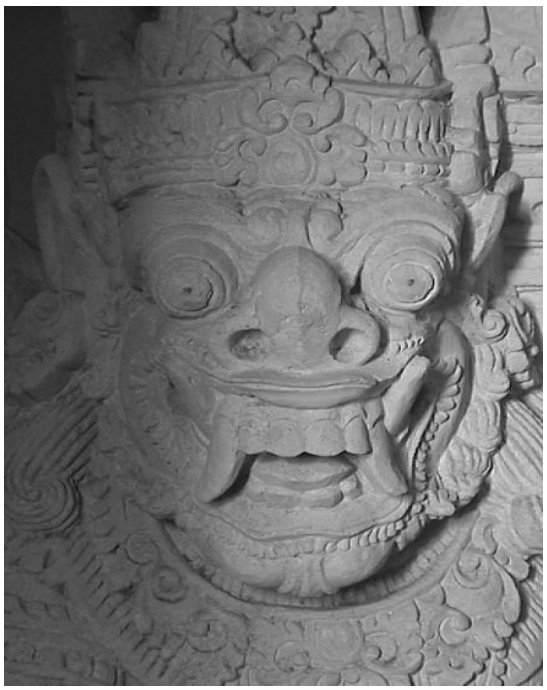
Praktikum ProcThreads

Prozesse,Threads und Dämonen

Frühlingssemester 2016

M. Thaler, J. Zeman



Überblick

Prozesse und Threads sind die ältesten und wichtigsten Abstraktionen, die von Betriebssystemen zur Verfügung gestellt werden. Sie erlauben auf einem single-CPU Prozessor die *pseudo*-gleichzeitige Verarbeitung mehrerer Programme, d.h. es stehen mehrere virtuelle CPUs zur Verfügung. Die gleichen Abstraktionen gelten auch für BS auf Multicore-Prozessoren (mehrere CPUs).

In diesem Praktikum werden wir uns mit Prozessen, Prozesshierarchien, Dämonen und Threads beschäftigen, um ein gutes Grundverständnis dieser Abstraktionen um Prozesse und Threads zu erhalten.

Inhaltsverzeichnis

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 zur 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 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.

1.3 Organisatorisches und Vorgehen

Das Praktikum besteht aus einer Anzahl Aufgabenstellungen, die Sie durcharbeiten werden. Die Dateien zu diesem Praktikum finden Sie im File **`ProcThreads.tgz`**. Nach dem Auspacken finden Sie in Ihrem Arbeitsverzeichnis den Ordner **`ProcThreads`**. Die Programme zu den einzelnen Aufgabenstellungen finden Sie in den entsprechenden Unterverzeichnissen.

Beantworten Sie zuerst die Fragen in de Aufgabenstellung und starten Sie die Programme erst, wenn Sie dazu in der Aufgabenstellung aufgefordert werden (nur so ist der Lerneffekt optimal). Implementieren bzw. erweitern Sie die Programme, verwenden Sie dazu die mitgelieferten Makefiles. Bei Problemen stehen wir Ihnen jederzeit gerne zur Verfügung.

In diesem Praktikum verwenden wir viele Systemfunktionen (System Calls), weitergehende Informationen zu diesen Funktionen finden Sie in den Manual Pages.

1.4 Zu Beachten

Sie arbeiten hier mit Systemfunktionen, die jederzeit aus verschiedensten 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 Systemfunktion **`perror()`** zur Verfügung: **`perror(message")`** gibt zuerst den String **`''message''`** aus, gefolgt vom Grund für den Fehler.

Beachten Sie in diesem Zusammenhang das Vorgehen in unseren Programmbeispielen: wir wenden die Überprüfung nicht überall konsequent an, um die Übersichtlichkeit und Lesbarkeit der Programme nicht allzu sehr zu beeinträchtigen.

2 Aufgaben

2.1 Betriebssystemressourcen mit top und htop anzeigen

Ziele

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

top

a) Starten Sie in einem Fenster **top**: sämtliche laufenden Prozesse werden angezeigt.

b) Geben Sie **u** gefolgt von Ihrem Benutzername ein: nur Ihre eigenen Prozesse werden angezeigt. Wenn Sie **top** mit **top -u $USER -d 1** starten, werden ebenfalls nur Ihre Prozesse angezeigt, zudem wird top nun alle Sekunden aufdatiert,

c) Wenn Sie den Befehl **f** eingeben, zeigt **top** eine Liste, wo Felder, Sortierkriterium und Darstellungsreihenfolge gewählt werden können. Mit Eingabe von q verlassen Sie das Menu.

