Work in progress

Multithreading

Ute Bormann, TI2

2023-10-13

Inhalt

1. Überblick Multithreading
2. Die Unix-Multithreading-Umgebung

Teil 1:
Überblick Multithreading

Wdh. Nebenläufigkeit

- Nichtdeterministische Abläufe
  - zwischen Prozessen im User-Mode (Shared Memory, gemeinsame Dateien,...)
  - im Betriebssystemkern (Prozesse vs. Interrupt-Handler)
- Nebenläufigkeit: Parallelität oder Quasi-Parallelität
- Gegenseitige Beeinflussung möglich/üblich durch
  - Konkurrenz von Prozessen (ggf. Interrupt-Handlern)
  - Kooperation von Prozessen (ggf. Interrupt-Handlern)
- Synchronisationsmaßnahmen erforderlich zur:
  - mehrseitigen Synchronisation
  - einseitigen Synchronisation
  - (u.U. Ressourcenverwaltung)

→
- Bisher konkurrierende/kooperierende sequentielle Prozesse
- Darüber hinaus: Nichtsequentielle Anwendungsprogrammierung

Nichtsequentielle Anwendungsprogrammierung

- Anwendung kann aus mehreren (miteinander verzahnten) Teilaufgaben bestehen

  ⇒ können (teilweise/weitgehend) unabhängig voneinander sein

  ⇒ keine feste Abarbeitungsreihenfolge nötig

Nichtsequentielle Anwendungsprogrammierung

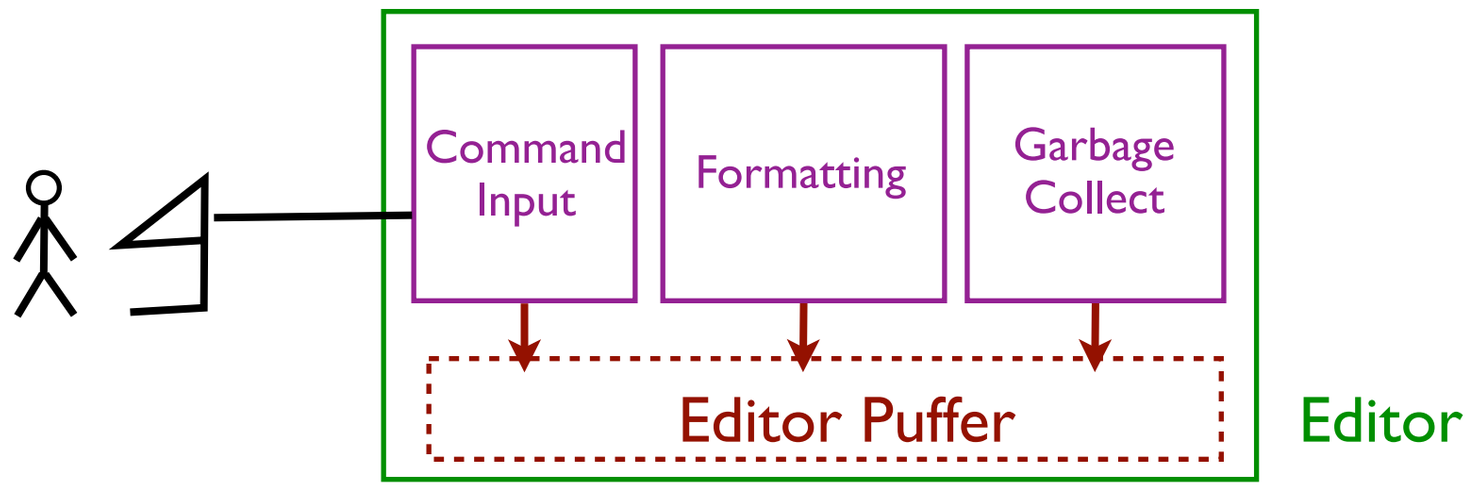
- Anwendung kann aus mehreren (miteinander verzahnten) Teilaufgaben bestehen

  ⇒ können (teilweise/weitgehend) unabhängig voneinander sein

  ⇒ keine feste Abarbeitungsreihenfolge nötig

- Beispiel: Realisierung eines komplexen Editors

⇒ verschiedene Benutzereingaben und damit zusammenhängende Berechnung



⇒ Alle Aktivitäten arbeiten (z.T.) auf gemeinsamen Daten

- Realisierung als ein einzelner (herkömmlicher) Prozess?

  ⇒ keine Nebenläufigkeit möglich

- Realisierung als mehrere Prozesse?

  ⇒ starke Abschottung (verschiedene Adressräume, Interprozesskommunikation nötig...)

