E Pthreads

E.1 Grundlagen

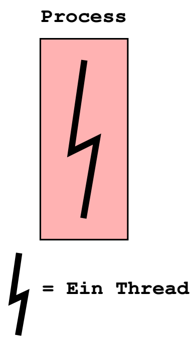
- Der Name steht für POSIX Threads.
  POSIX = Portable Operating System Interface)
- Verabschiedet im Juni 1995 als POSIX Standard 1003.1c
  - ◆ Leider ist der Standard nicht im Netz erreichbar, sondern muß für viel Geld bei IEEE erworben werden.Pthreads sind standardisiert und laufen deshalb unter fast jedem OS.
  - ◆ Pthreads sind unter fast jedem OS verfügbar.
- Bücher zur Vorlesung
  - ◆ “Pthreads Programming”. O’REILLY *Nutshell* Buch
  - ◆ “Threadtime, The Multithreaded Programming Guide”, Prentice Hall

1 Prozeßmodell

- Der einfädige (single-threaded) Prozeß

Program "hello.c"

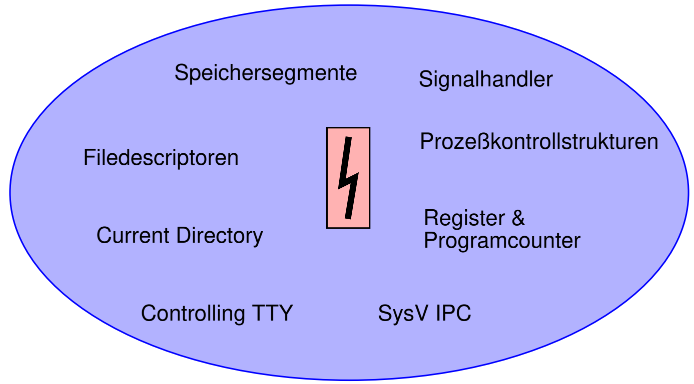
```
#include <stdio.h>
main()
{
 printf("hello PPS\n");
}

$ cc -o hello hello.c
$ ./hello
```



- ◆ Ein Prozeß ist die Instanz eines Programmes in Ausführung.

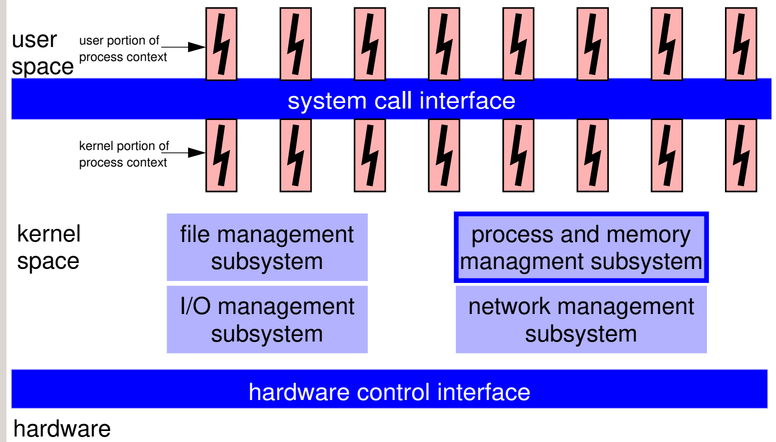
1 Prozeßmodell (2)

- Aufbau eines UNIX Prozesses:



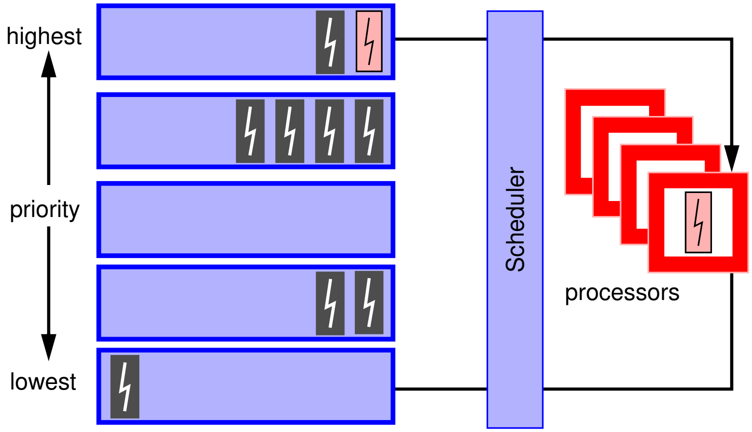
1 Prozeßmodell (3)

- Betriebssystem-Architektur für das Prozeßmodell



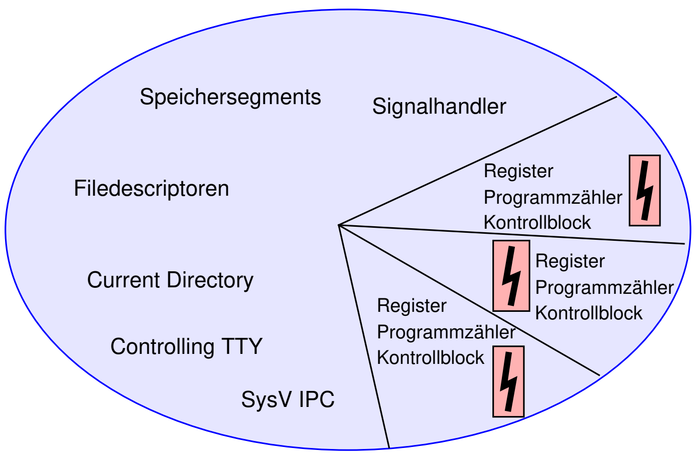
1 Prozeßmodell

- Scheduling



2 Threadmodell

- Aufspaltung eines Prozesses in mehrere Threads

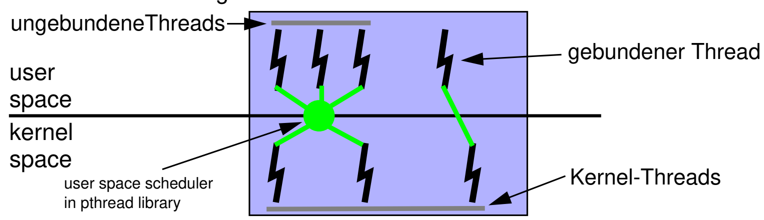


2 Threadmodell (2)

- Der Einsatz von Threads anstelle von Prozessen ist aus folgenden Gründen sinnvoll:
  - Die Prozeßerzeugung durch `fork()` ist sehr zeitaufwendig, da ein neuer Prozeßkontext angelegt werden muß.
  - Ein neuer Thread verbraucht weniger Systemressourcen als ein neuer Prozeß.
  - Der Scheduler kann wesentlich schneller zwischen zwei Threads eines Prozesses umschalten, da die gleichen Prozeßresourcen (insbesondere der gleiche Adreßraum) verwendet werden.
    ➜ Bessere Antwortzeiten (Beispiel: Apache vs. Netscape Web-Server)
  - SysV-IPC Mechanismen sind kompliziert, nicht an Prozesse gebunden (sie bleiben nach Prozeßende bestehen) und langsam (da ein Kernelaufruf getätigt werden muß).

2 Threadmodell (3)

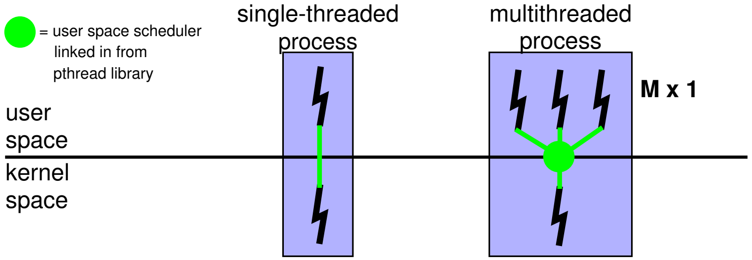
- Gebundene und Ungebundene User-Threads



  - Gebundene User-Threads werden direkt an Kernel-Threads gebunden. Aus Sicht der gebundenen User-Threads stellen Kernel-Threads virtuelle Prozessoren dar. Die Kernel-Threads konkurrieren mit allen Kernel-Threads des Systems um die verfügbaren physikalischen Prozessoren.
  - Ein ungebundener User-Thread kann von jedem Kernel-Thread eines Prozesses ausgeführt werden. Ungebundene User-Threads konkurrieren um die Kernel-Threads des Prozesses. Die Zuweisung erfolgt durch die Thread-Bibliothek (Pthreads Library).

