UNIVERSITÄT OSNABRÜCK

ESS

marcel.koeppen [☺] uni-osnabrueck.de
marcel.luetkedreimann [☺] uni-osnabrueck.de

Übung zur Vorlesung
Betriebssysteme
Wintersemester 2021/22

Programmierblatt 5: Threads mit `clone` Handarbeit, aber ohne Nadeln

➪ **Abgabe der Lösungen bis Montag, 24. Januar 14:00 im AsSESS**

Geben Sie Ihre Entwicklungsschritte jeweils in den angegebenen Dateien ab.

Bisher haben wir Threads immer mit der `pthread`-Bibliothek realisiert. Den zugrundeliegenden Linux-Systemdienst kann man allerdings auch direkt nutzen. Das wollen wir in dieser Aufgabe tun.

1 Threads erstellen (7 Punkte)

Unter Linux ist das Erzeugen von Prozessen und Threads sehr ähnlich implementiert. Für den Kernel sind beides „Tasks“, denen vom Scheduler CPU-Zeit zugeteilt wird. Deshalb nutzen `fork` und `pthread_create` intern denselben Systemdienst: **clone(2)**.

Als wesentlichen Unterschied zwischen Prozessen und Threads haben wir kennengelernt, dass sich die Threads eines Prozesses – bis auf den Stack – die Ressourcen des Prozesses teilen. Verschiedene Prozesse haben hingegen keine geteilten Ressourcen. Clone funktioniert nun so ähnlich wie fork: Ein Eltern-Task erzeugt einen Kind-Task. Welche Ressourcen sich Elter und Kind teilen, wird beim Aufruf von clone über einen Flag-Parameter eingestellt.

Da sich die Parameter des clone-Systemdienstes zwischen verschiedenen Kernel-Versionen und Systemarchitekturen unterscheiden, nutzen wir die Wrapper-Funktion aus der glibc, die eine einheitliche Schnittstelle zu clone bereitstellt.

Zunächst soll ein kleines Programm entstehen, das mit clone einen Thread erzeugt. Dieser Thread soll die Summe der ersten 5 Quadratzahlen berechnen, 2 Sekunden schlafen und beim Beenden die Summe als Exit-Status zurückliefern. Das Hauptprogramm soll die ID des Threads ausgeben, auf das Beenden des Threads warten und schließlich die berechnete Summe auf dem Bildschirm ausgeben.

1. Zuerst benötigen wir die Funktion, die im Thread ausgeführt werden soll. Implementieren Sie diese wie oben beschrieben. **Achtung:** Die Signatur der Thread-Funktionen ist bei `clone` anders als bei `pthread_create`: `int mein_thread(void *argument)`.
2. Als nächstes brauchen wir einen Stack. Reservieren Sie mit malloc dafür einen Speicherbereich mit einer angemessenen Größe. Mit diesem Parameter können Sie gern einmal experimentieren. Die Standard-Stackgröße des Systems in Kilobyte kann man sich in der Shell mit dem Befehl ulimit -s anzeigen lassen.
3. Wie schon erwähnt, werden beim Aufruf von clone einige Einstellungen über Flags vorgenommen, die mit dem Operator „|“ verodert werden. Die wichtigsten Flags sind CLONE_VM und SIGCHLD. CLONE_VM sorgt dafür, dass sich Eltern- und Kind-Task denselben Speicherbereich teilen. SIGCHLD stellt ein, dass beim Beenden des Threads ein SIGCHLD-Signal an den Eltern-Task gesendet wird. Mehr Informationen zu den verfügbaren Flags findet man in der Manpage zu clone.
4. Jetzt kann der neue Thread erzeugt werden: Rufen Sie clone mit der Thread-Funktion und den Flags auf. Außerdem müssen Sie die Adresse des Stack-Anfangs angeben. **Achtung: Auf x86 wächst der Stack von oben nach unten!** Speichern Sie die von clone zurückgegebene Thread-ID und geben Sie diese aus.

5. Ohne pthreads gibt es kein pthread_join, also müssen wir anders auf das Beenden des Threads warten: mit waitpid. Das funktioniert genau wie mit Prozessen. Warten Sie also auf das Ende des Threads, extrahieren Sie den Exit-Status des Threads und geben diesen auf der Konsole aus.

→ `clone_a1.c`

2 Stacknutzung messen (3 Punkte)

Nun wollen wir messen, wie viel Stack-Platz der Thread beim Berechnen der Summe benötigt. Dazu kann man das Watermarking-Verfahren benutzen: Der Stack wird vor dem Start des Threads mit einem Muster gefüllt. Nachdem der Thread beendet wurde, prüft man, wie groß der Bereich ist, in dem das Muster überschrieben wurde.

1. Füllen Sie den Stack nach dem Anfordern komplett mit dem Wert 0xAA (binär: 10101010).
2. Lassen Sie den Thread laufen und prüfen Sie nach dem Beenden des Threads, wie weit das Muster noch intakt ist. Geben Sie die Größe des **benutzten** Stack-Bereichs aus. Wenn Sie Ihren Stack auf diese Größe reduzieren, sollte das Programm weiter funktionieren.

→ `clone_a2.c`

Tipps zu den Programmieraufgaben:

- Kommentieren Sie Ihren Quellcode ausführlich, so dass wir auch bei Programmierfehlern im Zweifelsfall noch Punkte vergeben können!
- Denken Sie daran, dass viele Systemaufrufe fehlschlagen können! Fangen Sie diese Fehlerfälle ab (die Aufrufe melden dies über bestimmte Rückgabewerte, siehe die jeweiligen man-Pages), geben Sie geeignete Fehlermeldungen aus (z.B. unter Zuhilfenahme von **perror(3)**) und beenden Ihr Programm danach ordnungsgemäß.
- Die Programme sollen (soweit möglich) dem C11- und POSIX-Standard entsprechen sich mit dem gcc auf aktuellen Linux-Rechnern wie denen im CIP-Pool oder der BSVM übersetzen lassen. Z.B.:
  `gcc -std=c11 -Wall -Wextra -D_GNU_SOURCE -o clone clone_ax.c`
  Weitere (nicht zwingend zu verwendende) nützliche Compilerflags sind: `-Wpedantic -Werror -D_POSIX_SOURCE`
- Achten Sie darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt, z.B. durch Nutzung von `-Werror`.