Universität Heidelberg
Sommersemester 2022

AG Parallele und Verteilte Systeme (PVS)
Artur Andrzejak

2. Übung zur Vorlesung „Betriebssysteme und Netzwerke“ (IBN)

Abgabedatum: 10.05.2022, 11:00 Uhr

Aufgabe 1 (5 Punkte)

In this exercise you will create a shell script that organizes a folder full of images based on each image's creation date or last modified date. In detail, the script should work as follows:

- The script should accept as command line arguments a directory path, and an optional flag whether creation date or last modifed date should be used (think about a default value). Any non-directory path should be rejected by the script with a proper error message.
- The script should then inspect every `.jpeg` and `.png` file within this directory and create a subfolder per each year-month combination encountered as a creation date, or last modified date. Each subfolder should be named using the following format: YYYY-MM (e.g., 2019-01, 2019-02).
- Then, the script should move the files to their respective folder, based on each file's creation date/last modified date.

For instance, running the script with the input of a directory `dir` containing the following files (filename, creation date/ last modified date):

```
dir/
    pic1.jpeg 2019-03-21
    pic2.jpeg 2019-03-28
    pic3.png  2019-04-02
    other-file.txt  2019-04-04
```

... should yield the following file structure:

```
dir/
    other-file.txt 2019-04-04
    2019-03/
       pic1.jpeg 2019-03-21
       pic2.jpeg 2019-03-28
    2019-04/
       pic3.png  2019-04-02
```

Submit your code, a suitable test data set with at least 6 files (please only small size - 100 kB at most in total - use dummy `.jpeg` and `.png` files), and a description of the created file structure.

Aufgabe 2 (1 Punkt)

Lesen Sie den Abschnitt 4.4 des Buches *The Linux Kernel*[1] durch.

1. Wie groß ist die maximale Anzahl der geöffneten Dateien eines Prozesses laut diesem Text? Was ist diese Anzahl auf heutigen Linux-Systemen, bzw. kann sie verändert werden?
2. Weiterhin betrachten Sie den Quelltext der `task_struct` (PCB in Linux) des neuesten Linux-Kernels[2]. In welcher Zeile finden Sie (eine Referenz auf) die Datenstruktur mit den Informationen zu allen geöffneten Dateien eines Prozesses?

Aufgabe 3 (1 Punkt)

Wie viele Prozesse (inklusive des 1. Elternprozesses) werden durch das folgende Programm beim Aufruf mit dem Kommandozeilenargument 10 erzeugt? Begründen Sie Ihre Antwort durch die Erläuterungen, was in relevanten Teilen des Codes passiert.

```
int main(int argc, char* argv[]) {
  int i;
  for( i = 0; i < atoi(argv[1]); i++) {
    fork();
  }
  return 0;
}
```

Aufgabe 4 (1 Punkt)

Erinnern Sie sich an das Konzept der Prozesszustände. Finden Sie folgendes dazu heraus.

a) Mit welchem `bash`-Befehl kann man den Zustand laufender Prozesse erfahren?

b) In der Vorlesung wurden die Zustände aus UNIX vorgeführt und Übergänge zwischen diesen Zuständen diskutiert. Finden Sie heraus, wie die Zustände unter Linux benannt werden, und welchen Zuständen aus der Vorlesung sie entsprechen.

Aufgabe 5 (5 Punkte)

Bei dem Programm aus Vorlesung 4 („POSIX Pthreads – Beispiel“) kann die Reihenfolge der Ausführung der Aktionen „Erzeugung eines Threads“ und „Nachricht drucken“ zufällig miteinander verschachtelt sein.