2 Threadmodell (4)

- Many-to-One (M x 1)



Vorteile:

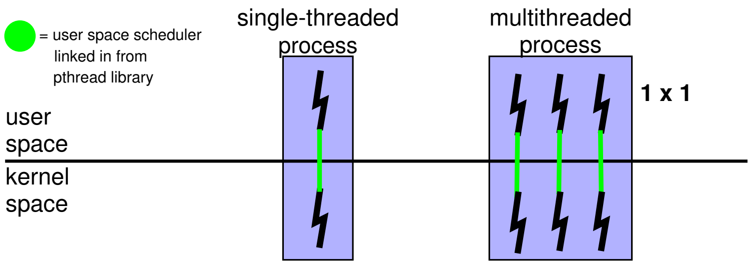
- Schnelle Thread Management Operationen
- Sparsamer Gebrauch von Systemressourcen
- Nebenläufiges Programmiermodell, das ohne Modifikation auch auf mehreren Prozessoren ablaufen kann (1 x 1, M x n Modell)

Nachteile:

- Keine echte Paralellität
- Verklemmung in Systemaufrufen möglich!
- Signalbehandlung ist kritisch
- Profiling einzelner Threads nicht möglich
- Debugging einzelner Threads nicht möglich

2 Threadmodell (5)

- One-to-One (1 x 1)



Vorteile:

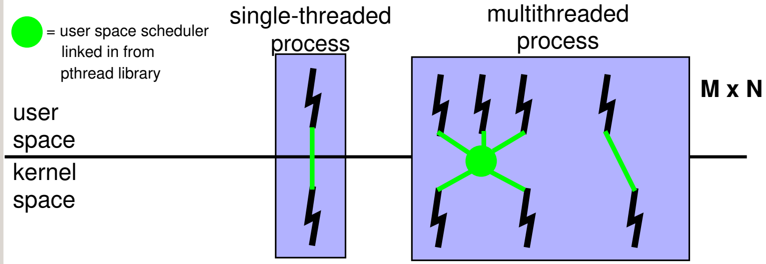
- Echte parallele Ausführung der Threads im Multiprozessor möglich
- Profiling einzelner Threads möglich
- Debugging einzelner Threads möglich

Nachteile:

- Aufwendige Threadverwaltung im OS-Kern
- Hoher Verbrauch an Systemressourcen

2 Threadmodell (6)

- Many-to-Many (M x N)



Vorteile:

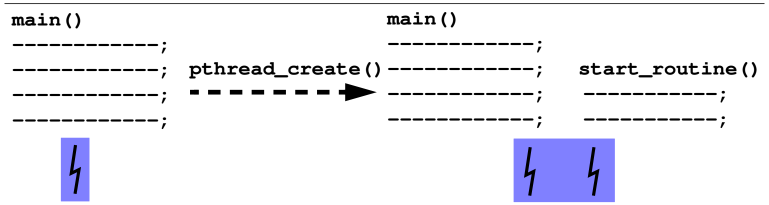
- Flexibles Modell
- Effiziente parallele Ausführung
- Verklemmung an Systemaufrufen kann durch Nachstarten von Kernel-Threads aufgelöst werden.

Nachteile:

- Scheduling auf zwei Ebenen (User- und Kernel-Ebene) steigert die Komplexität.
- Profiling und Debugging ungebundener Threads ist kritisch.

E.2 Basic Pthread Management

1 Erzeugung



◆ Ein neuer Thread kann durch

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

erzeugt werden. Der erzeugte Thread führt als erstes die Funktion `start_routine(arg)` auf. `thread` kann vom Vaterthread als *Handle* auf den erzeugten Thread verwendet werden.

2 Thread-Attribute

◆ Über das `attr` Argument können die Attribute des erzeugten Threads modifiziert werden. Ist `attr == NULL`, werden die Standardattribute gesetzt.

- Initialisieren des Attribut-Objekts:

```
ret_val = pthread_attr_init (&attr);
```

- Setzen der Stack-Größe

```
ret_val = pthread_attr_setstacksize(&attr,stacksize);
```

- Lesen der Stack-Größe

```
ret_val = pthread_attr_getstacksize(&attr,&stacksize)
```

◆ Es gibt noch weiter Parameter (Stackadresse, Joinable/Detached, ...).
➜ **man pthread_create**

3 Terminierung

■ Selbst-Terminierung:

◆ Ein Thread kann sich durch `void pthread_exit(void *retval)` beenden. Dies geschieht automatisch, wenn der Thread aus der Funktion `start_routine` zurückspringt.

■ Warten auf die Terminierung eines Thread

◆ Mit der Funktion
`pthread_join(pthread_t thread, void **retvalp)`
kann der Vaterthread auf die Beendigung eines erzeugten Threads warten. `thread` ist dabei der durch den Aufruf von `pthread_create` erzeugte Handle.
Über `retvalp` wird der Exitstatus des Threads zurückgeliefert. Interessiert der Status nicht, kann als `retvalp` auch `NULL` übergeben werden.

4 Rückgabewerte

◆ Die Pthread Funktionen weichen von ihrer Aufrufsemantik von den Standard Kernaufrufen ab. Zwar wird bei einem erfolgreichen Aufruf auch `0` zurückgeliefert, im Fehlerfall wird aber auf die Verwendung von `errno` verzichtet und der Fehlercode direkt zurückgeliefert.

◆ Die anderen Systemaufrufe setzen immer noch die `errno` Variable. Die Fehlerwerte der Pthreads Funktionen lassen sich über `perror()` oder `strerror()` als Fehlermeldung ausgeben.

◆ Da `errno` für jeden Thread verschieden sein muß, ist es normalerweise keine Variable mehr, sondern wird über ein `#define` auf einen Funktionsaufruf abgebildet.

■ Setzen des Parallelitätsgrades in SUN Solaris 2.5/6

◆ Unter Solaris muß der Parallelitätsgrad (Anzahl der Kernelthreads im M x N Modell) über `thr_setconcurrency(n)` konfiguriert werden.

◆ Das Defaultverhalten ist, daß nur **ein** Kernelthread läuft.

5 Beispiel zur Thread-Erzeugung

■ Beispiel 1: Multiplikation Matrix mit Vektor

```
#include <stdio.h>
#include <pthread.h>
double a[100][100], b[100], c[100];

main() {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,(void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}
```