- Realisierung als ein einzelner (herkömmlicher) Prozess?

  ⇒ keine Nebenläufigkeit möglich

- Realisierung als mehrere Prozesse?

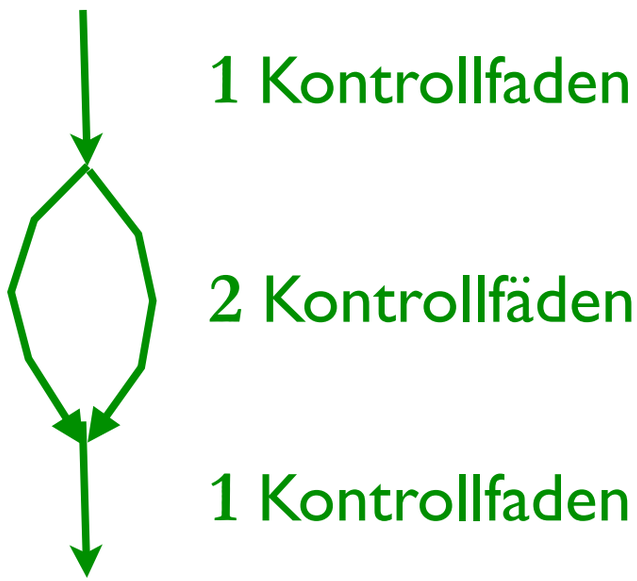
  ⇒ starke Abschottung (verschiedene Adressräume, Interprozesskommunikation nötig...)

- Stattdessen: Mehrere Threads in einem Prozess (Multithreading)

  ⇒ Mehrere Kontrollfäden in gemeinsamer Hülle

Multithreading

- Jeder Thread hat eigenen Ausführungszustand (PC, Stack, ...)
- Gemeinsame „Umgebung“ innerhalb des Prozesses (Adressraum, offene Dateien, ...)
- Beispiel:



- Nichtdeterministische Abarbeitung
- Dennoch in vielen Fällen weiterhin I/O-Determinismus gefordert

  ⇒ Synchronisationsmaßnahmen erforderlich

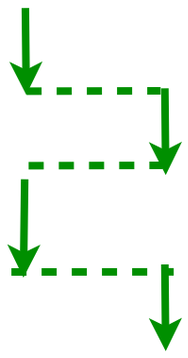
- Multithreading gewinnt zunehmend praktische Bedeutung in Anwendungsprogrammen

Sichtbarkeit von Threads

a) Nur im Anwendungsprozess sichtbar (reine „User Threads“)

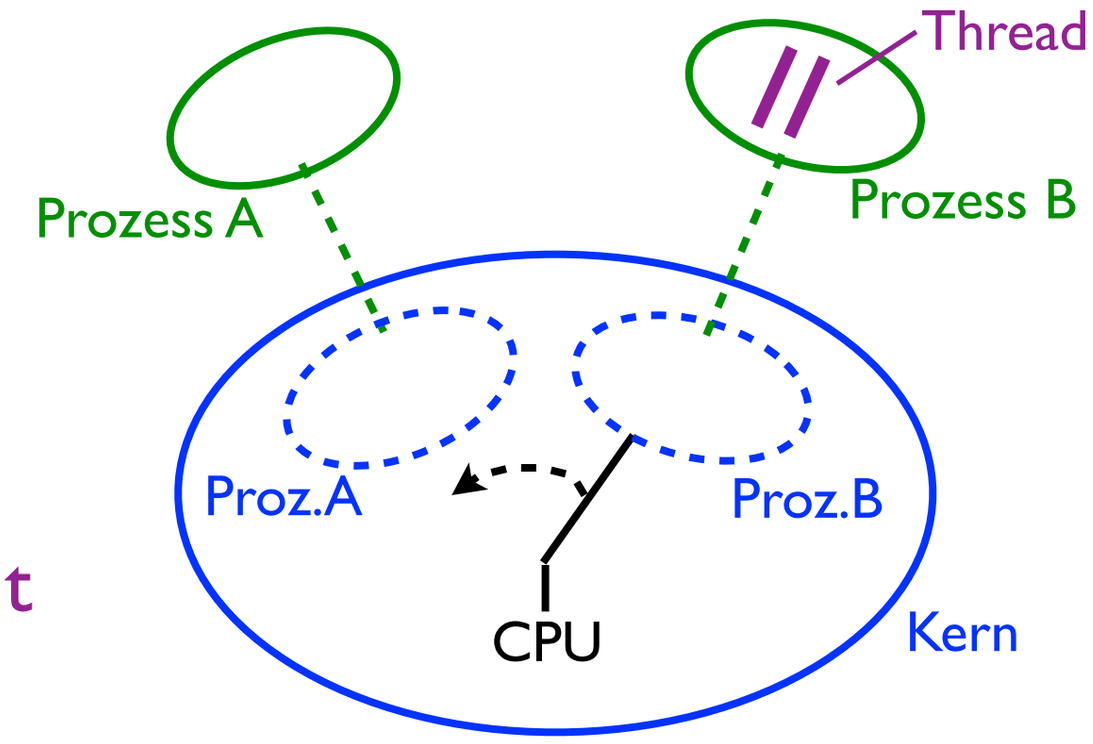
   ⇒ Betriebssystemkern erfährt nichts davon

