Universität Bremen

Betriebssysteme

Prozesse und Threads

Dr. Bernhard J. Berger
AG Sichere Systeme

Universität Bremen

Themenübersicht

- Rückblick
  - Was sind Prozesse?
  - Unix-Prozessmodell
  - Prozess Systemcalls
- Prozesskontext
- Prozesszustand

Prozesse

- Klassifikation
- LWPs
  - Thread Kontext
  - LWP Kontext
  - POSIX Threads
- Prozessmodelle
- User Threads
  - Thread Kontext
- Literaturhinweise

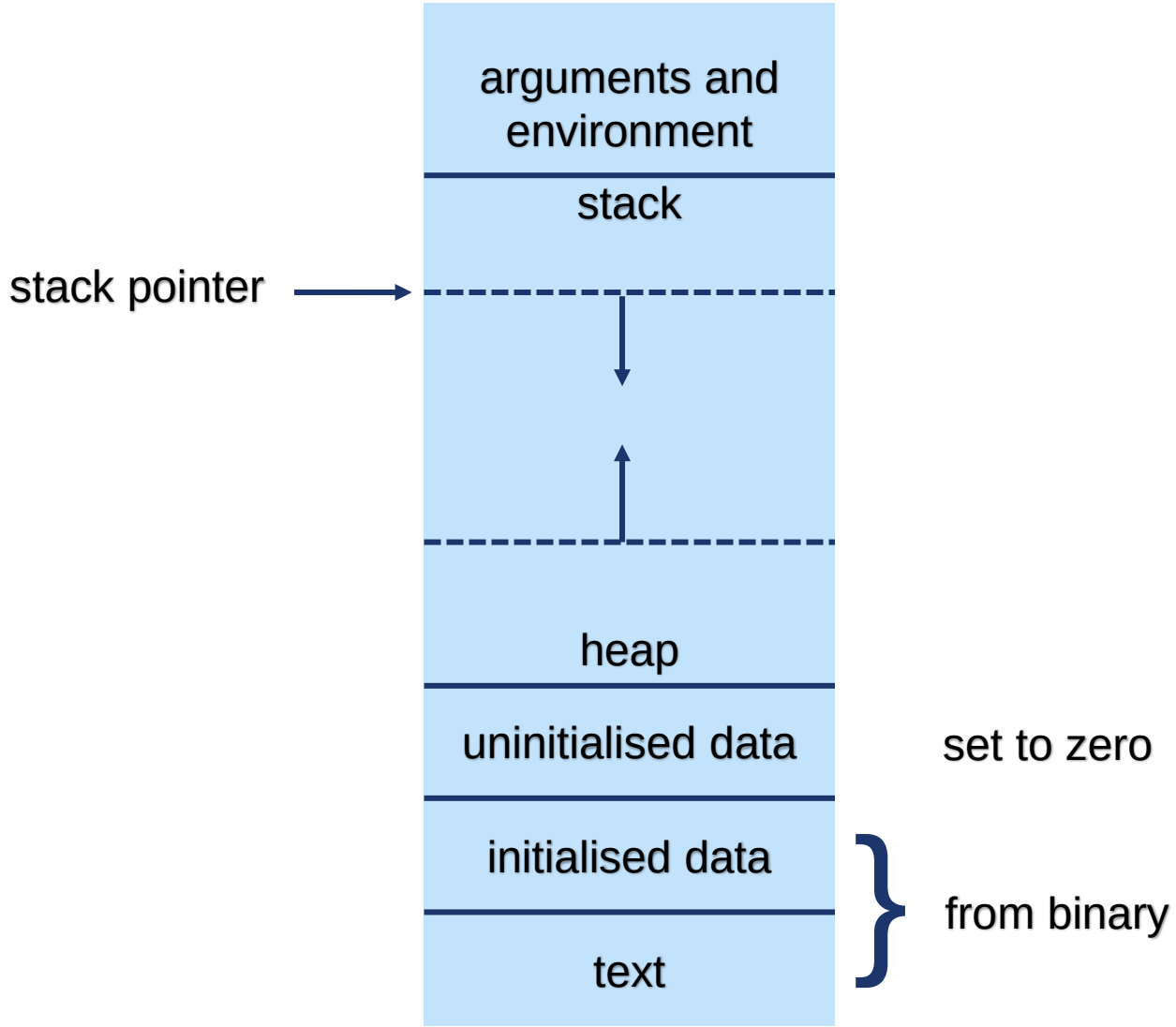
Threads

- Pthread
  - Erzeugen
  - Warten auf Thread
  - Weitere Funktionen
- User Threads
  - Abfragen
  - Erzeugen
  - Setzen
  - Tauschen
- LWP Pattern