5 Beispiel zur Thread-Erzeugung (2)

◆ Multiplikationsfunktion **mult()**

```
void *mult(void *cp){
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
    sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```


5 Beispiel zur Thread-Erzeugung (3)

■ Ergebnisse: 1000x1000 Matrix, `mult` *k*-fach wiederholt

k	Anzahl der Prozessoren (*nproc*)				
	–	1	2	3	4
1	0.6	1.1	0.8	0.8	0.8
2	0.9	1.3	0.9	0.8	0.8
10	2.9	3.4	2.0	1.4	1.2
100	26.3	26.7	13.3	9.2	7.1

■ => Laufzeit setzt sich ungefähr folgendermaßen zusammen:
- 0.3s Initialisierung,
- 0.4s Starten und Beenden von 1000 Threads,
- (*k* x 0.26s) /*nproc* reine Rechenzeit.

6 Identifikation

■ Ein Thread kann seinen Thread-Handle über `pthread_t pthread_self(void)` erfragen.

■ Es ist nicht garantiert, daß man Thread-Handles mit == auf Gleichheit überprüfen kann.
➜ Die Funktion `pthread_equal(pthread_t t1, pthread_t t2)` liefert `TRUE` zurück, wenn die beiden Threads identisch sind.

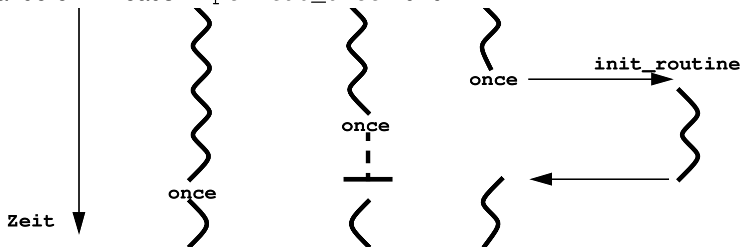
7 Initialisierungen

■ Initialisierungen dürfen häufig nur genau einmal durchgeführt werden.

■ Die Pthread Bibliothek erleichtert diese Aufgabenstellung mit der `pthread_once()` Funktion.

■ Die übergebene Funktion wird genau einmal vom dem Thread ausgeführt, der das erste Mal `pthread_once` aufruft.

■ Arbeitet gerade ein Thread die Initialisierungsfunktion ab, blockieren alle anderen Threads im `pthread_once` Aufruf.



7 Initialisierungen (2)

- Syntax:
```
pthread_once(pthread_once_t *once_block,
             void (*init_routine)(void))
```

- Der `once_block` wird von der Pthreads Bibliothek zur Speicherung von Statusinformation benötigt. Er muß *statisch* mit `PTHREAD_ONCE_INIT` initialisiert werden.

- Leider kann der Initialisierungsfunktion kein Argument mitgegeben werden.

8 Threadlokale globale Variablen

- Threadlokale Variablen sind normalerweise nur Variablen, die auf dem Programmstack abgelegt wurden.

- Es ist manchmal wünschenswert, threadspezifische Variablen zu verwenden, die unabhängig von der Aufrufhierarchie (Stack) sind.

- Threadlokale Variablen werden durch globale *Keys* implementiert. Jedem Key kann ein Wert zugewiesen werden, der threadlokal gespeichert wird.
  - ◆ Ein Key kann mit der Funktion
```
pthread_key_create(pthread_key_t *key,
                   void (*destructor)(void *))
```
    angelegt werden. Dieser Aufruf bewirkt folgendes:
    - Das System legt Speicherplatz für den Wert des Keys an.
    - Außerdem kann dem System eine Funktion mitgeteilt werden, die beim Überschreiben und Löschen eines Wertes aufgerufen werden soll. Hierdurch lassen sich auch komplizierte Strukturen als Wert speichern.

8 Threadlokale globale Variablen (2)

- ◆ Soll ein Wert an einen Key zugewiesen werden, ist
```
pthread_setspecific(pthread_key_t key,
                    void *value)
```
  aufzurufen. Hierbei wird der Wert `value` unter dem Key `key` gespeichert.
- ◆ Der gespeicherte Wert kann mit
```
void *pthread_getspecific(pthread_key_t key)
```
  abgefragt werden.
- ◆ Wird ein globaler Key nicht mehr benötigt, sollte
```
pthread_key_delete(pthread_t key)
```
  aufgerufen werden.

E.3 Synchronisation

- Da die einzelnen Threads gleichzeitig ablaufen, kann es zu ungewollten Effekten beim Zugriff auf den Speichern kommen (sogenannte *Race Conditions*).

Beispiel: Erhöhung eines Zählers

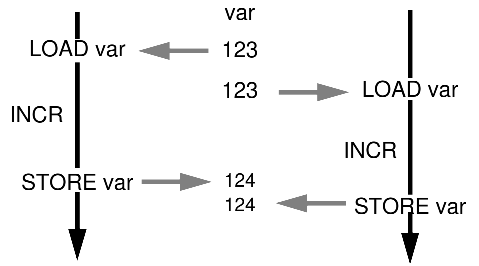
```
int var;
void *incr(void *arg) {
  int i;
  for (i = 0; i < 10000; i++)
    var++;
  return NULL;
}
.......
pthread_create(&son1, NULL, incr, NULL);
pthread_create(&son2, NULL, incr, NULL);
pthread_join(son1, NULL);
pthread_join(son2, NULL);
printf("Result: %d\n", var);
```

E.3 Synchronisation (2)

- Dreimaliges Aufrufen des Programms auf einer SUN ergab folgende Werte:

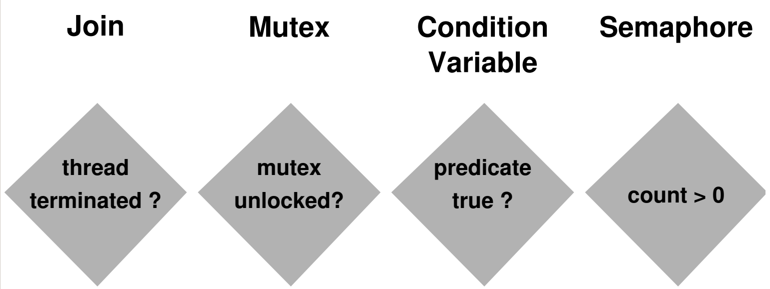
  Result: 13950
  Result: 13881
  Result: 13638

  Ursache für dieses Verhalten: Die `var++` Anweisung besteht aus einer Lese- und einer Schreiboperation. Versucht der zweite Thread zwischen diesen beiden Operationen, den Zähler zu erhöhen, geht ein `var++` verloren:



E.3 Synchronisation (3)

1 Mechanismen zur Thread-Koordination

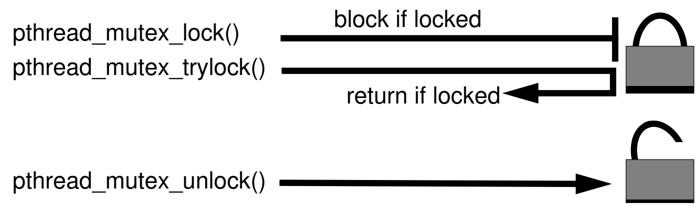


2 Mutex

- Programmabschnitte, in denen sich zu einem Zeitpunkt nur ein Thread befinden darf, werden als *critical Section* bezeichnet.
  Beispiel: Erhöhung eines Zählers

- Der Pthread Standard bietet die Möglichkeit, kritsche Abschnitte durch einen *Mutex* zu sperren.
  In einem gesperrten Bereich kann sich höchstens ein Thread aufhalten. Will zusätzlich ein zweiter Thread den Bereich betreten, muß dieser solange warten, bis der erste Thread den Bereich “entriegelt” hat.

- Ein Mutex ist also mit einem Türschloß vergleichbar.



2 Mutex (2)

- Erzeugen eines Mutexes:
  - Beim Kompilieren (statisch), durch
    `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
  - Beim Programmlauf (dynamisch), durch
    ```
    pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr)
    ```
    `mutex` muß auf einen gültigen Speicherbereich zeigen.
  - Beim dynamischen Aufruf können wie bei der Erzeugung von Threads Attribute gesetzt werden. Für einen Standardmutex kann dieses Argument `NULL` sein.

- Vernichten eines Mutexes:
  - Wird ein Mutex nicht mehr benötigt, kann dies dem System mit `pthread_mutex_destroy()` mitgeteilt werden.
    Die Pthread Bibliothek kann dann die zugeordneten Resourcen freigeben.

2 Mutex (3)

- Ein Mutex kann durch
  `pthread_mutex_lock(pthread_mutex_t *mutex)`
  verriegelt werden. Nach diesem Aufruf ist sichergestellt, daß der aktuelle Thread als einziges den Mutex belegt hat.

- Nach dem kritischem Codeabschnitt kann mit
  `pthread_mutex_unlock(pthread_mutex_t *mutex)`
  der Mutex entriegelt werden.

- Soll nicht die erfolgreiche Verriegelung abgegewartet werden, kann
  `pthread_mutex_trylock(pthread_mutex_t *mutex)`
  verwendet werden. Diese Funktion liefert einen Fehler (**`EBUSY`**) zurück, wenn der Mutex durch einen anderen Thread belegt ist.

2 Mutex (4)

- Beispiel:

```
int var;
pthread_mutex_t varlock = PTHREAD_MUTEX_INITIALIZER;
...
    for (i = 0; i < 10000; i++) {
        pthread_mutex_lock(&varlock);
        var++;
        pthread_mutex_unlock(&varlock);
    }
```


3 Condition-Variablen

- Häufig muß ein Programm auf das Eintreffen bestimmter Ereignisse warten. Dieses Warten sollte nicht aktiv geschehen (*polling*), da sonst wertvolle Rechenzeit ungenutzt bleibt.

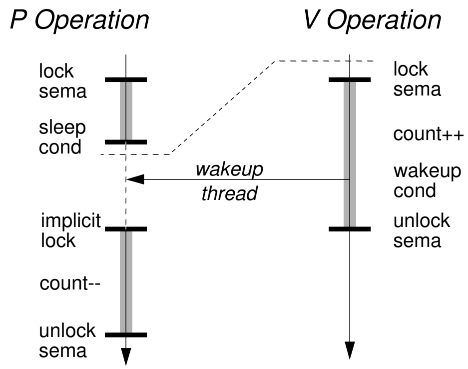
- Stattdessen sollte es eine Möglichkeit geben, daß sich Threads “schlafen legen” (*sleep*). Wenn ein anderer Thread feststellt, daß das gewünschte Ereignis eingetroffen ist, kann der schlafende Thread geweckt werden (*wakeup*).

- Normalerweise bezieht sich das Ereignis auf eine Struktur, die durch ein Mutex gelockt werden muß. Deshalb muß der Sleep-Aufruf den Mutex automatisch freigeben. Nach dem Wakeup muß dann ein implizites Lock durchgeführt werden.

- Im Prinzip ist die Idee der Condition-Variablen identisch mit dem `sleep()/wakeup()` Mechanismus, der im UNIX Kern verwendet wird. Allerdings wird dort anstelle der Mutexe ein `splhi()` Aufruf verwendet, der die Interrupts sperrt.

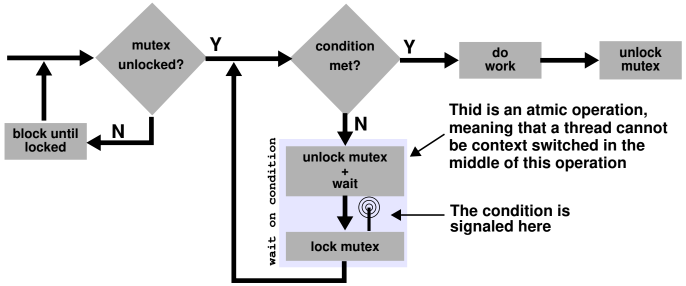
3 Condition-Variablen (2)

- Beispiel: P/V Semaphore.
  Die kritische Struktur ist der Zähler. Das Ereignis, auf das gewartet werden muß, ist “count > 0”.



3 Condition-Variablen (3)

- Wenn mehr als ein Thread eine P-Operation durchführen kann, muß nach jedem *Wakeup* die Bedingung neu getestet werden und gegebenenfalls wieder ein *Sleep* durchgeführt werden.




```
lock associated mutex
while (predicate is not true)
       wait on condition variable
do work
unlock associated mutex
```


3 Condition-Variablen (4)

- Erzeugen einer Condition-Variablen
  - ◆ Eine Condition-Variable wird ähnlich wie ein Mutex erzeugt:
  - ◆ Statisch durch

    `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  - ◆ Dynamisch mit

    ```
    pthread_cond_init(pthread_cond_t *cond,
                const pthread_condattr_t *attr)
    ```
  - ◆ Wie bei der Mutexerzeugung können wieder Attribute angegeben werden. Sollen die Standardattribute verwendet werden, kann `NULL` verwendet werden.
- Vernichten einer Condition-Variablen
  - ◆ Mit der Funktion

    `pthread_cond_destroy(pthread_cond_t *cond)`

    kann dem System mitgeteilt werden, daß diese Condition-Variable nicht mehr verwendet wird.

3 Condition-Variablen (5)

- Synchronisation (1):
  - ◆ Mit

    ```
    pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex)
    ```

    kann sich ein Thread auf der Condition-Variablen `cond` blockieren. `mutex` gibt hierbei den Mutex an, der implizit freigegeben werden soll.
  - ◆ Von diesem Aufruf gibt es auch eine Version mit Zeitbegrenzung: Mit

    ```
    pthread_cond_timedwait(pthread_cond_t *cond,
                  pthread_mutex_t *mutex,
                  const struct timespec *abstime)
    ```

    kann ein Zeitpunkt angegeben werden, zu dem der Aufruf mit einem Fehler (**`ETIMEDOUT`**) abgebrochen wird.

3 Condition-Variablen (6)

- Synchronisation (2):
  - ◆ Das Eintreffen eines Ereignisses kann durch

    `pthread_cond_signal(pthread_cond_t *cond)`

    signalisiert werden. Das System weckt hierbei höchstens einen Thread auf (abhängig von der Schedulingpriorität).
  - ◆ Sollen alle Threads aufgeweckt werden, die sich an der Condition-Variablen blockiert haben, ist dies mit

    `pthread_cond_broadcast(pthread_cond_t *cond)`

    möglich.
  - ◆ Beispiel in Pseudo-Code:

    ```
    lock associated mutex
    set predicate to true
    signal condition variable (wake-up one or all)
    unlock associated mutex
    ```

3 Condition-Variablen (7)

■ Beispiel

◆ s soll eine P/V Semaphore implementiert werden.

```
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;
} mysem_t;
```

◆ Initialisierung:

```
void mysem_init(mysem_t *ms, int init) {
    pthread_mutex_init(&ms->mutex, NULL);
    pthread_cond_init(&ms->cond, NULL);
    ms->count = init;
}
```

