

Übungen zu Systemprogrammierung 1 (SP1)

Ü5 – Threads und Koordinierung

Christian Eichler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2017 – 11. bis 15. Dezember 2017

http://www4.cs.fau.de/Lehre/WS17/V_SP1



Agenda

- 6.1 Threads
- 6.2 Schnittstelle
- 6.3 Koordinierung
- 6.4 Aufgabe 5: patric
- 6.5 Gelerntes anwenden



Agenda

- 6.1 Threads
- 6.2 Schnittstelle
- 6.3 Koordinierung
- 6.4 Aufgabe 5: patric
- 6.5 Gelerntes anwenden



Motivation von Threads

- UNIX-Prozesskonzept (Ausführungsumgebung mit einem Aktivitätsträger) für viele heutige Anwendungen unzureichend
 - keine parallelen Abläufe innerhalb eines logischen Adressraums auf Multiprozessorsystemen
 - typische UNIX-Server-Implementierungen benutzen die `fork`-Operation, um einen Server-Prozess für jeden Client zu erzeugen
 - Verbrauch unnötig vieler System-Ressourcen
 - zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adressraums nützlich
- Lösung: bei Bedarf weitere Aktivitätsträger in einem UNIX-Prozess erzeugen

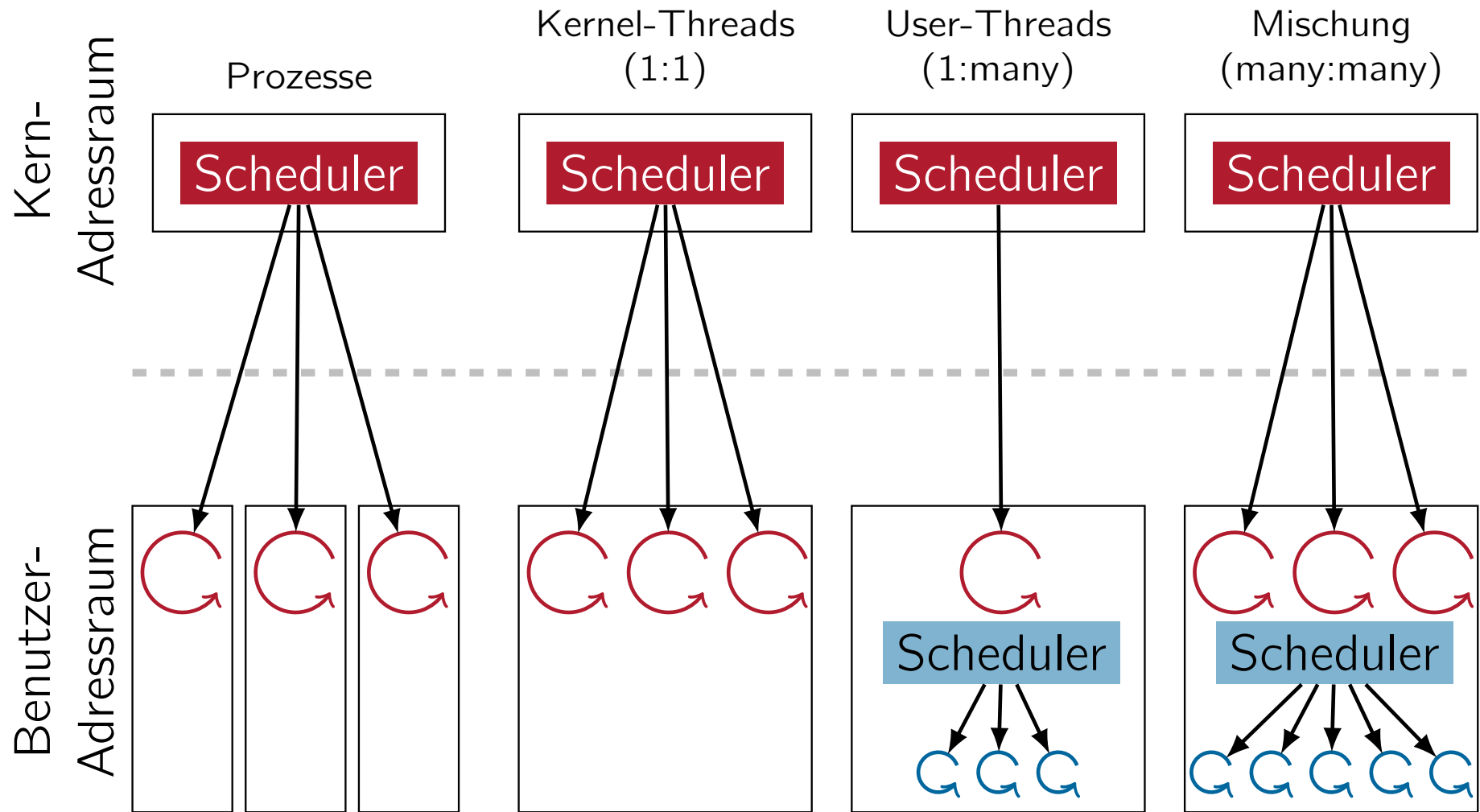


- User-Threads (“federgewichtig”)
 - Realisierung auf Anwendungsebene
 - Systemkern sieht nur **einen** Kontrollfluss
 - + Erzeugung von Threads und Umschaltung extrem billig
 - Systemkern hat kein Wissen über diese Threads
 - in Multiprozessorsystemen keine parallelen Abläufe möglich
 - wird **ein** User-Thread blockiert, sind **alle** User-Threads blockiert
 - Scheduling zwischen den Threads schwierig

- Kernel-Threads (“leichtgewichtig”)
 - + Gruppe von Threads nutzt gemeinsam die Betriebsmittel eines Prozesses
 - + jeder Thread ist als eigener Aktivitätsträger dem Betriebssystemkern bekannt
 - Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei „schwergewichtigen“ Prozessen, aber erheblich teurer als bei User-Threads



Arten von Threads



Agenda

6.1 Threads

6.2 Schnittstelle

6.3 Koordinierung

6.4 Aufgabe 5: patric

6.5 Gelerntes anwenden



■ Thread erzeugen

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

- thread Thread-ID
- attr Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.
- Nach der Erzeugung führt der Thread die Funktion `start_routine` mit Parameter `arg` aus
