

Rechnerarchitektur SoSe 2019

Multithreading

Jian-Jia Chen

TU Dortmund

Teilweise basierend auf Material von Michael Engel, Gernot A. Fink und
R. Yahyapour

02/08 April 2019

Multithreading: Einleitung

- Generelle **Eigenschaft** komplexer Rechnerarchitekturen: **Latenzen** entstehen durch ... (Wir besprechen die Themen später)
 - Speicherzugriffe (d.h. *cache misses*)
 - Synchronisationsoperationen
 - Kohärenzprotokolle
 - Speicherkonsistenz
- Die generellen **Ziele** des Architekturentwurfs: *latency hiding*, d.h. "Verstecken" / Unsichtbarmachen auftretender Latenzen
- Naheliegende Idee: Wartezeiten ausnutzen zur Ausführung eines anderen **Threads** (dt. Faden), d.h. eines parallelen Kontrollflusses
⇒ **Multithreading**

Zwei mögliche Sichten auf Multithreading:

1. *Hardware-Ebene:*

Wie wird die parallele Ausführung mehrerer Kontrollflüsse erreicht / verwaltet?

2. *Software-Ebene:*

Wie werden parallele Kontrollflüsse innerhalb eines Programms definiert?

Wie können diese parallelen Kontrollflüsse kommunizieren bei der gemeinsamen Bearbeitung eines Problems?

Multithreading: Einleitung II

Zwei mögliche Sichten auf Multithreading:

1. *Hardware-Ebene:*

Wie wird die parallele Ausführung mehrerer Kontrollflüsse erreicht / verwaltet?

2. *Software-Ebene:*

Wie werden parallele Kontrollflüsse innerhalb eines Programms definiert?

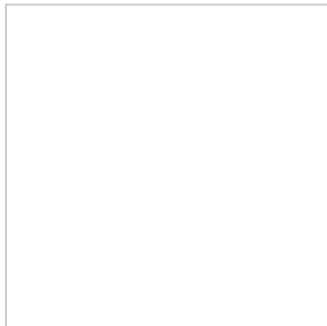
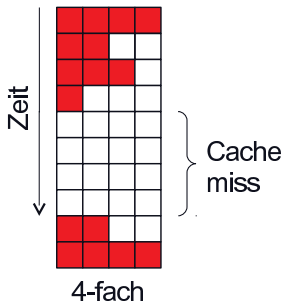
Wie können diese parallelen Kontrollflüsse kommunizieren bei der gemeinsamen Bearbeitung eines Problems?

Betrachten zunächst Hardware-Ebene

1. Hardware-Ebene

Multithreading: Ausführungsplattform

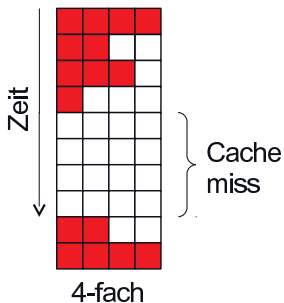
- Verbreitete Prozessorarchitektur: Uniprozessor mit mehreren funktionalen Einheiten, d.h. **superskalar**



Superskalar: Geringe Ressourcenauslastung (hier: 40%)

Multithreading: Ausführungsplattform

- Verbreitete Prozessorarchitektur: Uniprozessor mit mehreren funktionalen Einheiten, d.h. **superskalar**
- Einfaches Multithreading: Ausführung eines anderen Threads während (längerer) Wartezeiten



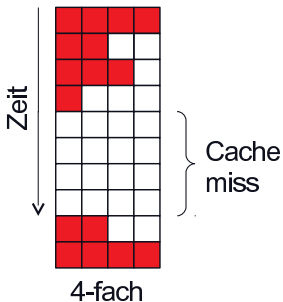
Superskalar: Geringe Ressourcenauslastung (hier: 40%)



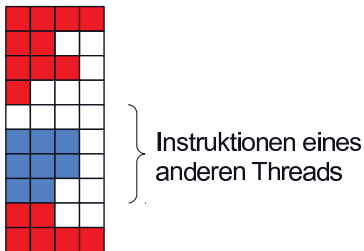
MT: Verbesserte Ressourcenauslastung (hier: 60%)

Multithreading: Ausführungsplattform

- Verbreitete Prozessorarchitektur: Uniprozessor mit mehreren funktionalen Einheiten, d.h. **superskalar**
- Einfaches Multithreading: Ausführung eines anderen Threads während (längerer) Wartezeiten



Superskalar: Geringe Ressourcenauslastung (hier: 40%)



MT: Verbesserte Ressourcenauslastung (hier: 60%)

Multithreading: Vorteile?

