscht werden.

8.3. SOCKETS

Ein Socket **repräsentiert einen Endpunkt auf einer Maschine**. Kommunikation findet im Regelfall zwischen zwei Sockets statt (UDP, TCP über wie Unix-Domains-Sockets). Sockets benötigen für Kommunikation einen Namen (Besitz IP, Port & Portnummer).

int socket(domain, type, protocol); Erzeugt einen neuen Socket als `Socket`. `domain` sind zurzeit `AF_INET` (IPv4, AF_INET6, AF_NETLINK), `type` (SO_DGRAM, SO_STREAM, protocol), `protocol` = 0 Default=Protocol.

socket(domain, type, protocol); send (mit Shared Memory bleibt über System).

int shm_unlink (const char *name); Löscht das Shared Memory mit dem name. (bleibt vorhanden, solange noch von Prozessen.

int mmap (void *address, size_t length); Entfernt das Mapping.

`void *addr = mmap(..., map shared memory into virt. address space of process 0, ...` void void *hostaddr (0 because nobody cares) size_of_shared_memory, size_t length (same length as used in `fruncate`) MAP_SHARED | PROT_READ, ...); `int execve (const char *file, ..., void *args, void *envp);`

`struct sockaddr_in ip_addr; ip_addr.sin_port = htons (443); // default HTTPS port`

`inet_ntop (AF_INET, "192.168.0.1", &ip_addr.sin_addr.s_addr);`

`IP_ADDRESS = 0x00000001; // IP Address in memory: 0x00 0x00 0x00 0x01`

`fd = socket (domain, type, protocol);`

`int bind (int socket, const struct sockaddr *local_address, socklen_t addrlen);`

Bindet den Socket an die angegebene, unbekannte lokale Adresse, wenn noch gebunden.

Blockiert, bis der Vorgang abgeschlossen ist.

`int connect (int socket, const struct sockaddr *remote_addr, socklen_t addrlen);`

Verbindet den Socket an eine neue unbekannte lokale Adresse, wenn noch gebunden. Blockiert, bis Verbindung steht oder ein Timeout eintritt.

`int listen (int socket, int backlog);` Markiert den Socket als bereit zum Empfang von Verbindungen. Erzeugt eine Warteschlange, die so viele Verbindungsanfragen aufnehmen kann, wie backlog angibt.

`int accept (int socket, struct sockaddr *remote_addr, socklen_t *addrlen);`

Ergebnis: Ein neuer Socket und die Adresse des Verbindungsanfrager.

`int read (int socket, void *buffer, size_t bytes);`

Lesen von Shared Memory. (read vor write)

`int write (int socket, void *buffer, size_t bytes);`

Schreiben von Shared Memory. (write vor read)

`int shutdown (int socket, int how);`

Warten auf eine Verbindung. (how = 1: SHUT_RD (Keine Schreib-Zugriffe mehr), SHUT_WR (Keine Schreib-Zugriffe mehr), SHUT_RDWR (Keine Lese- oder Schreib-Zugriffe mehr))

9. MESSAGE PASSING AND SHARED MEMORY

Shared Memory ist schneller zu realisieren, aber schwer warbar. **Message-Passing** erfordert mehr Engineering-Aufwand, schlussendlich aber in Mehr-Prozessor-Systemen bald performanter.

9.1. VERGLEICH MESSAGE-QUEUES UND PIPES

Message-Queues

- bidirektional

- Daten sind in einzelnen Messages organisiert

- beliebiger Zugriff

- Haben immer einen Namen

9.1. POSIX API

Das OS benötigt eine **Datei S**, das Informationen über den gemeinsamen Speicher verwaltet und eine **Mapping Table** je Prozess.

int ftruncate (int fd, off_t length); Setzt Größe der Datei. Muss **zwingend** nach Shared-Memory erstellt gesetzt werden, um entsprechende Vizes zu erhalten. Wird für Shared Memory mit ganzzähligen Vielfachen der Page-/Framegröße verwendet.

int close (int fd); Schließt die Datei. Shared Memory bleibt **über** im System.

int shm_unlink (const char *name); Löscht das Shared Memory mit dem name. (bleibt vorhanden, solange noch von Prozessen).

int munmap (void *address, size_t length); Entfernt das Mapping.

`void *addr = mmap(..., map shared memory into virt. address space of process 0, ...` void void *hostaddr (0 because nobody cares) size_of_shared_memory, size_t length (same length as used in `fruncate`) MAP_SHARED | PROT_READ, ...); `int execve (const char *file, ..., void *args, void *envp);`

`struct sockaddr_in ip_addr; ip_addr.sin_port = htons (443); // default HTTPS port`

`inet_ntop (AF_INET, "192.168.0.1", &ip_addr.sin_addr.s_addr);`

`IP_ADDRESS = 0x00000001; // IP Address in memory: 0x00 0x00 0x00 0x01`

`fd = socket (domain, type, protocol);`

`int bind (int socket, const struct sockaddr *local_address, socklen_t addrlen);`

Bindet den Socket an die angegebene, unbekannte lokale Adresse, wenn noch gebunden.

Blockiert, bis der Vorgang abgeschlossen ist.

`int connect (int socket, const struct sockaddr *remote_addr, socklen_t *addrlen);`

Verbindet den Socket an eine neue unbekannte lokale Adresse, wenn noch gebunden. Blockiert, bis Verbindung steht oder ein Timeout eintritt.