- Im Fehlerfall wird **errno nicht gesetzt**, aber ein Fehlercode als Ergebnis zurückgeliefert.
 - Um `perror(3)` verwenden zu können, muss der Rückgabewert erst in der **errno** gespeichert werden.

■ Eigene Thread-ID ermitteln

```
pthread_t pthread_self(void);
```

- Die Funktion kann nie fehlschlagen.



- Thread beenden (bei Rücksprung aus `start_routine` oder):
`void pthread_exit(void *retval)`
 - Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join(3)`)
- Auf Thread warten, Ressourcen freigeben und Rückgabewert abfragen:
`int pthread_join(pthread_t thread, void **retvalp)`
 - Wartet auf den Thread mit der Thread-ID `thread` und liefert dessen Rückgabewert über `retvalp` zurück.
- Ressourcen automatisch bei Beendigung freigeben:
`int pthread_detach(pthread_t thread)`
 - Die mit dem Thread `thread` verbundenen Systemressourcen werden bei dessen Beendigung automatisch freigegeben. Der Rückgabewert der Thread-Funktion kann nicht abgefragt werden.



Beispiel: Matrix-Vektor-Multiplikation

```
static double a[100][100], b[100], c[100];

int main(int argc, char *argv[]) {
    pthread_t tids[100];
    ...
    for(int i = 0; i < 100; i++)
        pthread_create(&tids[i], NULL, mult, (void *) i);
    for(int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
    ...
}

static void *mult(void *cp) {
    int i = (int) cp;
    double sum = 0;
    for(int j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return NULL;
}
```

- Casts zwischen `int` und Zeiger (bei Parameterübergabe für `pthread_create()`) problematisch – **nicht zu Hause nachmachen!**



Parameterübergabe bei `pthread_create()`

- Generischer Ansatz mit Hilfe einer Struktur für die Argumente

```
struct param {  
    int index;  
};
```

- Für jeden Thread eine eigene Argumenten-Struktur anlegen

- Speicher je nach Situation auf dem Heap oder dem Stack allozieren

```
int main(int argc, char *argv[]) {  
    pthread_t tids[100];  
    struct param args[100];  
  
    for(int i = 0; i < 100; i++) {  
        args[i].index = i;  
        pthread_create(&tids[i], NULL, mult, &args[i]);  
    }  
    for(int i = 0; i < 100; i++)  
        pthread_join(tids[i], NULL);  
    ...  
}
```