- Nebenläufigkeit erfordert prozessinternes Scheduling und Dispatching



⇒ aufwendig und selten sinnvoll

oder keine „echte“ Nebenläufigkeit



b) Auch im Betriebssystem sichtbar

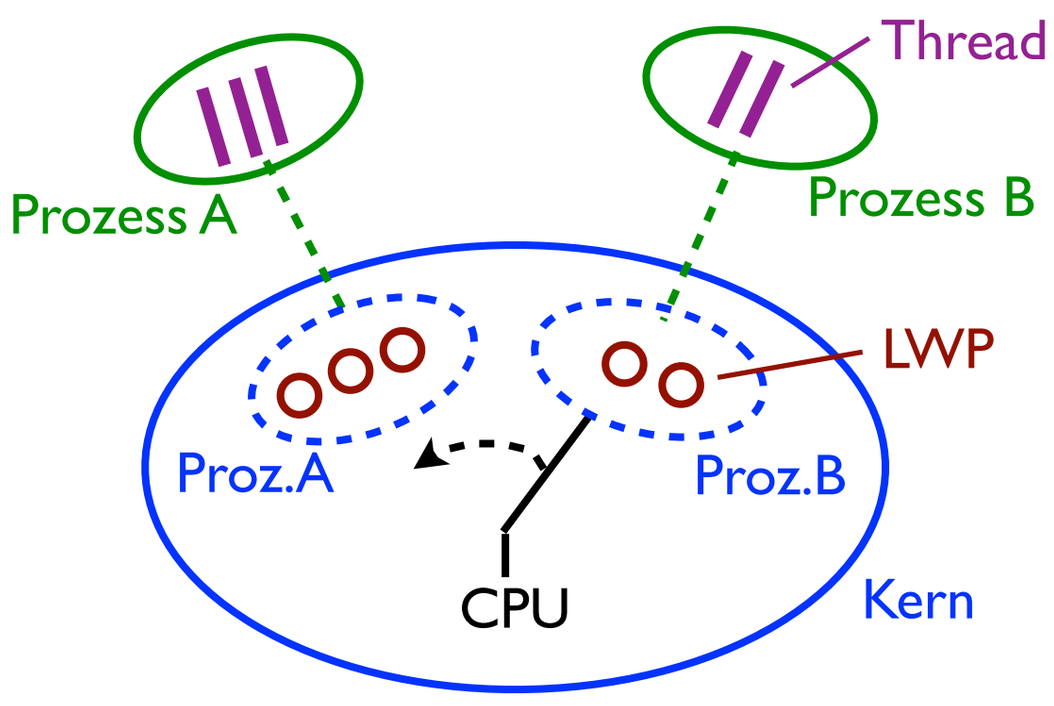
- Threads sind Grundlage des BS-internen Schedulings

  ⇒ Prozess wird zur passiven Hülle (Adressraum)

- Bei Mehrprozessorsystem: Verteilung auf CPUs

⇒ Light-Weight-Prozesse (LWPs) / Kernel Threads

- Ein (oder mehrere) User-Level-Threads werden einem LWP zugeordnet
- LWPs sind Grundlage des Schedulings



⇒ nur in dieser Kombination in unseren Übungen interessant

- Kernel Threads auch für Betriebssystem-interne Aufgaben nutzbar

- Aufteilung des Kontrollflusses in mehrere Fäden muss im Programm notiert werden, z.B. spezielle Programmiersprachennotation:

```
h:=x;  conc v:=h || w:=h  end conc …
```

- Aufteilung des Kontrollflusses in mehrere Fäden muss im Programm notiert werden, z.B. spezielle Programmiersprachennotation:

```
h:=x;  conc v:=h || w:=h  end conc …
```

- Java-Threads (s. PI1)