```
Resourcenfreigabe
    void mysem_destroy(mysem_t *ms) {
        pthread_mutex_destroy(&ms->mutex);
        pthread_cond_destroy(&ms->cond);
    }
```


3 Condition-Variablen (8)

◆ P Operation:

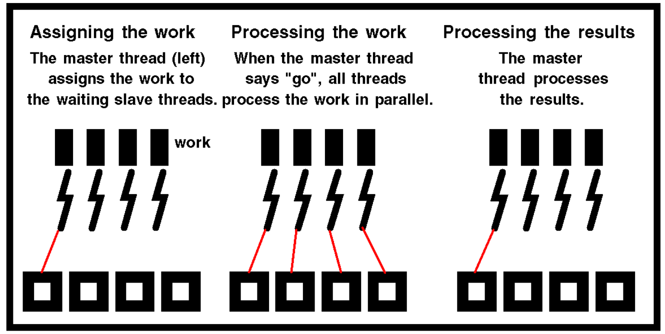
```
void mysem_p(mysem_t *ms) {
    pthread_mutex_lock(&ms->mutex);
    while (ms->count <= 0) {
    pthread_cond_wait(&ms->cond,
                      &ms->mutex);
    }
    ms->count--;
    pthread_mutex_unlock(&ms->mutex);
}
```

◆ V Operation:

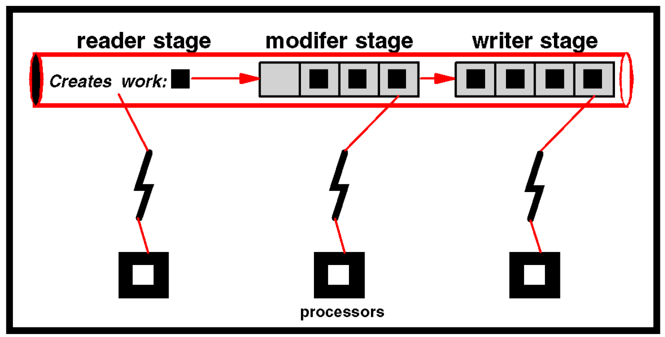
```
void mysem_v(mysem_t *ms) {
    pthread_mutex_lock(&ms->mutex);
    ms->count++;
    pthread_cond_signal(&ms->cond);
    pthread_mutex_unlock(&ms->mutex);
}
```


E.4 Parallelisierungsmodelle

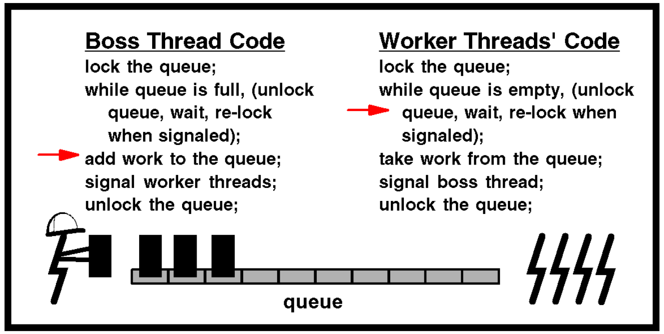
1 Master-Slave Modell



2 Pipeline Modell



3 Boss-Worker Modell



E.5 Attribute

- Bei einigen Pthread Aufrufen (`pthread_create`, `pthread_mutex_init`, `pthread_cond_init`) kann eine Attributstruktur als Parameter spezifiziert werden, mit dem das neue Objekt konfiguriert wird.
- Das Erzeugen/Setzen/Abfragen von Attributen geschieht immer nach dem gleichen Schema:
  - ◆ Initialisierung der Attributstruktur.
  - ◆ Abfragen oder Setzen der gewünschten Werte.
  - ◆ Erzeugung des neuen Objektes unter Benutzung eines Zeiger auf die Attributstruktur als Parameter.
  - ◆ Gegebenenfalls Freigabe der Attributstruktur.

E.5 Attribute (2)

- Initialisieren der Attributstruktur:
```
pthread_XXXattr_init(pthread_XXXattr_t *attr)
```
- Auslesen eines einzelnes Attributs:
```
pthread_XXXattr_getATTR(pthread_XXXattr_t *attr,
                        ATTRTYP *pointer)
```
- Setzen eines einzelnes Attributs:
```
pthread_XXXattr_setATTR(pthread_XXXattr_t *attr,
                        ATTRTYP value)
```
- Freigeben eines einzelnes Attributs:
```
pthread_XXXattr_destroy(pthread_XXXattr_t *attr)
```
- Attributierbare Objekte: `XXX`:
  - Threads: `XXX` = “” (Leerstring)
  - Mutex: `XXX` = “`mutex`”
  - Conditional: `XXX` = “`cond`”

E.6 Thread Scheduling

1 Festlegung des Thread-Modells (1 x 1, M x 1, M x N)

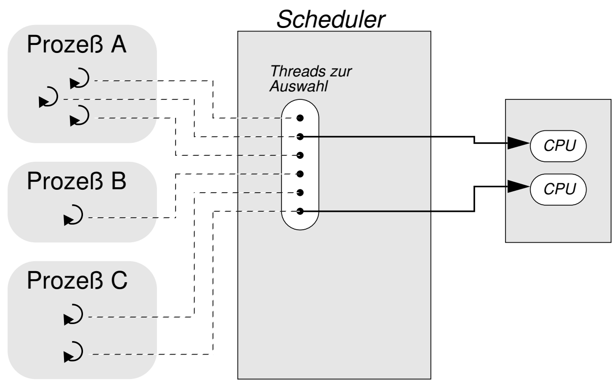
- Das Modell für die Bindung an die Kernel-Threads wird durch den Scheduling-Scope festegelegt. Der gewünschte Scope für einen Thread kann durch ein Threadattribut der Funktion `pthread_create` mitgeteilt werden:

  ATTR=`scope`, ATTRTYP=`int`

  - ◆ `PTHREAD_SCOPE_SYSTEM`: Die Schedulingstrategie soll für alle laufenden Threads des Systems (also aus allen Prozessen) gelten.
    Es handelt sich hier um einen gebundenen Thread (1 x 1 Modell).
  - ◆ `PTHREAD_SCOPE_PROCESS`: Die Schedulingstrategie soll nur für die Threads eines Prozesses gelten. Zwischen den Prozessen wird mit dem standard Systemscheduler umgeschaltet.
    Es handelt sich hier um einen ungebundenen Thread (M x 1, M x N Modell)

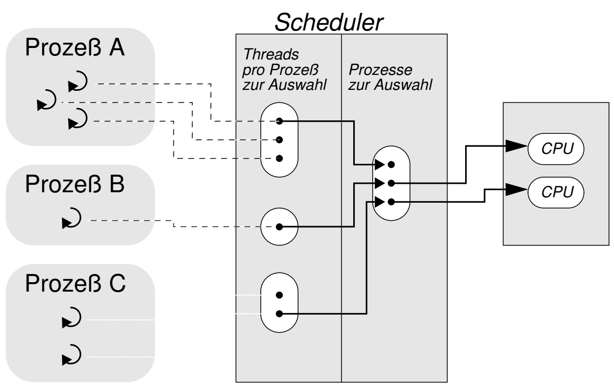
1 Scheduling Scope (2)

- Beispiel für System-Scope



1 Scheduling Scope (3)

- Beispiel für Prozeß-Scope (M x 1)



2 Scheduling Strategie

- Die gewünschte Scheduling Strategie für einen Thread kann durch ein Threadattribut der Funktion `pthread_create` mitgeteilt werden:

  ATTR=`schedpolicy`, ATTRTYP=`int`

  - `SCHED_FIFO`: Ein Thread läuft bis er stirbt oder sich blockiert. Wenn er, nachdem er sich blockiert hat, weiterlaufen kann, wird er am Ende der Queue für seine Priorität eingehängt.
    Dieser Attributwert ist nur für gebundene Threads anwendbar und für Echtzeitanwendungen vorgesehen (In Solaris nicht implementiert).
  - `SCHED_RR`: Ähnlich wie `SCHED_FIFO`, allerdings wird der Thread auf jeden Fall nach Ablauf eines festen Quantums unterbrochen.
    Dieser Attributwert ist nur für gebundene Threads anwendbar und für Echtzeitanwendungen vorgesehen (In Solaris nicht implementiert).
  - `SCHED_OTHER`: Eine beliebige Strategie. Dies ist die Standardeinstellung!
    Dieser Attributwert ist nur für Threads im Time-Sharing Betrieb anwendbar.

3 Threadpriorität

- Die gewünschte Priorität eines **ungebundenen Threads** kann durch ein Attribut der Funktion `pthread_create` mitgeteilt werden:
  ATTR=`schedparam`, ATTRTYP=`struct sched_param`
  Die Struktur wird auch beim `set` Aufruf als Pointer übergeben!
  Die verwendeten Parameter sind abhängig von der eingestellten Schedulingstrategie. POSIX.1b erzwingt genau ein Attribut für `sched_param`:

```
    struct sched_param {
        int sched_priority;
    };
```

  Um den Bereich abzufragen, in dem `sched_priority` gültig ist, können die POSIX.1b Aufrufe

```
sched_get_priority_max(int schedpolicy)
sched_get_priority_min(int schedpolicy)
```

  verwendet werden.

- Die Priorität von **gebundenen Threads** kann durch einen Systemaufruf `priocntl()` geändert werden.

4 Scheduling Parameter

- Die Schedulingparameter von laufenden ungebundenen Threads können nachträglich angepaßt werden. Dies geschieht über

```
pthread_getschedparam(pthread_t thread,
                      int *policy,
                      struct sched_param *param)
```

  und