Parameterübergabe bei `pthread_create()`

```
static void* mult(void *arg) {
    struct param *par = (struct param *) arg;

    double sum = 0;
    for(int j = 0; j < 100; j++) {
        sum += a[par->index][j] * b[j];
    }
    c[par->index] = sum;
    return NULL;
}
```

- Zugriff auf den threadspezifischen Parametersatz über (gecasteten) Parameter (`void *arg` → `struct param *par`)



Agenda

6.1 Threads

6.2 Schnittstelle

6.3 Koordinierung

6.4 Aufgabe 5: patric

6.5 Gelerntes anwenden



Koordinierung – Motivation

Was macht das Programm? Welches Problem kann auftreten?

```
static double a[100][100], sum;

int main(int argc, char *argv[]) {
    pthread_t tids[100];
    struct param args[100];

    for(int i = 0; i < 100; i++) {
        args[i].index = i;
        pthread_create(&tids[i], NULL, sumRow, &args[i]);
    }
    for(int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
}

static void *sumRow(void *arg) {
    struct param *par = (struct param *) arg;
    double localSum = 0;
    for(int j = 0; j < 100; j++)
        localSum += a[par->index][j];
    sum += localSum;
    return NULL;
}
```



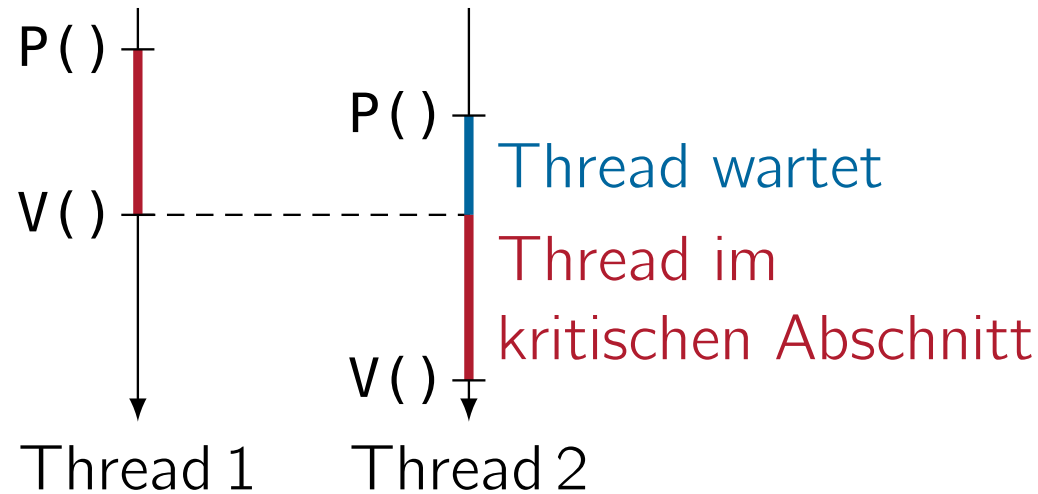
Semaphore

- Zur Koordinierung von Threads können Semaphore verwendet werden
- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
 - Implementierung durch den Systemkern
 - komplexe Datenstrukturen, aufwändig zu programmieren
 - für die Koordinierung von Threads viel zu teuer
- Stattdessen Verwendung einer eigenen Semaphorimplementierung mit atomaren $P()$ - und $V()$ -Operationen
 - Datenstruktur mit (atomarer) Zählervariable
 - $P()$ dekrementiert Zähler und blockiert Aufrufer, falls Zähler == 0
 - $V()$ inkrementiert Zähler und weckt ggf. wartende Threads
 - Mehr Details: s. Vorlesung B | VI.1, Seite 22f.