Threads in Java (Beispiel)

```
class MyClass implements Runnable {
  public MyClass(int val1, float val2) {
    .. Konstruktor ..
  }
  public void run() {
    .. do something
  }
}

public static void main(String[] args) {
  MyClass obj = new MyClass(12, 25.35);
  Thread thr = new Thread(obj);
  thr.start();
}
```

- Aufteilung des Kontrollflusses in mehrere Fäden muss im Programm notiert werden, z.B. spezielle Programmiersprachennotation:

  h:=x; conc v:=h || w:=h end conc …

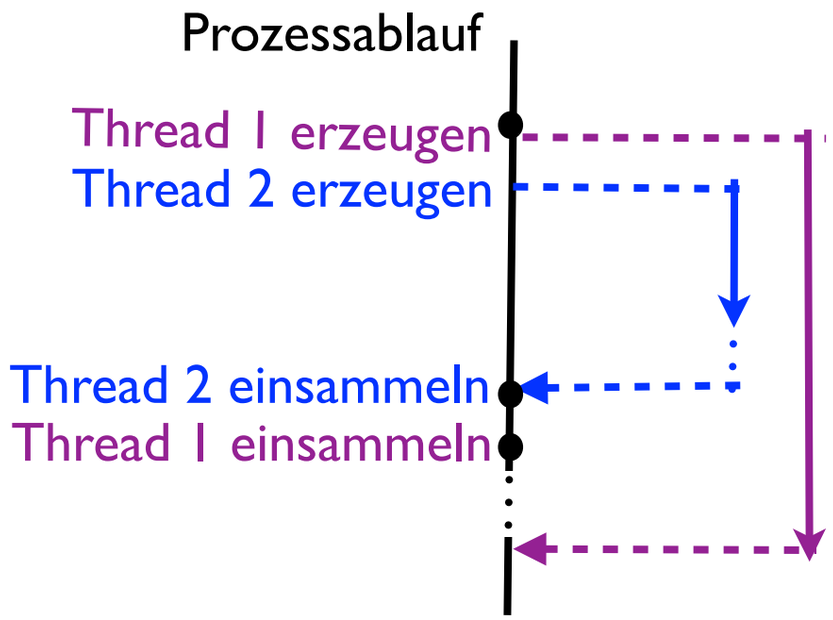
- Java-Threads (s. PI1)
- Lange Jahre keine Threads in C++
- Seit Mitte 90er Jahre konkrete Multithreading-Umgebung in SunOS5.4 verfügbar
- Standardisierte Schnittstelle dafür: Pthreads
  Allerdings:
  - Notation nicht optimal

Fragen – Teil 1

- Was ist ein *Thread*? Skizziere ein sinnvolles Anwendungsbeispiel für die Verwendung mehrerer Threads innerhalb eines Prozesses.
- Grenze den Thread-Begriff gegen den Unix-Prozess-Begriff ab (Adressraum, Zustandsinformationen, etc.).

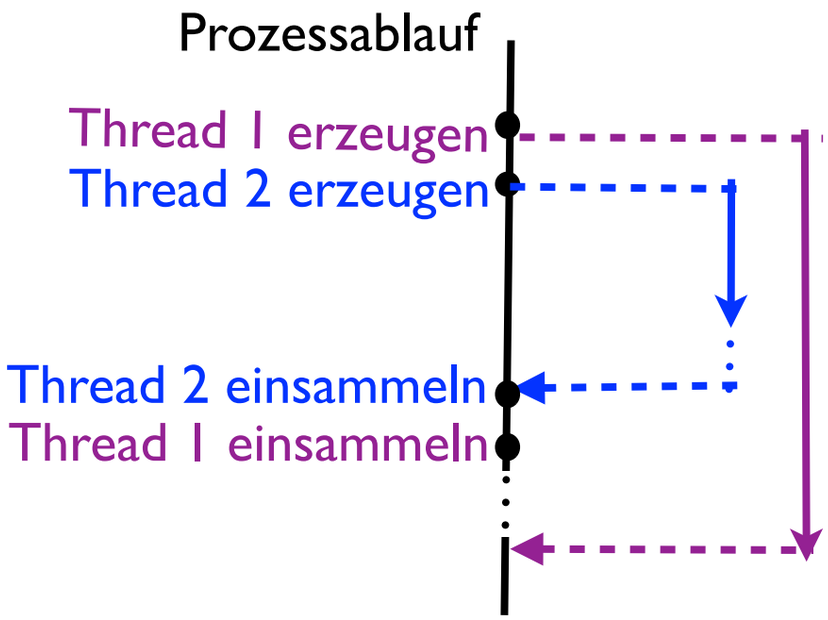
Teil 2:
Die Unix-Multithreading-Umgebung

