Prozesse

und

Threads

Gliederung

Prozesse und Threads

Übersicht:

- **Prozesse:**
  - Wozu Prozesse ?
  - Was ist ein Prozeß
  - Verwaltung von Prozessen
  - Prozeßzustände
  - Prozesslisten
  - Prozesshierarchien
  -
  - Praxisbeispiele + Ergänzungen

- **Threads:**
  - Was ist ein Thread ?
  - Thread-Realisierungen
  - Prozesse und Threads erzeugen
  - Unterschiede Prozesse / Threads
  - ULT
  - KLT
  - Thread-Zustände
  - Prozesse und Threads
  - Praxisbeispiele + Ergänzungen
  - Prozesserzeugung
  - Threads im Kernel

Wozu Prozesse ?

- **Single-Tasking / Multitasking:**
  - Wie viele Programme laufen gleichzeitig ?
    - Beispiel MS-DOS: 1 Programm
      - Betriebssystem startet und aktiviert Shell: COMMAND.COM
      - Eingabe von Befehlen
      - falls kein interner Befehl:
        - Programm laden und ausführen
        - Return zu COMMAND.COM nach Ende

  ==> Kein Wechsel zwischen Programmen
  ==> Keine Nutzung mehrerer CPUs möglich

- **Single-Processing / Multi-Processing:**

  Neuere BS fähig zu Multi-Tasking/-Processing und können mehrere CPUs nutzen (Concurrency)

==> Konzept eines **Prozesses** notwendig

Was ist ein Prozeß ?

Prozess:

- ein Programm ausgeführt/laufend im Rechner (Task)
- abgeschottet von anderen Prozessen
- erfordert zusätzliche Verwaltungsdaten
- das BS wählt Prozesse aus und weist ihnen CPUs zu (Scheduling)

Weitere Vorteile von Prozessen:

- übersichtlichere Programmstruktur
- „Besitz“ von Ressourcen (ressource ownership)
- Privilegien- und Rechtevergabe

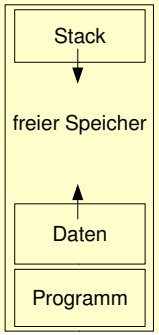
Prozess im Detail

- eigener (virtueller) Adressraum
- Programmkode
- aktuelle Daten:
  - Programmvariable
  - Konstanten
- Befehlszähler Programm-Counter (PC)
- Stacks und Stack-Pointer
- Inhalt der Hardware-Register (Prozess-Kontext)
- Heap-Speicher

Verwaltung von Prozessen

Prozesslisten/-tabelle:

- Informationen über alle Prozesse und ihre Zustände
- je Prozess ein Process-Control-Block (PCB):
  - Identifier (PID)
  - Registerwerte inkl. PC
  - Speicherbereich
  - Liste offener Dateien und Sockets
  - Informationen wie
    - Vater-PID
    - letzte Aktivität
    - Gesamtlaufzeit
    - Priorität
    - ...



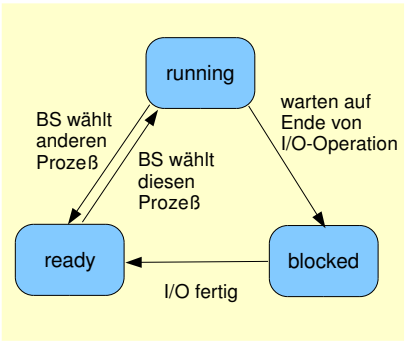
- Daten: dynamisch erzeugt
- Stack: Verwaltung der Funktionsaufrufe
- Stack und Daten „wachsen aufeinander zu“

Prozeßzustände

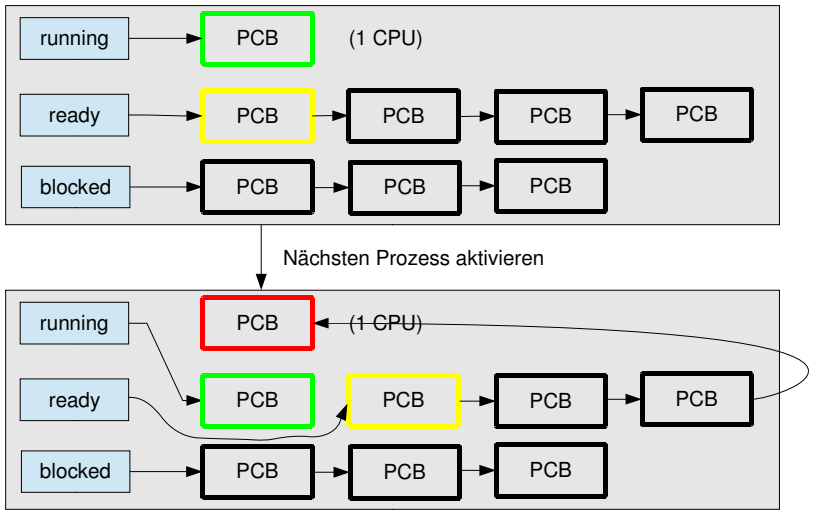
Prozesszustände:

- laufend / running: gerade aktiv
- bereit / ready: würde gerne laufen
- blockiert/blocked: wartet auf I/O
- suspendiert: vom Anwender unterbrochen
- schlafend/sleeping: wartend auf Signal (IPC)
- ausgelagert / swapped: Daten nicht im RAM

Zustandsübergänge:



Prozeßlisten



Prozeßhierarchien

- Prozesse erzeugen einander
- Erzeuger heißt Vaterprozess (parent)
  der andere Kindprozess (child)
- Kinder sind selbständig (eigener Adreßraum etc.)
- Nach Prozess-Ende: Rückgabewert an Vater

Beispiel:

- Start von emacs auf in der bash-Shell (cygwin)
- PID = 5016
- PPID = 4908 = PID der bash (Vater)

```
schnoerr@clsnb: ~ 3% emacs test.txt &
[1] 5016
schnoerr@clsnb: ~ 4% ps
    PID   PPID   PGID    WINPID   TTY     UID    STIME COMMAND
   4908   1560   4908      2956  pty1    1001 02:24:51 /usr/bin/bash
   2820   4908   2820      3260  pty1    1001 02:31:37 /usr/bin/ps
   5016   4908   5016      1036  pty1    1001 02:31:33 /hd/Textverarbeitung/emacs/bin/emacs
```


Praxisbeispiele und Ergänzungen

Praxis: einige Kommandos

Beispiel: &, ps, pstree, jobs, fg, bg, ctrl-Z

```
schnoerr@clsnb: ~ 16% emacs test.txt &
[3] 6032
schnoerr@clsnb: ~ 17% ps
    PID   PPID   PGID    WINPID   TTY     UID    STIME COMMAND
   6032   5016   6032      5148  pty1    1001 10:10:43 /hd/Textverarbeitung/emacs/bin/emacs
   5016   5152   5016      5060  pty1    1001 10:02:28 /usr/bin/bash
   6012   5016   6012      1092  pty1    1001 10:10:52 /usr/bin/ps
   5152   2260   5152      2460  pty0    1001 10:02:27 /usr/bin/rxvt
   2260    932   2260      5528  pty0    1001 10:01:49 /usr/bin/bash
    932      1    932       932  ?       1001 10:01:49 /usr/bin/mintty
schnoerr@clsnb: ~ 18% pstree -Glsp
?(1)---mintty(932)---bash(2260)---rxvt(5152)---bash(5016)-|-emacs(6032)
                                                          --pstree(5284)
schnoerr@clsnb: ~ 19% ps | grep emacs
   6032   5016   6032      5148  pty1    1001 10:10:43 /hd/Textverarbeitung/emacs/bin/emacs
schnoerr@clsnb: ~ 20% jobs
[3]+  Running                 emacs test.txt &
schnoerr@clsnb: ~ 21% fg 3
emacs test.txt
```

ctrl-Z: suspendiert Job im Vordergrund, mit bg dann wieder „running“

Praxis: Signale

Signale:

```
schnoerr@clsnb: ~ 26% kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGEMT       8) SIGFPE       9) SIGKILL     10) SIGBUS
11) SIGSEGV     12) SIGSYS      13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGURG      17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGPWR      30) SIGUSR1
31) SIGUSR2     32) SIGRTMAX

schnoerr@clsnb: ~ 30% kill -s 9 6032
schnoerr@clsnb: ~ 31%
[1]+  Killed                  emacs test.txt
```

- Signale:
  - STOP: unterbrechen
  - CONT: fortsetzen
  - TERM: beenden
  - KILL: abschießen
- disown: Verbindung zum Vater lösen

- weitere Infos: ManPages
  - man signal
  - man kill

Praxis: fork (1)

Neuer Prozess: fork()

```
int pid = fork()                                    //Kindprozess (Child) erzeugen
if ( pid == 0 ) {
      printf( "Ich bin das Kind mit pid=%d\n", getpid() );
} else if ( pid > 0 ) {
      printf( "Ich bin der Vater, mein Kind hat die pid=%d\n", pid );
} else {
      printf( "Error: fork() war nicht erfolgreich\n" );
}
```

```
> pstree | grep simple
      ... -bash---simplefork---simplefork

> ps -w | grep simple
   25684 pts/16 S+          0:00 ./simplefork
   25685 pts/16 S+          0:00 ./simplefork
```


Praxis: fork (2)

Start eines Programms mit fork() und exec():

```
int pid = fork()                                    //Kindprozess (Child) erzeugen
if ( pid == 0 ) {
      execl( "/usr/bin/gedit", "/etc/fstab", (char *)0 );   //Kind startet gedit
} else if ( pid > 0 ) {
      printf( "Ich bin der Vater, es sollte der Editor starten\n", pid );
} else {
      printf( "Error: fork() war nicht erfolgreich\n" );
}
```

- Syntax: exec(filename, argv, envp) <-> Child: main(argc, argv)
- Andere Betriebssysteme of nur „spawn", z.B.
  WinExec("notepad.exe", SW_NORMAL); //Sohnprozess abspalten

Praxis: fork (3)

