

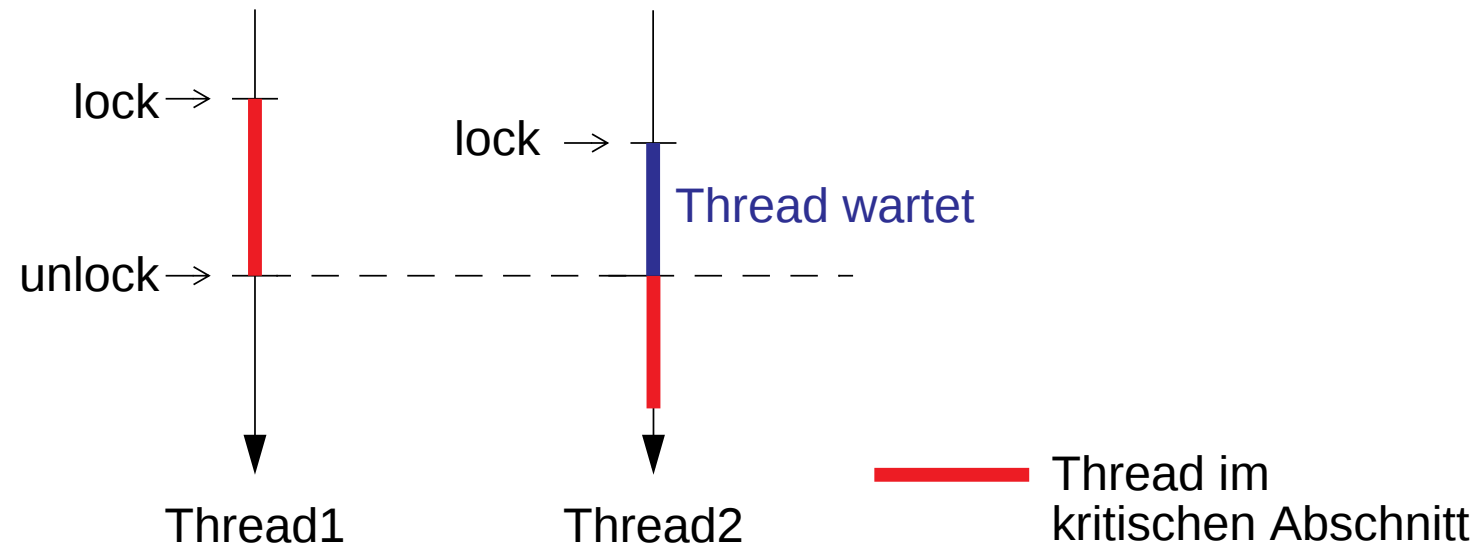
# U4 POSIX-Threads

---

- Mutexe
- Synchronisierung mit Mutexen
- Bedingungsvariablen (Condition Variables)
- Aufgabe 8: mt-httpd

# U4-1 Mutexe

- Koordinierung von kritischen Abschnitten



# U4-1 Mutexe

## ■ Schnittstelle

### ◆ Mutex erzeugen

```
pthread_mutex_t m1;  
s = pthread_mutex_init(&m1, NULL);
```

### ◆ Lock & unlock

```
s = pthread_mutex_lock(&m1);  
... kritischer Abschnitt  
s = pthread_mutex_unlock(&m1);
```

### ◆ Mutex zerstören und Ressourcen freigeben

```
s = pthread_mutex_destroy(&m1);
```

# U4-2 Synchronisierung mit Mutexen

- ...am Beispiel einer Semaphore-Implementierung
- Welches Problem kann hier auftreten?

```
static pthread_mutex_t m1;  
static int a;  
  
void P() {  
    while ( a == 0 ) {  
        // wait for changes  
    }  
    pthread_mutex_lock(&m1);  
    a--;  
    pthread_mutex_unlock(&m1);  
}
```

```
void V() {  
    pthread_mutex_lock(&m1);  
    a++;  
    pthread_mutex_unlock(&m1);  
}
```

# U4-2 Synchronisierung mit Mutexen

- Problem: Mehrere Threads warten gleichzeitig in der Schleife
  - ◆ **a** wird mehrmals heruntergezählt
  - ◆ Prüfung der Bedingung ist ebenfalls Teil des kritischen Abschnittes
- Welches Problem ist nun vorhanden?

```
static pthread_mutex_t m1;  
static int a;  
  
void P() {  
    pthread_mutex_lock(&m1);  
    while ( a == 0 ) {  
        // wait for change  
    }  
    a--;  
    pthread_mutex_unlock(&m1);  
}
```

```
void V() {  
    pthread_mutex_lock(&m1);  
    a++;  
    pthread_mutex_unlock(&m1);  
}
```

