Praktikum ProcThreads

Prozesse, Threads und Dämonen

Frühlingssemester 2012
M. Thaler

Inhalt

1. Einleitung

1.1 Ziele

In diesem Praktikum werden Sie sich mit Prozessen, Prozesshierarchien, Dämonen und Threads beschäftigen. Sie erhalten einen vertieften Einblick und Verständnis für die Erzeugung, Steuerung und Terminierung von Prozessen unter Unix/Linux und Sie werden die unterschiedlichen Eigenschaften von Prozessen und Threads kennenlernen. **Wichtig:** Wir werden selbstverständlich nicht alle Aspekte im Zusammenhang mit Prozessen behandeln können und verweisen auf die entsprechende Literatur, z.B. Helmut Harold [1].

1.2 Organisatorisches

Das Praktikum besteht aus einer Anzahl Aufgabenstellungen die Sie durcharbeiten werden. `Die Programme finden Sie beim` **`Ihrem Dozenten`** auf der Homepage oder auf Olat.

Laden Sie die Datei in Ihr Arbeitsverzeichnis. Die Datei ist mit `tar` archiviert und mit `zip` komprimiert. Die Datei können Sie mit **`tar -xvzf ProcThreads.tar.gz`** auspacken. In Ihrem Arbeitsverzeichnis wird der Ordner `ProcThreads` angelegt. Die Programme zu den einzelnen Aufgabenstellungen finden Sie im Verzeichnis `./ProcThreads`. Die Programmlistings sind auch den Aufgabenstellungen beigefügt. Beantworten Sie zuerst die Fragen und lassen Sie die Programme erst laufen, wenn Sie dazu in der Aufgabenstellung aufgefordert werden (nur so ist der Lerneffekt optimal). Implementieren resp. erweitern Sie die Programme, verwenden Sie dazu die mitgelieferten makefiles. Bei Problemen stehen wir Ihnen jederzeit gerne zur Verfügung.

1.3 Durchführung und Leistungsnachweis

Es gelten grundsätzlich die Vorgaben Ihres Dozenten zur Durchführung der Praktika und zu den Leistungsnachweisen im Kurs BSy .
Die Inhalte des Praktikums gehören zum Prüfungsstoff.

1.4 Aufbau des Praktikums

In Abschnitt 2 finden Sie die Aufgabenstellungen. Wir schlagen aber vor, dass Sie zuerst den Überblick zu Prozessen in Anschnitte 3 durchlesen, kurz die Systemfunktionen in Abschnitt 4 überfliegen und dann zu den Aufgabenstellungen übergehen. Beim Durcharbeiten der Aufgaben können Sie dann auf die einzelnen Funktionen zurückkommen.

1.5 Zu Beachten

Sie arbeiten hier mit Systemaufrufen, die jederzeit aus irgendwelchen Gründen nicht erfolgreich abgeschlossen werden können. Guter Programmierstil verlangt (speziell im Zusammenhang mit Systemprogrammierung), dass Sie die Rückgabewerte der System Calls überprüfen und falls notwendig eine entsprechende Fehlermeldung ausgeben. Dazu steht die Systemfunk-

tion `perror("MyMessage")` zur Verfügung. Diese Funktion gibt zuerst "`MyMessage`" aus, gefolgt vom Grund für den Fehler. Beachten Sie in diesem Zusammenhang das Vorgehen in unseren Programmbeispielen: wir wenden dieses Vorgehen aber nicht überall konsequent an, um die Übersichtlichkeit und Lesbarkeit Programme nicht allzusehr zu beeinträchtigen.

Grundsätzlich verwenden wir den C++-Compiler (g++) für die Übersetzung der Programme (restriktivere Typenüberprüfung), die Beispielprogramme in diesem Praktikum sind aber im C-Stil geschrieben.

2. Aufgabenstellungen

Zu jeder Aufgabenstellung finden Sie im Verzeichnis `./ProcThreads` entsprechende Unterverzeichnisse (`./Aufgabe_4.1 ... ./Aufgabe_4.11`) mit den jeweils aufgeführten Dateien.

2.1 Aufgabe 1: Anzeige der Betriebssystemressourcen mit top

Ziele

- Das Systemprogramm **`top`** kennen und bedienen lernen
- Das Konfigurationsfile für **`top`** einrichten

Aufgaben

- Starten Sie in einem beliebigen Fenster **`top`**: es werden sämtliche laufenden Prozesse angezeigt.
- Geben Sie **`u`** (username) ein, dann Ihren Username: es werden nur Ihre eigenen Prozesse angezeigt. Wenn Sie `top` mit **`top –u $USER`** starten, werden ebenfalls nur Ihre Prozesse angezeigt.
- Wenn Sie den Befehl **F** eingeben, zeigt **`top`** eine Liste, in der das Sortierkriterium für die Liste gewählt werden kann (default: CPU Zeit). Eingabe des entsprechenden Buchstabens wählt das neue Sortierkriterium: wählen Sie ein **`a`** (sortieren nach PID). Mit Eingabe von RETURN verlassen Sie das Menu.
  Hinweis: die Sortierung erfolgt defaultmässig in absteigender Richtung (das gilt auch für alphabethische Grössen), durch Eingabe eines **R** im Hauptmenu kann diese Reihenfolge umgekehrt werden.
- Wenn Sie im Hauptfenster den Befehl **f** zeigt **`top`** eine Liste an, in der Sie die Paramter wählen können, die Sie gerne anzeigen möchten.
- Mit **`h`** (help) können Sie eine Liste der unterstützten Befehle abfragen.
- Verlassen Sie `top` mit **`q`** (quit) und kopieren Sie die Datei `.toprc` mit dem vordefinierten Befehl `setToprc` in Ihr Home Direcory. Die Datei `.toprc` enthält Formatierungsanweisungen für die Ausgabe auf dem Bildschirm. Starten Sie `top` erneut: die Tabelle hat nun weniger Einträge, ist aber für unsere Zwecke übersichtlicher. Die Datei lässt sich mit dem Befehl `delToprc` entfernen.
- **`top`** ist nun so konfiguriert, dass z.B. auch die Threads angezeigt werden und neben der prozentuellen Auslastung der CPU(s) auf welchem Core der Job läuft.

Programme/Dateien

```
setToprc:      cp –i toprc ~/.toprc

delToprc:      rm –i ~/.toprc
```

2.2 Aufgabe 2: Prozesse erzeugen mit fork(): was läuft ab?

Ziele

- Verstehen, wie mit `fork()` Prozesse erzeugt werden
- Einfache Prozesshierarchie kennenlernen
- Verstehen, wie ein Programm, das `fork()` aufruft, durchlaufen wird

Aufgaben

- Studieren Sie zuerst das Programm ProcA2.c und versuchen Sie zu verstehen was geschieht.
- Schreiben Sie den Ablauf (was wird ausgegeben) auf. Starten Sie nun das Programm und vergleichen Sie die Ausgabe mit ihren Notizen?
- Was ist gleich, was anders... wieso?

