

Übungen zu Systemprogrammierung 2 (SP2)

Ü 6 – Mehrfädige Programme

Christoph Erhardt, Jens Schedel, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2012/13 – 07. bis 11. Januar 2013

http://www4.cs.fau.de/Lehre/WS12/V_SP2



Agenda

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



Agenda

6.1 Hinweise zur Evaluation

6.2 Thread-Pool-Entwurfsmuster

6.3 Zusammenspiel von BS-Konzepten

6.4 Aufgabe 5: mother



- Bei Kommentaren, die sich auf einen bestimmten Übungsleiter beziehen, bitte dessen Namen **in jedem Feld** voranstellen
 - Kommentarfelder werden in der Auswertung durcheinandergewürfelt
- Frage „eigener Aufwand zur Vor- und Nachbereitung“:
 - Bitte nach Vorlesung und Übung auftrennen
 - Übung: den jeweiligen Wochenaufwand durch 2 teilen (rechnerisch: 2 x 1 Stunde Übung (Tafel + Rechner) pro Woche, Angabe je 45 Minuten)
 - Vorlesung: den jeweiligen Wochenaufwand nicht teilen (1 x 90 Minuten Vorlesung pro Woche, Aufwandsangabe je 90 Minuten)



Agenda

6.1 Hinweise zur Evaluation

6.2 Thread-Pool-Entwurfsmuster

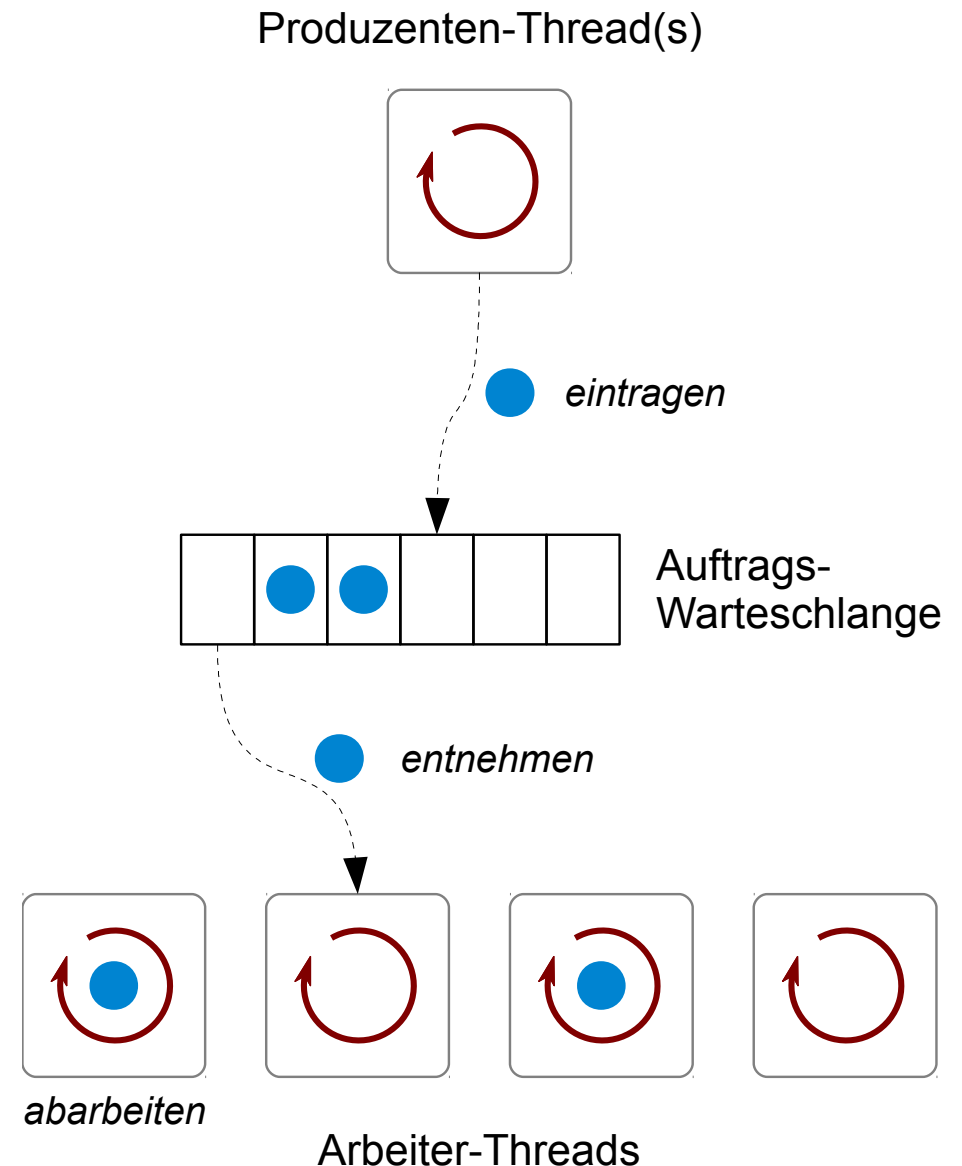
6.3 Zusammenspiel von BS-Konzepten

6.4 Aufgabe 5: mother



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



Agenda

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



- 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:
 - Von einem Thread blockierte Signale, die ...
 - an diesen gerichtet sind, werden verzögert
 - an dessen Prozess gerichtet sind, werden von anderem Thread bearbeitet
 - Statt `sigprocmask(2)` muss `pthread_sigmask(3)` benutzt werden:
 - Verhalten von `sigprocmask(2)` in mehrfädigem Prozess ist undefiniert



- 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



- 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("foo.txt", O_RDONLY | O_CLOEXEC);
```
 - Nachträgliches Setzen mit `fcntl(2)`:

```
fcntl(fd, F_SETFD, FD_CLOEXEC);
```



Agenda

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



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