Betrachtung der Latenz: Einzel vs. gesamt

Multithreading führt i.d.R. zu ...

- *höherer* Latenz für (jeden) einzelnen Thread aber ...
- höherem Durchsatz und damit *geringerer* Gesamtlatenz.

Beispiel

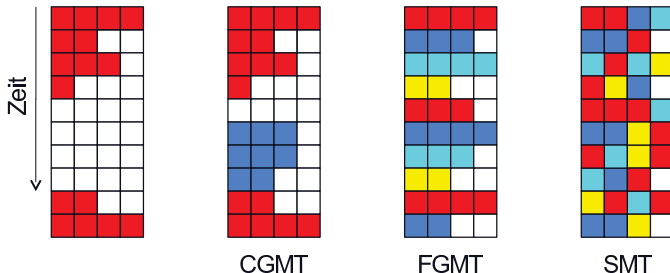
- Thread A: individuelle Latenz=10s, mit Thread B=15s
- Thread B: individuelle Latenz=20s, mit Thread A=25s
- Sequentiell (A → B): 30s
- Parallel (A || B): 25s
- ⚡ Einzellatenz erhöht: +5s
- ✓ Gesamtlatenz reduziert: -5s

Multithreading

Arten von Multithreading

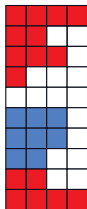
- Grobgranulares Multithreading (CGMT)
[engl. *coarse-grained multithreading*]
- Feingranulares Multithreading (FGMT)
[engl. *fine-grained multithreading*]
- Simultanes Multithreading (SMT)

Quelle: R. Yahyapour, ehem. TU Dortmund



Grobgranulares Multithreading

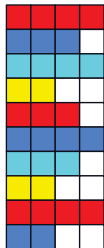
- Wechsel der Threads bei längeren Verzögerungen (z.B. L2-Cache Miss)
- Vorteile
 - Kleine Verzögerung bei Threadwechsel tolerierbar
 - Kritische Threads priorisierbar
- Nachteil
 - Kurze Stalls (pipeline draining) müssen toleriert werden
- Scheduling
 - Wahl eines bevorzugten Threads (z.B. A)
 - Wechsel zu Thread B bei L2-Miss von Thread A
 - Rückkehr zu Thread A, wenn Daten in L2
- Behandlung von Pipeline Stalls
 - Leerung bei Threadwechsel
 - Tolerierbare Latenz $> 2 \times$ Pipeline Länge



Quelle: R. Yahyapour, ehem. TU Dortmund

Feingranulares Multithreading

- Wechsel der Threads bei jeder Instruktion
- Vorteil
 - Jegliche Prozessorverfügbarkeit nutzbar
- Nachteil
 - Höhere Latenz einzelner (kritischer) Threads
- Umsetzung
 - Rudentabelle (round robin)
 - Auslassen wartender Threads
- Voraussetzung
 - Keine Zusatzverzögerung (pipeline stall)
 - Threadwechsel nach jedem Taktzyklus

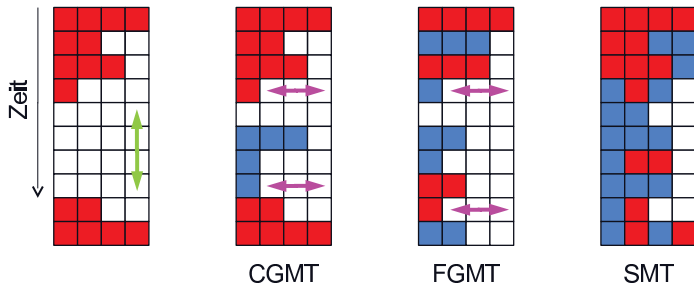


Quelle: R. Yahyapour, ehem. TU Dortmund

Verbesserungen durch CGMT & FGMT

- Höhere “vertikale” Auslastung \rightarrow mehr Takte genutzt
- Keine “horizontale” Verbesserung \rightarrow ungenutzte Slots

Quelle: R. Yahyapour, ehem. TU Dortmund



Simultanes Multithreading

z.B. Hyperthreading [Intel, erstmals Pentium 4, heute Core i7]

- Eigenschaften moderner superskalarer Prozessoren¹
 - Mehrere funktionale Einheiten
 - Register Renaming
 - Dynamic Scheduling
- Vorteil
 - Instruktionen unterschiedlicher, unabhängiger² Threads simultan ausführbar
- Nachteil
 - Höhere Latenz einzelner (kritischer) Threads
 - Implementierungsaufwand!