   Hinweis: die Sortierung erfolgt defaultmässig in absteigender Richtung (gilt auch alphabetisch). Durch Eingabe eines **R** im Hauptmenu kann diese Reihenfolge umgekehrt werden.

d) Mit **h** (help) können Sie eine Liste der unterstützten Befehle abfragen.

e) Mit **H** können Sie die Anzeige von Threads ein- und ausschalten (Toggle-Funktion).

f) Verlassen Sie top mit **q** (quit) und kopieren Sie das vorbereitete Konfigurationsfile **.toprc** mit dem Befehl **setToprc** in Ihr Home Directory: existiert .toprc, wird eine Sicherungskopie erstellt.

   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.

   top ist nun so konfiguriert, dass z.B. auch die Threads angezeigt werden und neben der prozentualen Auslastung der CPU(s) auf welchem Core der Job läuft.

   Hinweis: **.toprc** kann mit dem Befehl **rstToprc** entfernt bzw. widerhersgestellt werden.

g) Programme / Dateien

```
setToprc:  set .toprc
rstToprc:  remove / restore .toprc
```

htop (nur Linux)

Das Programm htop zeigt, ähnlich wie top, die laufenden Prozesse auf allen verfügbaren CPUs und die Speicherauslastung an. Da alle CPUs einzeln dargestellt werden, eignet sich htop speziell für Multi-Core Prozessoren.

a) Starten Sie das Programm mit **htop -u $USER -d 10**: damit werden nur Ihre eigenen Prozesse dargestellt und die Anzeige wird alle Sekunden aufdatiert.

b) Weitere Konfigurationsmöglichkeiten entnehmen Sie bitte dem Manual.

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

Ziele

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

Aufgaben

a) Studieren Sie zuerst das Programm `ProcA2.c` und beschrieben Sie was geschieht.

b) Notieren Sie sich, was ausgegeben wird. Starten Sie 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 Prozess mit `execl()` erzeugen und ausführen

Ziele

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

Aufgaben

a) Studieren Sie zuerst die Programme `ProcA3.c` und `ChildProcA3.c`.

b) 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.

c) Benennen Sie `ChildProcA3.e` auf `ChildProcA3.f` um (Shell Befehl `mv`) und überlegen Sie, was das Programm nun ausgibt. Starten Sie `ProcA3.e` und vergleichen Sie Ihre Überlegungen mit der Programmausgabe.

d) Nennen Sie das Kindprogramm wieder `ChildProcA3.e` und geben Sie folgenden Befehl ein: `chmod -x ChildProcA3.e`. Starten Sie wiederum `ProcA3.e` und analysieren Sie die Ausgabe von `perror("...")` ? 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 Prozesshierarchie: was `fork()` alles kann

Ziele

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

Aufgaben

a) 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.

b) Mit dem Befehl **`ps f`** oder **`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`**) erfragen. Wenn Sie nun den Befehl "`pstree -n -p PID`" eingeben, wird nur die Prozesshierarchie ausgehend von PID angezeigt: `-n` sortiert die Prozesse numerisch, `-p` zeigt für jeden Prozess die PID an.

   Hinweis: alle erzeugten Prozesse müssen *arbeiten*, damit die Darstellung gelingt. Wie wird das in unten stehendem Programm erreicht?

Programm (ohne header)

Datei: `ProcA4.cq`

```
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 Zeitlicher Ablauf: wer macht wann was?

Ziele

- Verstehen, wie Kind- und Elternprozesse zeitlich ablaufen.

Aufgaben

a) Studieren Sie Programm `ProcA5.c`. Starten Sie nun mehrmals hintereinander das Programm `ProcA5.e` und vergleichen Sie die jeweiligen Outputs (leiten Sie dazu auch die Ausgabe auf verschiedene Dateien um). Was schliessen Sie aus dem Resultat?

Anmerkung: `selectCPU(0)` erzwingt die Ausführung des Eltern- und Kindprozeses auf CPU 0 (siehe Modul `setCPU.c`). Die Prozedur `justWork(HARD_WORK)` simuliert CPU-Load durch den Prozess (siehe Modul `workerUtils.c`).

Programm (ohne header)

Datei: `ProcA5.c`