`int listen (int socket, int backlog);` Markiert den Socket als bereit zum Empfang von Verbindungen. Erzeugt eine Warteschlange, die so viele Verbindungsanfragen aufnehmen kann, wie backlog angibt.

`int accept (int socket, struct sockaddr *remote_addr, socklen_t *addrlen);`

Ergebnis: Ein neuer Socket und die Adresse des Verbindungsanfrager.

`int read (int socket, void *buffer, size_t bytes);`

Lesen von Shared Memory. (read vor write)

`int write (int socket, void *buffer, size_t bytes);`

Schreiben von Shared Memory. (write vor read)

`int shutdown (int socket, int how);`

Warten auf eine Verbindung. (how = 1: SHUT_RD (Keine Schreib-Zugriffe mehr), SHUT_WR (Keine Schreib-Zugriffe mehr), SHUT_RDWR (Keine Lese- oder Schreib-Zugriffe mehr))

9.2. VERGLEICH MESSAGE-PASSING AND SHARED MEMORY

Shared Memory ist schneller zu realisieren, aber schwer warbar. **Message-Passing** erfordert mehr Engineering-Aufwand, schlussendlich aber in Mehr-Prozessor-Systemen bald performanter.

9.3. VERGLEICH MESSAGE-QUEUES UND PIPES

Message-Queues

- unidirektional

- übermittelt Bytestrom an Daten

- FIFO-Zugriff

- Müssen keinen Namen haben

Pipes

- bidirektional

- Daten sind in einzelnen Messages organisiert

- beliebiger Zugriff

- Haben immer einen Namen

Beispiel

Encoding of U+10'437 (U+00 0000 000 000 1111):

1. Code-Point 0 minus 1 00 000, rechnen und in Binär umwandeln

$P = 10437_b = 0 = 10437 - 100 000_b = 0437_b = 00 0000 000 000 1111;$

2. Obere & untere 10 Bits in Hex umwandeln

00 01 37_b

3. Oberer Wert mit D0 00, und unterer Wert mit DC 00 addieren, um Code-Units zu erhalten

$U_1 = 001_000_b + D0\ 00_b = 08100_b = 0137_b + DC\ 00_b = 0D37_b$

4. Zu BELE Zusammensetzen

$B_E = 0B\ 0137\ 0D37_b = 010\ 8E37\ 0D37_b$

10.4. UTF-8

Jede UML umfasst 8 Bit, ein CP benötigt 1 bis 4 Bytes. Encoding muss Endianless nicht berücksichtigen.

OS-Messe-Queue mit variabler Länge, haben mindestens 32 Prioritäten und können **synchron** oder **asynchron** verwendet werden.

`mqd_t mq_open (const char *name, int flags, mode_t mode, struct mq_attr *attr);`

öffnet eine Message-Queue mit systemweitem Name, returnt Message-Queue-Descriptor. (name mit / beginnen, flags & mode wie bei Dateien, mq_attr: Diverse Konfiguration & Queue-Status, lesen und schreiben der Attribute mq_setattr(), mq_seterr());

`int mq_close (mqd_t queue);` Schließt die Queue mit dem Descriptor queue für diesen Prozess. Name wird sofort entfernt und Queue kann anschließend **nicht mehr geöffnet** werden.

`int mq_send (mqd_t queue, const char *msg, size_t msg_size, unsigned int priority);`

Sendet die Nachricht, die Adresse msg beginnt und Length Bytes lang ist, in die Queue.

`int mq_receive (mqd_t queue, const char *msg, size_t msg_size, unsigned int priority);`

Kopiert die nächste Nachricht aus der Queue in den Puffer, an der Adresse msg beginnt und length Bytes lang. Blockiert, wenn die Queue leer ist.

9.1. SHARED MEMORY

Frames des Hauptspeichers werden **zwei (oder mehr) Prozessen P1 und P2 zugänglich** gemacht.

In P1 wird Page 1 von Frame 1 abgebildet. In P2 wird Page 1 von **dieselben** Frame 1 abgebildet. Beide Prozesse können beliebig auf dieselben Daten zugreifen. Im Shared Memory müssen **relative Adressen** verwendet werden.

Leben mehrere Prozesse diese Pipe, ist unklar, wer die Daten erhält.

12.1. NOTATION

Extreite Trees

indexete Adressierung

direkte Blöcke: indexete => Blocknummer

Name: Index der 1. Logischen Blöcke aller Kinder

Blattblöcke: <1 logischer Block, <1 physischer Block

Header: indexete => Anzahl Einträge, <fewer>

Beispiel-Berechnung: 4MB grosse, konsekutiv gespeicherte Datei, 4KB Blöcke ab Block 10 000.

4 MB = $2^{22} B$, 4 KB = $2^{12} B$, $2^{22-12} = 2^{10} = 4096$ Blöcke von 10 000 bis 13 FF_b

0 → 10 000, ... 1 → 10 020, ... B₃ → 10 080, ... C₁ → 10 040, ... D₀, nach Startblock

14 000, ... 10 000, ... B₀ → 10 000, ... 14 000, ... F₃ → 13 FF_b

Extreite Trees

Header: 0 → 1 (L0)

Einträge: 1 → (0, 10 000, 4 000)

12.2. JOURNALING

Wenn Date