Programm (ohne header)

Datei: ProcA2.c

```
int main(void) {

    pid_t  pid;
    int    status;
    int    i;

    i = 5;

    printf("\n\ni vor fork: %d\n\n", i);

    pid = fork();
    switch (pid) {
      case -1:
        perror("Could not fork");
        break;
      case 0:
        i++;
        printf("\n... ich bin das Kind %d mit i %d, ", getpid(),i);
        printf("meine Eltern sind %d \n", getppid());
        break;
      default:
        i--;
        printf("\n... wird sind die Eltern %d mit i %d ", getpid(), i);
        printf("und Kind %d,\n   unsere Eltern sind %d\n", pid, getppid());
        wait(&status);
        break;
    }
    printf("\n. . . . . und wer bin ich ?\n\n");
    exit(0);
}
```

2.3 Aufgabe 3: Prozess erzeugen und ausführen mit execl()

Ziele

- An einem Beispiel die Funktion `execl()` kennenlernen
- Verstehen, wie nach `fork()` ein neues Programm gestartet wird

Aufgaben

- Studieren Sie zuerst die Programme ProcA3.c und ChildProcA3.c.
- Starten Sie ProcA3.e und vergleichen Sie die Ausgabe mit der Ausgabe unter Aufgabe 2.2. Diskutieren und erklären Sie was gleich ist und was anders.
- Benennen Sie **ChildProcA3.e** auf **ChildProcA3.f** um (Shell Befehl `mv`) und überlegen Sie sich, was das Programm nun ausgibt,. Starten Sie ProcA3.e und vergleichen Sie Ihre Überlegungen mit der Programmausgabe.
- Nennen Sie das das Kindprogramm wieder ChildProcA3.e und geben Sie folgenden Befehl ein: `chmod -x ChildProcA3.e`. Starten Sie ProcA3.e und analysieren Sie die Ausgabe von `perror("...")` aus? Wieso verwenden wir `perror()`?

Programme (ohne Header)

Datei: ProcA3.c

```
int main(void) {

    pid_t  pid;
    int    status;
    int    i, retval;
    char   str[8];

    i = 5;
    printf("\n\ni vor fork: %d\n\n", i);

    pid = fork();
    switch (pid) {
      case -1:
        perror("Could not fork");
        break;
      case 0:
        i++;
        sprintf(str, "%d",i);    // convert integer i to string str
        retval = execl("./ChildProgA3.e", "ChildProgA3.e", str, NULL);
        if (retval < 0) perror("execl not successful");
        break;
      default:
        i--;
        printf("\n... wir sind die Eltern %d mit i %d ", getpid(), i);
        printf("und Kind %d,\n   unsere Eltern sind %d\n", pid, getppid());
        wait(&status);
        break;
    }
    printf("\n. . . . . und wer bin ich ?\n\n");
    exit(0);
}
```

Datei: ChildProcA3.c

```
int main(int argc, char *argv[]) {

      int   i;

      if (argv[1] == NULL) {
            printf("argument missing\n");
            exit(-1);
      }
      else
            i = atoi(argv[1]);  // convert string argv[1] to integer i
                                // argv[1] is a number passed to child

    printf("\n... ich bin das Kind %d mit i %d, ", getpid(),i);
    printf("meine Eltern sind %d \n", getppid(), i);
    exit(0);
}
```

2.4 Aufgabe 4: Prozesshierarchie: was fork() alles kann

Ziele

- Verstehen, was fork() wirklich macht
- Verstehen, was Prozesshierarchien sind

Aufgaben

- Studieren Sie zuerst Programm ProcA4.c und zeichnen Sie die entstehende Prozesshierarchie (Baum) von Hand auf. Starten Sie das Programm und verifizieren Sie ob Ihre Prozesshierarchie stimmt.
- Mit den Befehlen **`ps f`** resp. **`pstree`** können Sie die Prozesshierarchie auf dem Bildschirm ausgeben. Damit die Ausgabe von **`pstree`** übersichtlich ist, müssen Sie in dem Fenster, wo Sie das Programm **ProcA4.e** starten, zuerst mit dem Befehl **`ps`** die **`PID`** der Shell (sehr wahrscheinlich ist das die **`bash`**) festellen. Wenn Sie dann den Befehl **`pstree PID`** in einem beliebigen Fenster eingeben, wird nur die Prozesshirarchie von **`PID`** ausgehend angezeigt.
  Hinweis alle erzeugten Prozesse müssen *arbeiten*, damit die Darstellung gelingt, wie wird das in unten stehendem Programm erreicht?

Programm (ohne Header)

`Datei: ProcA4.c`

```
int main(void) {
    fork();
    fork();
    fork();
    fork();
    printf("PID: %d\t PPID: %d\n", getpid(), getppid());
    sleep(10);  // keep processes in system to display their "stammbaum"
    exit(0);
}
```

2.5 Aufgabe 5: Zeitlicher Ablauf: wer macht wann was?

Ziele

- Verstehen, wie Kind- und Elternprozess zeitlich synchronisiert sind

Aufgaben

- Studieren Sie Programm ProcA5.c. Starten Sie nun mehrmals hintereinander das Programm ProcA5.e und vergleichen Sie die jeweiligen Outputs (Leiten Sie dazu die Ausgabe auf verschiedene Dateien um). Was schliessen Sie aus dem Resultat?
  Anmerkung: `startWorker()` erzeugt einen Prozess, der CPU-Zeit konsumiert, siehe Modul `workerUtils.c`.

Programm (ohne Header)

Datei: ProcA5.c

```
#include "workerUtils.h"

#define HARD_WORK 20000000
#define ITERATIONS 20

//***************************************************************************
// Function: main(), parameter: none
//***************************************************************************

int main(void) {

    pid_t  pid, worker1, worker2;
    int    i;

    worker1 = startWorker();   // start CPU load -> worker processes with
    worker2 = startWorker();   // randomized load to force context switches
    pid = fork();
    switch (pid) {
      case -1:
        perror("Could not fork");
        break;
      case 0:
        for (i = 0; i < ITERATIONS; i++) {
            justWork(HARD_WORK);
            printf("%d \t\tChild\n", i);
        }
        break;
      default:
        for (i = 0; i < ITERATIONS; i++) {;
            justWork(HARD_WORK);
            printf("%d \tMother\n", i);
        }
        stopWorker(worker1);        // stop (kill) worker process 1
        stopWorker(worker2);        // stop (kill) worker process 2
        break;
    }
    printf("I go it ...\n");
    exit(0);
}
```

2.6 Aufgabe 6: Was geschieht mit verwaisten Kindern?

Ziele

- Verstehen, was mit verwaisten Kindern geschieht
- oder wie werden die Kinder "erwachsen"?

Aufgabe

- Studieren Sie Programm ProcA6.c.
- Starten Sie nun ProcA6.e: der Elternprozess terminiert: was geschieht mit dem Kind?
- Was geschieht, wenn der Kindprozess vor dem Elternprozess terminiert? Ändern Sie dazu im **`sleep()`** Befehl die Zeit von 2s auf 12s und verfolgen Sie mit `top` das Verhalten der beiden Prozesse.

Programm (ohne Header)

`Datei: ProcA6.c`

```
int main(void) {

    pid_t  pid;
    int    i;

    pid = fork();
    switch (pid) {
      case -1:
        perror("Could not fork");
        break;
      case 0:
        printf("\n... ich bin das Kind \n", getpid());
        for (i = 0; i < 10; i++) {
            usleep(500000);                         // slow down a bit
            printf("Mein Elternprozess ist %d\n", getppid());
        }
        printf("... so das wars\n");
        break;
      default:
        sleep(2);
        exit(0);                                    // terminate
        break;
    }
}
```

2.7 Aufgabe 7: Zombies[1] auch in Unix . . .?

Ziel

- Verstehen, was ein Zombie ist
- Möglichkeit kennenlernen, um Zombies zu verhindern

Aufgaben

- Studieren Sie das Programm ProcA7.c
- Starten Sie **`top -d 1`**, in einem Fenster das immer gut sichtbar ist. Zeigen Sie ausschliesslich Ihre eigenen Prozesse an (Befehl u, dann Ihr username). Geben Sie nun als Befehl den Grossbuchstaben **F** ein, damit wird ein neues Fenster geöffnet, wo das Sortierkrterium für die top-Tabelle gewählt werden kann. Wählen Sie als Sortierkriterium die PID (Default ist die konsumierte CPU-Zeit): unter Linux müssen Sie dazu ein kleines **a** und dann **RETURN** eingeben.
- Starten Sie `eeeeeeee.e` und verfolgen Sie im "top-Fenster" was geschieht. Der etwas seltsame Programmname hilft bei der Visualisierung.
- Manchmal möchte man nicht auf die Terminierung eines Kindes warten (mit **`wait()`**, resp. **`waitpid()`**). Überlegen Sie sich, wie Sie in diesem Fall verhindern können, dass ein Kind zum Zombie wird.