```
#define ITERATIONS 20
#define WORK_HARD  2000000

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

int main(void) {
    pid_t   pid;
    int     i;

    pid = fork();
    selectCPU(0);                               // select CPU 0
    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);
            fflush(stdout);                      // force output
        }
        break;
      default:
        for (i = 0; i < ITERATIONS; i++) {;
            justWork(HARD_WORK);
            printf("%d \tMother\n", i);
            fflush(stdout);                      // force output
        }
    }
    printf("I go it ...\n");
    if (pid > 0)                                // wait for child to terminate
        waitpid(pid, NULL, 0);
    exit(0);
}
```

2.6 Was geschieht mit verwaisten Kindern?

Ziele

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

Aufgabe

a) Analysieren Sie Programm `ProcA6.c`: was läuft ab und welchen Ausgabe erwarten Sie?

b) Starten Sie `ProcA6.e`: der Elternprozess terminiert: was geschieht mit dem Kind?

c) 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, speziell auc die Spalte **S**.

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 Zombies auch in Unix . . .?

Ziel

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

Aufgaben

a) Analysieren Sie das Programm `ProcA7.c`

b) Starten Sie `top -u $USER -d 1`, in einem Fenster das immer gut sichtbar ist.

Alternativ können Sie auch das Script **`mtop`** bzw. **`mtop aaaa.e`** verwenden, es stelle das Verhalten der Prozesse dynamisch dar. Hinweis: <defunct> = zombie.

c) Starten Sie `aaaa.e` und verfolgen Sie im "top-Fenster" was geschieht: beachten Sie speziell die Spalte **`S`** und den Eintrag oben rechts in der zweiten Zeile: **`zombie`**.

d) In gewissen Fällen will man nicht auf die Terminierung eines Kindes mit `wait()`, bzw. `waitpid()` warten. Überlegen Sie sich, wie Sie in diesem Fall verhindern können, dass ein Kind zum Zombie wird.

Programm (ohne header)

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(8);                           // parent process sleeps for 8 sec
    wait(NULL);                         // consult manual for "wait"
    sleep(2);
    wait(NULL);
    sleep(2);
    wait(NULL);
    sleep(2);
    exit(0);
}
```

2.8 Wie Sie auf die Kinder warten können!

Ziele

- Verstehen, wie Informationen zu Kindprozessen abgefragt werden kann.
- Die Befehle `wait()` und `waitpid()` verwenden können.

Aufgaben

a) Starten Sie das Programm **`ProcA8.e`** und analysieren Sie wie die Ausgabe im Hauptprogramm zustande kommt und was im Kindprozess `ChildProgA8.c` abläuft.

b) Starten Sie nun `ProcA8.e` mit **`ProcA8.e 1`**. Der Parameter bewirkt, dass im Kindprozess ein "Segmentation Error" erzeugt wird, also eine Speicherzugriffsverletzung. Welches Signal wird durch die Zugriffsverletzung an das Kind geschickt? Diese Informtion finden Sie im Manual mit **`man 7 signal`**.

Schalten Sie nun `core dump` ein (siehe `README`) und starten Sie `ProcA8.e 1` erneut und analysieren Sie die Ausgabe. Hinweis: der Dump wird im File `core` abgelegt und kann mit dem `gdb` (GNU-Debugger) gelesen werden (siehe `README`).

c) Wenn Sie **`ProcA8.e 2`** starten, sendet das Kind das Signal 30 an sich selbst. Was geschieht?

d) Wenn Sie **`ProcA8.e 3`** starten, sendet `ProcA8.e` das Signal `SIGABRT` (abort) an das Kind: was geschieht in diesem Fall?

e) Mit **`ProcA8.e 4`** wird das Kind gestartet und terminiert nach 5s. Analysieren Sie wie in `ProcA8.e` der Lauf- bzw. Exit-Zustand des Kindes abgefragt wird (siehe dazu auch `man 3 exit`).

