Semaphore: P: (SEM *sem, unsigned n) { pthread_mutex_lock (&sem -> mutex); while (sem -> value < n) { pthread_cond_wait (&sem -> cond, &sem -> mutex); } sem -> value -= n; pthread_mutex_unlock (&sem -> mutex); } **V(SEM *sem, unsigned n)** { pthread_mutex_lock (&sem -> mutex); sem -> value += n; pthread_cond_signal (&sem -> cond); pthread_mutex_unlock (&sem -> mutex);

Parameter verarbeiten: for (int i = 1; i < argc; i++) { myfunction (argv[i]); } **Server Socket (wartet auf eingehende Nachricht):** struct sigaction action = { .sa_handler = SIG_IGN, }; sigemptyset (&action.sa_mask); sigaction (SIGPIPE, &action, NULL); int server_sock = socket (AF_INET6, SOCK_STREAM, 0); if (server_sock == -1) { die ("socket"); } struct sockaddr_in6 name = { .sin6_family = AF_INET6, .sin6_port = htons (portNr), .sin6_addr = in6addr_any }; if (bind (server_sock, (struct sockaddr *) &name, sizeof (name)) == -1) { die ("bind") } if (listen (server_sock, SOMAXCONN == -1) { die ("listen"); } // Infra anand while (1) { int client_sock = accept (server_sock, NULL, NULL); if (client_sock == -1) { perror ("accept"); } else { myfunction (); }}

Thread Pool aufsetzen: pthread_t workers [THREADS]; for (int i = 0; i < THREADS; i++) { int tmp_errno = pthread_create (&workers[i], NULL, &worker, &// Aufgabe z.B. &params); if (tmp_errno != 0) { errno = tmp_errno; die ("pthread create"); }} worker function: returns NULL

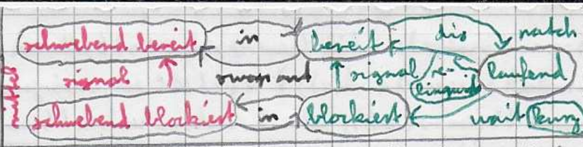
Signalbehandlung: struct sigaction = { .sa_handler = &my function oder SIG_IGN, .sa_flags = SA_RESTART, }; sigemptyset (&action.sa_mask); sigaction (SIGINT, &action, NULL); **Passives warten auf SIGINT:** sigset_t mask; sigfillset (&mask) sigdelset (&mask, SIGINT); sigsuspend (&mask) **Auf Threads warten:** for (int i = 0; i < THREADS; i++) { int tmp_errno = pthread_join (threads[i]); if (tmp_errno != 0) { errno = tmp_errno; die ("pthread_join"); } **Listen Socket:** int listenSock = socket (AF_INET6, SOCK_STREAM, 0); if (listenSock < 0) die ("socket"); struct sockaddr_in6 name = { .sin6_family = AF_INET6, .sin6_port = htons (port), .sin6_addr = in6addr_any }; if (bind (listenSock, (struct sockaddr *) &name, sizeof (name)) < 0) die ("bind"); if (listen (listenSock, SOMAXCONN) < 0) die ("listen"); return listenSock; **FILE erzeugen:** r = read w = write FILE *rx = fdopen (Socket, "r") if (rx == NULL) { die ("fdopen"); } // falls sowohl r als auch w notwendig: int Socket2 = dup (Socket) if (Socket2 < 0) die ();