Programm

Datei: ProcA7.c

```
int main(void) {
      pid_t pid;
      int   j;
      for (j = 0; j < 3; j++) {          // generate 3 processes
            pid = fork();
            switch (pid) {
              case -1:
                  perror("Could not fork");
                  break;
              case 0:
                  sleep(j+2);             // process j sleeps for j+2 sec
                  exit(0);                // then exits
                  break;
              default:                    // parent
                  break;
            }
      }
      sleep(6);   // parent process sleeps for 6 sec
      wait(NULL); // consult manual for "wait"
      sleep(2);
      wait(NULL);
      sleep(2);
      wait(NULL);
      sleep(2);
      exit(0);
}
```

[1] Leider werden in neueren Linux-Versionen Zombi-Prozesse nur noch als "defunct" markiert.

2.8 Aufgabe 8: Prozessräume und was sie nach dem fork()'en enthalten

Ziele

- Verstehen, wie Prozessräume vererbt werden
- Unterschiede zwischen dem Prozessraum von Eltern und Kindern erfahren

Aufgaben

- Studieren Sie Programm ProcA8_1.c und überlegen Sie, wie die Ausgabe des Programms aussieht
  - Starten Sie ProcA8_1.e und überprüfen Sie Ihre Überlegungen.
  - Waren Ihre Überlegungen richtig? Falls nicht, was haben Sie falsch gemacht?
  - Untersuchen Sie die Ausgabe, wenn Kind und Eltern in verschiedener Reihenfolge ausgeführt werden (verwenden Sie dazu `wait()` und `sleep()`): ändert sich etwas?
- Studieren Sie das Programm ProcA8_2.c und überlegen Sie, wie die Ausgabe in der Datei `./AnyOutPut.txt` aussieht. Wer schreibt alles in diese Datei (vor `fork()` geöffnet), wieso ist das soo?).
  - Starten Sie ProcA8_2.e und überprüfen Sie Ihre Überlegungen.
  - Waren Ihre Überlegungen diesmal auch richtig? Falls nicht, was haben Sie falsch gemacht?
  - Untersuchen Sie die Ausgabe, wenn Kind und Eltern in verschiedener Reihenfolge ausgeführt werden (verwenden Sie die System Calls `wait()` und `sleep()`): ändert sich etwas?
  - Kind und Elternprozess seien rechenintensive Prozesse (dies kann durch Einfügen eines leeren for-statements mit 10^6-10^8 Iteration simuliert werden). Starten Sie in einem zusätzlichen Fenster `top` mit einem Delay von 1s (dies simuliert einen hochprioritären Prozess, der Ihre Prozesse sporadisch unterbricht). Untersuchen Sie nun die Ausgabe des Programms: starten Sie dazu das Programm mehrmals hintereinander (ev. mit einer verschiedenen Anzahl von Iterationen: `ANZAHL`). Was stellen Sie fest?
- Studieren Sie nun Programm ProcA8_3.c und überlegen Sie wieder, wie die Ausgabe aussieht
  - Starten Sie ProcA8_3.e und überprüfen Sie Ihre Überlegungen.
  - Waren Ihre Überlegungen diesmal auch richtig? Falls nicht, was haben Sie falsch gemacht?

Programme (ohne Header)

nächste Seite

Datei: Proc8_1.c

```
// globaler array

#define ARRAY_SIZE 8
char GArray[ARRAY_SIZE][ARRAY_SIZE];

int main(void) {

    pid_t  pid;
    int    i,j;


    for (i = 0; i < ARRAY_SIZE; i++)
        for (j = 0; j <  ARRAY_SIZE; j++)
            GArray[i][j] = '-';

    printf("Array vor fork()\n\n");
        for (i = 0; i < ARRAY_SIZE; i++) {
            for (j = 0; j <  ARRAY_SIZE; j++)
                printf("%c ", GArray[i][j]);
        printf("\n");
    }

    pid = fork();
    switch (pid) {
      case -1:
            perror("Could not fork");
            break;
      case 0:   // --- child fills upper half of array ---
            for (i =  ARRAY_SIZE / 2; i < ARRAY_SIZE; i++)
                for (j = 0; j <  ARRAY_SIZE; j++)
                    GArray[i][j] = 'c';
            break;
      default:  // --- parent fills lower half of array ---
            for (i =  0; i < ARRAY_SIZE / 2; i++)
                for (j = 0; j <  ARRAY_SIZE; j++)
                    GArray[i][j] = 'p';
        break;
    }

    if (pid == 0)
        printf("\nKinderarray\n\n");
    else
        printf("\nElternarray\n\n");
    for (i = 0; i < ARRAY_SIZE; i++) {
        for (j = 0; j <  ARRAY_SIZE; j++)
            printf("%c ", GArray[i][j]);
        printf("\n");
    }
    exit(0);
}
```

Datei: ProcA8_2.c

```
#include "workerUtils.h"

#define ANZAHL      15
#define WORK_HARD   1000000

//*************************************************************************
**
// Function: main(), parameter: none
//*************************************************************************
**

int main(void) {

    FILE   *fdes;
    pid_t  pid, worker;
    int    i,j;

    worker = startWorker();         // start CPU load to force context
                                    // switches
    fdes = fopen("AnyOutPut.txt", "w");
    if (fdes == NULL) perror("Cannot open file");

    usleep(500000);

    pid = fork();
    switch (pid) {
        case -1:
            perror("Could not fork");
            break;
        case 0:
            for (i = 1; i <= ANZAHL; i++) {
                fprintf(fdes, "Fritzli\t%d\n", i);
                fflush(fdes);       // make sure date is written to file
                justWork(WORK_HARD);
            }
            break;
      default:
            for (i = 1; i <= ANZAHL; i++) {
                fprintf(fdes, "Mami\t%d\n", i);
                fflush(fdes);       // make sure date is written to file
                justWork(WORK_HARD);
            }
            wait(NULL);
            stopWorker(worker);
            break;
    }
    printf("We are done\n");
    exit(0);
}
```

Datei: ProcA8_3.c

```
int main(void) {

    pid_t    pid;

    printf("\n");
    printf("\nHallo, I am on the way to fork now, ......lo");

    pid = fork();
    switch (pid) {
        case -1:
            perror("Could not fork");
            break;
        case 0:
            printf("ok: I am the child\n");
            break;
        default:
            printf("ok: I am the parent\n");
            wait(NULL);
            break;
    }
    printf("\nclear ?\n\n");
    exit(0);
}
```

2.9 Aufgabe 9: Threads, was ist anders?

Ziele

- Den Unterschied zwischen Thread und Prozess kennenlernen
- Problemstellungen um Threads kennenlernen
- Linuxspezifische Implementation kennen lernen

Aufgaben

- Studieren Sie Programm ProcA9.c und überlegen Sie sich, wie die Programmausgabe aussieht. Vergleichen Sie Ihre Überlegungen mit denjenigen bei Aufgabe 4.8 zum Programm ProcA8_1.e
  - Starten Sie ProcA9.e und vergleichen das Resultat mit Ihren Überlegungen.
  - Was ist anders als bei Programm ProcA8_1.e?
- Setzen Sie in der Thread-Routine vor `pthread_exit()` eine unendliche Schleife ein, z.B. `while(1){};`
  - Starten Sie das Programm und beobachten Sie das Verhalten mit `ps -f` und `top`. Was beobachten Sie und was schliessen Sie daraus? Hinweis: wenn Sie in `top` den Buchstaben `H` eingeben, werden auch die Threads dargestellt.
  - Kommentieren Sie im Hauptprogram to beiden `pthread_join()` aus und fügen Sie dafür ein `sleep(5)` ein: start Sie das Program. Was geschieht? Erklären Sie das Verhalten