¹Details will be discussed later.

²Auflösung von Abhängigkeiten durch Dynamic Scheduling

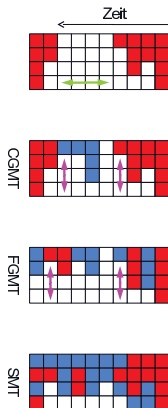
Multithreading: Vergleich

Quelle: R. Yahyapour, ehem. TU Dortmund

Grundlage: Superskalarer Prozessor

Wieso?

- ohne MT: Verschiedene Ereignisse (z.B. Cache Miss)
⇒ ungenutzte Prozessoren
- CGMT: Einzelne funktionale Einheiten ungenutzt
- FGMT: Kein Ausgleich zwischen verschiedenen Instruktionen, erfordert sehr viele Threads
- SMT: Besserer Ausgleich zwischen verschiedenen Threads



Implementierung — Herausforderungen

- Große Anzahl physikalischer Register
- Unabhängige Renaming Tabellen (je Thread)
- Separate Program Counter
- Unabhängiger Abschluss der Instruktionen unterschiedlicher Threads
⇒ Erfordert separate Umordnungsspeicher
- Tiefe Pipeline führt zu Leistungseinbußen
- Großes Registerfile mit mehreren Kontexten
- Geringer Overhead im Taktzyklus
(betrifft Instruction Issue, Instruction Completion)
- Cache-Konflikte bei SMT

GCMT — Anzahl Threads

- Wieviele Threads sollte ein Prozessor für GCMT unterstützen?
- Einfaches Modell (Auslastung U):

$$U = \frac{\text{Busy}}{\text{Busy} + \text{Kontextwechsel} + \text{Unbenutzt}}$$

- Relevante Parameter:
 - N : Anzahl der (unterstützten) aktiven Threads pro Prozessor (*N-way multithreading*)
 - R : (durchschnittl.) Länge der ununterbrochen ausgeführten Instruktionen (ohne stalls; *run length*)
 - L : (durchschnittl.) Latenz, d.h. Dauer des Blockadeereignisses
 - C : Overhead für Kontextwechsel

GCMT — Anzahl Threads II

Beispiel zur Auslastungsberechnung (Culler/Singh/Gupta 1999)

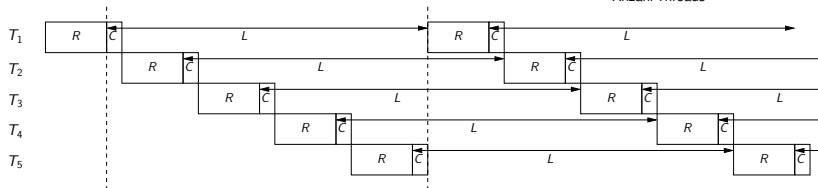
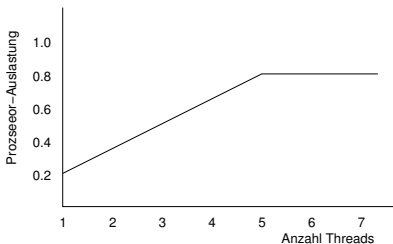
Parameter:

$$R = 40,$$

$$L = 210,$$

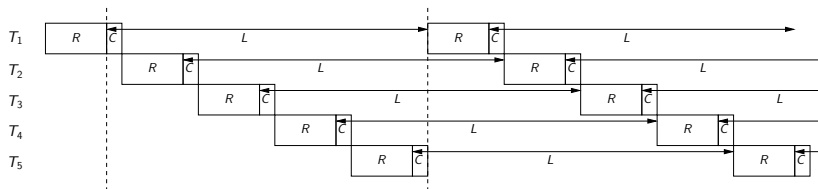
$$C = 10$$

(Taktzyklen)



GCMT — Anzahl Threads II

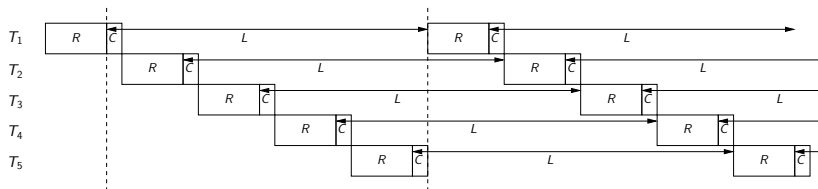
Beispiel zur Auslastungsberechnung (Culler/Singh/Gupta 1999)