Kindprozess (ohne header)

Datei: `ChildProgA8.c`

```
int main(int argc, char *argv[]) {
    int i = 0, *a = NULL;
    if (argc > 1)
        i = atoi(argv[1]);  // convert string argv[1] to integer i
            // argv[1] is a number passed to child
    printf("\n*** I am the child having job nr. %d ***\n\n", i);
    switch(i) {
        case 0: exit(0);                // exit normally
                break;
        case 1: *a     = i;             // force segmentation error
                break;
        case 2: kill(getpid(), 30);     // I send signal 30 to myself
                break;
        case 3: sleep(5);               // sleep and wait for signal
                break;
        case 4: sleep(5);               // just sleep
                exit(222);              // then exit
                break;
        default:
                exit(-1);
    }
    exit(0);
}
```

Hauptprogramm (ohne header)

Datei: `ProcA8.c`

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

int main(int argc, char *argv[]) {
    pid_t  pid;
    int    status, retval, whatToDo = 0;
    char   str[8];

    if (argc > 1)
      whatToDo = atoi(argv[1]);       // get job number for child

    pid = fork();                       // fork child
    switch (pid) {
      case -1:
        perror("Could not fork");
        break;
      case 0:
        sprintf(str, "%d",whatToDo);
        retval = execl("./ChildProgA8.e", "ChildProgA8.e", str, NULL);
        if (retval < 0) perror("\nexecl not successful");
        break;
      default:
        if (whatToDo <= 3) {
            if (whatToDo == 3) {
                sleep(1);
                kill(pid, SIGABRT);             // send signal SIGABTR to child
            }
            wait(&status);
            if (WIFEXITED(status))
                printf("Child exits with status    %d\n", WEXITSTATUS(status));
            if (WIFSIGNALED(status)) {
                printf("Child exits on signal      %d\n", WTERMSIG(status));
                printf("Child exits with core dump %d\n", WCOREDUMP(status));
            }
        } else {
            usleep(500*1000);
            while (!waitpid(pid, &status, WNOHANG)) {
                printf(". . . child is playing\n");
                usleep(500*1000);
            }
            printf("Child has exited with 'exit(%d)'\n", WEXITSTATUS(status));
        }
        break;
    }
    exit(0);
}
```

2.9 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

a) Analysieren Sie Programm `ProcA9_1.c`: was gibt das Programm aus?

   - Starten Sie `ProcA9_1.e` und überprüfen Sie Ihre Überlegungen.
   - Waren Ihre Überlegungen richtig? Falls nicht, was könnten Sie falsch überlegt haben?

b) Analysieren Sie Programm `ProcA9_2.c`: was gibt das Programm aus?

   - Starten Sie `ProcA9_2.e` und überprüfen Sie Ihre Überlegungen.
   - Waren Ihre Überlegungen richtig? Falls nicht, was könnten Sie falsch gemacht haben?
   - Kind und Eltern werden in verschiedener Reihenfolge ausgeführt: ist ein Unterschied ausser der Reihenfolge festzustellen?

c) Analysieren Sie Programm ProcA9_3.c und überlegen Sie, was in die Datei `AnyOutPut.txt` geschrieben wird, wer schreibt alles in diese Datei (sie wird ja vor `fork()` geöffnet) und wieso ist das so?.

   - Starten Sie `ProcA9_3.e` und überprüfen Sie Ihre Überlegungen.
   - Waren Ihre Überlegungen richtig? Falls nicht, wieso nicht?

Programme (ohne header), auch nächste Seiten

Datei: `Proc9_1.c`

```
int main(void) {
    pid_t    pid;

    printf("\n");
    printf("\n\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");
            break;
    }
    printf("\nclear ?\n\n");
    exit(0);
}
```

Datei: ProcA9_2.c

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

int main(void) {
    pid_t  pid;
    int    i,j;
    // flip coin to select "child first" or "parent first"
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srandom(tv.tv_usec);                          // evaluate seed
    int head = ((int)(random()) >> 7) & 0x1;    // flip coin
    // fill global array with '-' and print array value
    for (i = 0; i < ARRAY_SIZE; i++)
        for (j = 0; j <  ARRAY_SIZE; j++) {
            GArray[i][j] = '-';
            printf("%c ", GArray[i][j]);
        }
        printf("\n");
    }
    fflush(stdout);

    pid = fork();
    switch (pid) {
      case -1:  perror("Could not fork");
                break;
      case 0:   // --- child fills upper half of array with 'c'
                if (head) usleep(WAIT_TIME);
                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 with 'p'
                if (! head) usleep(WAIT_TIME);
                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");
    }
    fflush(stdout);
    if (pid > 0) wait(NULL);
    exit(0);
}
```

Datei: ProcA9_3.c

```
#define ANZAHL      15
#define WORK_HARD   10000000

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

int main(void) {
    FILE   *fdes;
    pid_t  pid;
    int    i;

    launchWorkLoad();               // 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);
            }
            fflush(stdout);
            stopWorkLoad();
            break;
    }
    printf("We are done\n");
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        printf("See file AnyOutPut.txt\n");
    }
    fflush(stdout);
    exit(0);
}
```

2.10 Threads, was ist anders?

Ziele

- Den Unterschied zwischen Thread und Prozess kennenlernen.
- Problemstellungen um Threads kennenlernen.
- Die PThread-Implementation kennen lernen.