Datei: ProcA9.c

```
// globaler array
#define ARRAY_SIZE 8
char    GArray[ARRAY_SIZE][ARRAY_SIZE];

void *ThreadF(void *letter) {
    int    i,j;
    int    LowLim, HighLim;
    char   letr;

    letr =  *(char *)letter;
    if ( letr == 'p') {     // paremeter = p: fill lower half of array
        LowLim = 0; HighLim = ARRAY_SIZE / 2;
    }
    else {            //  paremeter != p: fill upper half of array
        LowLim = ARRAY_SIZE / 2; HighLim = ARRAY_SIZE;
    }
    for (i = LowLim; i < HighLim; i++) {    // fill corresponding half
        for (j = 0; j <  ARRAY_SIZE; j++)
            GArray[i][j] = letr;
    }
    for (i = 0; i < ARRAY_SIZE; i++) {    // print whole array
        for (j = 0; j <  ARRAY_SIZE; j++)
            printf("%c", GArray[i][j]);
        printf("\n");
    }
    printf("\n");
    pthread_exit(0);
}
```

```
//*************************************************************************
// Function: main(), parameter: none
//*************************************************************************

int main(void) {

    pthread_t   thread1, thread2;
    int         i,j;
    int         pthr;
    char        let1 = 'p';
    char        let2 = 'c';

    for (i = 0; i < ARRAY_SIZE; i++)
        for (j = 0; j <  ARRAY_SIZE; j++)
            GArray[i][j] = '-';

    printf("\nArray vor Threads\n\n");
        for (i = 0; i < ARRAY_SIZE; i++) {
            for (j = 0; j <  ARRAY_SIZE; j++)
                printf("%c", GArray[i][j]);
        printf("\n");
    }

    pthr = pthread_create(&thread1, NULL, ThreadF, (void *)&let1);
    if (pthr < 0) perror("Could not create thread");

    pthr = pthread_create(&thread2, NULL, ThreadF, (void *)&let2);
    if (pthr < 0) perror("Could not create thread");

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("\n... nach Threads\n");
        for (i = 0; i < ARRAY_SIZE; i++) {
            for (j = 0; j <  ARRAY_SIZE; j++)
                printf("%c", GArray[i][j]);
        printf("\n");
    }
}
```

2.10 Aufgabe 10: . . . und wie schnell sind sie denn?

Ziele

- Kennenlernen einer Möglichkeit zur Bestimmung der benötigten Rechenzeit
- Verwendung der Funktion **`times()`**

Aufgaben

Studieren Sie Programm ProcA10_1.c, dann starten Sie ProcA10_1.e, und übergeben als Parameter einen Shellbefehl: z.B. **`ProcA10_1.e "ls -alR ~"`**. Experimentieren Sie auch mit verschiedenen anderen Shell Befehlen.
In Programm `ProcA10_2.c` ist ein *Playground* markiert, experimentieren Sie mit verschiedenen Funktionen, z.B. `sleep()`, oder ein sehr langen for-Schleife oder mit was immer Sie möchten.

Programme (ohne Header)

`Datei: ProcA10_1.c`

```
int main(int argc, char *argv[]) {

    struct tms    startT, endT;
    clock_t       StartTime, EndTime;
    double        Zeit, ticks, usr, sys, cusr, csys;

    ticks = sysconf(_SC_CLK_TCK);

    if (argc > 1 ) {
        StartTime = times(&startT);
        system(argv[1]);
        EndTime = times(&endT);
    }
    else {
        printf("Missing argument\n");
        exit(0);
    }

    Zeit = (double)(EndTime - StartTime) / ticks;
    usr  = (double)(endT.tms_utime - startT.tms_utime) / ticks;
    sys  = (double)(endT.tms_stime - startT.tms_stime) / ticks;
    cusr = (double)(endT.tms_cutime - startT.tms_cutime) / ticks;
    csys = (double)(endT.tms_cstime - startT.tms_cstime) / ticks;

    printf("\n\n");
    printf("Befehl:\t\t\t\t%s\n", argv[1]);
    printf("Uhrzeit:\t\t\t%4.3f\n", Zeit);
    printf("User CPU-time:\t\t\t%4.3f\n", usr);
    printf("System CPU-time:\t\t%4.3f\n", sys);
    printf("Children user CPU-time:\t\t%4.3f\n", cusr);
    printf("Children system CPU-time:\t%4.3f\n", csys);

    exit(0);
}
```

Datei: ProcA10_2.c

```
int main(int argc, char *argv[]) {

    struct tms  startT, endT;
    clock_t     StartTime, EndTime;
    double      Zeit, ticks, usr, sys, cusr, csys;

    ticks = sysconf(_SC_CLK_TCK);

    StartTime = times(&startT);

    // ************ playground
    sleep(3);
    // ************ playground

    EndTime = times(&endT);

    Zeit = (double)(EndTime - StartTime) / ticks;
    usr  = (double)(endT.tms_utime - startT.tms_utime) / ticks;
    sys  = (double)(endT.tms_stime - startT.tms_stime) / ticks;
    cusr = (double)(endT.tms_cutime - startT.tms_cutime) / ticks;
    csys = (double)(endT.tms_cstime - startT.tms_cstime) / ticks;

    printf("\n\n");
    printf("Befehl:\t\t\t\t%s\n", argv[1]);
    printf("Uhrzeit:\t\t\t%4.3f\n", Zeit);
    printf("User CPU-time:\t\t\t%4.3f\n", usr);
    printf("System CPU-time:\t\t%4.3f\n", sys);
    printf("Children user CPU-time:\t\t%4.3f\n", cusr);
    printf("Children system CPU-time:\t%4.3f\n", csys);

    exit(0);
}
```

2.11 Aufgabe 11: Mr. Daemon, what's the time please?

Ziele

- Problemstellungen um Dämonen kennenlernen:
  - wie wird ein Prozess zum Daemon?
  - wie erreicht man, dass nur ein Daemon vom gleichen Typ aktiv ist?
  - wie teilt sich ein Daemon seiner Umwelt mit?
  - wo lebt ein Daemon?

Einleitung

Zu dieser Aufgabe haben wir für Sie einen Daemon implementiert: `MrTimeDaemon` gibt auf Anfrage die Systemzeit Ihres Rechners bekannt. Abfragen können Sie diese Zeit mit dem Programm `WhatsTheTimeMr localhost`. Die Kommunikation zwischen den beiden Prozessen haben wir mit TCP/IP Sockets implementiert. Weitere Infos zum Daemon finden Sie nach den Aufgaben.

Aufgaben

- Für die folgende Aufgabe benötigen Sie mindestens zwei Fenster (Kommandozeilen-Konsolen). Übersetzen Sie die Programme mit `make` und starten Sie das Programm `PlapperMaul` in einem der Fenster. Das Programm schreibt (ca.) alle 1/2 Sek. "Hallo, ich bins.... Idi" plus seine Prozess-ID auf den Bildschirm. Mit dem Shell Befehl `ps` können Sie Ihren aktiven Prozesse auflisten, auch `PlapperMaul`. Überlegen Sie sich zuerst, was mit `PlapperMaul` geschieht, wenn Sie das Fenster schliessen: läuft `PlapperMaul` weiter? Was geschieht mit `PlapperMaul` wenn Sie sich ausloggen und wieder einloggen?
  Testen Sie Ihre Überlegungen, in dem Sie die entsprechenden Aktionen durchführen. Stimmen Ihre Überlegungen?
- Starten Sie nun das Programm resp. den Daemon `MrTimeDaemon`. Stellen Sie die gleichen Überlegungen an wie mit `PlapperMaul` und testen Sie wiederum, ob Ihre Überlegungen stimmen. Ob MrTimeDaemon noch läuft können Sie feststellen, indem Sie die Zeit abfragen oder mit dem Befehl "**`ps ajx | grep MrTimeDaemon`**" abfragen ob der Demaon noch läuft : was fällt Ihnen am Output auf? Was schliessen Sie aus Ihren Beobachtungen?
- Starten Sie `MrTimeDaemon` erneut, was geschieht?
- Stoppen Sie nun MrTimeDaemon mit "`killall MrTimeDaemon`".
- Fragen Sie die Zeit bei einem Ihrer Kollegen ab. Dazu muss beim Server (dort wo `MrTimeDaemon` läuft) ev. die Firewall angepasst werden. Folgende Befehle müssen dazu mit root-Privilegien ausgeführt werden:

  1. `iptables-save > myTables.txt` (sichert die aktuelle Firewall)
  2. `iptables -I INPUT 1 -p tcp --dport 65534 -j ACCEPT`
  3. `iptables -I OUTPUT 2 -p tcp --sport 65534 -j ACCEPT`

Nun sollten Sie über die IP-Nummer oder über den Namen auf den TimeServer mit WhatsTheTimeMr zugreifen können.
Die Firewall können Sie mit folgendem Befehl wiederherstellen:

```
iptables-restore myTables.txt
```

- Studieren Sie `MrTimeDaemon.c`, `Daemonizer.c` und `TimeDaemon.c` und analysieren Sie, wie die Daemonisierung abläuft. Entfernen Sie die Kommentarstrings "//@" im Macro `OutPutPIDs` (am Anfang des Moduls `Daemonizer.c`), übersetzen Sie die Programme mit `make` und starten Sie `MrTimeDaemon` erneut. Analysieren Sie die Ausgabe, was fällt Ihnen auf? Notieren Sie sich alle für die vollständige Daemonisierung notwendigen Schritte.
- Setzen Sie beim Aufruf von `Daemonizer()` in `MrTimeDaemon.c` anstelle von `lockFilePath` den Null-Zeiger `NULL` ein. Damit wird keine lock-Datei erzeugt. Übersetzen Sie die Programme und starten Sie erneut `MrTimedaemon`. Was geschieht resp. wie können Sie feststellen, was geschehen ist?
  Hinweis: lesen Sie das log-File: **`/tmp/timeDaemon.log`**

Wenn Sie noch Zeit und Lust haben: messen Sie die Zeit, zwischen Start der Zeitanfrage und Eintreffen der Antwort. Dazu müssen Sie die Datei `WhatsTheTimeMr.c` entsprechend anpassen.

Beschreibung des Daemons

- Der Daemon besteht aus den 3 Komponenten:
  - **Hauptprogramm: `MrTimeDaemon.c`**
    Hier werden die Pfade für die lock-Datei, die log-Datei und den "Aufenthaltsort" des Daemons gesetzt. Die lock-Datei wird benötigt um sicherzustellen, dass der Daemon nur einmal gleichzeitig laufen kann. In die lock-Datei schreibt der Daemon z.B. seine PID und sperrt sie dann für Schreiben. Wird der Daemon ein zweites Mal gestartet und will seine PID in diese Datei schreiben, erhält er eine Fehlermeldung und terminiert (es soll ja nur ein Daemon arbeiten). Terminiert der Daemon, wird die Datei automatisch freigegeben.

    Weil Daemonen sämtliche Kontakte mit ihrer Umwelt im Normalfall abbrechen und auch kein Kontrollterminal besitzen, ist es sinnvoll, zumindest die Ausgabe des Daemons in eine log-Datei umzuleiten. Dazu stehen einige Systemfunktionen für Logging zur Verfügung. Der Einfachheit halber haben wir hier eine normale Datei im Verzeichnis `/tmp` gewählt. **Anmerkung**: die Wahl des Verzeichnisses `/tmp` für die lock- und log-Datei ist für den normalen Betrieb ungünstig (sogar problematisch), weil der Inhalt dieses Verzeichnisses jederzeit gelöscht werden kann, resp. darf. Wir haben dieses Verzeichnis gewählt, weil wir die beiden Dateien nur für die kurze Zeit des Praktikums benötigen.

    Der Daemon erbt sein Arbeitsverzeichnis vom Elternprozesse, er sollte deshalb in ein "festes" Verzeichnis des Systems wechseln, um zu verhindern, dass er sich in einem

montierten (gemounteten) Verzeichnis aufhält, das dann beim Herunterfahren nicht demontiert werden könnte (wir haben hier wiederum `/tmp` gewählt).

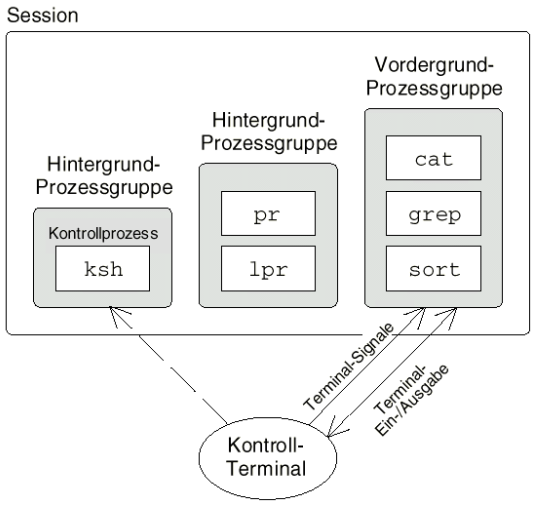
- **Daemonizer: `Daemonizer.c`**
  Der Daemonizer macht aus dem aktuellen Prozess einen Daemon. Z.B. sollte er Signale (eine Art Softwareinterrupts) ignorieren: wenn Sie die CTRL-C Taste während dem Ausführen eines Vordergrundprozess drücken, erhält dieser vom Betriebssystem das Signal SIGINT und bricht seine Ausführung ab (mehr dazu in der Literatur, z.B. [1]). Weiter sollte er die Dateierzeugungsmaske auf 0 setzen (Dateizugriffsrechte), damit kann er beim Öffnen von Dateien beliebige Zugriffsrechte verlangen (die Dateierzeugungsmaske erbt er auch vom Elternprozess). Die restlichen Aktivitäten werden Sie später in der Aufgabenstellung genauer untersuchen. Am Schluss startet der Daemonizer das eigentliche Daemonprogramm: `TimeDaemon`.
- **Daemonprogramm: `TimeDaemon.c`**
  Das Daemonprogramm wartet in einer unendlichen Schleife auf Anfragen zur Zeit und schickt die Antwort an den Absender zurück. Die Datenkommunikation ist, wie schon erwähnt, mit Sockets implementiert, auf die wir aber im Rahmen dieses Praktikums nicht weiter eingehen wollen (wir stellen lediglich einige Hilfsfunktionen zur Verfügung).

3. Prozesse und Threads unter Unix/Linux

3.1 Prozessidentifikation

Jeder Prozess unter Unix hat eine eindeutige Kennung in Form einer nichtnegativen ganzen Zahl, die Prozessnummer oder Process-ID genannt wird (Abkürzung PID). Bis auf eine Ausnahme hat jeder Prozess einen Elternprozess, von dem er erzeugt wurde. Die Prozess-ID des Elternprozesses wird Parent Process-ID oder kurz PPID genannt. Jeder Prozess hat noch weitere Kennungen wie User-ID und Group-ID auf die wir hier aber nicht weiter eingehen möchten.

Jeder Prozess unter Unix gehört zu einer **Prozessgruppe**. Jede Prozessgruppe besteht aus einem oder mehreren Prozessen. Jede Prozessgruppe kann einen **Prozessgruppenführer** (leader) haben, den man daran erkennt, dass seine Prozess-ID gleich wie seine Prozessgruppen-ID ist. Eine Prozessgruppe hört auf zu existieren, wenn sie keine Mitglieder mehr hat. Ein Prozess kann die Prozessgruppe wechseln (Details dazu bei Herold, [1]).