APIs

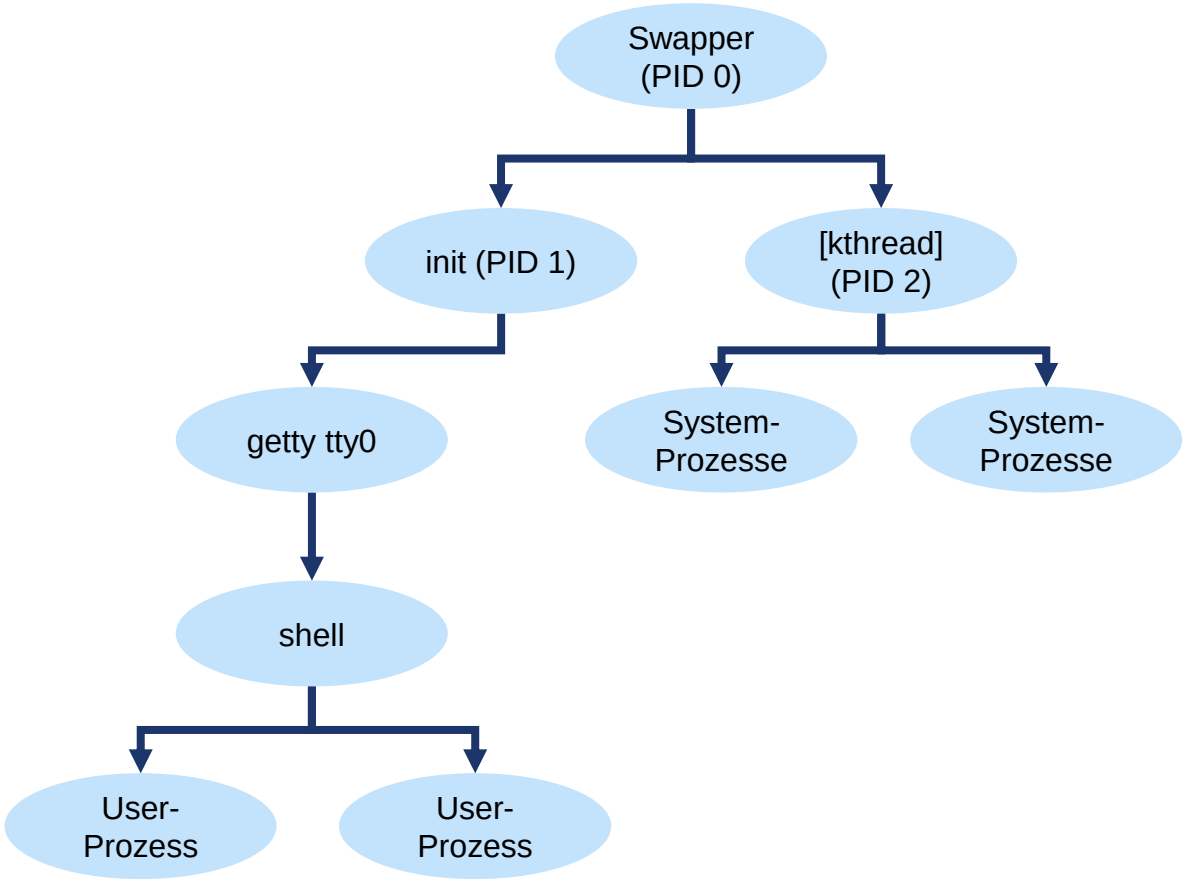
Rückblick – Was sind Prozesse

- Ein Prozess ist ein Programm in Ausführung
- Verwendet Ressourcen
  - CPU
  - Speicher
  - Geräte
- Ressourcenbedarf schwankt
- Hat einen Zustand
  - Wartend (Ready)
  - Laufend (Running)
  - Blockiert (Blocked)
- Betriebssystem verwaltet Ressourcen
  - Erlaubt Nutzung
  - Blockiert Nutzung