Aufgaben

a) Studieren Sie Programm `ProcA10.c` und überlegen Sie, wie die Programmausgabe aussieht. Vergleichen Sie Ihre Überlegungen mit denjenigen aus Aufgabe 2.9 b) (`ProcA9_2.e`).

- Starten Sie `ProcA10.e` und vergleichen das Resultat mit Ihren Überlegungen.
- Was ist anders als bei `ProcA9_2.e`?

b) Setzen Sie in der Thread-Routine vor dem Befehl `pthread_exit()` eine unendliche Schleife ein, z.B. `while(1) { };`

- Starten Sie das Programm und beobachten Sie das Verhalten mit `top`. Was beobachten Sie und was schliessen Sie daraus? Hinweis: wenn Sie in `top` den Buchstaben `H` eingeben, werden die Threads einzeln dargestellt.
- Kommentieren Sie im Hauptprogram die beiden `pthread_join()` aus und fügen Sie dafür ein `sleep(3)` ein: starten Sie das Program. Was geschieht? Erklären Sie das Verhalten

Programm (ohne header)

Datei: `ProcA10.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 own 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");
    fflush(stdout);
    pthread_exit(0);
}
```

```
int main(void) {
    pthread_t   thread1, thread2;
    int         i,j, pthr;
    char        letter1, letter2;

    selectCPU(0);                       // run on CPU 0

    // flip coin to select p or c first
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srandom(tv.tv_usec);                // evaluate seed
    int head = (int)(random()) >> 7;    // flip coin
    head &= 0x1;
    if (head) {
        letter1 = 'p';
        letter2 = 'c';
    }
    else {
        letter1 = 'c';
        letter2 = 'p';
    }

    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");
    }
    printf("\n");

    pthr = pthread_create(&thread1, NULL, ThreadF, (void *)&letter1);
    if (pthr != 0) perror("Could not create thread");
    pthr = pthread_create(&thread2, NULL, ThreadF, (void *)&letter2);
    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");
    }
}
```

2.11 . . . und wie schnell sind sie?

Ziele

- Eine Möglichkeit zur Bestimmung der benötigten Rechenzeit kennenlernen.
- Verwendung der Funktion `times()`.

Aufgaben

a) Studieren Sie Programm `ProcA11.c` und starten Sie anschliessend `ProcA11.e`. Analysieren die Zeitausgabe (welcher Programmteil ist für welche Zeiten verantwortlich?).

b) Starten Sie das Programm mit einem Argument: `ProcA11.e 1`. Nun wird zusätzliche CPU-Last erzeugt: was ändert sich wesentlich im Vergleich zu der Ausgabe von oben und wieso?

Programme (ohne header)

Datei: `ProcA11_1.c`

```
int main(int argc, char *argv[]) {
    struct tms    startT, endT;
    clock_t       StartTime, EndTime;
    ....
    if (argc > 1)
        load = atoi(argv[1]);
    else
        load = 0;
    if (load > 0)
        launchWorkLoad();
    ticks = sysconf(_SC_CLK_TCK);
    StartTime = times(&startT);
    for (i = 0; i < FORKS; i++) {
        pid = fork();
        if (pid == 0) {
            for (j = 0; j < WORK_LOAD; j++) {};
            exit(0);
        }
        else if (pid > 0)
            wait(NULL);
        else
            assert(pid >= 0);
    }
    for (i = 0; i < FORKS; i++)
        for (j = 0; j < WORK_LOAD; j++) {};

