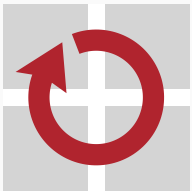
Übungen zu Systemprogrammierung 2

Ü6 – Mehrfädige Programme

Wintersemester 2023/24

Luis Gerhorst, Thomas Preisner, Jonas Rabenstein, Eva Dengler, Dustin Nguyen, Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Agenda

6.1 Thread-Pool-Entwurfsmuster

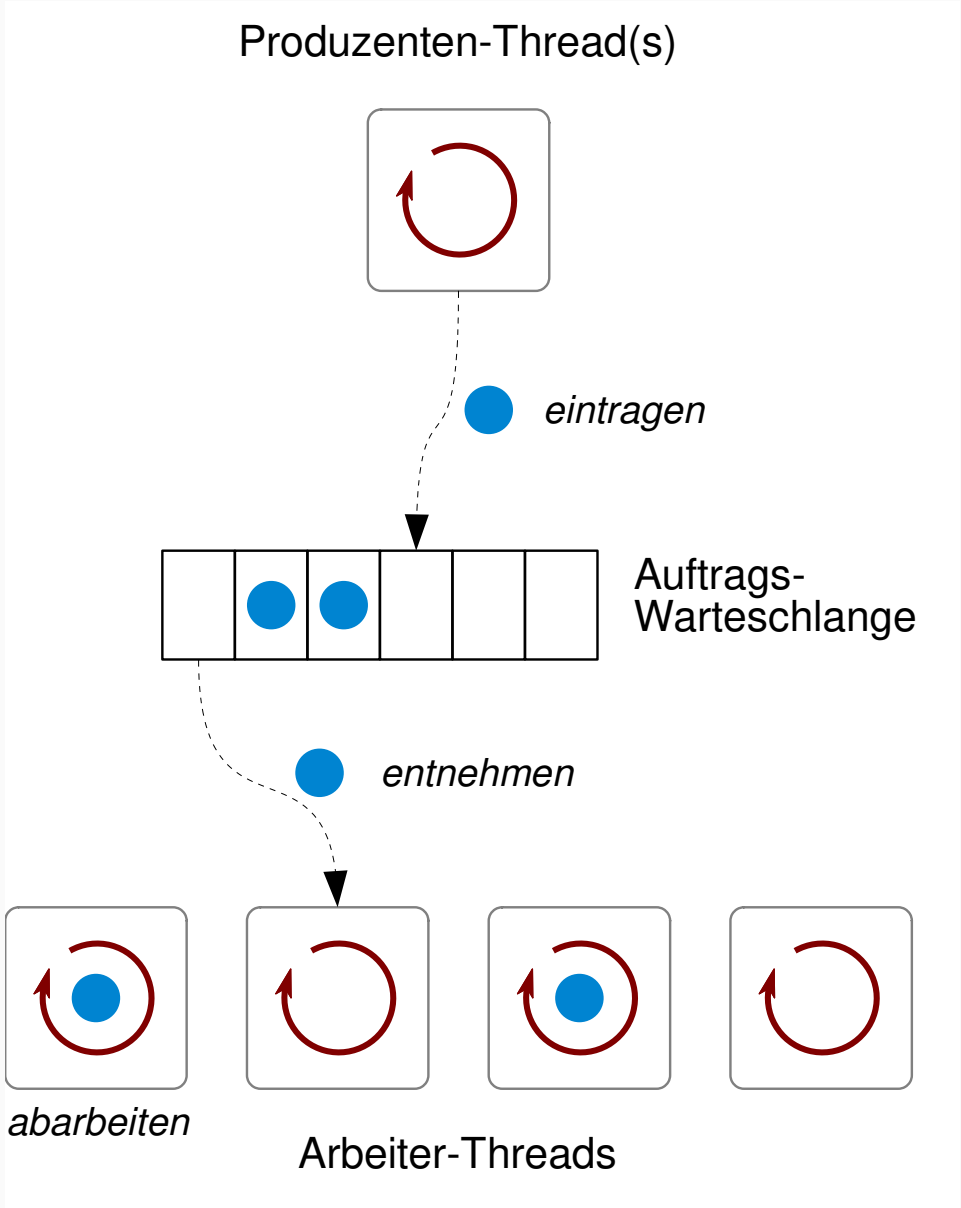
6.2 Zusammenspiel von BS-Konzepten

6.3 Aufgabe 5: `mother`

Agenda

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

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)

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

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
  - Neue Threads „erben“ Signalmaske des Erzeugers
  - 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

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(3)` auszuführen:
  - Beim Aufruf von `exec(3)` …
    - 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(3)` 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

Prozesse und offene Dateien

- Erinnerung: offene Dateien/Sockets/…
  - werden bei `fork(2)` an den neu erzeugten Kindprozess vererbt
  - bleiben bei `exec(3)` 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(3)` automatisch geschlossen
  - Sofortiges Setzen beim Öffnen einer Datei:

```c
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)`:

```c
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

Agenda

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