SPickzettel-Ideen

maxiro

C

Netzwerk

- Dienstleister: `socket, listen, bind, accept`

```
int sock = socket(AF_INET6, SOCK_STREAM, 0);
if (sock == -1) die("socket");
struct sockaddr_in6 addr = {
    .sin6_family = AF_INET6,
    .sin6_addr   = in6addr_any, // Eigentlich nicht nötig, 0-initialisiert.
    .sin6_port   = htons(PORT),
};

if (bind(sock, (struct sockaddr*) &addr, sizeof addr)) die("bind");

if (listen(sock, 0)) die("listen");

for(;;) {
  int cli = accept(sock, NULL, NULL);
  if(cli == -1) continue;
  // ...
}
```

- Klient: `socket, getaddrinfo, connect`

```
struct addrinfo *res, *iter;
int err, sock;

struct addrinfo inf = {
    .ai_socktype = SOCK_STREAM, // TCP
    .ai_family = AF_UNSPEC, // Beliebiges L3-Protokoll
    .ai_flags = AI_ADDRCONFIG, // v4 oder v6 automatisch
};

err = getaddrinfo(HOST, PORT, &inf, &res);
if (err == EAI_SYSTEM) {
  die("getaddrinfo");
} else if (err) {
  fprintf(stderr, "%s\n", gai_strerror(err));
  exit(EXIT_FAILURE);
}

for (iter = res; iter != NULL; iter = iter->ai_next) {
    sock = socket(iter->ai_family,
          iter->ai_socktype,
          iter->ai_protocol);
   if (sock == -1) die("socket"); // Oder perror, continue
   if (0 == connect(sock, iter->ai_addr, iter->ai_addrlen)) {
       break;
   }
    close(sock);
}
```

```
if (iter == NULL) {
    die("connect");
}

freeaddrinfo(res);
// Use sock ... 🤞
```

Faden

- Lebenszyklus: `pthread_{create,detach,join}`

```
static void* func(void* arg) {
  if (errno = pthread_detach(pthread_self())) die("pthread_detach");
}

pthread_t tid;
if (errno = pthread_create(&tid, NULL, &func, arg)) die("pthread_create");
void* retval;
if (errno = pthread_join(tid, &retval /*| NULL*/)) die("pthread_join");
```

- Mutex, Bedingungsvariable: `pthread_cond_{init,wait,signal,destroy}`, `pthread_mutex_{init,lock,unlock,destroy}`

```
pthread_mutex_t mut;
pthread_mutex_init(&mut, NULL);
pthread_mutex_lock(&mut);
pthread_mutex_unlock(&mut);
pthread_mutex_destroy(&mut);
// Return: 0 oder Fehler
```

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
pthread_cond_wait(&cond);
pthread_cond_signal(&cond);
pthread_cond_destroy(&cond);
// Return: 0 oder Fehler
```

Prozess

- Lebenszyklus: `fork, execve, wait, waitpid`

```
pid_t pid = fork();
if (pid == -1) // Fehler
  die("fork");
else if (pid == 0) { // Kind
  execvp(ch_argv[0], ch_argv); // oder
  execlp(ch_argv[0], ...)
  die("execve");
} else { // Elter
  int status;
  pid_t pid2 = waitpid(pid | -1, &status, WNOHANG | 0);
  if (pid2 == -1) die("waitpid"); // 0 bei WNOHANG: kein Zombie
  if (WIFEXITED(status)) ...
}
```

E/A

- Datei: `fopen, fget{s,c}, feof, ferror, fflush(stdout), fclose, fputs, fprintf`

```c
FILE* f = fopen(PATH, "r" | "w");
if (fprintf(f, "narf%s\n", "foo") < 0) die("fprintf");
if (!fgets(&buf, sizeof(buf), f)) die("fgets");
int c;

while ((c = fgetc(f)) != EOF)
  if (fputc(c, f) == EOF) die("fputc");
if (ferror(f)) die("fgetc");

if (fclose(f) < 0) die("fclose");
if (fflush(stdout) < 0) die("fflush");
```

- Dateideskriptor: `open, fdopen, close, stat {f,l}stat, dup, dup2`

```c
struct stat statbuf;
if (stat(PATH, &statbuf) == -1) die("stat"); // oder lstat
if (fstat(fd, &statbuf) == -1) die("stat");

FILE *rx, *tx;
if (!(rx = fdopen(fd, "r"))) die("fdopen");
int fd2 = dup(fd);
if (fd2 == -1) die("dup");
if (!(tx = fdopen(fd2, "w"))) die("fdopen");
```

- ~~Ordner: `opendir, readdir, closedir`~~
- Ordner nebenläufig sicher: `scandir, alphasort`

```c
struct dirent **nameList;
int nDirs = scandir(PATH, &nameList, NULL, alphasort);
struct dirent **nameList;
if (nDirs < 0) die("scandir");
for (int i = 0; i < nDirs; ++i) {
  // Use nameList[i]->d_name;
  free(nameList[i]);
}
free(nameList);
```

~~zusammenhängender Speicher (z. B. Text)~~

- Text: `str{dup,cpy,str,chr,cmp}`
- Speicher: `malloc, free, calloc, realloc, mmap`
- Sortieren: `qsort`

Signale

- `sig{empty,full}set, sigaddset, sigprocmask, sigsuspend`