Eine weitere Gruppierung sind sogenannte Sessions. Zu einer Session können eine oder mehrere Prozessgruppen gehören. Eine Session kann genau ein Kontrollterminal besitzen. Der Prozess, der die Verbindung zum **Kontrollterminal** eingerichtet hat, wird **Kontrollprozess** genannt (und ist Sessionführer). In einer Session gibt es maximal eine **Vordergrund**-Prozessgruppe, alle anderen Prozessgruppen sind **Hintergrund**-Prozessgruppen. Die Vordergrund-Prozessgruppe existiert genau dann, wenn die Session ein Kontrollterminal hat. Nur die Prozesse der Vordergrund-Prozessgruppe können mit dem Kontrollterminal kommunizieren, deshalb können auch nur diese Prozess mit CTRL-C (Signal SIGINT) abgebrochen werden.

3.2 Unix Prozesserzeugung, -hierarchie und -steuerung

Wie schon erwähnt stammen fast alle Prozesse von einem Erzeuger ab. Wie ist nun diese Prozesshierarchie aufgebaut? Dazu werden beim Start des Systems einige spezielle Prozesse eingerichtet: Prozess 0 und Prozess 1.

- Prozess 0 wird zur Bootzeit erzeugt und wird *Swapper* oder *Scheduler-Prozess* genannt. Dieser Systemprozess ist Teil des Kerns. Eine seiner Aufgaben ist es, Prozess 1, genannt *Init*, zu erzeugen. Alle weiteren Prozessen stammen in irgendeiner Form von Prozess 1 ab.
- Prozess 1 ist im Gegensatz zum Swapper ein normaler Prozess, allerdings mit Super-User Privilegien). Er ist verantwortlich für systemspezifische Initialisierungen , wobei er die Dateien `/etc/rconfig` und /etc/rc* liest und das System gemäss den dort gemachten Vorgaben konfiguriert. Beim Login eines neuen Benutzers ist er verantwortlich, die entsprechenden Prozesse für diesen Benutzer zu erzeugen.

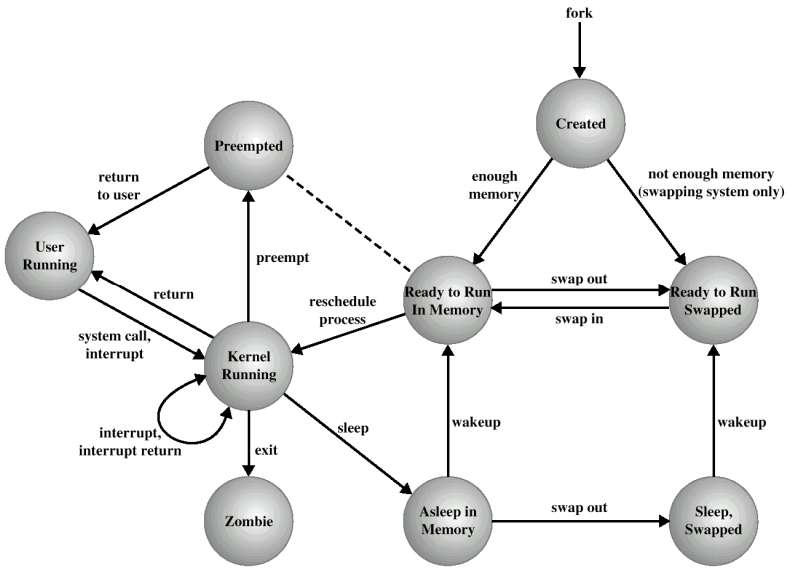
Unter Unix werden alle Prozesse (ausser Prozess 0) über die Systemfunktion **`fork()`** erzeugt. Die Ausführung resp. das Verhalten der Prozesse ist dann durch das Prozess-Zustandsdiagramm geregelt. Bei einem Uniprozessorsystem kann immer nur ein Prozess aktiv sein.

Terminiert ein Prozess, wird er zuerst zum ***Zombie***, d.h. das Prozessimage besteht nach wie vor, wartet aber darauf, bis es endgültig aus der Prozesshierarchie entfernt wird. Solange sich ein Prozess im Zustand Zombie befindet, stehen Informationen zu seiner Ausführung (z.B. verbrauchte Rechenzeit, benutzte Ressourcen, etc.) zur Verfügung und können für Statistikzwecke gelesen werden. Für die endgültige Entfernung eines Prozesses (Zombie) aus dem System ist grundsätzlich der Elternprozess verantwortlich. Dazu gibt es zwei Möglichkeiten:

- der Elternprozess selbst terminiert (oder wird terminiert)
- der Elternprozess wartet auf die Terminierung der Kinder (`wait()`, oder `waitpid()`)

Grundsätzlich sollten keine Zombieprozesse vorhanden sein, sie belegen Resourcen: das Prozessimage muss immer noch vom System verwaltet werden.

Damit Sie sich über den "Prozessablauf" im Klaren sind, hier nochmals das Zustandsdiagramm von Unix:



3.3 Dämon-Prozesse

Dämonen oder englisch Daemons sind eine spezielle Art von **Prozessen** (resp. Threads), die **vollständig unabhängig arbeiten**, d.h. ohne direkte Interaktion mit dem Anwender. Dämonen sind Hintergrundprozesse und terminieren i.A. nur, wenn das System heruntergefahren wird oder abstürzt. Dämonen erledigen meist Aufgaben, die periodisch ausgeführt werden müssen, z.B. Überwachung von Systemkomponenten, abfragen, ob neue Mails angekommen sind, etc. Ein typisches Beispiel unter Unix ist der Printer Daemon `lpd`, der periodisch nachschaut, ob ein Anwender eine Datei zum Ausdrucken hinterlegt hat, wenn ja, schickt er die Datei auf den Drucker. Hier wird eine weitere Eigenschaft von Daemons ersichtlich: meist kann nur ein Dämon pro Aufgabe aktiv sein: stellen Sie sich vor, was passiert, wenn zwei Druckerdämonen gleichzeitig arbeiten. Andererseits muss aber auch dafür gesorgt werden, dass ein Dämon wieder gestartet wird, falls er stirbt.

3.4 Threads unter Linux

Aus der Vorlesung wissen Sie, dass Threads sogenannte Leichgewichtsprozesse sind, die nicht einen eigenen Prozesskontext benötigen, sondern **in einem gemeinsamen Prozesskontext, parallel ablaufen**. Dabei bilden der Prozess und seine Threads eine Einheit, wenn z.B. der Prozess terminiert, terminieren auch die Threads.

Wir werden in diesem Praktikum die **POSIX-kompatible** Thread-Implementation (pthreads) unter Linux verwenden . Diese Linux-Implementation verwendet Kernel Threads. Dabei wird pro Thread eigentlich ein Prozess erzeugt, aber im Gegensatz zu Kindprozessen nutzen die Threads alle Resourcen gemeinsam. Selbstverständlich hat auch das Hauptprogramm `main()` als Haupt-Thread Zugriff auf alle Prozessressourcen (globale Variablen, offene Files, etc.). Wenn das Hauptprogramm terminiert (entspricht dem Prozess), terminieren auch die einzelnen Threads. Der Vorteil dieser Implementation ist, dass die Threads vom normalen Prozessscheduler bedient werden und mit `top` und `ps` beobachtet werden können. Nachteilig ist, dass ein Prozess erzeugt und verwaltet werden muss, was allerdings unter Linux sehr effizient abläuft (für die Thread-Erzeugung wird der System-Call **`clone`** verwendet, siehe Linux Manuals).

Wie möchten hier darauf hinweisen, dass Threads in anderen Betriebssystemen (z.B. Solaris, Windows NT, etc.), gänzlich anders implementiert sind. Dies hängt einerseits von der Betriebssystemunterstützung ab, andererseits auch von den verwendeten Bibliotheken. Selbstverständlich stehen auch unter Linux andere Thread-Bibliotheken zur Verfügung.