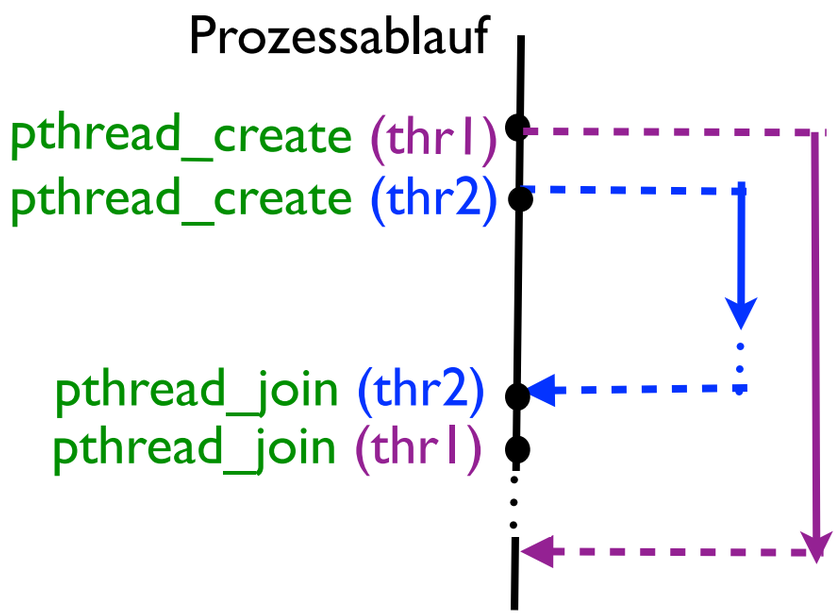
Die Unix-Multithreading-Umgebung



- Bei Erzeugung von Thread „Arbeit zuweisen“ (Prozedur mit Parametern):
- Thread arbeitet diese ab (oder vorzeitiges Ende:)
- Muss nach Beendigung von Prozess „wiedereingesammelt“ werden:
- Achtung: Nichteingesammelte Threads werden bei Prozessende vernichtet (evtl. noch nicht fertig)