Folgerung 1: Sättigung erreicht, falls $L = (N - 1) \cdot R + N \cdot C$

$$N_{\text{sat}} = \frac{R + L}{R + C} \quad U_{\text{sat}} = \frac{R}{R + C} = \frac{1}{1 + \frac{C}{R}}$$

Beispiel zur Auslastungsberechnung (Culler/Singh/Gupta 1999)



Folgerung 1: Sättigung erreicht, falls $L = (N - 1) \cdot R + N \cdot C$

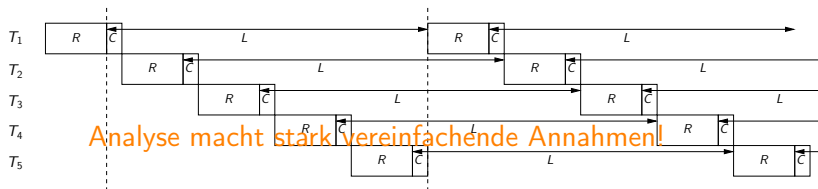
$$N_{\text{sat}} = \frac{R + L}{R + C} \quad U_{\text{sat}} = \frac{R}{R + C} = \frac{1}{1 + \frac{C}{R}}$$

Folgerung 2: Sonst $R + R \cdot (N - 1)$ Takte genutzt von $R + L$:

$$U_{\text{lin}} = \frac{NR}{R + L} = N \cdot \frac{1}{1 + \frac{L}{R}}$$

GCMT — Anzahl Threads II

Beispiel zur Auslastungsberechnung (Culler/Singh/Gupta 1999)



Folgerung 1: Sättigung erreicht, falls $L = (N - 1) \cdot R + N \cdot C$

$$N_{\text{sat}} = \frac{R + L}{R + C} \quad U_{\text{sat}} = \frac{R}{R + C} = \frac{1}{1 + \frac{C}{R}}$$

Folgerung 2: Sonst $R + R \cdot (N - 1)$ Takte genutzt von $R + L$:

$$U_{\text{lin}} = \frac{NR}{R + L} = N \cdot \frac{1}{1 + \frac{L}{R}}$$

Übung

Für die Ausführungszeiten der Instruktionen aller Threads einer Anwendung gilt diese Verteilung:

- 40% der Taktzyklen entfallen auf die Ausführung der Instruktionen im Prozessor.
- 30% der Taktzyklen sind Warte-Zyklen, die durch L1-Cache-Misses (aber L2-Cache-Hits) entstehen. Die L1-Miss-Bearbeitung benötigt 10 Taktzyklen.
- 30% der Taktzyklen sind Warte-Zyklen, die durch L2-Cache-Misses hervorgerufen werden (jeweils 30 Taktzyklen).

Im Durchschnitt tritt pro Thread nach jeweils 10 Rechen-Takten ein Cache-Miss auf. Beantworten Sie folgende Fragen unter der Annahme, dass ausreichend viele Threads sowohl von der Hardware als auch durch die Anwendung unterstützt werden.

- a) Wie hoch ist die maximale Auslastung, wenn ein Threadwechsel 5 Taktzyklen dauert?
- b) Wie lange darf ein Threadwechsel höchstens dauern, so dass die Auslastung mindestens 50% beträgt?

Auslösung des Kontextwechsels

- Cache Miss: Entdeckung des Miss (Hardware)
- Synchronisierung: Explizite Wechselinstruktion (Software)
- Lange Pipeline Stalls (z.B. Division): wie Synchron.
- Kurze Pipeline Stalls: kein Wechsel

Ziel: Frühe Entdeckung einer blockierenden Instruktion

Was passiert (bei Kontextwechsel) mit Instruktionen in der Pipeline?