```
pthread_setschedparam(pthread_t thread,
                      int policy,
                      struct sched_param *param)
```

- `policy` und `param` entsprechen den Parametern, wie sie bei den Threadattributen werden (policy: `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`, param: `sched_priority`).

5 Vererbung von Scheduling Parametern

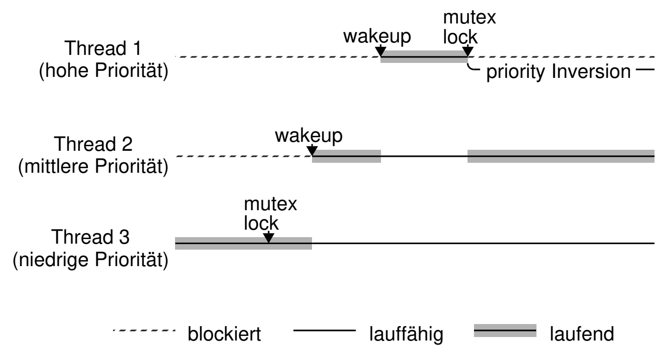
- Ein Attribut kann festlegen, ob die Schedulingparameter aus der Attributstruktur übernommen werden sollen, oder ob diese ignoriert werden und die Parameter vom Vaterthread kopiert werden sollen.
  ATTR=`inheritsched`, ATTRTYP=`int`
  - ◆ `PTHREAD_INHERIT_SCHED`: Kopiere die Parameter vom Vaterthread
  - ◆ `PTHREAD_EXPLICIT_SCHED`: Übernehme die Parameter aus dem Prozeßattributobjekt.

6 Priority Inversion

- Über die Mutexattribute lassen sich Strategien zur Vermeidung von *Priority Inversion* konfigurieren. Außerdem ist einstellbar, daß Mutexe auch über Prozeßgrenzen hinweg funktionieren sollen.
- Priority Inversion bedeutet, daß ein Thread mit hoher Priorität nicht arbeiten kann, weil ein anderer mit niedriger Priorität eine Resource hält.
- Dies kann in Realtime-Anwendungen zu drastischen Problemen führen.

6 Priority Inversion (2)

- Beispiel:



6 Priority Inversion (2)

- Strategie 1: Priority Ceiling
  - Hier wird jedem Mutex eine eigene Priorität zugeordnet. Wenn ein Thread einen kritischen Abschnitt betritt und einen Mutex lockt, bekommt er automatisch die Priorität zugeteilt. Beim Unlock des Mutex wird seine alte Priorität restauriert.
  - Diese Strategie ist verwendbar, wenn die Konstante `_POSIX_THREAD_PRIO_PROTECT` definiert ist.
  - Priority Ceiling wird gewählt, indem das Attribut ATTR=`protocol`, ATTRTYPE=`int` in den Mutexattributen auf `PTHREAD_PRIO_PROTECT` gesetzt wird.
    Der Attributwert `PTHREAD_PRIO_NONE` schaltet Priority Ceiling wieder ab.
  - Die Priorität des Mutexes sollte so hoch gewählt werden, wie das Maximum der Prioritäten aller Threads, die den Mutex belegen können.
  - Die Priorität des Mutexes ist mit dem Attribut ATTR=`prioceiling`, ATTRTYPE=`int` einstellbar.

6 Priority Inversion (3)

- Strategie 2: Priority Inheritance
  - Hier wird die Priorität des Threads, der den Mutex belegt hat, dynamisch so verändert, daß sie immer so groß ist wie das Maximum der Prioritäten aller Threads, die auf den Mutex warten.
  - Diese Strategie ist verwendbar, wenn die Konstante `_POSIX_THREAD_PRIO_INHERIT` definiert ist.
  - Priority Inheritance wird gewählt, indem das Attribut ATTR=`protocol`, ATTRTYPE=`int` in den Mutexattributen auf `PTHREAD_PRIO_INHERIT` gesetzt wird.

7 Entkopplung von Threads (detached threads)

- Mit `pthread_detach(pthread_t thread)` kann der Pthread-Bibliothek mitteilen werden, daß der Exitstatus des Threads `thread` nicht gespeichert werden soll.
  ➜ Automatische Freigabe alle Thread-Ressourcen.
  ➜ Ein `pthread_join` ist nicht mehr möglich.
  ➜Die ID eines Thread kann nach seiner Terminierung für einen neuen Thread wiederverwendet werden! Falls unklar ist,ob ein entkoppelter Thread schon terminiert ist, darf seine Handle nicht mehr verwendet werden.
- Die Entkopplung kann auch direkt über ein Attribut beim Aufruf von `pthreat_create` geschehen.
  - ATTR=`detachstate`, ATTRTYP=`int`
    - `PTHREAD_CREATE_JOINABLE`: Auf diesen Thread kann ein `join()` durchgeführt werden, d.h., das System muß sich den “Exitstatus” merken.
    - `PTHREAD_CREATE_DETACHED`: Dieser Thread soll völlig unabhängig vom Vaterthread laufen.

7 Entkopplung von Threads (2)

- Soll ein Server für jede Anfrage einen neuen Thread starten (ala *inetd*), ist es nicht sinnvoll, daß der erzeugt Thread noch Verbindung zu seinem Vaterthread hat.
- Beispiel – Server Applikation:

```
main() {
    ...
    for (;;) {
        pthread_t worker;
        req = malloc(sizeof *req);
        wait_for_request(&req);
        pthread_create(&worker, NULL,
                       process, req));
        pthread_detach(worker);
    }
}
```

E.7 UNIX Interaktion

- UNIX kennt traditionell nur Prozesse als Aktivitätsträger, die alle einen eigenen Adreßraum besitzen. Daraus resultieren eine Reihe von Problemen bezüglich Threads:
  - ◆ Können sich mehrere Threads gleichzeitig in einer Bibliotheksfunktion befinden?
  - ◆ Was passiert, wenn eine Cancellation geschieht, während der Thread eine Bibliotheksfunktion abarbeitet?
  - ◆ UNIX Signale: Wenn ein Prozeß ein Signal empfängt, welche Folgen entstehen für die Threads?
  - ◆ Was passiert bei *fork()*, *exec()* und *exit()*?
  - ◆ Gibt es eine Synchronisation über Prozeßgrenzen hinweg?
  - ◆ Was passiert, wenn eine Funktion einen Systemaufruf tätigt, der sich blockiert?

1 Bibliotheksaufrufe

- Durch die Verwendung von Threads kann es passieren, daß sich mehrere Aktivitätsträger gleichzeitig in einer Funktion befinden. Wenn die Funktion Datenstrukturen manipuliert, kann es zu Fehlern kommen, wenn diese nicht mit Mutexen oder anderen Locks versehen wurden.
- Funktionen, die so gesichert wurden, nennt man (*multi-*) *thread-safe* (*MT-safe*) oder *reentrant*.
- Alle POSIX-Funktionen sind MT-safe, bis auf zwei Klassen von Ausnahmen:
  - ◆ Funktionen, deren Spezifikation dies nicht zuläßt.
  - ◆ Funktionen, die aus Geschwindigkeitsgründen nicht mit Locks versehen wurden.

1 Bibliotheksaufrufe (2)

- Funktionen, die von ihrer Spezifikation nicht threadsafe gemacht werden können, sind solche, die Daten in einen statischen Puffer hinterlegen und häufig einen Zeiger auf sie zurückliefern. Dies sind zum Beispiel `rand()`, `getpw*()`, `ctime()`, `strtok()`...
- Für viele dieser Funktionen spezifiziert der Pthread Standard ein zweites Interface, bei dem der Puffer über zusätzliche Parameter übergeben werden kann. Diese Funktionen enden alle auf `_r` (wie reentrant).
- Beispiel:

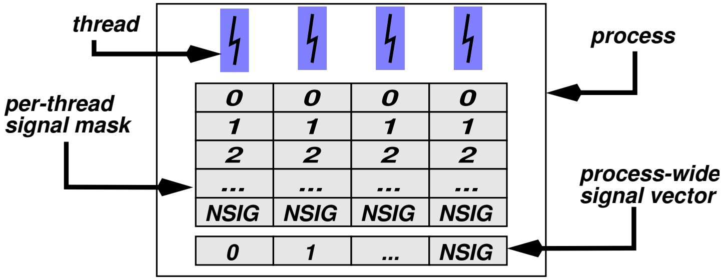
```
char *ctime_r(const time_t *clock,
              char *buf, int buflen)
```


1 Bibliotheksaufrufe (3)

- Die Standard-IO Funktionen `getc/getchar/putc/putchar` wurden zwar threadsafe gemacht, es existiert aber noch eine Variante, bei der auf das Lock verzichtet wird.
- Diese Funktionen enden auf `_unlocked`. Sie eignen sich vor allem dann, wenn sie für eine Zeitlang häufig hintereinander aufgerufen werden. In diesem Fall ist es sinnvoll, vor dem ersten Aufruf das `FILE` zu locken, die Aufrufe durchzuführen, und danach ein Unlock aufzurufen.
- Die `FILE`-Struktur kann mit `flockfile(FILE *stream)`, `funlockfile(FILE *stream)` und (nicht blockierend) mit `ftrylockfile(FILE *stream)` gesperrt und freigegeben werden.

2 Signale

- Um kompatibel zu UNIX zu bleiben, spezifizierten die Pthread Entwickler, daß Signale sich auf ganze Prozesse beziehen. Ein Signalhandler gilt also für alle Threads, es kann der standard POSIX Aufruf *sigaction()* verwendet werden.
- Allerdings erlaubt der Pthread Standard jedem Thread, Signale auszumaskieren. Dies erlaubt den Benutzer, die Aufgaben des Signalhandlings an einzelne Threads zu verteilen.



2 Signale (2)

- Die Art, Signale zu maskieren, entspricht der Technik, die POSIX für Prozesse spezifiziert hat:
  - ◆ Es gibt eine Signalmaske die alle Signale enthält.
  - ◆ Diese kann gelesen/verändert/gesetzt werden.
- Setzen der Threadsignalmaske (analog zu `sigprocmask`):