[1] http://www.tldp.org/LDP/tlk/tlk-toc.html
[2] http://elixir.free-electrons.com/linux/latest/source/include/linux/sched.h

1. Schreiben Sie das Programm so um, dass die Ausführung die folgende *strikte* Reihenfolge einhält: „Erzeugung von Thread 1“; „Nachricht von Thread 1 drucken“; „Beenden von Thread 1“; „Erzeugung von Thread 2“; . . . , usw. Testen Sie Ihr Programm anschließend. Bitte reichen Sie Ihr Programm und eine Beispielausgabe ein.
2. Was ist (bei Ihrer Implementierung und Ihrem System) der maximale Wert $N_{\max}$ der Konstante *NUM_ THREADS*, bei dem die Ausführung des Programms etwa 10 Sekunden dauert (eine Näherung reicht)?
3. Wie lange benötigt ein Programm mit einem einzigen Thread (d.h. ein „normales“ sequentielles Programm), das die Funktion *TaskCode* genau $N_{\max}$ mal in einer einfachen Schleife aufruft? Können Sie daraus quantitative Aussagen über die Zeit der Ausführung von (gewissen) *pthread_ **-Funktionen machen?

Aufgabe 6 (1 Punkt)

Beim Online-Spiel *The Deadlock Empire*[3] nehmen Sie die Rolle des Schedulers ein, der die Ausführung von Threads steuert. Lösen Sie die 3 Aufgaben (Levels) aus dem Kapitel *Unsynchronized Code*. Geben Sie an, wie Sie beide Threads in die kritische Region manövrieren bzw. wie Sie die Assertion auslösen.

Aufgabe 7 (2 Punkte)

Betrachten Sie die Implementation von Locks mittels des Hardware-Befehls „Test and Set Lock”. Beschreiben Sie folgende Situationen Zeile für Zeile anhand des Assemblercodes aus der Vorlesung, und beschreiben Sie dabei den Inhalt des Registers `RX` und der Variable `LOCK`. Beachten Sie dabei, dass der betrachtete Thread auch mit anderen Threads interagiert und gehen Sie darauf mit Hilfe von Beispielen ein.

1. Die gesamte Routine `enter_region` wird genau einmal durchlaufen.

2. Die Schleife in `enter_region` wird beim zweiten Durchlauf verlassen.

Zur Erinnerung: Der Befehl `CMP RX, #0` vergleicht den Inhalt von Register `RX` mit der Zahl `0`. Der Befehl `JNE` führt einen bedingten Sprung (hier: zum Label `enter_region`) aus bei Nicht-Gleichheit (`NE`, non-equality) bei der letzten `CMP`-Operation. Der Befehl RET verursacht, dass die aktuelle Routine (hier: `enter_region`) verlassen wird und man zur Stelle ihres Aufrufs zurückkehrt.

Aufgabe 8 (Bonus, 4 Punkte)

Verallgemeinern Sie Peterson's Algorithmus so, dass er mit `n` Prozessen funktioniert. Das heißt, die n gleichzeitig laufenden Prozesse wissen von einander und zeigen ein ähnlich „höfliches“ Verhalten

[3] https://deadlockempire.github.io/

wie im Original-Algorithmus, das sicherstellt, dass sich immer höchstens ein Prozess in *seinem* kritischen Bereich befindet. Es genügt, wenn Sie Ihre Implementation in Pseudo-Code angeben. Erläutern Sie, wie ihre Implementation funktioniert. Stellen Sie insbesondere dar, warum bei Ihrer Implementation a) die kritischen Bereiche voreinander geschützt sind, b) jeder Prozess irgendwann seinen kritischen Bereich betritt (d.h. es darf auch nicht zu Deadlocks kommen).

Hinweis für eine mögliche Implementation: Stellen Sie sich vor, es gibt `n` „Warteräume". Erst, wenn ein Prozess erfolgreich alle Warteräume durchlaufen hat, betritt er seinen kritischen Bereich. Für jeden Raum gibt es eine Variable, die anzeigt, welcher der `n` Prozesse in diesem Raum als *wartend* gilt. Dieser Prozess kann nicht aus diesem Raum weitergehen, solange es Prozesse im selben oder einem der nächsten Räume gibt. Ansonsten geht er einen Raum weiter. Betritt ein Prozess einen neuen Raum, setzt er die *wartend*-Variable dieses Raumes auf seine ID (und befreit damit einen anderen Prozess aus dieser Rolle, dessen ID dort vorher gespeichert war). Welcher Prozess in welchem Raum ist, und welcher der Prozesse in einem Raum der *Wartende* ist, muss allen Prozessen bekannt sein, also in gemeinsam geteilten Speicher abgelegt.