## U4-2 Synchronisierung mit Mutexen

- Deadlock, da in kritischem Bereich gewartet wird
  - ◆ Kein anderer Thread kann kritischen Abschnitt betreten
  - ◆ Freigabe des Mutexes notwendig

```
static pthread_mutex_t m1;
static int a;

void P() {
    pthread_mutex_lock(&m1);
    while ( a == 0 ) {
        pthread_mutex_unlock(&m1);
        // wait for change
        pthread_mutex_lock(&m1);
    }
    a--;
    pthread_mutex_unlock(&m1);
}
```

```
void V() {
    pthread_mutex_lock(&m1);
    a++;
    pthread_mutex_unlock(&m1);
}
```

- Zur Vermeidung von aktivem Warten wird sleep/wakeup-Mechanismus benötigt.

## U4-3 Synchronisierung mit Mutexen und Sleep/Wakeup-Mechanismus

- Zur Vermeidung von aktivem Warten wird Pseudo-Funktion `wait_for_change()` eingesetzt, diese blockiert solange bis `signal_change()` aufgerufen wird.

```
static pthread_mutex_t m1;
static int a;

void P() {
    pthread_mutex_lock(&m1);
    while ( a == 0 ) {
        pthread_mutex_unlock(&m1);
        wait_for_change();
        pthread_mutex_lock(&m1);
    }
    a--;
    pthread_mutex_unlock(&m1);
}
```

```
void V() {
    pthread_mutex_lock(&m1);
    a++;
    signal_change();
    pthread_mutex_unlock(&m1);
}
```

- Welches Problem kann hier auftreten?

# U4-3 Synchronisierung mit Mutexen und Sleep/Wakeup-Mechanismus

## ■ Lost-Wakeup-Problem

- ◆ Aufwecksignal kann verloren gehen
- ◆ Freigabe des Mutex und Schlafenlegen muss atomar sein

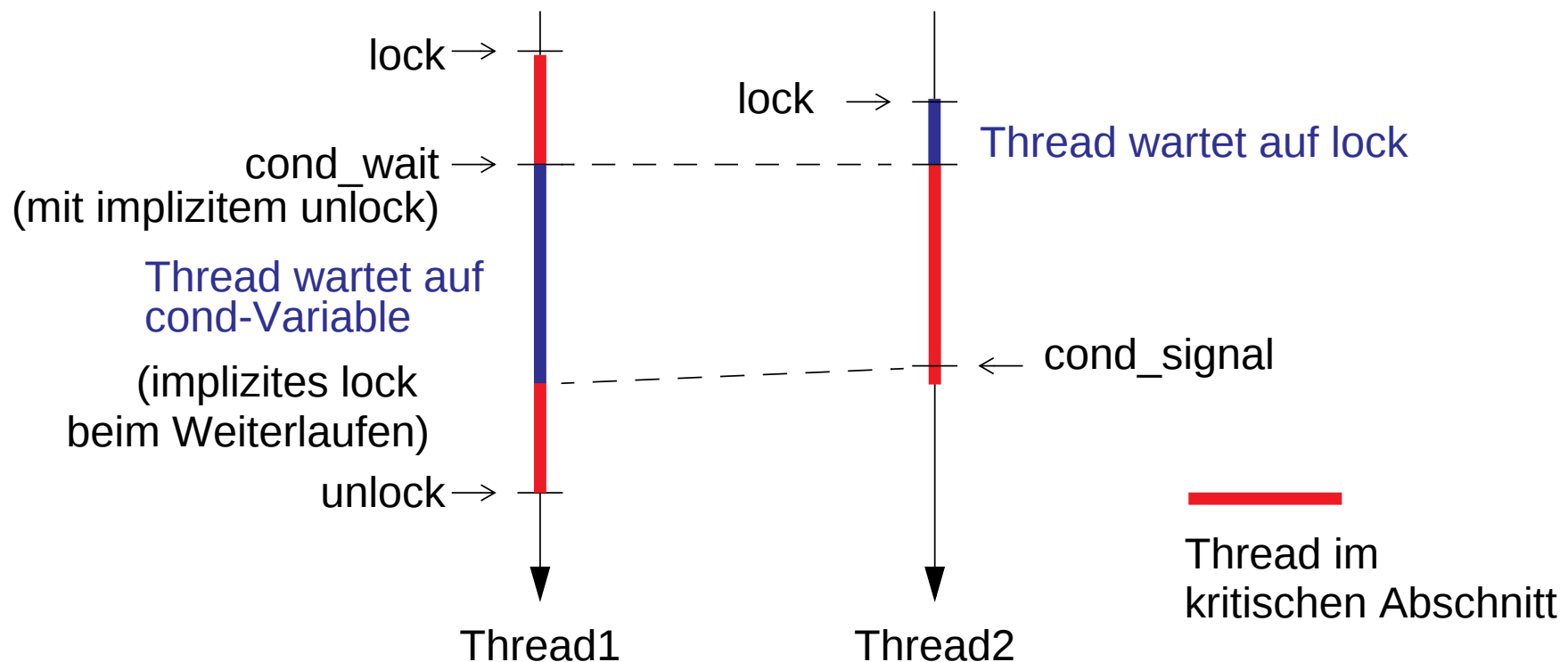
```
static pthread_mutex_t m1;
static int a;

void P() {
    pthread_mutex_lock(&m1);
    while ( a == 0 ) {
        // atomic start
        pthread_mutex_unlock(&m1);
        wait_for_change();
        pthread_mutex_lock(&m1);
        // atomic end
    }
    a--;
    pthread_mutex_unlock(&m1);
}
```

```
void V() {
    pthread_mutex_lock(&m1);
    a++;
    signal_change();
    pthread_mutex_unlock(&m1);
}
```