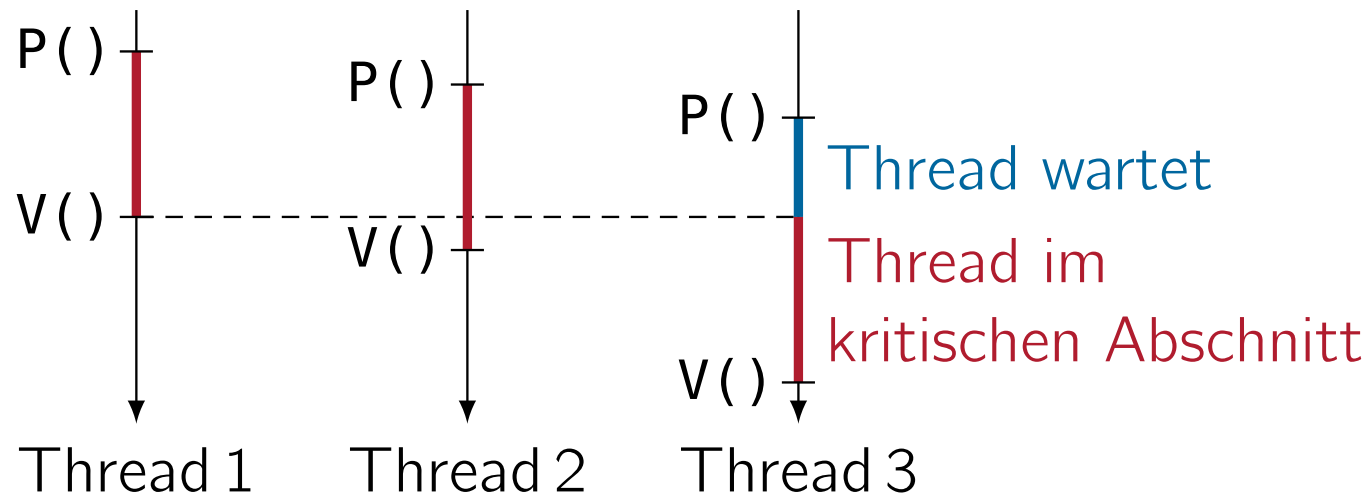
Gegenseitiger Ausschluss

- Spezialfall des zählenden Semaphors: Binärer Semaphor
 - Initialisierung des Semaphors mit 1
- Beispiel: Schreibender Zugriff auf ein gemeinsames Datum



