SNP: Prozesse und Threads

Quelle: https://www.wikiwand.com/de/Ein-Mann-Orchester

1 Übersicht

In diesem Praktikum werden wir uns mit Prozessen, Prozesshierarchien und Threads beschäftigen, um ein gutes Grundverständnis dieser Abstraktionen zu erhalten. Sie werden bestehenden Code analysieren und damit experimentieren. D.h. dies ist nicht ein «Codierungs»-Praktikum, sondern ein «Analyse»- und «Experimentier»-Praktikum.

1.1 Nachweis

Dieses Praktikum ist eine leicht abgewandelte Variante des ProcThreads Praktikum des Moduls BSY, angepasst an die Verhältnisse des SNP Moduls. Die Beispiele und Beschreibungen wurden, wo möglich, eins-zu-ein übernommen.

Als Autoren des BSY Praktikums sind genannt: M. Thaler, J. Zeman.

2 Lernziele

In diesem Praktikum werden Sie sich mit Prozessen, Prozesshierarchien 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.

- Sie können Prozesse erzeugen und die Prozesshierarchie erklären
- Sie wissen was beim Erzeugen eines Prozesses vom Elternprozess vererbt wird
- Sie wissen wie man auf die Terminierung von Kindprozessen wartet
- Sie kennen die Unterschiede zwischen Prozessen und Threads

3 Aufgaben

Das Betriebssystem bietet Programme um die aktuellen Prozesse und Threads darzustellen.

Die Werkzeuge kommen mit einer Vielzahl von Optionen für die Auswahl und Darstellung der Daten, z.B. ob nur Prozesse oder auch Threads aufgelistet werden sollen, und ob alle Prozesse oder nur die «eigenen» Prozesse ausgewählt werden sollen, etc.

Siehe die entsprechenden **`man`** Pages für weitere Details.

Eine Auswahl, welche unter Umständen für die folgenden Aufgaben nützlich sind:

`ps`	Auflisten der Prozess Zustände zum gegebenen Zeitpunkt.
`pstree`	Darstellung der gesamten Prozesshierarchie.
`top`	Wie **`ps`**, aber die Darstellung wird in Zeitintervallen aufdatiert.
`htop`	Wie **`top`**, aber zusätzlich dazu die Auslastung der CPU in einem System mit mehreren CPUs.
`lscpu`	Auflisten der CPUs.
`cat /proc/cpuinfo`	Ähnlich zu **`lscpu`**, aber mit Zusatzinformationen wie enthaltene CPU Bugs (z.B. ***`bugs: cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit`***)

3.1 Aufgabe 1: Prozess mit fork() erzeugen

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 **`ProcA1.c`** und beschrieben Sie was geschieht.

fork() parent-child
printf…
parent wartet auf child
beide letztes printf…

b) Notieren Sie sich, was ausgegeben wird. Starten Sie das Programm und vergleichen Sie die Ausgabe mit ihren Notizen? Was ist gleich, was anders und wieso?

i vor fork: 5
... wir sind die Eltern 14771 mit i=4 und Kind 14772,
unsere Eltern sind 2659
... ich bin das Kind 14772 mit i=6, meine Eltern sind 14771
. . . . . und wer bin ich ?
. . . . . und wer bin ich ?

3.2 Aufgabe 2: Prozess mit fork() und exec(): Programm Image ersetzen

Ziele

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

Aufgaben

a) Studieren Sie zuerst die Programme **`ProcA2.c`** und **`ChildProcA2.c`**.

b) Starten Sie **`ProcA2.e`** und vergleichen Sie die Ausgabe mit der Ausgabe unter Aufgabe 1. Diskutieren und erklären Sie was gleich ist und was anders.

Nach dem fork() wird im child Prozess execl() ausgeführt und somit alle Code Zeilen nach execl() nicht mehr ausgeführt.

c) Benennen Sie **`ChildProcA2.e`** auf **`ChildProcA2.f`** um (Shell Befehl **`mv`**) und überlegen Sie, was das Programm nun ausgibt. Starten Sie **`ProcA2.e`** und vergleichen Sie Ihre Überlegungen mit der Programmausgabe.