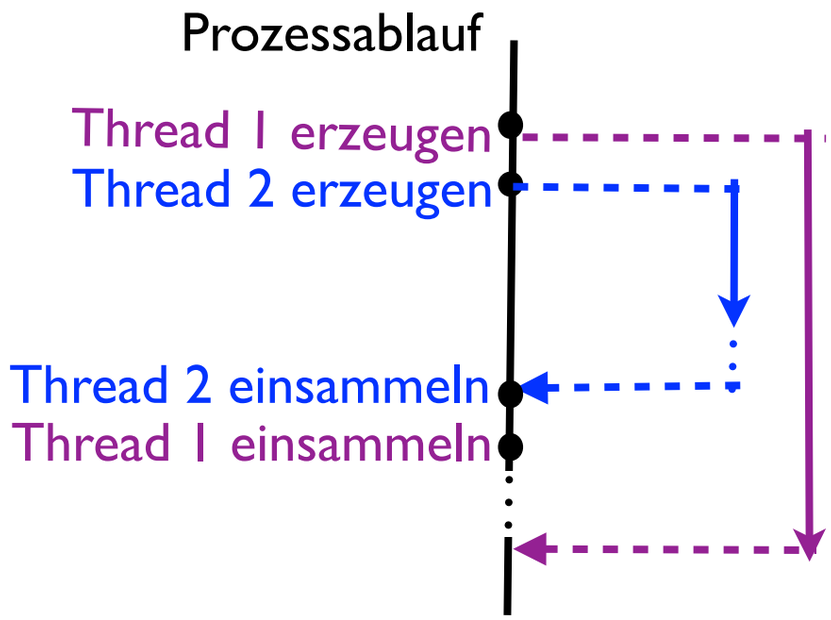
Die Unix-Multithreading-Umgebung

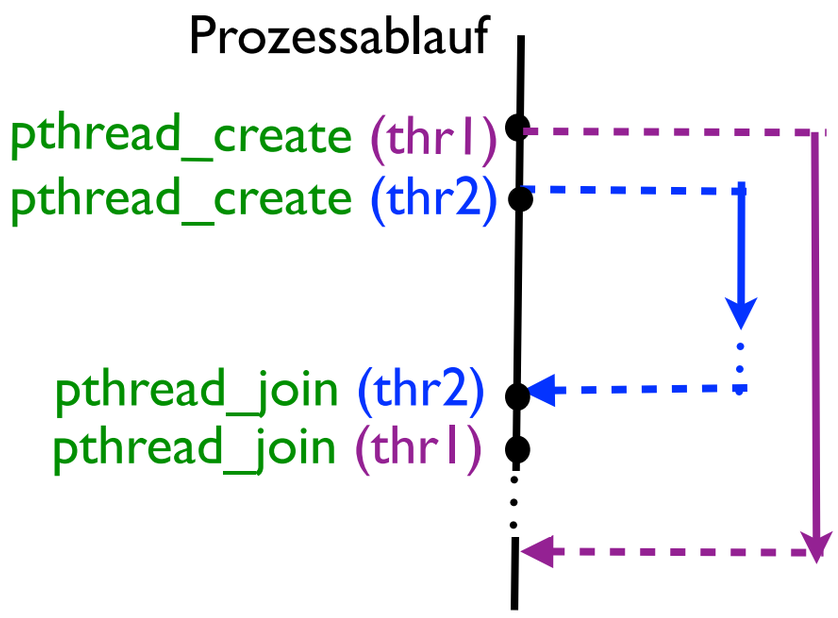




- Bei Erzeugung von Thread „Arbeit zuweisen“ (Prozedur mit Parametern): pthread_create()
- Thread arbeitet diese ab (oder vorzeitiges Ende: pthread_exit())
- Muss nach Beendigung von Prozess „wiedereingesammelt“ werden: pthread_join()
- Achtung: Nichteingesammelte Threads werden bei Prozessende vernichtet (evtl. noch nicht fertig)

Die Unix-Multithreading-Umgebung





- Bei Erzeugung von Thread „Arbeit zuweisen“ (Prozedur mit Parametern): pthread_create()
- Thread arbeitet diese ab (oder vorzeitiges Ende: pthread_exit())
- Muss nach Beendigung von Prozess „wiedereingesammelt“ werden: pthread_join()
- Achtung: Nichteingesammelte Threads werden bei Prozessende vernichtet (evtl. noch nicht fertig)
- Modell erinnert stark an Erzeugung von Kindprozessen in Unix (fork(), exit(), wait())
- Aber:
  - fork() erzeugt Kopie des Adressraums
  - Kindprozesse können nach Beendigung des Vaters u.U. weiterlaufen

- Deklaration:
  pthread_t Thread_Name;

- Threads erzeugen:
  pthread_create(&Thread_Name, Attribute, Prozedur, Parameter);
  (bei uns: Attribute=NULL ⇒ Default)

  Beispiel: pthread_create(&t1, NULL, &drucken, "Hello World");

- Threads „einsammeln“:
  pthread_join(Thread_Name, Variable für Rückgabewert);
  (bei uns: Variable=NULL ⇒ ignorieren)

  Beispiel: pthread_join(t1, NULL);