Rückblick – Unix Prozessmodell

- Zwei Kategorien
  - Systemprozesse
  - User-Prozesse
- Prozesse können Kindprozesse erzeugen
- Prozesse ergeben eine Hierarchie
  - ps -eaf
  - pstree
- Eineindeutige Prozess-ID (pid)
- Prozess-ID des Elternprozess: ppid



Rückblick – Prozess Systemcalls

- Prozess-IDs
  - `getpid (2)`
  - `getppid (2)`
- Prozesseigentümer
  - `getuid (2)`
- Kindprozess starten
  - `fork (2)`
  - `vfork (2)`
- Prozess beenden
  - `exit (3)`
  - `_exit (2)`
- Prozesssynchronisation
  - `wait (2)`
- Programm ausführen
  - `exec* (2)`

```c
#include <stdio.h>
#include <unistd.h>

int main () {
  printf("Elternprozess: PID %d PPID %d\n", getpid(), getppid());

  const int pid = fork(); /* Neuen Prozess erstellen. */

  if (pid == 0) {
    printf("Kindprozess: PID %d PPID %d\n", getpid(), getppid());
  } else if (pid > 0) {
    printf("Elternprozess: Kind-PID %d\n", pid);
  } else {
    printf("Ein Fehler ist aufgetreten!\n");
  }
}
```

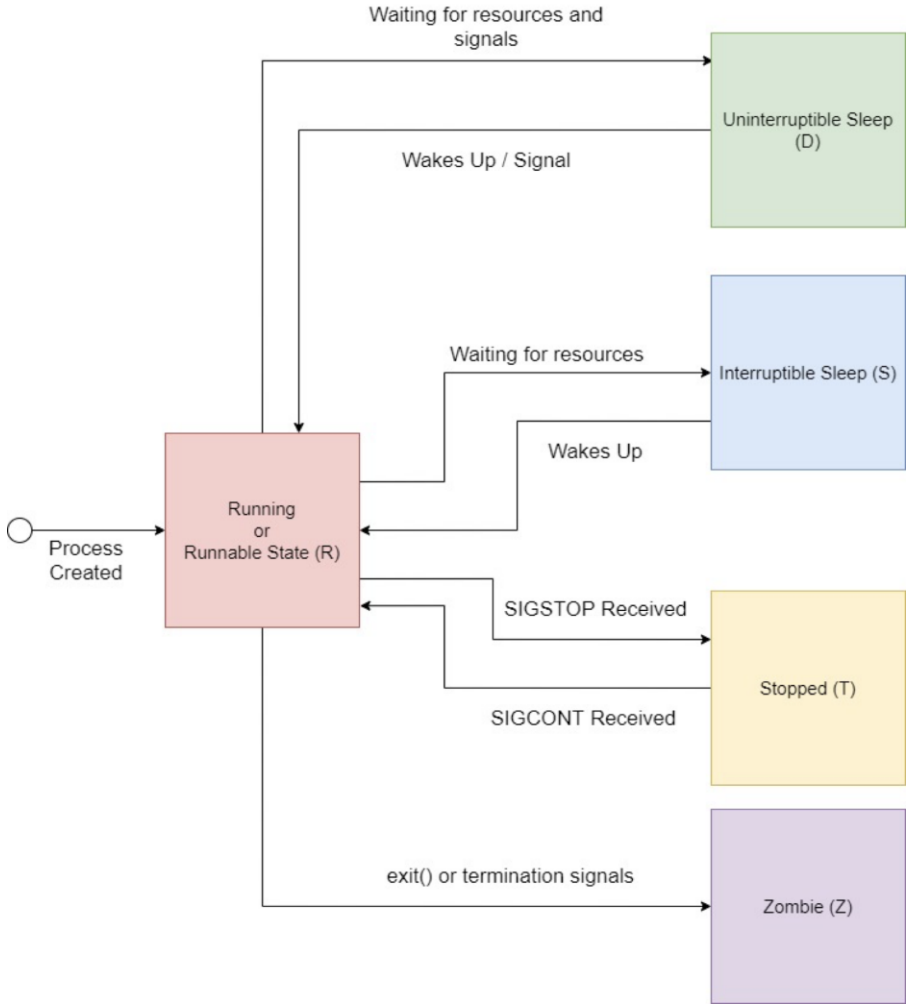
Achtung

Der aufrufende Prozess wird kopiert. Der Aufruf von `fork` kehrt zweimal zurück.