```
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset)
```

- Wenn `oset` gesetzt ist, wird hier die alte Maske hinterlegt.
- Ist `set` gesetzt, wird die aktuelle Maske verändert. `how` gibt an, in welcher Weise dies geschehen soll:
  - ◆ `SIG_BLOCK`: die Maske wird zu der aktuellen Maske hinzugefügt.
  - ◆ `SIG_UNBLOCK`: die Maske wird von der aktuellen entfernt.
  - ◆ `SIG_SETMASK`: die Maske wird übernommen.

2 Signale (3)

- Beim Eintreffen eines Signals wählt das System einen Thread aus, der den Signalhandler aufruft.
  - ◆ Wenn das Signal eindeutig einem Thread zugeordnet werden kann (z.B. bei Exceptions (`SEGV`...)), wird dieser Thread verwendet.
  - ◆ Andernfalls wird ein beliebiger Thread ausgewählt, der das Signal nicht blockiert hat.
- Es ist auch möglich, ein Signal direkt an einen Thread des gleichen Prozesses zu schicken:
  `pthread_kill(pthread_t thread, int sig)`
  `thread` ist durch `pthread_create` oder `pthread_self` erhaltene Threadkontrollblock.

2 Signale (4)

- Für einen Signalhandler gelten einige Einschränkungen: Er darf nur Funktionen aufrufen, die *asynchronous signal-safe* sind. Das bedeutet, daß die Funktion gefahrlos unterbrochen werden kann, ohne daß Inkonsistenzen oder Deadlocks eintreten. Async signal-safe ist somit eine Einschränkung von MT-safe.
- Der Standard führt eine Reihe von Funktionen auf, die signal-safe sein müssen. Dies sind praktisch nur Kernaufrufe.
- *Keine* der Pthread Funktionen muß signal-safe sein! Das bedeutet, daß in einem Signalhandler keine Mutexe oder ähnliches verwendet werden dürfen.

2 Signale (5)

- Diese Einschränkung ist nicht ganz so schlimm, wie sie sich zuerst anhört. Man löst das Problem dadurch, indem man einen Thread startet, der nur für die Annahme des gewünschten Signals zuständig ist. Alle Threads maskieren dieses Signal aus.

- Dieser Thread ruft die POSIX Standardfunktion `sigwait()` auf, die solange wartet, bis ein Signal eintrifft, und dann zurückkehrt:
  `int sigwait(const sigset_t *set, int *sig)`
  `set` gibt an, auf welche Signale gewartet werden soll. In `sig` wird bei Eintreffen die Signalnummer gespeichert.

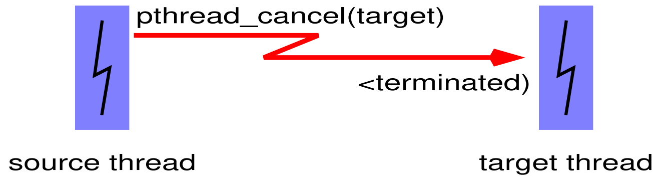
2 Signale (6)

- Beispiel: `SIGCHLD` Handler

```
sigset_t chldset;
pthread_t chldthr;
...
void *chldhand(void *arg) {
    int sig;
    for (;;) {
    sigwait(&chldset, &sig);
    waitpid()
    ...
}
...
    sigemptyset(&chldset);
    sigaddset(&chldset, SIGCHLD);
    pthread_sigmask(SIG_BLOCK, &chldset, NULL);
    pthread_create(&chldthr, NULL, chldhand, NULL);
```


3 Abbruch eines Threads

- Manche Probleme erfordern die Möglichkeit, laufende Threads abzubrechen (sogenannte *Thread Cancellation*).
  Beispiel: Falls ein Thread in einem Suchalgorithmus erfolgreich war, ist es nicht mehr nötig, daß alle andern Threads weitersuchen.



- Ein Thread kann mit
  `pthread_cancel(pthread_t thread)`
  versuchen, den durch `thread` beschriebenen Thread abzubrechen.

3 Abbruch eines Threads (2)

- Der Pthread Standard ermöglicht drei Arten, mit denen ein Thread auf eine Abbruchaufforderung reagieren kann:
  - Er kann sie ignorieren `(PTHREAD_CANCEL_DISABLE)`
  - Er kann sofort abbrechen.
    `(PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_ASYNCHRONOUS)`
  - Er kann sich die Aufforderung merken und beim nächsten Cancellation-Point abbrechen.
    `(PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_DEFERRED)`
    Dies ist standardmäßig eingestellt.

Cancelstate	Canceltype
PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_ASYNCHRONOUS oder PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_DISABLE	--

3 Abbruch eines Threads (3)

- Der Empfang von Cancelversuchen kann mit

```
pthread_setcancelstate(int state,
                       int *oldstate)
```

ein- und ausgeschaltet werden. (ähnlich wie die Signalmaske)
Gültige Werte für `state` sind die vordefinierten Werte:

  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`

- Ob ein Cancelversuch sofort wirksam wird oder erst an einem Cancellation-Point, ist mit

```
pthread_setcanceltype(int type,
                      int *oldtype)
```

konfigurierbar.
Auch hier gibt es zwei vordefinierte Werte:

  - `PTHREAD_CANCEL_ASYNCHRONOUS`
  - `PTHREAD_CANCEL_DEFERRED`

3 Abbruch eines Threads (4)

- Da ein Thread wichtige Resourcen belegt haben kann (z.B. Mutexe), ist es aber meist sinnvoll, ihn nur an bestimmten Stellen im Programm abbrechen zu lassen. Diese Stellen werden *Cancellation-Points* genannt.

- Ein Cancellation-Point kann durch den Aufruf von

```
void pthread_testcancel(void)
```

markiert werden. Zusätzlich sind folgende Aufrufe automatisch gültige Cancellation-Points:

  - pthread_cond_[timed]wait
  - pthread_join
  - Systemfunktionen, die längere Zeit warten können, dürfen (und müssen teilweise) auch Cancellation-Points sein.

3 Abbruch eines Threads (5)

- Beispiel:

```
/* make shute we cannot be canceled asynchonously */
type = PTHREAD_CANCEL_DEFERRED;
(void) pthread_setcanceltype(type, &oldtype);

/* do some buffer processing */
(void) pthread_mutex_lock(&write_mtx);
...
(void) pthread_mutex_unlock(&write_mtx);

/* act on cancellation requests before writing database */
pthread_testcancel();

/* don't process cancellation requests */
state = PTHREAD_CANCEL_DISABLE;
(void) pthread_setcancelstate(state, &ostate);
...
write_into_database(buffer);
...
/* restore cancelability state and type */
(void) pthread_setcancelstate(oldstate, &state);
(void) pthread_setcanceltype(oldtype, &type);
```


4 Der Cleanup Stack

- Problem: Verklemmung durch belegte Locks nach einer Cancellation

```
/* make shute we cannot be canceled asynchonously */
type = PTHREAD_CANCEL_DEFERRED;
(void) pthread_setcanceltype(type, &oldtype);

/* do some buffer processing                     */
(void) pthread_mutex_lock(&write_mtx);
if (global_data_ptr != NULL) {
   write(fd, global_data_ptr, sizeof(global_data_ptr));
   free(global_data_ptr);
   global_data_ptr = NULL;
(void) pthread_mutex_unlock(&write_mtx);
```

cancel

  - Der Mutex wird durch den abgebrochenen Thread nicht freigegeben.
    Ein anderer Thread, der diesen Code-Bereich durchlaufen wird, wird für immer auf die Freigabe des Mutex warten.
    ➜ Die Anwendung verklemmt sich!

4 Der Cleanup Stack (2)

- Problem: Inkonsistente Daten nach einer Cancellation

```
/* make shute we cannot be canceled asynchonously */
type = PTHREAD_CANCEL_ASYNCHRONOUS;
(void) pthread_setcanceltype(type, &oldtype);