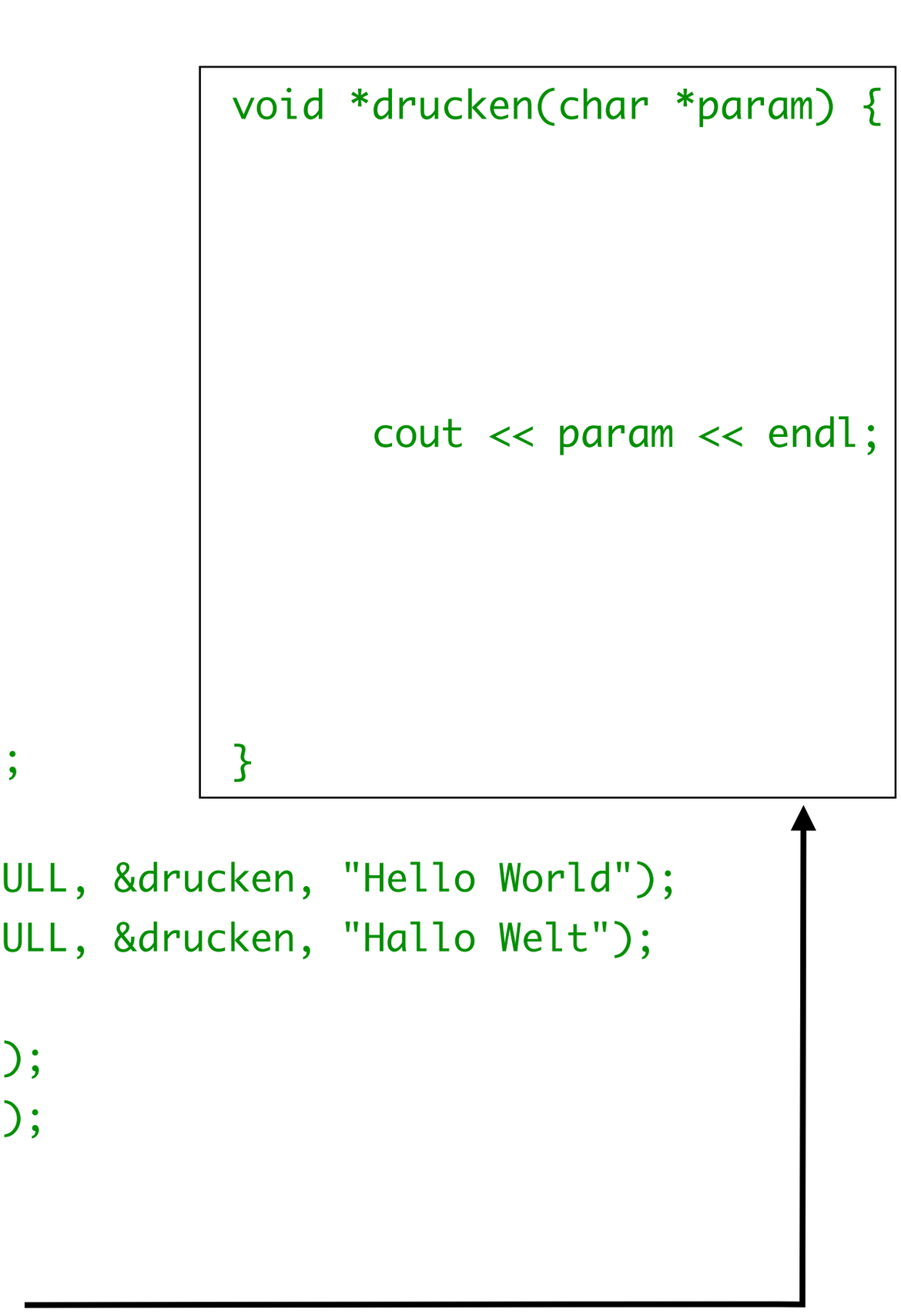
Beispielprogramm:

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

void *drucken(char*);




main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}

void *drucken(char *param) {
  cout << param << endl;
}
```

Beispielprogramm:

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

void *drucken(char*);

main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {

        cout << param << endl;

}
```



Beispielprogramm:

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

void *drucken(char*);

int said = 0;


main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {
  for (;;) {

    if (said < 10) {
      said++;

      cout << param << endl;
    } else {

      break;
    }
  }
}
```

Beispielprogramm:

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

void *drucken(char*);

int said = 0;


main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {
  for (;;) {

    if (said < 10) {
    said++;

      cout << param << endl;
    } else {

      break;
    }
  }
}
```

- Deklaration:
  pthread_t Thread_Name;

- Threads erzeugen:
  pthread_create(&Thread_Name, Attribute, Prozedur, Parameter);
  (bei uns: Attribute=NULL ⇒ Default)

  Beispiel: pthread_create(&t1, NULL, &drucken, "Hello World");

- Threads „einsammeln“:
  pthread_join(Thread_Name, Variable für Rückgabewert);
  (bei uns: Variable=NULL ⇒ ignorieren)

  Beispiel: pthread_join(t1, NULL);

→ - Schutz von kritischen Abschnitten:
  pthread_mutex_t Mutex_Variable; (Deklaration einer Mutex-Variablen)
  pthread_mutex_lock(Mutex_Variable);
  pthread_mutex_unlock(Mutex_Variable);

Beispielprogramm:

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

void *drucken(char*);

int said = 0;
pthread_mutex_t *said_mutex;

main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {
  for (;;) {
    pthread_mutex_lock(said_mutex);
    if (said < 10) {
      said++;
      pthread_mutex_unlock(said_mutex);
      cout << param << endl;
    } else {

      break;
    }
  }
}
```

Beispielprogramm:

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

void *drucken(char*);

int said = 0;
pthread_mutex_t *said_mutex;

main() {
  pthread_t english, deutsch;

  pthread_create(&english, NULL, &drucken, "Hello World");
  pthread_create(&deutsch, NULL, &drucken, "Hallo Welt");
  ...
  pthread_join(english, NULL);
  pthread_join(deutsch, NULL);
}
```

```
void *drucken(char *param) {
  for (;;) {
    pthread_mutex_lock(said_mutex);
    if (said < 10) {
      said++;
      pthread_mutex_unlock(said_mutex);
      cout << param << endl;
    } else {
      pthread_mutex_unlock(said_mutex);
      break;
    }
  }
}
```