```c
static void sigchld_handler(int signum) {
  pid_t pid = 0;
  int stat;
  while ((pid = waitpid(-1, &stat, WNOHANG)) > 0) // Use pid, stat
}
struct sigaction act = {
    .sa_handler = sigchld_handler,
    .sa_flags = SA_RESTART};
struct sigaction ign_act = { .sa_handler = SIG_IGN | SIG_DFL };

sigset_t sigchld_set, old_set;
if (sigemptyset(&sigchld_set)) die("sigemptyset");
if (sigaddset(&sigchld_set, SIGCHLD)) die("sigaddset");
```

```c
if (sigaction(SIGCHLD, &act, NULL)) die("sigaction");
if (sigsuspend(&old_set) && errno != EINTR) die("sigsuspend");
if (sigprocmask(SIG_BLOCK, &sigchld_set, &old_set)) die("sigprocmask");
```

Atomare Variablen

- `atomic_{int,long,uint64}_t, atomic_fetch_add, atomic_compare_exchange_strong`

```c
_Atomic int i;
atomic_init(&i, 42);
int old, tmp;
atomic_fetch_add(&i, 1);
do {
  old = atomic_load(&i);
  tmp = (old + 1) % N;
  ...
} while(!atomic_compare_exchange_strong(&i, &old, tmp));
```

Implizit

- `struct`-Initializer-Syntax mindestens einmal verwenden, z. B. bei `sigaction`.

Theorie

Dateisysteme

FAT

RAID n

Aus dem Gedächtnis:

- 0, nur Striping, keine Redundanz, schneller lesen und schreiben 🚀
- 1, Zwei Kopien, wenn eine Platte ausfällt, hat man noch die andere. Schneller lesen
- 4, Eine designierte Paritätsplatte, die mehr belastet wird. Schneller, toleriert einen Ausfall Schneller.
- 5, Paritäts- und Datenblöcke verteilt, schnell, gleichmäßige Ausnutzung. Toleriert einen Ausfall. Schneller.
- 6, Wie 5, nur Paritätsblöcke doppelt. Toleriert zwei Ausfälle. Schneller.

schneller als normal, da mehrererererere festplatten

Scheduler

FCFS

First come first served. Zyklische Warteschlange. Konvoieffekt.

Round-Robin

FCFS mit Zeitscheiben. Konvoieffekt. (Stand vorher bei Dateisystemen) (Es fühlt sich gerade an wie ein StuvePad)

Virtual-Round-Robin

Round-Robin aber Prozesse, die gerade E/A beendet haben, werden bevorzugt. Keine Bevorzugung rechenintensiver Prozesse.

MLQ

Altklausuren

Ws22/23: Was passiert bei Seitenfehler?

- Prozessorzustand sichern
- Prozess auf blockiert setzen
- Einlagern der Seite veranlassen
- Ausführung fortsetzen
- Wenn Seite eingelesen: Prozess auf bereit setzen
- Prozess einlasten, zustand laufend

Ws 21/22: Notwendig für Verklemmung
- iterative Anforderung
- kein Ent 
- Gegenseitiger Ausschluss

Hinreichend:
- zirkuläres Warten

Semaphore: Atomare Operationen auf Zähler
- P: versuche, Zähler zu verringern. Warte, wenn das den Zähler unter 0 bringen würde.
- V: erhöhe den Zähler

externe Fragmentierung: Lücken sind frei, aber nicht vergeben werden, da zu klein. Dagegen: Verschmelzung

interne Fragmentierung: Prozess bekommt mehr Speicher, als er angefragt hat. Rest ist nicht nutzbar. Dagegen: Kleinere Seiten

Problem: Zugriff auf unbelegten Bereich

Journalling: Transaktional, vor Änderung protokollieren, nach Änderung protokollieren. Beim Hochfahren nach Absturz Journal abwickeln und unfertige Transaktionen abschließen/rückgängig machen. ⇒ Konsistenter Zustand, Konsistente Metadaten

SS 21: einseitige Sync.: Ein Prozess ist warten, andere schreiten fort. mehrseitige Sync.: Alle Prozesse bis auf einen warten, reihenfolge unbestimmt.

Wiederverwendbar: Beliebig oft verwendbar, z. B. CPU, Mutex, etc. Konsumierbar: Wird einmal verarbeitet, z. B. Signale, Traps

Compare-And-Exchange-Schleife. Berechne neuen Wert, versuche Wert zu schreiben, beginne erneut, wenn der geteilte Speicher währenddessen nebenläufig verändert wurde.

Vorteil:
- Schneller bei wenigen nebenläufigen Zugriffen
- Benötigt keine Betriebssystemunterstützung

Nachteil:
- Rechenintensiv bei nebenläufiger Veränderung
- Benötigt Hardwareunterstützung

Prozesse:
- eigener Adressraum
- Scheduling-Strategie im Systemkern

Kernel-Threads:
- geteilter Adressraum
- Nachteil: Höhere Kosten zur Erstellung im Vergleich zu User-Threads (Systemaufruf)

User-Threads

- Blockierender Systemaufruf blockiert alle User-Threads
- Scheduling-Strategie im *userspace*