Der Prozesskontext

- **Definition**
  - Der **Prozesskontext** besteht aus allen notwendigen Daten, um den Prozesszustand zu speichern, sodass die Verarbeitung zu einem späteren Zeitpunkt mithilfe dieser Daten wiederhergestellt werden kann.
- **Nutzung des Kontext**
  - Der Prozesskontext wird gespeichert, wenn
    - der Prozess die CPU freigibt ohne sich zu beenden
    - der Prozess vom Scheduler unterbrochen wird
- **Datenstruktur zum Speichern und Wiederherstellen des Prozesskontext**
  - Alle Kontextinformationen können über den Prozesstabelleneintrag (**process table entry**) – auch **process control block** oder task **control block** – erreicht werden
    - In Linux ist jeder Eintrag der Prozesstabelle vom Typ `struct task_struct`
    - In Linux folgt Multi-Core Processing dem Prinzip des **Symmetric Multi Processing (SMP)**
      - Linux verwaltet eine eigene Prozesstabelle für jeden CPU-Kern
      - Für jeden CPU-Kern läuft ein eigener Scheduler
      - Der Zugriff auf die Prozesstabellen muss nur synchronisiert werden, wenn ein Prozess zwischen den Kernen verschoben wird, um ein **Load-Balancing** durchzuführen

Prozesszustände



Quelle: https://www.baeldung.com/linux/process-states

Universität Bremen

Themenübersicht

- Rückblick
  - Was sind Prozesse?
  - Unix-Prozessmodell
  - Prozess Systemcalls
- Prozesskontext
- Prozesszustand

Prozesse

- Klassifikation
- LWPs
  - Thread Kontext
  - LWP Kontext
  - POSIX Threads
- Prozessmodelle
- User Threads
  - Thread Kontext
- Literaturhinweise

Threads

- Pthread
  - Erzeugen
  - Warten auf Thread
  - Weitere Funktionen
- User Threads
  - Abfragen
  - Erzeugen
  - Setzen
  - Tauschen
- LWP Pattern

APIs

Klassifikation

- **Light-Weight Processes (LWPs)**
  - Verwaltet vom Betriebssystem - “Nur ein Prozess”
  - Kann alles tun, was auch ein Prozess kann – core reservation
  - Teilen sich einen Kontext mit dem Prozess, der sie erzeugt hat
  - LWPs werden auch **kernel-level threads** genannt
- **User Threads**
  - Verwaltet durch eine Thread-Bibliothek in einer Applikation
  - Das Betriebssystem hat keine Kenntnis
  - Keinen Zugriff auf Ressourcen, die durch den Kernel verwaltet werden
- Wir betrachten LWPs

Randnotiz: In einiger Literatur wird der Begriff Kernel-Threads für Threads verwendet, die durch vom Kernel verwendet werden. Betriebssysteme wie Solaris nutzen Multi-Threaded Kernel.

LWP – LWP Kontext

- Spezialregister
  - Stackzeiger
  - Programmzähler
  - Instruktionszeiger
  - Programstatus
- LWP Zustand (runnable, sleeping, signal mask, …)
- CPU auf der der LWP läuft (wird auf exakt einem Kern ausgeführt)
- Scheduling-Informationen (Policy, Priorität, nice Werte, verwendete CPU-Werte)
- Stack

Randnotiz: In Linux ist der task control block eines LWP ein `struct task_struct entry` und wird mit den „realen“ Prozessen in der Prozesstabelle verwaltet

LWP – Thread Kontext

- LWPs laufen im Kontext eines Prozess
- Threads sind für das Betriebssystem sichbar und werden separate auf CPUs verteilt
- Wenn ein LWP in den Status sleep wechselt aber der Prozess noch ausgeführt wird, dann wird nur der LWP-spezifische Kontext gespeichert
- LWPs haben einen eigenen
  - Stack
  - Programmzähler
  - Scheduling-Informationen