- Aufteilung des Kontrollflusses in mehrere Fäden muss im Programm notiert werden, z.B. spezielle Programmiersprachennotation:

  h:=x; conc v:=h || w:=h end conc …

- Java-Threads (s. PI1)
- Lange Jahre keine Threads in C++
- Seit Mitte 90er Jahre konkrete Multithreading-Umgebung in SunOS5.4 verfügbar
- Standardisierte Schnittstelle dafür: Pthreads
  Allerdings:
  - Notation nicht optimal
  - Konzept ist flexibler, als wir es benötigen

- Aufteilung des Kontrollflusses in mehrere Fäden muss im Programm notiert werden, z.B. spezielle Programmiersprachennotation:

  h:=x; conc v:=h || w:=h end conc …

- Java-Threads (s. PI1)
- Lange Jahre keine Threads in C++
- Seit Mitte 90er Jahre konkrete Multithreading-Umgebung in SunOS5.4 verfügbar

- Standardisierte Schnittstelle dafür: Pthreads
  Allerdings:
  - Notation nicht optimal
  - Konzept ist flexibler, als wir es benötigen

    ⇒ C++-Hülle dafür geschrieben…

    lock() und unlock() als Memberfunktionen der Klasse Mutex

    ⇒ Nutzung: obj.lock() bzw. obj.unlock()

- Aufteilung des Kontrollflusses in mehrere Fäden muss im Programm notiert werden, z.B. spezielle Programmiersprachennotation:

  h:=x; conc v:=h || w:=h end conc …

- Java-Threads (s. PI1)
- Lange Jahre keine Threads in C++
- Seit Mitte 90er Jahre konkrete Multithreading-Umgebung in SunOS5.4 verfügbar
- Standardisierte Schnittstelle dafür: Pthreads
  Allerdings:
  - Notation nicht optimal
  - Konzept ist flexibler, als wir es benötigen
    ⇒ C++-Hülle dafür geschrieben…

→ - C++11 hat eigenes Thread-Konzept
  (in etwa vergleichbar mit Java-Threads)

Threads in C++11

```
#include <thread>

void run(optional args...) {
    /* do something */
}

int main() {

  std::thread t(run, args...);

  /* ... */

  t.join();
  return 0;
}
```

Übersetzen mit

```
c++ -std=c++11 -pthread ...
```

Kleine Aufgabe

- Gegeben sei der folgende Programmauszug. Hat er Nebenläufigkeitsprobleme (Einprozessorsystem)? Falls ja: welche? Falls nein: warum nicht?

```
...
int fd;
int main() {
  fd = open (“/bla”, ...);
  int c1, c2;
  pthread_t t1, t2;
  pthread_create(&t1, 0, func, &c1);
  pthread_create(&t2, 0, func, &c2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
 cout << (c1 + c2) << endl;
...
```

```
void *func(int *cp) {
  int count = 0;
  char buf[1];
  for (;;) {
    int i = read (fd, buf, 1);
    if (i > 0) {
       ... Verarbeiten des Zeichens
       count++;
    } else break;
  };
  *cp = count;
  return 0;
}
```

Fragen – Teil 2

- Die Routinen `pthread_create()`, `pthread_join()`, `pthread_exit()` realisieren die Erzeugung und Termination von Threads in der Unix-Multithreading-Umgebung. Vergleiche ihre Funktionalität mit den Systemaufrufen zur Erzeugung und Termination von Prozessen (`wait()`, `fork()` und `exit()`). Warum arbeitet `pthread_create()` deutlich anders als `fork()`?

Zusammenfassung

- Nicht-sequentielle Anwendungsprogrammierung
- Multithreading
- User Threads vs. Kernel Threads
- Threads in Java
- Threads in Unix: Pthreads
- Threads in C++11

Multithreading – Fragen

1. Was ist ein *Thread*? Skizziere ein sinnvolles Anwendungsbeispiel für die Verwendung mehrerer Threads innerhalb eines Prozesses.

2. Grenze den Thread-Begriff gegen den Unix-Prozess-Begriff ab (Adressraum, Zustandsinformationen, etc.).

3. Die Routinen `pthread_create()`, `pthread_join()`, `pthread_exit()` realisieren die Erzeugung und Termination von Threads in der Unix-Multithreading-Umgebung. Vergleiche ihre Funktionalität mit den Systemaufrufen zur Erzeugung und Termination von Prozessen (`wait()`, `fork()` und `exit()`). Warum arbeitet `pthread_create()` deutlich anders als `fork()`?