

# Übungen zu Systemprogrammierung 2

## Ü5 – Mehrfädige Programme

---

Wintersemester 2025/26

Jürgen Kleinöder, Thomas Preisner, Tobias Häberlein, Ole Wiedemann

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



**Lehrstuhl für Informatik 4**  
Systemsoftware

**FAU**  
Friedrich-Alexander-Universität  
Technische Fakultät



- 5.1 Thread-Pool-Entwurfsmuster
- 5.2 Zusammenspiel von BS-Konzepten
- 5.3 Aufgabe 5: mother



5.1 Thread-Pool-Entwurfsmuster

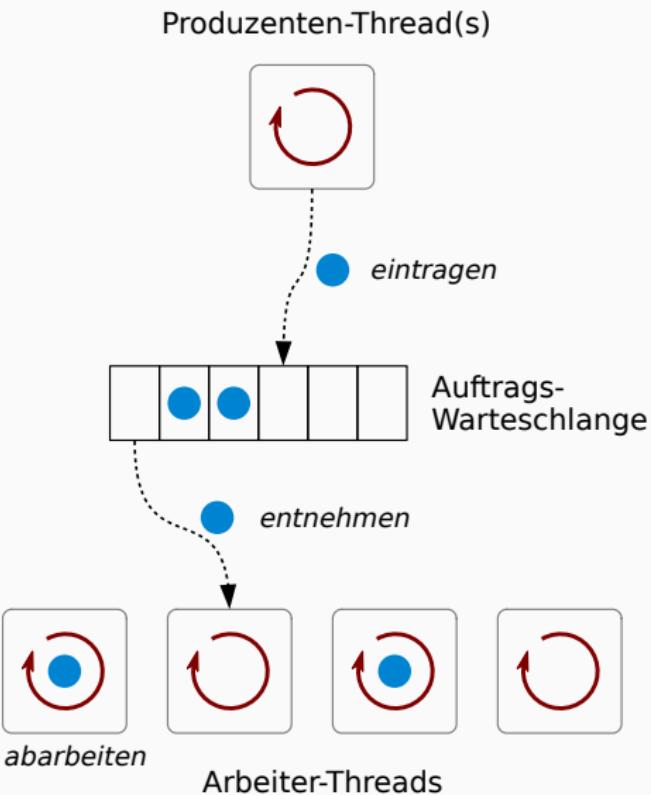
5.2 Zusammenspiel von BS-Konzepten

5.3 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



5.1 Thread-Pool-Entwurfsmuster

5.2 Zusammenspiel von BS-Konzepten

5.3 Aufgabe 5: mother



- Signale können ...
  - an einen Thread gerichtet sein:
    - Synchron auftretende Signale (z. B. `SIGSEGV`, `SIGPIPE`)
    - Signale, die mit `pthread_kill(3p)` geschickt wurden
  - an einen Prozess gerichtet sein:
    - Alle anderen Signale (z. B. mit `kill(3p)` 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(3p)` muss `pthread_sigmask(3p)` benutzt werden:
    - Verhalten von `sigprocmask(3p)` in mehrfädigem Prozess ist undefined
  - 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



- Verwendung von `fork(3p)` in mehrfädigen Prozessen grundsätzlich problematisch:
  - Bei `fork(3p)` 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(3p)` auszuführen:
  - Zwischen `fork(3p)` und `exec(3p)` dürfen im Kind nur async-signal-safe Funktionen verwendet werden
    - siehe `signal-safety(7)`
  - Beim Aufruf von `exec(3p)` ...
    - werden alle Mutexe und Bedingungsvariablen zerstört
    - verschwinden alle Threads – bis auf den aufrufenden



- Verwendung von (static) globalen Variablen in einer Funktion ist problematisch, wenn die Funktion von mehreren Threads gleichzeitig aufgerufen wird
  - Beispiel: `strtok(3p)`
  - Ohne Synchronisierung: Race-Conditions
  - Zustand wird unter Umständen wechselseitig überschrieben
  - Darf daher nicht in mehreren Threads gleichzeitig verwendet werden
- POSIX definiert deswegen einige Funktionen, die als thread-sichere Alternative verwendet werden können:
  - enden meist auf `_r`
  - Beispiel: `strtok_r(3p)`

```
char *state; // wird verwendet, um den Zustand zu speichern
strtok_r(str, " ", &state);
```



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

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



- Close-on-exec-Flag für Dateideskriptoren, Fortsetzung
  - Alternativ: Setzen mit fcntl(3p):

```
int flags = fcntl(fd, F_GETFD, 0);      // Alte Flags holen
fcntl(fd, F_SETFD, flags | FD_CLOEXEC); // Neue Flags setzen
```

- dup(3p), dup2(3p) setzen Close-on-exec beim neuen Dateideskriptor zurück
- Statt dup(3p) + fcntl(3p) um Close-on-exec wieder zu setzen, kann auch Folgendes verwendet werden:

```
int new_fd = fcntl(fd, F_DUPFD_CLOEXEC, 0);
```



- 5.1 Thread-Pool-Entwurfsmuster
- 5.2 Zusammenspiel von BS-Konzepten
- 5.3 Aufgabe 5: mother

- Stark aufgebohrte Version der `sister`
- Änderungen:
  - Das Connection- und das Request-Modul sollen mithilfe eines Thread-Pools implementiert werden
  - in der `sister` wurden dafür Prozesse verwendet
- Neue Features:
  - Auflistung von Verzeichnisinhalten (alphabetisch sortiert)
  - Ausführen von Perl-Skripten
- Ziel der Aufgabe:
  - Wiederholung etlicher in den SP-Übungen gelernter Konzepte