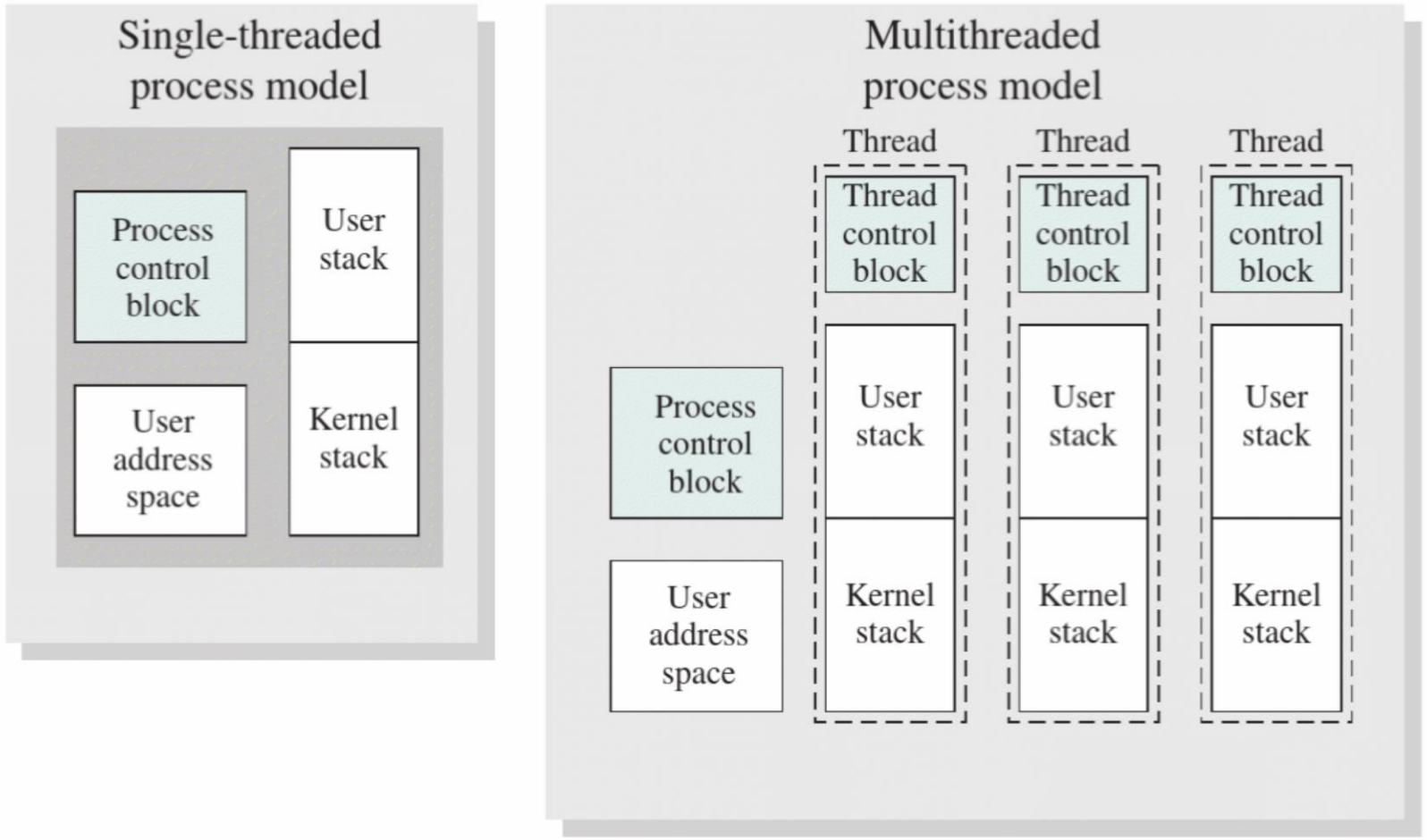
LWP – POSIX Threads

- Portable library for managing LWPs
  - Available on Unix-based systems
  - Windows
  - MacOS (Darwin)
- Part of the C library

Thread-Aufruf	Beschreibung
pthread_create	Erzeugt einen neuen Thread
pthread_exit	Beendet den aufrufenden Thread
pthread_join	Wartet auf die Beendigung eines bestimmten Threads
pthread_yield	Gibt die CPU frei, damit andere Threads laufen können
pthread_attr_init	Erzeugt und initialisiert eine Attributstruktur
pthread_attr_destroy	Löscht die Attributstruktur eines Threads