1. Instruktionen dürfen abschließen
⇒ *Gleichzeitig* Laden von Instruktionen des neuen Threads
2. Instruktionen dürfen abschließen
⇒ *Danach* Laden von Instruktionen des neuen Threads
(d.h. Pipeline läuft nach Abarbeitung d. Instruktionen leer)
3. Instruktionen werden gelöscht
⇒ *Dann* Laden von Instruktionen des neuen Threads

Vor- und Nachteile:

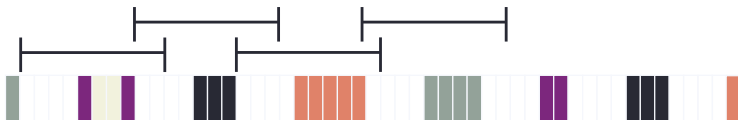
- 1.: Instruktionen verschiedener Threads i.d. Pipeline
⇒ Pipelineregister etc. müssen dafür ausgelegt werden
- 1. und 2.: Stalls in Instruktionen wg. Abhängigkeiten problematisch
- 3.: Keine Probleme mit Stalls, aber größerer Overhead
⇒ präferierte Lösung für CGMT

CGMT und Pipelining

Beispiel für CGMT (d.h. blockierend; Culler/Singh/Gupta 1999)



Quelle: R. Yahvabpour, ehem. TU Dortmund



Pipelining — CGMT vs. FGMT

Auswirkung auf Pipeline: CGMT vs. FGMT

- Pipeline (z.B. 7 Stufen)



⇒ Cache-Miss erst nach DF2 (d.h. in WB) erkannt

- Grobgranulares MT

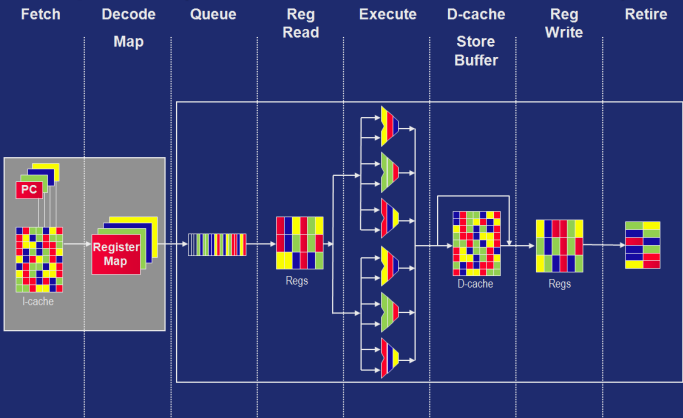


- Feingranulares MT (3 Threads A, B & C)



SMT Implementierungsbeispiel: Alpha 21464

SMT Pipeline (4 Thread, Double Execution)



<http://molesterwaterball.blogspot.com>
痴漢水球の部落格

Copy Right (c) 2010 "Molester Waterball" All rights reserved
Dedicated To **PCM ADV** Hong Kong

2. Software-Ebene

Definition

[Thread] Als Thread bezeichnet man einen unabhängigen Kontrollfluß innerhalb eines Programms / Prozesses. Mehrere Threads desselben Programms verwenden einen gemeinsamen Adressraum und können prinzipiell parallel ausgeführt werden.

- Ein Thread ist eine “leichtgewichtiger” Verwaltungseinheit (des Betriebssystems) als ein Prozess (ähnlich / gleich(?) *light-weight process*).
- Threads können über den gemeinsam genutzten Speicher kommunizieren.
- Die parallele Ausführung von Threads erfordert ggf. Synchronisation.

Thread-Programmierung II

- Historisch betrachtet ex. jeweils *unterschiedliche* Thread-Implementierungen einzelner Hersteller
⇒ Entwicklung portabler Thread-Anwendungen unmöglich!
- Für Unix-artige Betriebssysteme (*heute* z.B. Linux, MacOS):
Standardisierung des Thread Interfaces durch den **IEEE POSIX 1003.1c Standard** (1995)
⇒ Standardkonforme Implementierungen: **POSIX Threads**
- POSIX Threads (bzw. PThreads) definieren
 - eine Menge von C Datentypen und Bibliotheksfunktionen (i.d.R. realisiert als Include-Datei und Library)
 - sowie deren Verhalten.

Verwendung des POSIX Thread API:

Im Quellcode: `#include <pthread.h>`

Beim Compilieren und Linken: `-pthread`

Erzeugung von Threads:

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

- `pthread_create()` liefert den Rückgabewert 0 (erfolgreich) oder einen Fehlercode.
- Im Erfolgsfall enthält `thread` die ID des neu erzeugten Threads.
- Der neue Thread führt `start_routine()` aus, ...
- die mit dem Argument `arg` aufgerufen wird.
- Mit `attr` können Details des Threadverhaltens spezifiziert werden (z.B. das Scheduling).

In `start_routine()` sollten nur *reentrante* Funktionen verwendet werden!