4. Shell Befehle und Systemfunktionen

4.1 Shell Befehle

Im folgenden finden Sie ein Übersicht der wichtigsten UNIX Shell Befehle zur Steuerung und Überwachung von Prozessen. Für weitere Details möchten wir Sie auf die entsprechenden man-pages verweisen.

`ps` Prozesse anzeigen (single shot)
Mit **`ps`** lassen sich Daten zu den laufenden Prozessen in einer Tabelle anzeigen. Hilfreiche Optionen:
- `ps a`: zeigt sämtliche aktiven Prozesse im System an
- `ps f`: zeigt den Familienstammbaum der Prozesse an (Linux) (alternativ können Sie auch den Befehl `pstree` verwenden)
- `ps axj`: zeigt alle aktiven Prozesse an, die nicht mit einem Kontrollterminal verbunden sind (Dämonen)

`top` Prozesse dynamisch anzeigen (repetitiv)
Mit `top` lassen sich Informationen zu den laufenden Prozessen in einer Tabelle anzeigen. Die Tabelle kann konfiguriert werden und die Konfiguration in der Datei `.toprc` im Home Directory abgelegt werden. Ist `top` aktiv, können zudem Parameter verändert und angepasst werden, z.B. welche Benutzer angezeigt werden sollen, wie oft das Display erneuert werden soll, etc. Für weitere Informationen möchten wir auf die man-pages verweisen.
Hilfreiche Option: top -d x : aktualisiert die Information alle x Sekunden

`kill` Prozess terminieren
Mit "`kill PID`" wird der Prozesse mit Prozess-ID `PID` terminiert, die Prozess-ID kann z.B. mit `ps` oder `top` abgefragt werden. Kill schickt das SIGKILL Signal an den entsprechenden Prozess. Prozesse, die nicht mehr auf Signale reagieren, können mit `kill -9 PID` entfernt werden. Hintergrundprozesse können Sie mit dem Befehl **`jobs`** anzeigen, dabei steht vor jedem Prozess eine Nummer in eckigen Klammernt: diese Prozesse lassen sich mit **`kill %Nummer`** (z.B. `kill %1`) terminieren. Mit **"`killall foo`"** lassen sich alle Prozesse mit Name `foo` terminieren.

3.2 Systemfunktionen

3.2.1 Prozesse erzeugen: `fork()`

Erzeugt einen neuen Prozess.

<sys/types.h>, <unistd.h>

`pid_t fork (void)`

returns: bei Erfolg siehe unten, -1 bei Fehler

Die Systemfunktion `fork()` erzeugt einen neuen Prozess. Der Prozess der fork aufruft heisst Elternprozess (parent), der neu erzeugte Prozess heisst Kindprozess (child). Unmittelbar nach fork sind beide Prozesse sehr ähnlich, `fork()` erstellt im Wesentlichen eine Kopie des Elternprozesses. Beide Prozesse fahren dann im gleichen Programmcode mit der Instruktion nach `fork()` weiter, nun aber als eigenständige Prozesse mit verschiedenen Programmzählern. Beide haben die gleichen offenen Dateien, das gleiche working directory, etc., aber `fork()` liefert zwei verschiedene Rückgabewerte: im Elternprozess wird die PID des neu erzeugten Kindes zurückgegeben, im Kindprozess liefert `fork()` den Wert 0 (so kann festgestellt werden, ob man sich im Kind- oder Elternprozess befindet. Falls `fork()` nicht erfolgreich war, ist der Rückgabewert negativ: der Grund kann mit `perror()` angezeigt werden.

In der folgenden Liste sind die Unterschiede und Gemeinsamkeiten zwischen Eltern und Kindprozesse nach `fork()` und `exec()` aufgeführt (ohne Erklärung, `exec()` siehe 3.2.9):

Attribut	Vererbung bei fork()	Erhaltung bei exec()
Rückgabewert fork()	Eltern: PID des Kindes Kind: 0	---
Prozess-Identifier	nein	ja
Vaterprozess-Identifier	nein	ja
reale User-/Group-Identifier	ja	ja
effektive User-/Group-Identifier	ja	eventuell
Prozessgruppen-Identifier	ja	ja
Session-Identifier	ja	ja
Kontroll-Terminal	ja	ja
Arbeitsverzeichnis	ja	ja
Root-Directory	ja	ja
offene Dateidescriptoren	ja	eventuell
Maske der Zugriffsrechte	ja	ja
Datei-Locks	nein	ja
Signalmaske	ja	ja
hängige Signale	nein	ja
hängige Alarme	nein	ja
Umgebungsvariablen	ja	eventuell
Resourcen-Grenzwerte	ja	ja
Prozesszeiten	nein	ja

Für detailliertere Informationen siehe z. B. [1] Herold.

Typische Anwendungen von fork():

- Ein Prozess möchte ein anderes Programm ausführen (siehe Praktikum zu Shell). In diesem Fall muss der Programmcode des Kindes durch anderen Programmcode ersetzt werden → geschieht mit Hilfe der exec-System-Calls
- Ein Prozess möchte verschiedene Codestücke parallel ausführen, z.B. ein Server, der gleichzeitig mehrere Klienten bedienen muss, hier ist der Code für den Eltern und die Kindprozesse im gleichen Programm enthalten: Eltern und Kindprozess führen aber verschiedenen Code aus.

4.2.2 Prozesse beenden: `exit()`

Beendet den aktuellen Prozess.

<stdlib.h>

```
void exit(int status)
```

Beendet den aktuellen Prozess und erledigt Aufräumarbeiten. In `status` kann Information zur erfolgreichen (oder eben nicht) Durchführung des Prozesses übergeben werden (`status` = 0: Erfolg, `status` = -1: Misserfolg).

4.2.3 Information zur PID: `getpid(), getppid()`

Abfragen der PID des aktuellen Prozesses und des Elternprozesses.

<sys/types.h>,<unistd.h>

```
pid_t getpid(void);
pid_t getppid(void);
```

returns: bei Erfolg die PID, -1 bei Misserfolg

4.2.4 Prozessgruppen-ID abfragen: getpgrp()

Abfragen der Prozessgruppen-ID des aktuellen Prozesses.

<sys/types.h>,<unistd.h>

```
pid_t getpgrp(void);
```

returns: bei Erfolg die Prozessgruppen-ID, -1 bei Misserfolg

4.2.5 Neue Session einrichten: setsid()

Richtet eine neue Session ein.

<sys/types.h>,<unistd.h>

```
pid_t setsid(void);
```

returns: bei Erfolg die Prozessgruppen-ID, -1 bei Misserfolg

Ist der aufrufende Prozess kein Prozessgruppenführer, richtet `setsid()` eine neue Session ein. Der aufrufende Prozess wird Prozessgruppenführer und Session Leader und hat kein Kontrollterminal.

4.2.6 Auf Prozess warten: `wait()`

Auf die Beendigungen irgendeines Kindprozesses warten.

<sys/types.h>, <sys/wait.h>

```
pid_t wait(int *status)
```

returns: siehe unten

Mit `wait()` kann auf die Beendigung irgendeines Kindes gewartet werden, der aufrufende Prozesse blockiert, falls noch kein Kind terminiert hat. Der Prozess blockiert nicht, wenn kein Kind aktiv ist. Der Rückgabewert entspricht der PID des Kindes das terminiert hat, bei Fehler wird -1 zurückgegeben. In `status` wird der Beendigungsstatus des Kindprozesses abgelegt und kann über entsprechende Makros abgefragt werden (definiert in <sys/wait.h>). Mehr dazu in der entsprechenden Literatur. Wenn man am Wert von `status` nicht interessiert ist, kann ein NULL Zeiger übergeben werden.

4.2.7 Auf Prozess warten: `waitpid()`

Auf die Beendigung des Kindprozesses mit PID warten.