Quelle: Tanenbaum, Abbildung 2.4

Prozessmodelle



Quelle: Stallings, Abbildung 4.2

Universität Bremen

Themenübersicht

- Rückblick
  - Was sind Prozesse?
  - Unix-Prozessmodell
  - Prozess Systemcalls
- Prozesskontext
- Prozesszustand

Prozesse

- Klassifikation
- LWPs
  - Thread Kontext
  - LWP Kontext
  - POSIX Threads
- Prozessmodelle
- User Threads
  - Thread Kontext
- Literaturhinweise

Threads

- Pthread
  - Erzeugen
  - Warten auf Thread
  - Weitere Funktionen
- User Threads
  - Abfragen
  - Erzeugen
  - Setzen
  - Tauschen
- LWP Pattern

APIs

Pthread – Create an LWP

```c
#include <pthread.h>

int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg);
```

Pthread – Create an LWP

```
#include <pthread.h>
```

Zeiger auf Variable zur Identifikation des Threads, wird durch `pthread_create()` gesetzt

```
int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg);
```

Pthread – Create an LWP

```
#include <pthread.h>
```

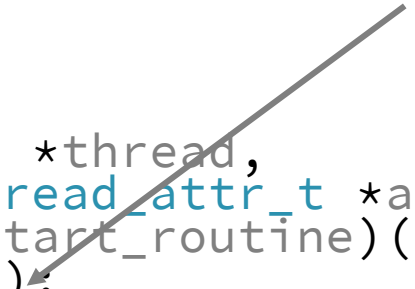
Zeiger auf Thread-Attributspezifikation, z.B. Scheduling Policy. Dieser Wert kann NULL sein

```
int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg);
```

Pthread – Create an LWP

```
#include <pthread.h>
```

Zeiger auf eine Datenstruktur, die als Parameter für den LWP genutzt werden soll.



```
int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg);
```

Pthread – Create an LWP

```
#include <pthread.h>


int
pthread_join(pthread_t thread,
             void **value_ptr);
```

Identifikationsvariable des Threads (von `pthread_create()`)

Zeiger auf die Rückgabe des LWP.

Pthread – Weitere Funktionen

- Pthread-Funktionen für
  - LWPs beenden
  - Attribut-Management
  - Synchronisation zwischen LWPs
  - Verwaltung von Thread-spezifischen Key-Value-Paaren
- Für mehr Informationen: `man 3 pthread`

User Context – Erinnerung

- Spezialregister
  - Stackzeiger
  - Programmzähler
  - Instruktionszeiger
  - Programstatus
- LWP Zustand (runnable, sleeping, signal mask, …)
- CPU auf der der LWP läuft (wird auf exakt einem Kern ausgeführt)
- Scheduling-Informationen (Policy, Priorität, nice Werte, verwendete CPU-Werte)
- Stack

User Context – Abfragen

- Speichert den aktuellen Kontext in der Datenstruktur in `ucp`
- **Hinweise:** Dies kann in der `main()`-Function geschehen, was nützlich ist, um neue LWP-Kontexte mit `makecontext()` zu erstellen

```
#include <ucontext.h>

int
getcontext(ucontext_t *ucp);
```

User Context – Erzeugen

- Modifiziert den Kontext, der durch `ucp` referenziert wird
- Kontext führt `func()` aus
- **Hinweise:** Vor dem Aufruf von `makecontext()` müssen folgende Daten initialisiert sein:
  - Stack Pointer
  - Stackgröße
  - Kontext der `return`-Funktion

```
#include <ucontext.h>

void
makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
```

User Context – Setzen

- Stellt gespeicherten Kontext wieder her

```
#include <ucontext.h>

int
setcontext(ucontext_t *ucp);
```