FILE löschen: fclose (rx) **Zeilen von FILE einlesen:** char buf [n]; while (fgets (buf, sizeof (buf), stream) != NULL) { size_t len = strlen (buf); if (len > sizeof (buf) - 2 && buf [len-1] != "\n") { int c; while ((c = fgetc (stream)) != EOF) { if (c == '\n') { break; }} if (ferror (stream)) { die ("fgets"); }} if (buf [len-1] == '\n') buf [len-1] = '\0'; } // Zeile in s verarbeiten (s = char *s) **Client Socket DNS Anfrage + verbinden mit URL:** char *url; char *port struct addrinfo hints = { .ai_socktype = SOCK_STREAM, ai_family = AF_UNSPEC, ai_flags = AI_ADDRCONFIG, }; struct addrinfo *head; int ret_addrinfo = getaddrinfo (url, port, &hints, &head); if (ret_addrinfo == EAI_SYSTEM) { die ("getaddrinfo"); } else if (ret_addrinfo != 0) { fprintf (stderr, "getaddrinfo error: %s\n", gai_strerror (ret_addrinfo)); exit (EXIT_FAILURE); } int sock; struct addrinfo *curr; for (curr = head; curr != NULL; curr -> curr -> ai_next) { sock = socket (curr -> ai_family, curr -> ai_socktype, curr -> ai_protocol); if (sock == -1) { perror ("socket") continue; } int ret_con = connect (sock, curr -> ai_addr, curr -> ai_addrlen); if (ret_con == -1) (perror ("connect")) else if (ret_con == 0) { break; } else if (close (sock) == -1) { perror ("close"); }}} if (curr == NULL) { fprintf ("keine Adressen gefunden\n"); exit (EXIT_FAILURE); } **Jobprozess SIGKILL schicken** void * timeout (void * void Job) JobInfo *jobinfo = (JobInfo *) voidJob; sleep (10); if (kill (jobInfo -> pid, SIGKILL) == -1) { perror ("kill"); pthread_exit (NULL); } **Ganzes Directory / Verzeichnis durchgehen:** char *path = "." DIR *dir = opendir (path); if (dir == NULL) { die ("opendir"); } struct dirent *dirent; while (errno = 0, (dirent = readdir (dir)) != NULL) { if (strcmp (dirent -> d_name, ".") == 0 || strcmp (dirent -> d_name, "..") { continue; } char npath [strlen (path) + strlen (dirent -> d_name) + 2]. if (sprintf (npath, "%s/%s", path, dirent -> d_name) < 0 { perror ("sprintf"); continue; } struct stat info; if (lstat (npath, &info) == -1 { perror ("lstat") continue; } if (S_ISREG (info.st_mode)) { // regular file } else if (S_ISDIR (info.st_mode)) { // directory }} if (errno) { die ("readdir"); } if (closedir (dir) == -1 { die ("closedir"); }

CAS: atomic_int a = ATOMIC_VAR_INIT (10); int old, local; do { old = atomic_load (&a); local = old - 2; } while (! atomic_compare_exchange_strong (&a, &old, &local)); **Warten auf bestimmten Child:** pid_t pid = fork (); ... int wstatus; while (waitpid (pid, &wstatus, 0) != -1) { if (WIFEXITED (wstatus) || WIFSIGNALED (wstatus)) { return; }} perror ("waitpid"); **Warten auf alle Childs:** int status; pid_t child = waitpid (-1, &status, WNOHANG); if (child == -1) { perror ("waitpid"); else if (child == 0) { // Childs nicht terminiert } else { ... // childs done } // extra Status ausgeben: if (! WIFEXITED (status)) { if printf (stderr, "%i\n", WEXITSTATUS (status)) < 0) { perror ("fprintf"); }}} **FORK:** pid_t p = fork (); if (p == -1) { die ("fork"); } else if (p == 0) { // Kind } else { // Elternprozess } **Pattern Matching:** int fn_res = fnmatch ("*.txt", basename (path), FNM_PERIOD); if (!fn_res) { // path ends with .txt } else if { fn_res == FN_NOMATCH } { // no match } { else die () } **Input mit Trennzeichen trennen:** char input [ ] char delim [ ] = "-" // Trennzeichen hier - char * args [strlen (input) / 2 + 1]; args [0] = strtok (input, delim); int index = 1; while ((args [index++] = strtok (NULL, delim)) != NULL); **Pointer zum 1. Char eines Strings:** if (strchr (mytext, '\n') ! null) {...} **A an B anhängen (Strings):** char full_path (strlen (path) + strlen (address) + 1]; strcpy (full_path, path); strcat (full_path, address); **Makefile:** CC = gcc CFLAGS = ...Phony: all clean all: patriots clean: rm -f patriots patriots.o %.o: %.c $(CC) $(CFLAGS) -c -o $@ $<

%: %.o $(CC) $(CFLAGS) -o $@ $^ $< patriots.o : patriots.c triangle.h bbuffer.h

patriots : patriots.o triangle.o bbuffer.o sem.o



schwebend bereit: Exemplar des Prozess ausgelagert,
schwebend blockiert: ausgelagert, wartet auf Ereignis
bereit zur Ausführung laufend hat CPU bekommen
blockiert wartet auf bestimmtes Ereignis