<sys/types.h>, <sys/wait.h>

```
pid_t waitpid(pit_t pid, int *status, int option);
```

returns: siehe unten

`Waitpid()` wartet im Gegensatz zu `wait()` auf die Terminierung des Kindes mit der Prozess-ID **`pid`**. Mit der Option `option` kann angegeben werden, dass der aufrufende Prozess nicht blockiert, wenn das Kind noch nicht terminiert hat (option: WNOHANG). Auch der Wert von pid wird zu Kontrollzwecken verwendet: pid > 0: warten auf Beendigung des Kindes mit pid, pid =-1: arbeitet wie wait (Weitere Möglichkeiten: pid = 0 und < -1: siehe Lit.). Für detailliertere Informationen möchten auch hier auf die Literatur [1] und man-pages verweisen.

4.2.8 Prozess schlafen legen: `sleep(), usleep()`

Versetzt den aktuellen Prozess für eine bestimmte Zeitdauer in den Schlafzustand.

<unistd.h>

```
unsigned int sleep(unsigned int sec);
void     usleep(unsigned int usec);
```

returns: 0 oder Anzahl nicht geschlafener Sekunden

Die beiden Prozeduren suspendierenden aktuellen Prozess für sec Sekunden resp. usec Mikrosekunden.

4.2.9 Programm ausführen: `exec()`

Aktuellen Prozess mit einem neuen Programm "überlagern".

<unistd.h>

```
int execl(const char *pfad, const char *arg0,..., NULL);
int execv(const char *pfad, const char *argv[]);
int execle(const char *pfad, const char *arg0,..., NULL,
           char *const envp[]);
int execve(const char *pfad, const char *argv[],
           char *const envp[]);
int execlp(const char *filenam, const char *arg0,..., NULL);
int execvp(const char *filenam, const char *argv[]);
```

returns: -1 bei Misserfolg, keine Rückkehr bei Erfolg

Der Systemaufruf `fork()` erzeugt zwar einen neuen Prozess, aber das ausgeführte Programm ist im Eltern- und Kindprozess gleich (Ausnahme: die Abschnitte wo der Rückgabewert von `fork()` als Bedingung für die Programmausführung verwendet wird). Soll ein neues Programm (eine andere ausführbare Datei) ausgeführt werden, muss dieses mit `exec()` gestartet werden, das den Programmcode, die Datensegmente, sowie Heap und Stack des aktuellen Programmes mit den Daten des neuen Prozesses überschreibt.

Der Unterschied bei den 6 Varianten liegt vor allem im Aufruf des Programmes und der Übergabe der Parameter. Wir möchten hier auf das Manual verweisen.

4.2.10 Shell Befehle ausführen: `system()`

Aus dem aktuellen Programm den angegebenen Shell Befehl oder eine ausführbare Datei starten. Das aufrufende Programm wertet auf die Beendigung des aufgerufenen Befehls).

<stdlib.h>

```
int system(const char *shellcmd);
```

returns: siehe Literatur (-1 bei Misserfolg)

Der Aufruf `system()` führt das angegebenen Shell Befehl oder eine ausführbare Datei aus. Z.B. zeigt `system("ps -a")` alle aktiven Prozesse an. Intern ruft `system()` die Funktionen `fork()`, `exec()` und `waitpid()` auf.

4.2.11 System- und Benutzerzeit abfragen: `times()`

Zeitinformationen zum Eltern- und Kindprozess abfragen.

<sys/times.h>

```
clock_t times(struct tms *cpu_zeit);
```

returns: Systemzeit, -1 bei Misserfolg

`times()` schreibt die zur Verfügung stehenden Systemzeiten in die Datenstruktur **`tms`**:

```
struct tms {
  clock_t  tms_utime;   // Benutzer CPU-Zeit
  clock_t  tms_stime;   // System CPU-Zeit
  clock_t  tms_cutime;  // Benutzer CPU-Zeit beendeter Kinder
  clock_t  tms_cstime;  // System CPU-Zeit beendeter Kinder
};
```

`times()` gibt die Anzahl Clock-Ticks seit dem System Start zurück (-1 bei Misserfolge), dieser absolute Wert ist nicht sehr informativ, man arbeitet deshalb meist mit relativen Werten. Die Anzahl clock ticks pro Sekunde ist von der Implementation abhängig und kann mit Hilfe der Funktion `sysconf(_SC_CLK_TCK)` abgefragt werden (auf 80x86 basierten Systemen normalerweise = 100). Ein Beispiel zur Anwendung finden Sie in der letzten Aufgabenstellung.

4.2.12 Thread Managment

4.2.12.1 Kompilation von Programmen mit Threads

Beim Kompilieren muss die Pthread-Bibliothek explizit angegeben werden:

```
g++ -lpthread foo.c -o foo.e
```

4.2.12.2 Erzeugung eines Threads: `pthread_create()`

Funktion zur Erzeugung eines Threads:

<pthread.h>

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*routine)(void *), void *arg);
```

returns: 0 bei Erfolg, -1 bei Misserfolg

`*tid`	Ein Resultatparameter, in dem der Identifier, des Threads zurückgegeben wird.
`attr`	Ein Zeiger auf ein Attributobjekt, wo der Stack, Prioritäten, Scheduling-Policy, etc. festgelegt werden. Ist `attr` ein NULL Zeiger, werden Defaultwerte festgelegt.
`routine`	Ist eine Funktion, die im Thread gestartet wird, die als Parameter einen void Zeiger enthält. Diesem Zeiger wird der Wert von arg zugewiesen, der auf eine beliebige Datenstruktur zeigen kann. Achtung: alle Threads nutzen den Datenbereich gemeinsam, d.h. für jeden Thread muss eine individuelle Datenstruktur definiert werden (falls sich die Daten in der Datenstruktur unterscheiden).
`arg`	Zeiger auf eine Datenstruktur (auch einzelner Wert), der an die Threadfunktion übergeben wird.

4.2.12.3 Auf Terminierung eines Threads warten: pthread_join()

Funktion zum Warten auf Terminierung eines Threads:

```
<pthread.h>
int pthread_join(pthread_t th, void **th_ret);
returns: 0 bei Erfolg, -1 bei Misserfolg
```

Auf Terminierung des Threads mit Thread-ID **th** warten.

th Thread ID

****th_ret** Rückgabewert des Threads, der Terminiert. Entweder Wert der mit pthread_exit() zurückgegeben wird oder PTHREAD_CANCELED falls der Thread abgebrochen wurde.

4.2.12.4 Thread terminieren: pthread_exit()

Funktion zur Erzeugung eines Threads:

```
<pthread.h>
int pthread_exit(void *retval);
returns: 0 bei Erfolg, -1 bei Misserfolg
```

***retval** Rückgabewert des Threads, oft wird 0 (NULL) eingesetzt: normale Terminierung

4.3 C++ Terminal-Ausgabe

Im ersten C-Programmbeispiel wurde für die Bildschirmausgabe die **printf(…)**-Anweisung verwendet. In einigen Beispielen dieses Praktikum wird die **bequemere C++** Alternative mit **IO-Streams** verwendet. Der Hauptvorteil ist, dass man sich nicht um die Typen der ausgegebenen Variablen kümmern muss (im Gegensatz zu den in printf(…)-Anweisung).

Beispiel: Ausgabe der Variablen i (=10) und a[i] (=21.3456) , i: Typ integer, Arrayelement a[i]: Typ float, geforderte Ausgabe: `"Resultat a[10] = 21.3456"`

Printf:

```
#include <stdio.h>
. . .
printf(”Resultat a[%d] = %f \n”, i, a[i]);
```

IO-Streams:

```
#include <iostream>
using namespace std;
. . .
cout << ”Resultat a[” << i << ”] = ” << a[i] << endl;
```

5 Literatur

[1] H.Herold, *Linux-Unix Systemprogrammierung*, 3.Auflage 2004, Addison Wesley.

[2] M. Bach, Unix, *Wie funktioniert das Betriebssystem*, Hanser, 1991

[3] T. Wagner, D. Towsley, *Getting Started with POSIX Threads*, Univ. of Massachusetts at Amherst, July, 1995.