    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;
    printf("\nElapsed:\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);
    if (load > 0)
        stopWorkLoad();
    exit(0);
}
```

2.12 Mr. Daemon, what's the time please?

Ziele

– Problemstellungen um *Daemons* 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

Für diese Aufgabe haben wir 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

a) 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 0.5 Sek. "Hallo, ich bins.... Pidi" plus seine Prozess-ID auf den Bildschirm. Mit dem Shell Befehl `ps` können Sie Ihre 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?

b) Starten Sie nun das Programm bzw. 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 den Befehl "**`ps ajx | grep MrTimeDaemon`**" eingeben: was fällt Ihnen am Output auf? Was schliessen Sie aus Ihren Beobachtungen?

c) Starten Sie MrTimeDaemon erneut, was geschieht?

d) Stoppen Sie nun `MrTimeDaemon` mit **`killall MrTimeDaemon`**.

e) 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 Rechner-Namen auf den TimeServer mit **`WhatsTheTimeMr`** zugreifen können.

Die Firewall können Sie mit folgendem Befehl wiederherstellen:

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

f) Studieren Sie `MrTimeDaemon.c`, `Daemonizer.c` und `TimeDaemon.c` und analysieren Sie, wie die Daemonisierung abläuft. Entfernen Sie die Kommentare 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 alle für die vollständige Daemonisierung notwendigen Schritte.

g) 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 bzw. 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 der "Aufenthaltsort" des Daemons gesetzt. Die lock-Datei wird benötigt um sicherzustellen, dass der Daemon nur einmal gestartet werden 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 problematisch, weil der Inhalt dieses Verzeichnisses jederzeit gelöscht werden kann, bzw. 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 vom Elternprozess). Am Schluss startet der Daemonizer das eigentliche Daemonprogramm: `TimeDaemon.e`.

Daemonprogramm: TimeDaemon.c

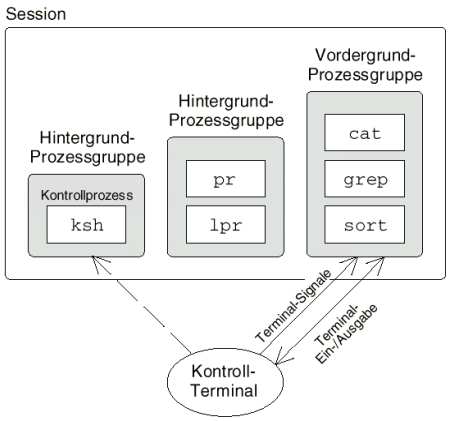
Das Daemonprogramm wartet in einer unendlichen Schleife auf Anfragen zur Zeit und schickt die Antwort an den Absender zurück. Die Datenkommunikation ist, wie schon erwähnt, mit Sockets implementiert, auf die wir aber im Rahmen dieses Praktikums nicht weiter eingehen wollen (wir stellen lediglich Hilfsfunktionen 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. Zu jedem Prozess gibt es 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. Die Prozessgruppen können 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.

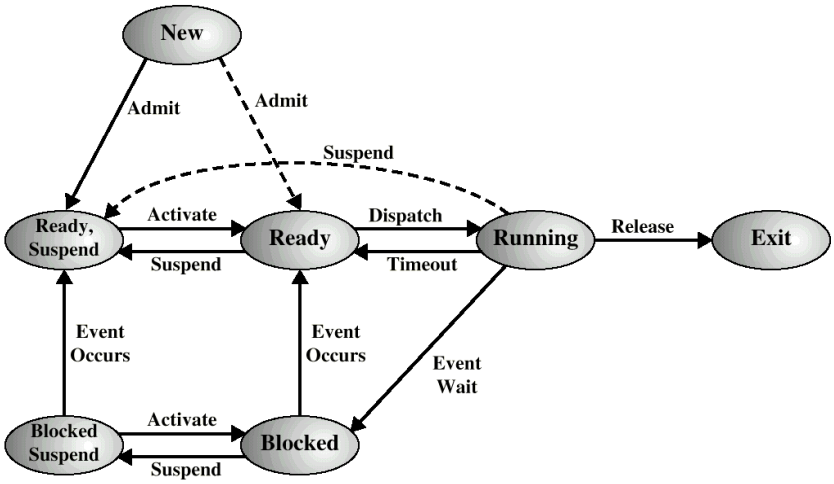
- Prozess 0 wird zur Bootzeit erzeugt und wird Swapper oder Scheduler-Prozess genannt. Dieser Systemprozess ist Teil des Kerns. Eine seiner Aufgaben ist es, Prozess 1, genannt Init, zu erzeugen. Alle weiteren Prozessen stammen in irgendeiner Form von Prozess 1 ab.
- Prozess 1 ist im Gegensatz zum Swapper 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 bzw. 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 mit 7 States:



3.3 Dämon-Prozesse

Dämonen oder englisch Daemons sind eine spezielle Art von Prozessen (bzw. 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 verwenden in diesem Praktikum die POSIX-kompatiblen *pthreads*. Diese Linux-Implementation unterstützt 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 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.