Beenden von Threads:

```
void pthread_exit ( void *retval );  
void pthread_join ( pthread_t thread, void **retval );  
int pthread_cancel ( pthread_t thread );
```

- `pthread_exit()` beendet den aktuellen Thread und liefert den Rückgabewert `retval`.
 - Äquivalent zu `return(retval)` in `start_routine()`.
- `pthread_join()` wartet, dass `thread` sich beendet und kopiert dessen Rückgabewert in `retval` (sofern nicht `NULL`).
- `pthread_cancel()` *versucht* `thread` (asynchron) zu beenden.

POSIX Threads: Beispiel

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *arg)
{
    long tid;
    tid = (long)arg;
    printf("this is thread number %ld!\n", tid);
    pthread_exit(NULL);
}
```

nach: Barney: "POSIX Threads Programming", Lawrence Livermore National Laboratory

POSIX Threads: Beispiel (cont.)

```
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL,
                           PrintHello, (void *)t);
        if (rc) {
            printf("error creating thread: %d\n", rc);
            exit(-1);
        }
    }
}
```

nach: Barney: "POSIX Threads Programming", Lawrence Livermore National Laboratory

POSIX Threads: Beispiel (cont.)

```
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL,
                           PrintHello, (void *)t);
        if (rc) {
            Hack: Übergabe der Thread-Id t!
            printf("error creating thread: %d\n", rc);
            exit(-1);
        }
    }
}
```

nach: Barney: "POSIX Threads Programming", Lawrence Livermore National Laboratory

POSIX Threads: Beispiel (cont.)

```
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL,
                           PrintHello, (void *)t);
        if (rc) {
            printf("error creating thread: %d\n", rc);
            exit(-1);
        }
    }
}
```

Hack: Übergabe der Thread-Id t!

Wie sieht die Programmausgabe aus?

nach: Barney: "POSIX Threads Programming", Lawrence Livermore National Laboratory

POSIX Threads: Argumentübergabe

Startroutine des neuen Threads wird mit bei `pthread_create()` angegebenem Argument aufgerufen, d.h.

```
pthread_create(&thread, NULL, start, (void *)arg);
```

führt zu einem Aufruf von

```
start(arg)
```

sobald `thread` gescheduled wird.



Datentyp des Arguments ist *generischer Pointer* (d.h. `(void *)`)!

POSIX Threads: Beispiel (2)

Saubere Übergabe der Thread-ID:

```
int thread_data[NUM_THREADS];

void *PrintHello(void *arg) {
    int *my_data;
    ...
    my_data = (int *)arg;
    taskid = *my_data;
    ...
}

int main (int argc, char *argv[]) {
    ...
    thread_data[t] = t;
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data[t]);
    ...
}
```

nach: Barney: "POSIX Threads Programming", Lawrence Livermore National Laboratory

POSIX Threads: Beispiel (2, cont.)

```
struct thread_data {
    int id;
    char *message;
} thread_data[NUM_THREADS];

void *PrintHello(void *arg) {
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *)arg;
    taskid = my_data->id;
    msg = my_data->message;
    ...
}

int main (int argc, char *argv[]) {
    ...
    thread_data[t].id = t;
    thread_data_array[t].message = ...
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data[t]);
    ...
}
```

... und Übergabe einer Datenstruktur:

Gegenseitiger Ausschluss mit Mutexen

```
int pthread_mutex_init (    pthread_mutex_t *mutex,  
                           pthread_mutexattr_t *attr);  
pthread_mutex_destroy (    pthread_mutex_t *mutex);  
  
int pthread_mutex_lock (    pthread_mutex_t *mutex);  
int pthread_mutex_unlock (  pthread_mutex_t *mutex);
```

- Mutexe müssen vor Gebrauch initialisiert werden.
- Funktionspaar `pthread_mutex_lock()` und `pthread_mutex_unlock()` realisieren *blockierendes* lock/unlock.

PThread Mutexe: Beispiel

FIFO-Puffer-Manipulation (für Erzeuger-Verbraucher-Kommunikation)

```
int rfifo_push(rfifo_t *fifo, rfifo_packet_t *packet) {
    int status;

    pthread_mutex_lock(&(fifo->lock));

    while (fifo->elements >= fifo->size)
        /* wait for buffer space to become available */;

    fifo->buffer[fifo->head] = packet; /* insert element into queue */
    fifo->head = (fifo->head + 1) % fifo->size;
    status = ++(fifo->elements);

    pthread_mutex_unlock(&(fifo->lock));

    return(status);
}
```

PThread Mutexe: Beispiel

FIFO-Puffer-Manipulation (für Erzeuger-Verbraucher-Kommunikation)