Einseitige synchro: Wirkung auf einen Prozess läuft normal weiter Mehrseitig: Wirkung auf alle Prozesse erfolgt logisch oder bedingt / konsumierbare Betriebsmittel | blockierend oder nicht / wiederverwendbar

Betriebsmittel wiederverwendbar: Begrenzter Zugriff / existieren dauerhaft / teil oder unteilbar / belegt, benutzt, freigegeben z.B. CPU / RAM / kritischer Abschnitt Konsumierbar: unbegrenzt Zugriff / existieren vorrübergehend / produziert, empfangen, benutzt, zerstört z.B. Signale, Traps

Verklemmung: beteiligte Zustände warten wechselseitig auf Bedingung, die nur von anderen Prozessen in der Gruppe hergestellt werden können. Notwendig: Ausschließlichkeit in Betriebsmittelnutzung / Nachforderung von Betriebsmittel / Unentziehbarkeit von Betriebsmittel / Hinreichend: zirkuläres warten

Optimistische synchro: Vorteil + Parallel + Verklemmungsvorbeugung Nachteil: ABA Problem komplexe Algorithmen Bsp: Lesen aus Ringbuffer mit read index, dieser muss mit sync compare bool and swap atomar geprüft und geändert werden blockierend: Semaphoren / kritischer Abschnitt / - Effizienz Verklemmungsgefahr. aktives warten: beim warten nachfragen, keine Unterstützung durch BS notwendig CPU Zeitverschwendung bei Umschaltzeit < erwartete Wartezeit passiv: schläft bis zur Benachrichtigung

Semaphore: ganzzahlige Variable P verringert Wert um 1 falls Wert davor 0: Prozess blockiert / V: Wert+1 zufälliger wartender Prozess wird aktiviert / logisch und physisch jeweils unteilbar

interne Fragmentierung: angeforderte Größe < bekommene Größe -> ungenutzter Speicher / BS machtlos hier
externe Fragmentierung: angeforderter Speicher > größter freier Speicherbereich -> eig freier Speicher (fast) nie benutzt / aufwendig vermeidbar z.B. verschmelzen 1.A. Nelsoneffekt: Zugriff auf den zu viel belo. Speicher

Speicherzugriff Auslagerung erkennen: MMU liest beim Deskriptor das Presentbit aus (Bit nicht gesetzt) (Pagefault)
1. MMU löst Trap aus 2. BS löst Handler für Pagefault aus 3. Freien Speicher suchen / reservieren od. andere Seite auslagern 4. Seiteneinlagerung starten 5. Prozesszustand blockiert 6. einlagern erfolgt -> Interrupt 7. Präsenzbit auf 1 setzen 8. Prozesszustand bereit 9. Nach Einladung: Befehl der Trap auslöst wiederholen

Benutzerorientierte Kriterien: Vom Nutzer wahrgenommenes Verhalten (akzeptanz) z.B. Antwortzeit minimieren / Durchlauf: Minimale Zeit zwischen Prozess Start / Ende / vorhersagbarkeit unabhängig von Systemlast systemorientiert: effizient und effektive Auslastung der Betriebsmittel z.B. Durchsatz möglichst viele Prozesse pro Zeit beenden / Gerechtigkeit: Gleichberechtigung der Prozesse / Dringlichkeit: Bevorzugung Prozesse hoher Priorität

kooperative Einplanung: Planung abhängiger Prozess, CPU wird Prozessen nicht entzogen, laufender Prozess gibt CPU nur mit Systemaufruf ab z.B. First Come First Serve bei: Batch Betrieb präemtive Einplanung: BS entzieht Prozessen die CPU z.B. Mehrbenutzer / Timesharing Systemen z.B. RR