# U4-4 Bedingungsvariablen - Condition-Variables

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



# U4-4 Bedingungsvariablen

■ ...am Beispiel der Semaphoren Implementierung

```
static pthread_cond_t c1;
static pthread_mutex_t m1;
static int a;

int main () {
    pthread_cond_init(&c1, NULL);
    // ...
}

void P() {
    pthread_mutex_lock(&m1);
    while ( a == 0 ) {
        pthread_cond_wait(&c1, &m1);
    }
    a--;
    pthread_mutex_unlock(&m1);
}
```

```
void V() {
    pthread_mutex_lock(&m1);
    a++;
    pthread_cond_broadcast(&c1);
    pthread_mutex_unlock(&m1);
}
```

# U4-4 Bedingungsvariablen

---

- Realisierung `cond_wait()`
  - ◆ Thread reiht sich in Warteschlange der Bedingungsvariablen ein
  - ◆ Thread gibt Mutex frei
  - ◆ Thread gibt Prozessor auf
  - ◆ Nach Signalisierung wird Thread wieder laufbereit (`cond_signal()` S.U.)
  - ◆ Thread muss kritischen Abschnitt neu betreten (`lock()`)
  
- Realisierung `cond_broadcast()/cond_signal()`
  - ◆ Aufwecken eines (oder mehrerer) Threads, aus der Warteschlange der Bedingungsvariablen
  
- Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden (siehe Seite 8)

## U4-4 Bedingungsvariablen

---

- Bei `pthread_cond_signal()` wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher
  - eventuell Prioritätsverletzung wenn nicht der höchstpriorre gewählt wird
  - Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben
- Mit `pthread_cond_broadcast()` werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt, bleibt der Thread solange blockiert

# U4-5 Aufgabe 8: Webserver

---

- Mehrfädiger Webserver
  - ◆ HTTP 0.9 GET-Anfragen
  - ◆ nur eine Anfrage pro Verbindung
- Hauptfaden nimmt Verbindungen entgegen
- Arbeiterthreads behandeln diese Verbindungen
- Auftragsübergabe über einen Ringpuffer
  - ◆ Hauptfaden (Schreiber) fügt Filedeskriptoren der Verbindungen in den Ringpuffer ein
  - ◆ Arbeiterthreads (Leser) entnehmen Filedeskriptoren aus dem Ringpuffer, bearbeiten die Anfrage und schließen die Verbindung
  - ◆ Danach entnimmt ein freier Arbeiterthread den nächsten Filedeskriptor

# 1 Semaphore-Modul

---

- Synchronisation von Überlauf/Unterlauf des Ringpuffers
- Zählende P/V-Semaphoren zur Synchronisation von POSIX-Threads (siehe Vorlesung ab 11-39)

# 2 Ringpuffer-Modul

---

- Ringpuffer zur Verwaltung von int-Werten
- Randbedingungen: ein Produzent, mehrere Konsumenten
- blockierende Synchronisation bei Überlauf/Unterlauf
- nicht-blockierende Synchronisation der Konsumenten mittels CAS (siehe Vorlesung ab C | X-4, S. 17ff)