# F.8 MACH — Tasks und Threads

## 1 Motivation

UNIX-Prozeßkonzept ist für viele heutige Anwendungen unzureichend

- in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adreßraum benötigt
- zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adreßraums nützlich
- typische UNIX-Server-Implementierungen benutzen die *fork*-Operation, um einen Server für jeden Client zu erzeugen
  - ➥ Verbrauch unnötig vieler System-Ressourcen (Datei-Deskriptoren, Page-Table, Speicher, ...)




## 2 Vergleich von Prozeß- und Thread-Konzepten



- mehrere **UNIX-Prozesse** mit gemeinsamen Speicherbereichen

Bewertung:

+ echte Parallelität möglich
– viele Betriebsmittel zur Verwaltung eines Prozesses notwendig; Prozeßumschaltungen aufwendig → teuer
– innerhalb einer solchen Prozeßfamilie wäre häufig ein anwendungsorientiertes Scheduling notwendig: schwierig realisierbar




## 2 Vergleich von Prozeß- und Thread-Konzepten (2)



- ***User-Level-Threads*** (Koroutinen) — Realisierung von Threads auf Benutzerebene innerhalb eines Prozesses

Bewertung:

+ Erzeugung von Threads und Umschaltung extrem billig
+ Verwaltung und Scheduling anwendungsorientiert möglich
– Systemkern hat kein Wissen über diese Threads
  - ➥ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
  - ➥ in Multiprozessorsystemen keine parallelen Abläufe möglich
  - ➥ wird eine Koroutine wegen eines *page faults* oder in einem Systemaufruf blockiert, ist der gesamte Prozeß blockiert




## 2 Vergleich von Prozeß- und Thread-Konzepten (3)



- ***Kernel-Threads***: leichtgewichtige Prozesse *(lightweight processes)*

Bewertung:

+ eine Gruppe leichtgewichtiger Prozesse nutzt gemeinsam eine Menge von Betriebsmitteln
+ jeder leichtgewichtige Prozeß ist aber als eigener Aktivitätsträger dem Betriebssystemkern bekannt
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack

... Bewertung *Kernel-Threads (lightweight processes)*

+ Umschalten zwischen zwei leichtgewichtigen Prozessen einer Gruppe ist erheblich billiger als eine normale Prozeßumschaltung
  - ➥ es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
  - ➥ Adreßraum muß nicht gewechselt werden
  - ➥ alle Systemressourcen bleiben verfügbar
- Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei "schwergewichtigen" Prozessen, aber erheblich teurer als bei *user-level-Threads*
- Verwaltung und Scheduling meist durch Kern vorgegeben





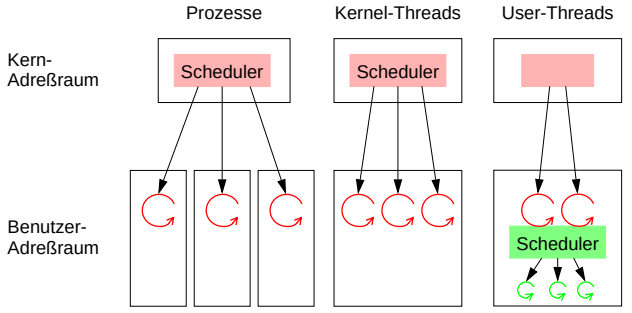

■ Vergleich

| | Prozesse | Kernel-Threads | User-Threads |
|---|---|---|---|
| Kosten | – teuer | O mittel | + billig |
| Betriebssystemeingliederung | + gut | + gut | – schlecht |
| Interaktion untereinander | – schwierig | + einfach | + einfach |
| Benutzerkonfigurierbarkeit | – nein | – nein | + ja |
| Gerechtigkeit | – nein | + ja | ± teils |

- Gerechtigkeit bedeutet:
  wie kommt das System damit klar, wenn eine Anwendung eine große Anzahl von Aktivitätsträgern erzeugt, eine andere dagegen eine geringe — werden Zeitscheiben an Anwendungen oder an Aktivitätsträger vergeben?











▲ ***Task:*** Betriebsumgebung (System-Resourcen) für Aktivitätsträger
- ➤ virtueller Adreßraum
- ➤ Zugriffsrechte (= Portrechte)
- ➤ Betriebsmittel-Informationen
- ➤ kein Programm-Ablauf und keine Register

▲ ***Thread:*** Aktivitätsträger und seine Ablaufumgebung
- ➤ Registersatz
- ➤ Stack
- ➤ Programmzähler

◆ Innerhalb einer *Task* können beliebig viele *Threads* existieren

◆ In einer echten Multiprozessorumgebung können verschiedene *Threads* einer *Task* parallel auf mehreren Prozessoren ablaufen

◆ Traditioneller UNIX-Prozeß ≙ MACH-*Task* mit einem *Thread*

## F.9 UNIX — Prozesse, LWPs & Threads

- Thread-Konzept zunehmend auch in UNIX-Systemen realisiert
  - ◆ Solaris
  - ◆ HP UX
  - ◆ Digital UNIX
  - ◆ Linux
  - ◆ ...
- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE POSIX Standard P1003.4a
- Pthreads-Implementierungen aber sehr unterschiedlich!
  - ➤ reine User-level-Threads (Linux, HP-UX)
  - ➤ reine Kernel-Threads (MACH, KSR-UNIX, Digital UNIX)
  - ➤ parametrierbare Mischung (Solaris)
- Daneben z. T. auch andere Thread-Bibliotheken (z. B. Solaris-Threads)




## 1 pthread-Benutzerschnittstelle

F.9 UNIX — Prozesse, LWPs & Threads

- **Pthreads-Schnittstelle (Basisfunktionen)**:

| | |
|---|---|
| ***pthread_create*** | Thread erzeugen & Startfunktion angeben |
| ***pthread_exit*** | Thread beendet sich selbst |
| ***pthread_cancel*** | Anderen Thread beenden |
| ***pthread_join*** | Auf Ende eines anderen Threads warten |
| ***pthread_self*** | Eigene Thread-Id abfragen |
| ***pthread_yield*** | Prozessor zugunsten eines anderen Threads aufgeben |




## 1 pthread-Benutzerschnittstelle (2)

F.9 UNIX — Prozesse, LWPs & Threads

- Threaderzeugung

```
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.

Thread wird erzeugt und ruft Funktion **start_routine** mit Parameter **arg** auf

- Thread beenden (bei return aus **start_routine** oder):

```
void pthread_exit(void *retval)
```

- Auf Thread warten und exit-Status abfragen:

```
pthread_join(pthread_t thread, void **retvalp)
```




## 1 pthread-Benutzerschnittstelle (3)

F.9 UNIX — Prozesse, LWPs & Threads

- Beispiel (Multiplikation Matrix mit Vektor)

```
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);
...
}

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;
}
```