User Threads: extrem billig / Systemkern ~~kennt sie nicht~~ hat kein Wissen über sie / Ein user Thread blockiert -> alle blockiert / Multiprozessorsysteme keine parallelen Abläufe / Scheduling schwierig
Kernel Threads: billig / Gruppe von Threads nutzt zusammen Betriebsmittel eines Prozesses / jeder Thread ist BS separat bekannt / großer overhead / besseres Scheduling / langsamer / ineffizient
Prozess: keine Parallelen Abläufe innerhalb eines logischen Adressraums auf Multiprozessorsystem fork Operation verbraucht unnötig viel Systemressourcen / generell teuer

reelle Adressraum: physische Speicheradressen im Hauptspeicher durch Hardware begrenzt logischer / virtueller Adressraum: Adressen vom Programm verwendet, CPU erstellt diese werden mit MMU zu reelen umgewandelt
Verklemmungsvorbeugung: 1von: nicht blockierende Synchro / Alle Betriebsmittel Atomar anfordern / BM mit Ordnung zuteilen / BM virtualisieren | bietet Schutz und Sicherheit Verklemmungsverhinderung: verhindert zirkuläres warten vor jeder BM zuteilung: Prüfe ob Verklemmung entsteht / Bankiers Algorithmus

Strategien Seitenverdrängung: Least frequently Used: ersetzt wird die am seltesten referenzierte Seite first in first out: Ersetzt wird die zuerst eingelagerte noch verbleibende Seite

Monitor: abstrakte Datentyp Zugriffsoperatoren implizit synchronisiert. Mehrseitige synchro an Schnittstelle / einseitig innerhalb Hansen: blockierende Bedingungsvariablen / Monitor verlassen nachdem Geber alle nehmer bereit gesetzt hat mit Hoare: wie Hansen aber verlassen nach genau einem fortfahren von Signalnehmer. Mesa: nicht block / beliebig

Seitentabellen: 1. pro Stufe 8 bit 12 Hex => offset letzte 4 Hex 2. In Seitentabelle 1 die ersten 2 Hex suchen zur entsprechenden Tabelle gehen 3. Dort Hex 3/4 suchen und dann gefundene Adresse + Offset berechnen

INODES: 1. Jede INODE Nummer bekommt Eintrag rechts hinter mit 1. Spalte 3. SP 5. SP Beachte: keine Doppelung bei St. ino 2. jeder Ordner erhält Datenblock, zeilenweise Einträge der Form <inode num> <name> analog zu ls output ~~erhält~~ enthält 3. Reguläre Dateien bekommen? 4. symbolische Links (Pfeil bei ls output) bekommen Datenblock der rechten Pfeil ... enthält Ordner: 1. Spalte startet mit d Datei: 1. SP "-"

Dateisystem

Raid 0: Daten auf mehreren Platten speed / kein save

Raid 1: Daten auf 2 Platten 1 Platte kann ausfallen / + lesen aber -2 Speicher

Raid 4: Daten auf vielen Platten 1 Paritätsplatten XOR 1 Platte ausfall + lesen Plattenzahl beliebig > 3 Schreiben -> Paritätsplatten Änderung -> hohe Belastung

Raid 5: wie 4 aber verteilte Paritätsblöcke -> last gut verteilt

Raid 6: 2 Paritätsblöcke -> Ausfall 2 Platten möglich

Journaling File Für Einzelvorgang einer Transaktion: Logeintrag dann Änderung am Dateisystem -> Bei Boot prüfe ob geloggte Änderungen vorhanden falls nein undo / redo + Speed Absturz - Speed allgemein

CHDCK: MMU + Präsenzbit notwendig

Zugriff ungültige virt Seite: Segfault

Traps: interne Ursache / synchron / vorhersagbar / reproduzierbar / Bug in Code Interrupt: externe Ursache / asynchron / unvorhersagbar / nicht reproduzierbar Grund: E/A Operation

Seitenflattern: ständiges ein und auslagern von Seiten wenn Hauptspeicher zu klein blockiert Prozessor

Mantelprozeduren: 1. Zustand speichern 2. Aktivierung von Unterbrechungen 3. run Unterbrechungsbehandlungsroutine 4. Zustand wiederherstellen 5. Rückkehr zur Unterbrechung

malloc (999999) Fehler malloc gibt Null / ENOMEM out of memory nicht genug speicher für pointer) MMU sendet trap an BS / BS sendet signal an Prozess (SIGSEGV)

Echtzeitbetrieb: wichtig: Dringlichkeit, termineinhaltung, vorhersagbarkeit Konflikt: Gerechtigkeit, Lastverteilt

Nicht optimale Ersetzungsstrategien: Lokalitätsprinzip Sequentielle Befehle nacheinander gespeichert