

Übungen zu Systemprogrammierung 2 (SP2)

Ü6 – Mehrfädige Programme

Andreas Ziegler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2016 – 09. bis 13. Januar 2017

http://www4.cs.fau.de/Lehre/WS16/V_SP2



Agenda

- 6.1 Hinweise zur Evaluation
- 6.2 Klausurvorbereitung
- 6.3 Thread-Pool-Entwurfsmuster
- 6.4 Zusammenspiel von BS-Konzepten
- 6.5 Aufgabe 5: mother



Agenda

- 6.1 Hinweise zur Evaluation
- 6.2 Klausurvorbereitung
- 6.3 Thread-Pool-Entwurfsmuster
- 6.4 Zusammenspiel von BS-Konzepten
- 6.5 Aufgabe 5: mother



Hinweise zur Evaluation

- Bei Kommentaren, die sich auf einen bestimmten Übungsleiter beziehen, bitte dessen Namen **in jedem Feld** voranstellen
 - Kommentarfelder werden in der Auswertung durcheinandergewürfelt



Klausurvorbereitung

- Ab übernächster Semesterwoche (ab 23.01.): Klausurvorbereitung in der Tafelübung zur Vorbereitung auf die Klausur
- Wir erarbeiten die Klausur vom Juli 2015 (SS 2015) gemeinsam
 - Angabe ist auf Übungsseite (SP2 ⇒ Übung ⇒ Folien) verlinkt
 - Eine Vorbereitung der Klausur im Vorfeld der Tafelübung wird erwartet
- **Voraussichtlicher** Klausurtermin: Mittwoch, 22.02.2017

26-Multithreading_handout



Agenda

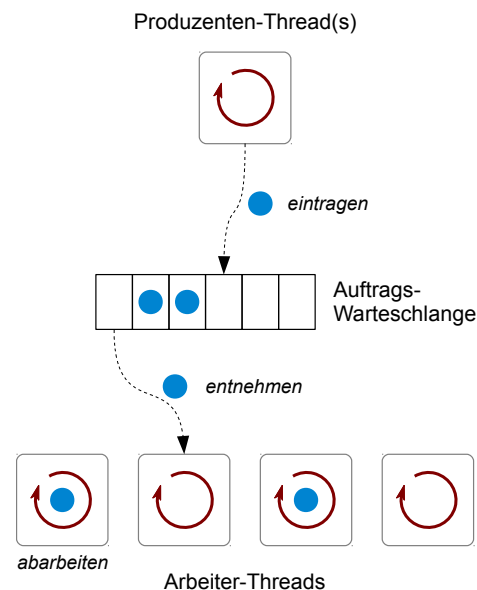
- 6.1 Hinweise zur Evaluation
- 6.2 Klausurvorbereitung
- 6.3 Thread-Pool-Entwurfsmuster
- 6.4 Zusammenspiel von BS-Konzepten
- 6.5 Aufgabe 5: mother

26-Multithreading_handout



Thread-Pool-Entwurfsmuster

- Feste Menge von Arbeiter-Threads:
 - laufen endlos
 - erhalten Aufträge zur Abarbeitung
- Verteilen der Aufträge mittels zentraler, synchronisierter Warteschlange (z. B. Ringpuffer)
- Vorteil: kein ständiges Erzeugen + Zerstören von Threads für Aufträge



26-Multithreading_handout



Agenda

- 6.1 Hinweise zur Evaluation
- 6.2 Klausurvorbereitung
- 6.3 Thread-Pool-Entwurfsmuster
- 6.4 Zusammenspiel von BS-Konzepten
- 6.5 Aufgabe 5: mother

26-Multithreading_handout



Threads und UNIX-Signale

- Signale können ...
 - an einen Thread gerichtet sein:
 - Synchron auftretende Signale (z. B. SIGSEGV, SIGPIPE)
 - Signale, die mit pthread_kill(3) geschickt wurden
 - an einen Prozess gerichtet sein:
 - Alle anderen Signale (z. B. mit kill(2) erzeugte Signale)
- Signalbehandlung gilt prozessweit:
 - An Thread gerichtete Signale werden von diesem bearbeitet
 - An Prozess gerichtete Signale werden von beliebigem Thread bearbeitet
- Signalmaske ist Thread-lokal:
 - Statt sigprocmask(2) muss pthread_sigmask(3) benutzt werden:
 - Verhalten von sigprocmask(2) in mehrfädigem Prozess ist undefiniert
 - Von einem Thread blockierte Signale, die ...
 - an diesen gerichtet sind, werden verzögert
 - an dessen Prozess gerichtet sind, werden von einem anderen Thread bearbeitet

Threads und Prozesse

- Verwendung von fork(2) in mehrfädigen Prozessen grundsätzlich problematisch:
 - Bei fork(2) wird nur der aufrufende Thread geklont; alle anderen Threads sind im Kind nicht mehr vorhanden
 - Gelockte Mutexe bleiben gelockt und können nicht freigegeben oder zerstört werden
 - Kind kann inkonsistenten Zustand kopieren
- Unproblematisch, wenn geforkt wird, um exec(2) auszuführen:
 - Beim Aufruf von exec(2) ...
 - werden alle Mutexe und Bedingungsvariablen zerstört
 - verschwinden alle Threads – bis auf den aufrufenden

Prozesse und offene Dateien

- Erinnerung: offene Dateien/Sockets/...
 - werden bei fork(2) an den neu erzeugten Kindprozess vererbt
 - bleiben bei exec(2) im neu geladenen Programm erhalten
- Dieses Verhalten ist unter Umständen unerwünscht!
 - Beispiel: Server will seine offenen Sockets nicht an ein von ihm gestartetes Programm weiterreichen
- Abhilfe: Close-on-exec-Flag für Dateideskriptoren
 - Dateideskriptoren, bei denen dieses Flag gesetzt ist, werden beim Aufruf von exec(2) automatisch geschlossen
 - Sofortiges Setzen beim Öffnen einer Datei:

```
int fd = open("index.html", O_RDONLY | O_CLOEXEC);
FILE *fp = fdopen(fd, "r");
```

Prozesse und offene Dateien, cont.

- Close-on-exec-Flag für Dateideskriptoren, Fortsetzung
 - Alternativ: Setzen mit fcntl(2):

```
int flags = fcntl(fd, F_GETFD, 0); // Alte Flags holen
fcntl(fd, F_SETFD, flags | FD_CLOEXEC); // Neue Flags setzen
```
 - dup(2), dup2(2) setzen Close-on-exec beim neuen Dateideskriptor zurück
 - Bei Verzeichnissen: opendir(3) setzt Close-on-exec automatisch

- 6.1 Hinweise zur Evaluation
- 6.2 Klausurvorbereitung
- 6.3 Thread-Pool-Entwurfsmuster
- 6.4 Zusammenspiel von BS-Konzepten
- 6.5 Aufgabe 5: mother



- Stark aufgebohrte Version der sister
- Neue Features:
 - Thread-Pool statt fork(2)
 - Auflistung von Verzeichnisinhalten (alphabetisch sortiert)
 - Ausführen von Perl-Skripten
- Ziel der Aufgabe:
 - Wiederholung etlicher in den SP-Übungen gelernter Konzepte