Wenn execl() fehlschlägt, werden die Code Zeilen nach execl() ausgeführt.

d) Nennen Sie das Kindprogramm wieder **`ChildProcA2.e`** und geben Sie folgenden Befehl ein: **`chmod -x ChildProcA2.e`**. Starten Sie wiederum **`ProcA2.e`** und analysieren Sie die Ausgabe von **`perror("...")`**. Wieso verwenden wir **`perror()`**?

Gleich wie bei c).

perror() schreibt eine Fehlermeldung entsprechend errno auf stderr, denn execl() setzt im Fehlerfall errno.

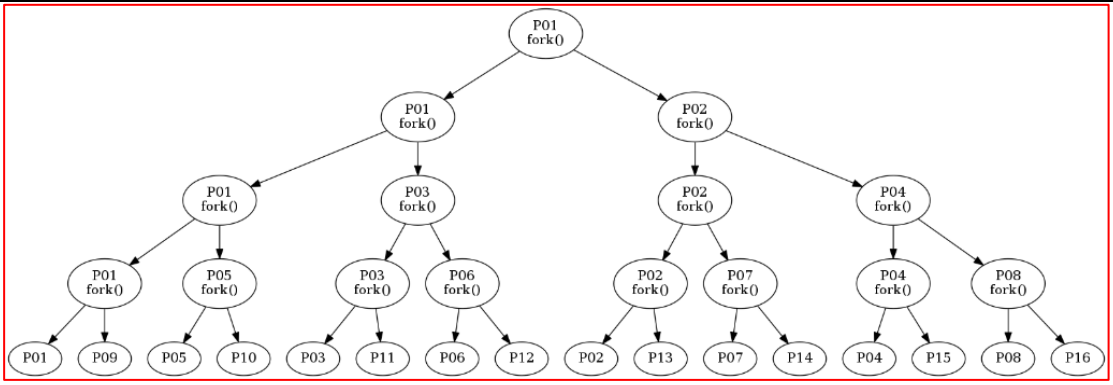
3.3 Aufgabe 3: Prozesshierarchie analysieren

Ziele

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

Aufgaben

a) Studieren Sie zuerst Programm **`ProcA3.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 **`ProcA3.e`** starten, zuerst die PID der Shell erfragen, z.B. über **`echo $$`**. Wenn Sie nun den Befehl **`pstree -n -p pid-von-oben`** eingeben, wird nur die Prozesshierarchie ausgehend von der Bash Shell angezeigt: **`-n`** sortiert die Prozesse numerisch, **`-p`** zeigt für jeden Prozess die PID an.

```
vagrant@snp:~$ pstree -n -p 2659
bash(2659)───ProcA3.e(15134)─┬─ProcA3.e(15135)─┬─ProcA3.e(15139)─┬─ProcA3.e(15146)───ProcA3.e(15149)
                             │                 │                 └─ProcA3.e(15147)
                             │                 ├─ProcA3.e(15140)───ProcA3.e(15142)
                             │                 └─ProcA3.e(15141)
                             ├─ProcA3.e(15136)─┬─ProcA3.e(15143)───ProcA3.e(15148)
                             │                 └─ProcA3.e(15144)
                             ├─ProcA3.e(15137)───ProcA3.e(15145)
                             └─ProcA3.e(15138)
vagrant@snp:~$
```

Hinweis: alle erzeugten Prozesse müssen arbeiten (d.h. nicht terminiert sein), damit die Darstellung gelingt. Wie wird das im gegebenen Programm erreicht?

sleep(10) erlaubt es, während 10 Sekunden das Kommando pstree aufzurufen.

3.4 Aufgabe 4: Zeitlicher Ablauf von Prozessen

Ziele

- Verstehen, wie Kind- und Elternprozesse zeitlich ablaufen.