/* do some buffer processing                      */
(void) pthread_mutex_lock(&write_mtx);
if (global_data_ptr != NULL) {
    write(fd, global_data_ptr, sizeof(global_data_ptr));
    free(global_data_ptr);
    global_data_ptr = NULL;
(void) pthread_mutex_unlock(&write_mtx);
```

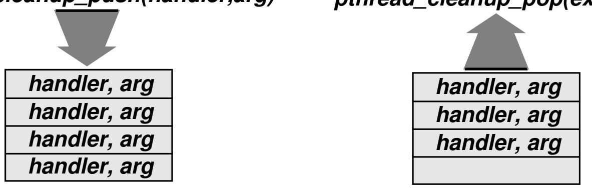
cancel

◆ Die Schreiboperation **write** wurde beendet. Durch den Abbruch des Threads wird aber der globale Zustand nicht mehr richtig gesetzt. Die Anwendung denkt, die Daten seien noch nicht geschrieben worden.
➜ Die Daten der Anwendung werden inkonsistent!

4 Der Cleanup Stack (3)

- Um die Speicherfreigabe und andere Aufräumarbeiten zu vereinfachen, hält die Pthread Bibliothek pro Thread einen sogenannten *Cleanup Stack*.
- Wenn ein Thread verstirbt (durch `pthread_exit`, Rücksprung aus der Startfunktion oder einer erfolgreichen Cancellation), werden alle Routinen aufgerufen, die auf den Stack geschoben worden sind (natürlich in LIFO Reihenfolge).
- Zu jedem Push-Aufruf muß es genau ein Pop-Aufruf geben, und zwar auf gleicher Programmblockebene. Dies erlaubt es, die beiden Funktionen als Macros zu definieren, die neue Variablen auf dem Stack anlegen.

pthread_cleanup_push(handler,arg) ***pthread_cleanup_pop(execute)***



4 Der Cleanup Stack (4)

- Push auf den Cleanup Stack:

```
void pthread_cleanup_push(
                          void (*routine)(void *),
                          void *arg)
```

Die Funktion muß vom Typ `void` sein. Das Argument `arg` wird beim Aufruf der Cleanupfunktion als Parameter übergeben.

- Pop der letzten Funktion:

```
void pthread_cleanup_pop(int execute)
```

Falls `execute` gesetzt ist, wird die gespeicherte Funktion aufgerufen.

4 Der Cleanup Stack (5)

```
void cleanup_func (char *ptr) {
 free(ptr);
}
...

void start_routine() {
 char *ptr
 ptr = (char *) malloc(1024);
 pthread_cleanup_push((void (*)())cleanup_func, (void *)ptr)

 /* Code to do sometinhg with pointer ptr        */
 ...
 /* Finish with ptr, free() it up an remove the */
 /* Cancellation cleanup handl                   */
 pthread_cleanup_pop(0);
 free(ptr);

 pthread_exit(0);
}
```

4 Der Cleanup Stack (6)

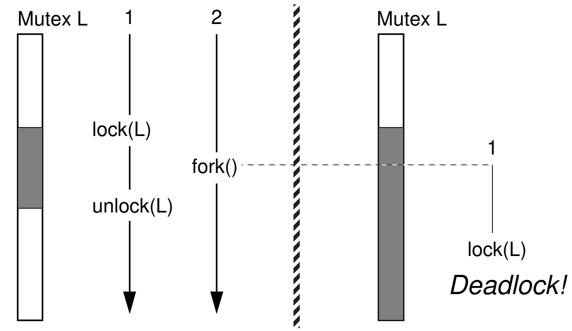
```
void cleanup_mutex (pthread_mutex_t *lock) {
 pthread_mutex_unlock(lock);
}

void start_routine() {
struct job_task *task;
for ( ; ; ) {
   (void) pthread_mutex_lock(task_lock);
  pthread_cleanup_push((void (*)())cleanup_mutex,
                       (void *)task_lock)
  while (task_queue_empty())
        pthread_cond_wait(task_cv, task_lock);
  pthread_cleanup_pop(0);
  task = task_dequeue();
  pthread_mutex_unlock(task_lock);
  /* Process the Task */
  ...
  free((void *)task);
 }
}
```


5 Fork/Exec/Exit

- Die Pthread Designer haben versucht, sich möglichst wenig von der Semantik der UNIX Prozesse zu entfernen.
- Ruft ein Thread in einem Programm, in dem mehrere Threads laufen, `fork()` auf, wird ein neuer Prozeß mit einem Abbild das Adreßraumes und genau einem Thread gestartet.
- Dieses führt zu zwei schwerwiegenden Problemen:
  - Der Kindprozeß erbt den Zustand aller Locks des Vaterprozesses, ohne über deren Status genaueres zu wissen.
  - Der Kindprozeß erbt den Datenbereich, also auch den von anderen Threads alloziierten Speicher. Ein sauberes Freigeben der unbenötigten Bereiche ist schwierig.
- Diese Probleme sind natürlich irrelevant, wenn nach dem `fork()` direkt ein `exec()` oder `exit()` aufgerufen wird.

5 Fork/Exec/Exit (2)

- Beispiel für einen Deadlock:



5 Fork/Exec/Exit (3)

- Die Pthread Bibliothek bietet zur Lösung dieses Problems sogenannte *fork-handling Stacks* an. Diese werden bei einem `fork()` Aufruf automatisch abgearbeitet.
- Es gibt drei unterschiedliche Stacks:
  - Der *prepare-Stack*: Die Funktionen auf diesem Stack werden automatisch vor dem eigentlichen fork() aufgerufen.
  - Der *parent-Stack*: Der Vater des erzeugten Prozesses ruft nach dem fork() automatisch alle Funktionen auf diesem Stack auf.
  - Der *child-Stack*: Der erzeugte Prozeß ruft alle auf diesen Stack gelegten Funktionen auf.

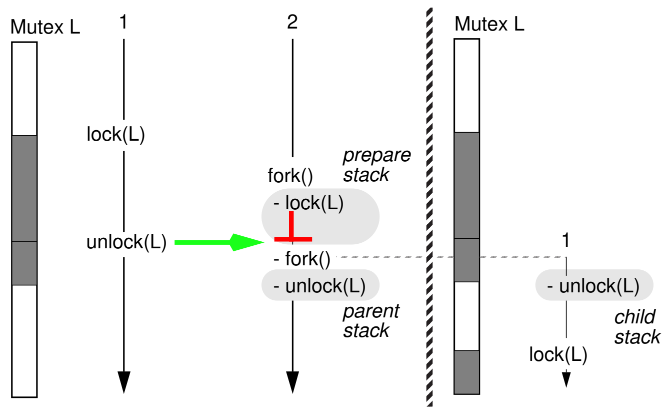
5 Fork/Exec/Exit (4)

- Die Syntax:
```
pthread_atfork(void (*prepare)(void),
               void (*parent)(void),
               void (*child)(void))
```
Ein Stack kann durch Angabe von `NULL` unverändert bleiben.

- Mit dieser Funktion ist das fork-Problem leicht lösbar:
  - ◆ Die Routinen auf dem prepare-Stack belegen alle Mutexe (in der richtigen Reihenfolge), die der erzeugte Prozeß benötigt.
  - ◆ Die Funktionen auf dem parent- und dem child-Stack geben die Mutexe wieder frei. Zusätzlich kann die child-Funktion noch weitere Resourcen freigeben.

5 Fork/Exec/Exit (5)

- Vermeidung des Deadlocks:



5 Fork/Exec/Exit (6)

- Da der Pthread Standard Implementierungen zuläßt, die auf einem Monoprozessor ohne Kernunterstützung laufen, muß ein Aufruf von `exec()` und `exit()` eine automatischen Terminierung aller Threads bewirken.

- Ein Aufruf von `exit()` bewirkt noch einige Aufräumaktionen (Flush aller geöffneten Files).

- Soll dies vermieden werden, kann `_exit()` aufgerufen werden. Dies ist der direkte Kernaufruf.

6 Interprozeß-Synchronisation

- Durch das Setzen von Attributen können Mutexe und Conditionals über Prozeßgrenzen hinweg verwenden werden.

- Der Mutex/Conditional muß hierzu in einem Shared-Memory-Bereich liegen, auf den die Prozesse zugreifen können (über `mmap()` oder SysV-IPC). Weiterhin ist diese Option nur gestattet, wenn die Konstante `_POSIX_THREAD_PROCESS_SHARED` definiert ist.

- Das benötigte Attribut heißt ATTR=`pshared`, ATTRTYPE=`int`.
  - ◆ `PTHREAD_PROCESS_SHARED`: ermöglicht diese Option,
  - ◆ `PTHREAD_PROCESS_PRIVATE`: verbietet sie.

E.8 Zusammenfassung

- Threads sind ein mächtiges Hilfmittel zur Parallelisierung eines Programmes.
- Der Vorteil des schnellen und uneingeschränkten Zugangs auf alle Prozeßresourcen (Speicher, Files) ist allerdings gleichzeitig das Hauptrisiko, da es leicht zu sehr schwer zu findende *Raceconditions* kommen kann.
- Man sollte sich immer genau überlegen, ob man auf Synchronisations- und Ausnahmebehandlungsroutinen wirklich verzichten kann.