```
int rfifo_push(rfifo_t *fifo, rfifo_packet_t *packet) {
    int status;

    pthread_mutex_lock(&(fifo->lock));

    while (fifo->elements >= fifo->size)
        /* wait for buffer space to become available */;

    fifo->buffer[fifo->head] = packet; /* insert element into queue */
    fifo->head = (fifo->head + 1) % fifo->size;
    status = ++(fifo->elements);

    pthread_mutex_unlock(&(fifo->lock));

    return(status);
}
```

Wie *sinnvoll* auf freiwerdenden Pufferplatz warten?

Benachrichtigung *und* gegenseitiger Ausschluss mit Conditions

```
int pthread_cond_init (    pthread_cond_t *cond,  
                          pthread_condattr_t *attr);  
  
int pthread_cond_signal ( pthread_cond_t *cond);  
int pthread_cond_wait (   pthread_cond_t *cond,  
                          pthread_mutex_t *mutex);
```

- Conditions müssen vor Gebrauch initialisiert werden.
- `pthread_cond_wait()` wartet auf die Signalisierung von `cond` **und gibt davor den gelockten mutex frei!**
- `pthread_cond_signal()` "weckt" einen Thread, der sich an `cond` blockiert hat. **Dieser erhält das Lock für den (dynamisch mit `cond` assoziierten) mutex!**



Tatsächliche Bedingung muss separat überprüft werden!

PThread Conditions: Beispiel

FIFO-Puffer-Manipulation (für Erzeuger-Verbraucher-Kommunikation)

```
int rfifo_push(rfifo_t *fifo, rfifo_packet_t *packet) {
    int status;

    pthread_mutex_lock(&(fifo->lock));

    while (fifo->elements >= fifo->size)
        pthread_cond_wait(&(fifo->changed), &(fifo->lock));

    fifo->buffer[fifo->head] = packet; /* insert element into queue */
    fifo->head = (fifo->head + 1) % fifo->size;
    status = ++(fifo->elements);

    pthread_cond_signal(&(fifo->changed));
    pthread_mutex_unlock(&(fifo->lock));

    return(status);
}
```

POSIX Semaphore

Verwendung des POSIX Semaphore API:



Im Quellcode: `#include <semaphore.h>`

Beim Linken: `-lrt` oder `-pthread`

... nicht direkt Teil des PThread-APIs

POSIX Semaphore

`int sem_wait(sem_t *sem);` d.h. Äquivalent zu $P(S)$
`int sem_post(sem_t *sem);` d.h. Äquivalent zu $V(S)$

- `sem_wait()` liefert 0 im Erfolgsfall.
-  `sem_wait()` kann durch Interruptbehandlung unterbrochen werden, muss dann erneut versucht werden!
-  Nur bei binären Semaphoren äquivalent zu lock/unlock!

Erzeuger-Verbraucher mit Semaphoren

```
sem_t empty, full, mutex;
```

... und endlichem Puffer!

```
void *producer(void *arg) {
    int i;
    for (i=0; i<loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int i, tmp;
    for (i=0; i<loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}
```

nach: Arpaci-Dusseau & Arpaci-Dusseau: Operating Systems, Lehrbuchentwurf

Erzeuger-Verbraucher mit Semaphoren

```
sem_t empty, full, mutex;           ... und endlichem Puffer!
sem_init(&empty, 0, MAX); // initially: MAX buffers empty
sem_init(&full, 0, 0);    // ... and 0 are full
sem_init(&mutex, 0, 1);  // mutex=1 because it is a lock
// ...

void *producer(void *arg) {        void *consumer(void *arg) {
    int i;                          int i, tmp;
    for (i=0; i<loops; i++) {      for (i=0; i<loops; i++) {
        sem_wait(&empty);           sem_wait(&full);
        sem_wait(&mutex);           sem_wait(&mutex);
        put(i);                     tmp = get();
        sem_post(&mutex);           sem_post(&mutex);
        sem_post(&full);            sem_post(&empty);
    }                                printf("%d\n", tmp);
}                                    }
}
```

nach: Arpaci-Dusseau & Arpaci-Dusseau: Operating Systems, Lehrbuchentwurf

Erzeuger-Verbraucher mit Semaphoren