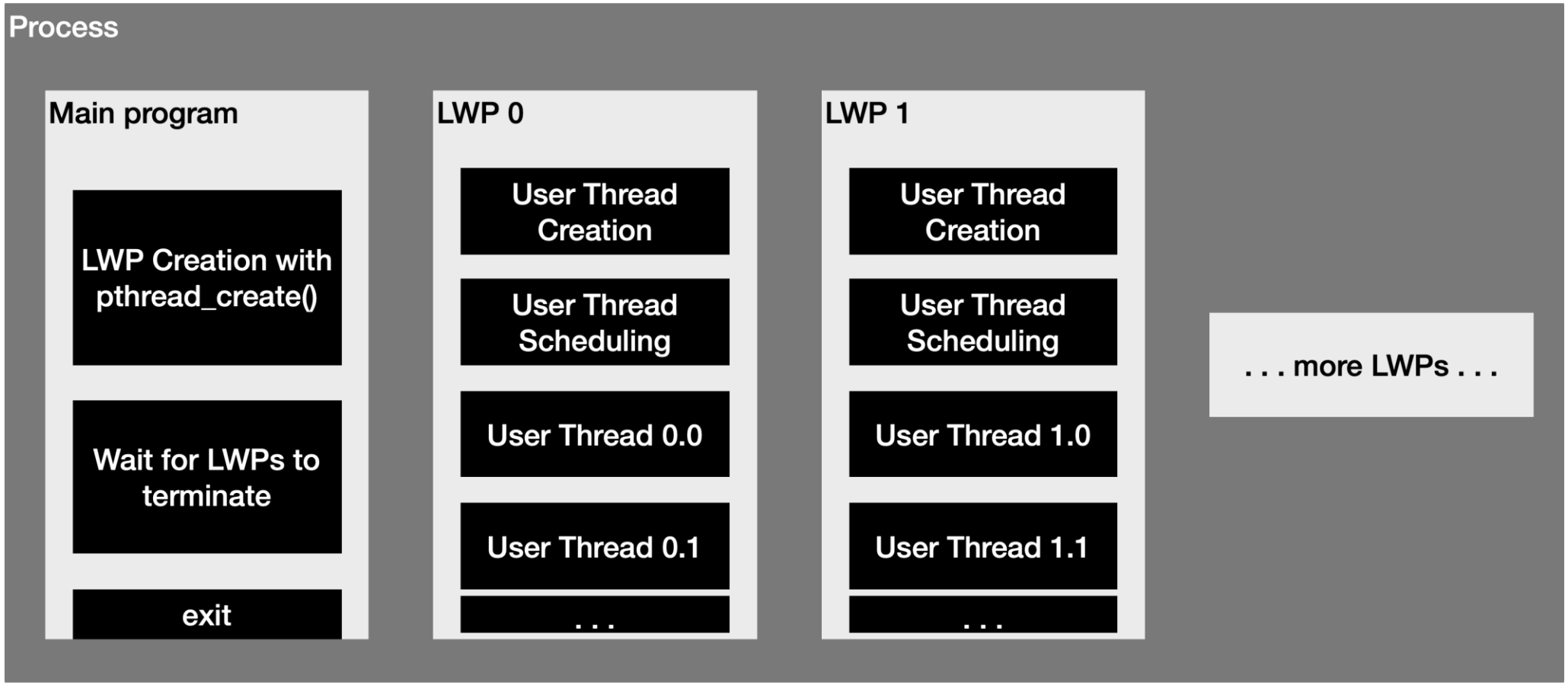
User Context – Austauschen

- Speichert den aktuellen Kontext in der Datenstruktur in `oucp`
- Macht den Kontext, der durch `oucp` referenziert wird zum aktiven Kontext

```c
#include <ucontext.h>

int
swapcontext(ucontext_t *oucp, ucontext_t *ucp);
```

LWP Pattern


Process
Main program
LWP Creation with pthread_create()
Wait for LWPs to terminate
exit
LWP 0
User Thread Creation
User Thread Scheduling
User Thread 0.0
User Thread 0.1
. . .
LWP 1
User Thread Creation
User Thread Scheduling
User Thread 1.0
User Thread 1.1
. . .
. . . more LWPs . . .

Universität Bremen

Themenübersicht

- Rückblick
  - Was sind Prozesse?
  - Unix-Prozessmodell
  - Prozess Systemcalls
- Prozesskontext
- Prozesszustand

Prozesse

- Klassifikation
- LWPs
  - Thread Kontext
  - LWP Kontext
  - POSIX Threads
- Prozessmodelle
- User Threads
  - Thread Kontext
- Literaturhinweise

Threads

- Pthread
  - Erzeugen
  - Warten auf Thread
  - Weitere Funktionen
- User Threads
  - Abfragen
  - Erzeugen
  - Setzen
  - Tauschen
- LWP Pattern

APIs