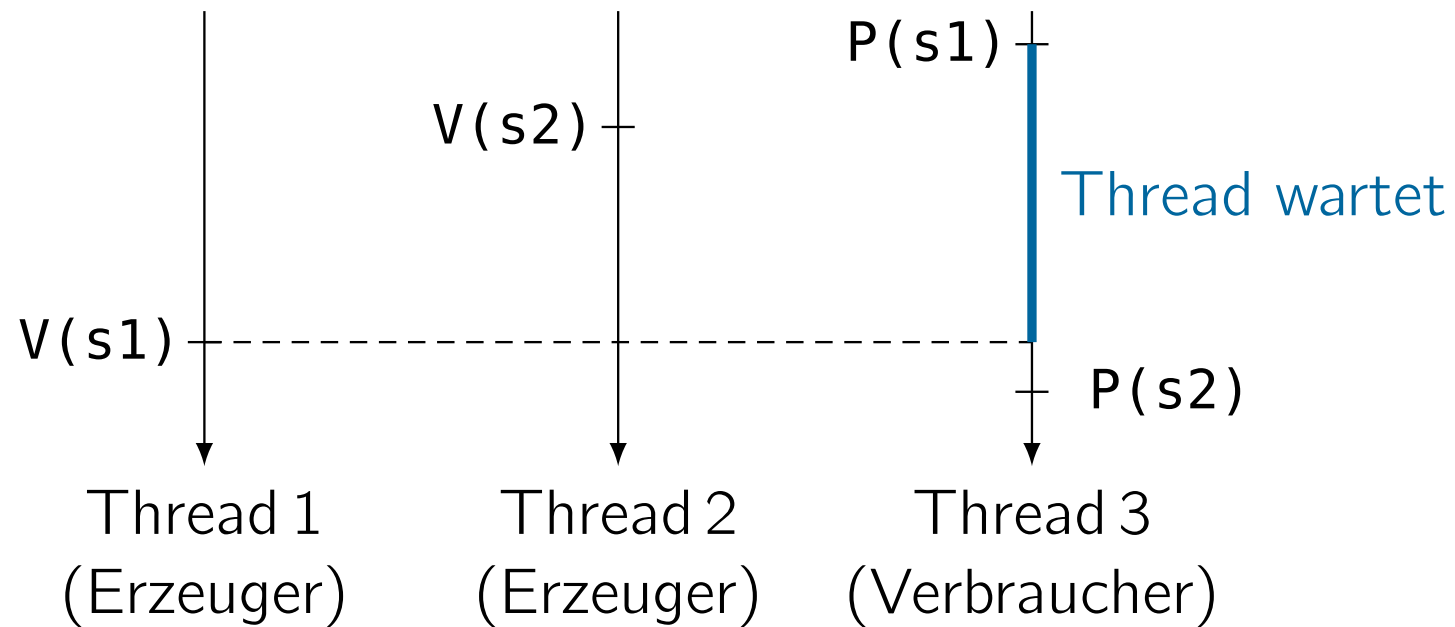
Limitierung von Ressourcen

- Verwendung eines zählenden Semaphors
- Beispiel: Nur zwei aktive Threads gleichzeitig gewünscht
 - Initialisierung des Semaphors mit 2



Signalisierung

- Benachrichtigung eines anderen Threads über ein Ereignis
- Beispiel: Bereitstellen von Zwischenergebnissen



SP-Semaphor-Modul

- Semaphor erzeugen

```
SEM* semCreate(int initVal)
```

- P/V-Operationen

```
void P(SEM *sem)
```

```
void V(SEM *sem)
```

- Semaphor zerstören

```
void semDestroy(SEM *sem)
```

- Semaphor-Modul und zugehörige Headerdatei befinden sich im pub-Verzeichnis.



Agenda

- 6.1 Threads
- 6.2 Schnittstelle
- 6.3 Koordinierung
- 6.4 Aufgabe 5: patric
- 6.5 Gelerntes anwenden



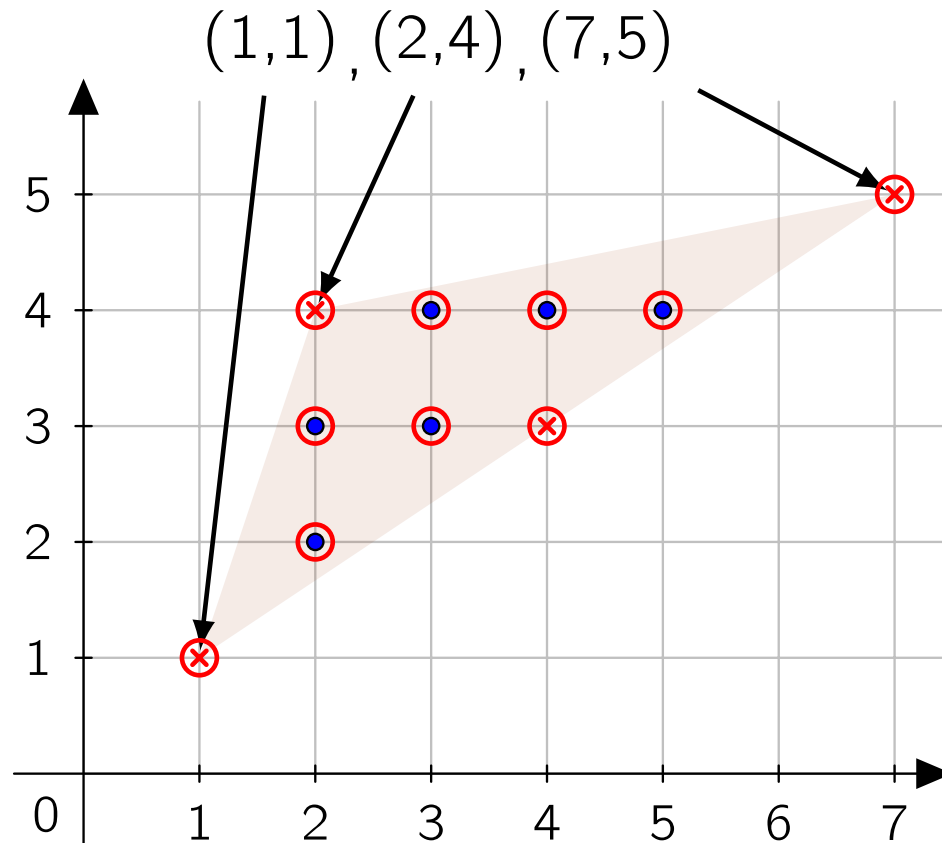
Funktionsweise der `patric`

(Paralleles) Bestimmen der Anzahl der Punkte mit ganzzahligen Koordinaten auf den Kanten / innerhalb vorgegebener Dreiecke