```
sem_t empty, full, mutex;           ... und endlichem Puffer!  
sem_init(&empty, 0, MAX); // initially: MAX buffers empty  
sem_init(&full, 0, 0);    // ... and 0 are full  
sem_init(&mutex, 0, 1);  // mutex=1 because it is a lock  
// ...  
void *producer(void *arg) {        void *consumer(void *arg) {  
    int i;                          int i, tmp;  
    for (i=0; i<loops; i++) {      for (i=0; i<loops; i++) {  
        sem_wait(&empty);          sem_wait(&full);  
        sem_wait(&mutex);          sem_wait(&mutex);  
        put(i);                    tmp = get();  
        sem_post(&mutex);          sem_post(&mutex);  
        sem_post(&full);           sem_post(&empty);  
    }                               printf("%d\n", tmp);  
}                                   }  
}
```

Bei mehreren Erzeugern/Verbrauchern
muss Verfügbarkeit in get()/put() erneut geprüft werden!

nach: Arpaci-Dusseau & Arpaci-Dusseau: Operating Systems, Lehrbuchentwurf

Erzeuger-Verbraucher mit Mutexen/Conditions

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

typedef struct {
    size_t size;           /* total length of buffer */
    size_t head;          /* index to first and ... */
    size_t tail;           /* ... last element in ring buffer */
    rfifo_packet_t **buffer; /* buffer holding data packets */
    size_t elements;       /* number of elements in buffer */
    pthread_mutex_t lock;  /* lock for buffer access */
    pthread_cond_t changed;
} rfifo_t;

void main(int argc, char **argv) {
    rfifo_t *fifo;
    pthread_t prod, cons;

    fifo = rfifo_make(FIFOSIZ);

    pthread_create(&prod, NULL, producer, fifo); /* create producer and ... */
    pthread_create(&cons, NULL, consumer, fifo); /* consumer thread with default attributes */

    pthread_join(prod, NULL); /* wait to finish, don't care for result */
    pthread_join(cons, NULL); /* wait to finish, don't care for result */
    ...
}
```

Erzeuger-Verbr. mit Mutexen/Conditions II

```
void producer(rfifo_t *fifo) {
    rfifo_packet_t *packet;

    do {
        packet = /* somehow create a packet */;
        rfifo_push(fifo, packet);
    }
    while (packet != NULL);

    pthread_exit(NULL); /* no return value */
}

int rfifo_push(rfifo_t *fifo,
              rfifo_packet_t *packet) {
    int status;

    pthread_mutex_lock(&(fifo->lock));

    while (fifo->elements >= fifo->size)
        pthread_cond_wait(&(fifo->changed),
                        &(fifo->lock));

    /* insert element */
    fifo->buffer[fifo->head] = packet;
    fifo->head = (fifo->head + 1) % fifo->size;
    status = ++(fifo->elements);

    pthread_cond_signal(&(fifo->changed));
    pthread_mutex_unlock(&(fifo->lock));
    return(status);
}
```

```
void consumer(rfifo_t *fifo) {
    rfifo_packet_t *packet;

    while (1) {
        packet = rfifo_pop(fifo);
        if (packet == NULL)
            pthread_exit(NULL);

        /* somehow process data */ ;
    }
}

rfifo_packet_t *rfifo_pop(rfifo_t *fifo) {
    rfifo_packet_t *packet;

    pthread_mutex_lock(&(fifo->lock));


    while (fifo->elements <= 0)
        pthread_cond_wait(
            &(fifo->changed), &(fifo->lock));

    /* retrieve element */
    packet = fifo->buffer[fifo->tail];
    fifo->tail = (fifo->tail + 1) % fifo->size;
    fifo->elements--;

    pthread_cond_signal(&(fifo->changed));
    pthread_mutex_unlock(&(fifo->lock));

    return(packet);
}
```

POSIX Threads: Zusammenfassung

- Standardisierte Programmierschnittstelle für thread-basierte Anwendungen auf Unix-artigen Betriebssystemen
- Definiert Funktionalität für:
 - Thread-Verwaltung (Erzeugen, Beenden)
 - Synchronisierung mit Hilfe von
 - Mutexen \Rightarrow gegenseitiger Ausschluss
 - Conditions \Rightarrow *zusätzlich* Benachrichtigung
 -  Semaphore *nicht* enthalten!
(separat als Weiterentwicklung von BSD-Semaphoren mit System V-Semantik)
 - detaillierte Kontrolle von Thread-Eigenschaften (Scheduling, Prioritäten, Unterbrechbarkeit ...)
- Ex. PThread Implementierungen für nicht-Unix-basierte Betriebssysteme (z.B. Windows)