Aufgaben

a) Studieren Sie Programm **`ProcA4.c`**. Starten Sie nun mehrmals hintereinander das Programm **`ProcA4.e`** und vergleichen Sie die jeweiligen Outputs (leiten Sie dazu auch die Ausgabe auf verschiedene Dateien um). Was schliessen Sie aus dem Resultat?

```
0       Mother
1       Mother
0               Child
1               Child
2       Mother
3       Mother
2               Child
3               Child
4               Child
4       Mother
5       Mother
5               Child
6               Child
6       Mother
7       Mother
7               Child
8       Mother
8               Child
9               Child
9       Mother
10      Mother
10              Child
11              Child
11      Mother
12      Mother
12              Child
13      Mother
13              Child
14              Child
14      Mother
15      Mother
15              Child
16              Child
16      Mother
17      Mother
17              Child
18              Child
18      Mother
19              Child
I go it ...
19      Mother
I go it ...
```

```
0               Child
0       Mother
1       Mother
1               Child
2       Mother
2               Child
3               Child
3       Mother
4       Mother
4               Child
5               Child
5       Mother
6       Mother
6               Child
7               Child
7       Mother
8       Mother
8               Child
9               Child
9       Mother
10              Child
11              Child
10      Mother
11      Mother
12      Mother
12              Child
13              Child
13      Mother
14              Child
14      Mother
15      Mother
15              Child
16              Child
16      Mother
17      Mother
17              Child
18      Mother
18              Child
19              Child
I go it ...
19      Mother
I go it ...
```

Es kann nicht vorausgesagt werden, in welcher Reihenfolge auch ganz einfache Prozesse abgearbeitet werden.

Hier wird CPU Last mittels for-Loop auf einer einzigen CPU simuliert.

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

3.5 Aufgabe 5: Waisenkinder (Orphan Processes)

Ziele

- Verstehen, was mit verwaisten Kindern geschieht.

Aufgaben

a) Analysieren Sie Programm **`ProcA5.c`**: was läuft ab und welche Ausgabe erwarten Sie?

Der Child Prozess schreibt während 5 Sekunden die Parent ID.

Der Parent Prozess terminiert nach 2 Sekunden – somit entsteht ein Orphan.

→ Der Orphan wird von Prozess mit ID 1 adoptiert.

b) Starten Sie ProcA5.e: der Elternprozess terminiert: was geschieht mit dem Kind?

```
... ich bin das Kind 15451
Mein Elternprozess ist 15450
Mein Elternprozess ist 15450
Mein Elternprozess ist 15450
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
... so das wars

*** and here my new parent ****

  PID TTY          TIME CMD
    1 ?        00:00:02 systemd
```

c) Was geschieht, wenn der Kindprozess vor dem Elternprozess terminiert? Ändern Sie dazu im **`sleep()`** Befehl die Zeit von 2 Sekunden auf 12 Sekunden und verfolgen Sie mit top das Verhalten der beiden Prozesse, speziell auch die Spalte S.

In top wird nichts sichtbar, aber mit pstree -n -p <BASH-ID> sieht man die Prozess Hierarchie während der Parent lebt, obwohl das Kind schon terminiert ist (Kind ist dann ein Zombi).

3.6 Aufgabe 6: Terminierte, halbtote Prozesse (Zombies)

Ziele

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

Aufgaben

1. Analysieren Sie das Programm **`ProcA6.c`**.

   Es werden vier Kinder erzeugt welche nach unterschiedlicher Zeit terminieren.
   Der Eltern Prozess wartet auf ein Kind nach dem andern – wenn ein Kind terminiert ist und der Eltern Prozess noch nicht darauf gewartet hat, ist das terminierte Kind ein Zombi.

2. Starten Sie das Script **`mtop`** bzw. **`mtop aaaa.e`**. Es stellt das Verhalten der Prozesse dynamisch dar.

   Hinweis: **`<defunct>`** = Zombie.