- Ziele der Aufgabe
 - Programmieren mit der `pthread`-Bibliothek
 - Erkennen von Nebenläufigkeitsproblemen
 - Einsatz geeigneter Koordinierungsmaßnahmen
- Zur Verwendung der Pthreads-Bibliothek ist die `gcc`-Option `-pthread` beim Übersetzen und Linken notwendig



patric: Eingabeformat am Beispiel



4 *boundary points* (x)
6 *interior points* (•)



Nutzung des **vorgegebenen Moduls** triangle zum Zählen der Punkte



triangle-Modul

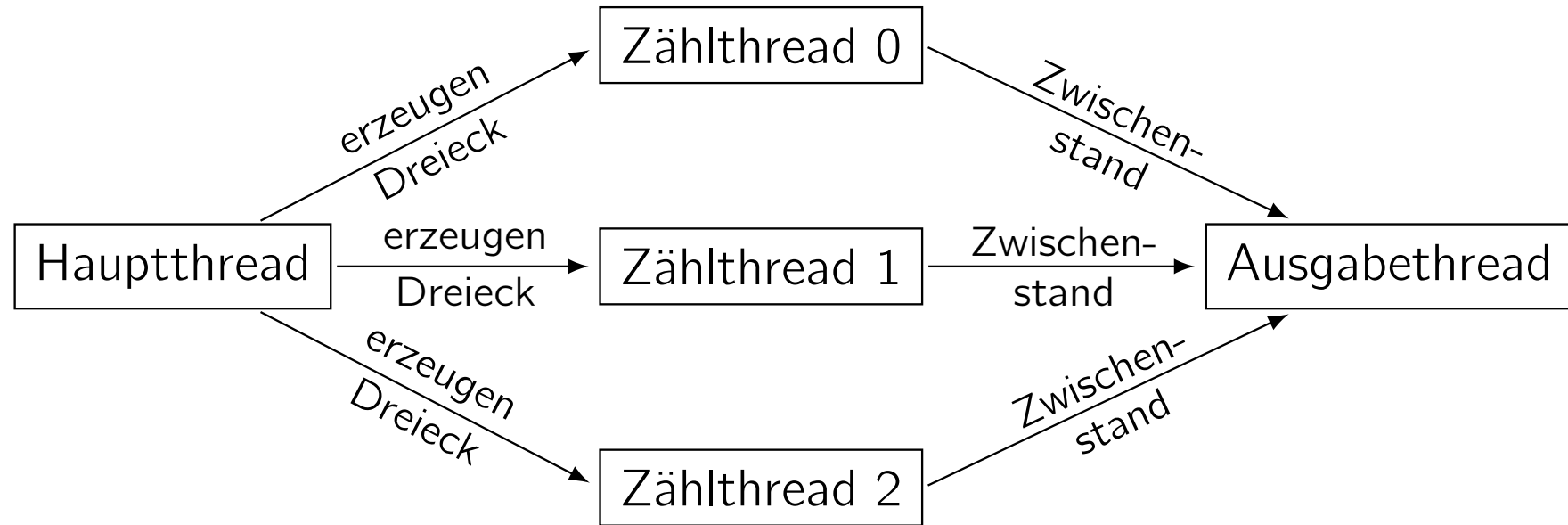
Enthält die vorgegebene Implementierung von

```
void countPoints(  
    struct triangle *tri,  
    void (*callback)(int boundary, int interior)  
);
```

- `tri`: Dreieck, dessen boundary/interior points gezählt werden
- `callback`: Funktion, die periodisch aufgerufen wird und die Anzahl der gefundenen Punkte seit dem letzten Aufruf übergeben bekommt.



Aufgaben der Threads



- *Hauptthread* (`main()`): Einlesen der Eingabedaten, Starten eines Zählthreads pro Dreieck
- *Zählthread*: Zählen der boundary/interior points
- *Ausgabethread*: Ausgeben des aktuellen Zwischenstands/des finalen Ergebnisses



Agenda

- 6.1 Threads
- 6.2 Schnittstelle
- 6.3 Koordinierung
- 6.4 Aufgabe 5: patric
- 6.5 Gelerntes anwenden



„Aufgabenstellung“

- Beispiel von Folie 6–12 mit Hilfe eines Semaphors korrekt synchronisieren