3. Starten Sie **`aaaa.e`** und verfolgen Sie im **`mtop`**-Fenster was geschieht. Was beachten Sie?

```
vagrant  15639  0.0  0.0   4384   796 pts/0    S+   18:59   0:00 ./aaaa.e
vagrant  15640  0.0  0.0      0     0 pts/0    Z+   18:59   0:00 [aaaa.e] <defunct>
vagrant  15641  0.0  0.0      0     0 pts/0    Z+   18:59   0:00 [aaaa.e] <defunct>
vagrant  15642  0.0  0.0   4384    72 pts/0    S+   18:59   0:00 ./aaaa.e
vagrant  15643  0.0  0.0  38380  3644 pts/1    R+   18:59   0:00 ps -ux
```

4. 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.

   1. parent fork and wait for child
   2. child fork grandchild which becomes the **«detached»** process
   3. child terminate which makes the started grandchild an **orphan**

3.7 Aufgabe 7: Auf Terminieren von Kindprozessen warten

Vorbemerkung: Diese Aufgabe verwendet Funktionen welche erst in der Vorlesung über *Inter-Process-Communication (IPC)* im Detail behandelt werden.

Sie können diese Aufgabe bis dann aufsparen oder die verwendeten Funktionen selber via **`man`** Pages im benötigten Umfang kennenlernen: **`man 2 kill`** und **`man 7 signal`**.

Ziele

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

Aufgaben

a) Starten Sie das Programm **`ProcA7.e`** und analysieren Sie wie die Ausgabe im Hauptprogramm zustande kommt und was im Kindprozess **`ChildProcA7.c`** abläuft.

Parent ohne Argument = Child moit Argument 0, somit Child mit exit 0.

b) Starten Sie **`ProcA7.e`** und danach nochmals mit **`1`** als erstem Argument. Dieser Argument Wert bewirkt, dass im Kindprozess ein "Segmentation Error" erzeugt wird, also eine Speicherzugriffsverletzung. Welches Signal wird durch die Zugriffsverletzung an das Kind geschickt? Diese Information finden Sie im Manual mit **`man 7 signal`**. Schalten Sie nun core dump ein (siehe README) und starten Sie **`ProcA7.e 1`** erneut und analysieren Sie die Ausgabe.

Core File enablen (nicht wie im README File angegeben… ☹):

```
ulimit -c unlimited
```

Hinweis: ein core Dump ist ein Abbild des Speichers z.B. zum Zeitpunkt, wenn das Programm abstürzt (wie oben mit der Speicher Zugriff Verletzung). Der Dump wird im File **`core`** abgelegt und kann mit dem **`gdb`** (GNU-Debugger) gelesen werden (siehe **`README`**). Tippen Sie nach dem Starten des Command Line UI des gdb **`where`** gefolgt von **`list`** ein, damit sie den Ort des Absturzes sehen. Mit **`quit`** verlassen Sie gdb wieder.

c) Wenn Sie **`ProcA7.e 2`** starten, sendet das Kind das Signal 30 an sich selbst. Was geschieht?

Terminiert mit Signal 30, kein Core Dump.

d) Wenn Sie **`ProcA7.e 3`** starten, sendet ProcA7.e das Signal SIGABRT (abort) an das Kind: was geschieht in diesem Fall?

Liste der Signale:

```
man 7 signal
```

Signal 6 = Abort → core dump (siehe obige man Page)

e) Mit **`ProcA7.e 4`** wird das Kind gestartet und terminiert nach 5 Sekunden. Analysieren Sie wie in ProcA7.e der Lauf- bzw. Exit-Zustand des Kindes abgefragt wird (siehe dazu auch **`man 3 exit`**).

Bessere man Page für Abfrage von Prozess Terminierung:

```
man 2 wait
```

Makros: WIF…, etc.

wait() blockiert bis ein Kind terminiert.

waitpid(…WNOHANG) gibt direkt 0 zurück wenn das Kind noch am Laufen ist.

3.8 Aufgabe 8: Kindprozess als Kopie des Elternprozesses

Ziele

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

Aufgaben

a) Analysieren Sie Programm **`ProcA8_1.c`**: was gibt das Programm aus?

- Starten Sie **`ProcA8_1.e`** und überprüfen Sie Ihre Überlegungen.
- Waren Ihre Überlegungen richtig? Falls nicht, was könnten Sie falsch überlegt haben?

Output wie erwartet:

1. gemeinsamer Code vor dem fork()
2. individueller Code innerhalb dem fork()-Switch
3. gemeinsamer Code nach dem Switch

b) Analysieren Sie Programm **`ProcA8_2.c`**: was gibt das Programm aus?

- Starten Sie **`ProcA8_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?

Nach dem fork() ist das gesamte Image eine Kopie, das beinhaltet auch die globalen Variablen – diese sind nicht «shared».

Nach dem fork() leben die Prozesse ihre «eigenen Leben», d.h. die Reihenfolge der Ausführung ist irrelevant.

c) Analysieren Sie Programm **`ProcA8_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 **`ProcA8_3.e`** und überprüfen Sie Ihre Überlegungen.
- Waren Ihre Überlegungen richtig? Falls nicht, wieso nicht?

Dasselbe File (der selbe File Deskriptor) wird von beiden Prozesse verendet da er vor dem fork() geöffnet wurde. Den Kernel interessiert nicht, welcher Prozess den (indirekt) via write() auf das File schreibt – es wird in der Reihenfolge der Aufrufe geschrieben. Da die Prozesse (quasi-) parallel laufen, wechselt der Scheduler zu beliebigen Zeitpunkten zwischen den Prozessen, und somit ist nicht vorhersagbar, in welcher Reihenfolge individuelle write() Aufrufe unter den Prozessen erfolgen.

3.9 Aufgabe 9: Unterschied von Threads gegenüber Prozessen

Ziele

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

Aufgaben

a) Studieren Sie Programm **`ProcA9.c`** und überlegen Sie, wie die Programmausgabe aussieht. Vergleichen Sie Ihre Überlegungen mit denjenigen aus Aufgabe 8.2 b) (**`ProcA8_2.e`**).

- Starten Sie **`ProcA9.e`** und vergleichen das Resultat mit Ihren Überlegungen.
- Was ist anders als bei **`ProcA8_2.e`**?

BTW: Fehler im Code: errno wird nicht gesetzt durch irgend eine pthread…() Funktion. Der Grund ist möglicherweise, weil errno eine «thread-local» Variable ist. Siehe man 3 errno: *[…] errno is thread-local; setting it in one thread does not affect its value in any other thread […]*

Die Globalen Variablen werden nun zwischen den Thread geteilt. Somit greifen beide auf dieselben Daten zu. Die Reihenfolgen der individuellen Zugriffe ist auch hier nicht vorhersagbar.

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()`** Aufrufe aus und starten Sie das Programm. Was geschieht? Erklären Sie das Verhalten.

Teil 1: beide Threads laufen mit 50% CPU: «busy wait».

Teil 2: main terminiert und somit auch die Threads

4 Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	**Kriterium**	**Punkte**
	Sie können die gestellten Fragen erklären.	
1	*Prozess mit fork() erzeugen*	1
2	*Prozess mit fork() und exec(): Programm Image ersetzen*	
3	*Prozesshierarchie analysieren*	1
4	*Zeitlicher Ablauf von Prozessen*	
5	*Waisenkinder (Orphan Processes)*	1
6	*Terminierte, halbtote Prozesse (Zombies)*	
7	*Auf Terminieren von Kindprozessen warten*	1
8	*Kindprozess als Kopie des Elternprozesses*	
9	*Unterschied von Threads gegenüber Prozessen*	