Warten auf Kind-Prozess: wait()

```
int pid = fork()                                    //Kindprozess (Child) erzeugen
int status;
if ( pid == 0 ) {
      printf( "Ich bin das Kind mit pid=%d\n", getpid() );
} else if ( pid > 0 ) {
      printf( "Ich bin der Vater, mein Kind hat die pid=%d\n", pid );
      wait( & status );                             //exit status des Kind-Prozesses
} else {
      printf( "Error: fork() war nicht erfolgreich\n" );
}
```

- siehe auch waitpid(): warten auf Kinder aus eigener Prozessgruppe

Praxis: fork (4)

Abfrage, ob Programmstart über fork(), exec() erfolgreich war:

```
#include <errno.h>

int pid = fork();
int errno2;
if ( pid == 0 ) {
      execl( "/bin/date", 0 );
      errno2 = errno;
      perror();
      printf( "Fehlerkode errno = %d\n", errno2);
} else ...
```

- perror(): Fehlermeldung in lesbarem Format
- errno: globale Fehlervariable

Praxis: fork (5)

Abbruch aller Kind-Prozesse:

1. Shell wird mit `exit` verlassen:

   -> Kind-Prozesse laufen weiter

2. Shell wird gewaltsam geschlossen
   (z.B. `kill`, Fenster schließen)

   --> Kind-Prozesse werden auch beendet

Threads

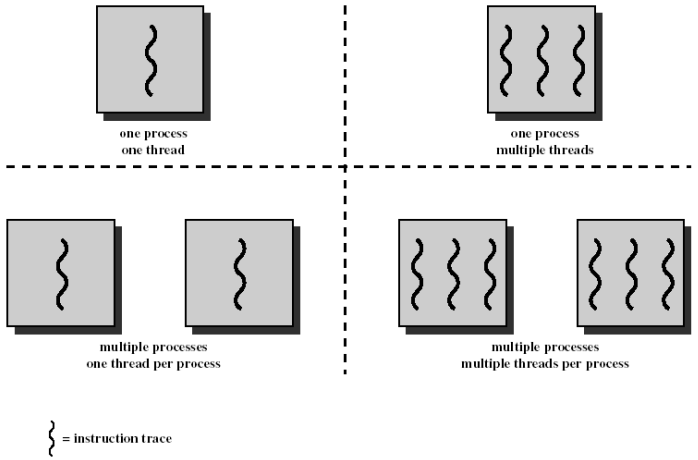
Was ist ein Thread ?

- Aktivitätsstrang in einem Prozess
- einer von mehreren
- gemeinsamer Zugriff auf Daten des Prozesses
- aber: separat pro Thread
  - Befehlszähler (PC)
  - Stack
  - Stack Pointer
  - Hardware-Register
- Kernel-Threads:
  - Prozess-Scheduler verwaltet diese
- User-Level-Threads:
  - User-Level-Bibliothek verwaltet diese

Thread im Detail

- eigener (virtueller) Adressraum
- Programmkode
- aktuelle Daten:
  - Programmvariable
  - Konstanten
- Befehlszähler
  Programm-Counter (PC)
- Stacks und Stack-Pointer
- Inhalt der Hardware-Register
  (Prozess-Kontext)
- Heap-Speicher

Prozesse und Threads



Prozesse und Threads erzeugen

Warum Threads ?

- Multi-Prozessor-System:
  mehrere Threads echt gleichzeitig aktiv
- ist ein Thread durch I/O blockiert, arbeiten andere weiter
- Logisch parallele Abläufe --> einfachere Programmstruktur mit Threads möglich
- aber:
  - gesonderte Behandlung gemeinsamer Daten notwendig
  - Compiler kann logische Fehler aus Nebenläufigkeit nicht erkennen
    --> Verantwortung des Entwicklers

Beispiele (1)

Zwei Aktivitätsstränge:

Ohne Threads:

```
while ( 1 ) {
    rechne_etwas();
    if ( benutzereingabe(x) ) {
        bearbeite_eingabe( x );
    }
}
```

Mit Threads:

T1:

```
while ( 1 ) {
    rechne_alles();
}
```

T2:

```
while ( 1 ) {
    if ( benutzereingabe(x) ) {
        bearbeite_eingabe( x );
    }
}
```


Beispiele (2)

Serverprozess, der viele Anfragen bearbeitet:

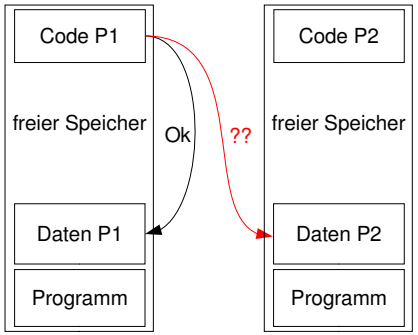
- Prozess öffnet Port
- Für jede eingehende Verbindung:
  - erzeuge Thread, der die Anfrage bearbeitet
- Nach Verbindungsabbruch: Thread beenden
- Vorteil:
  - keine aufwendige Erzeugung von Prozessen notwendig!

Unterschiede Prozesse / Threads (1)

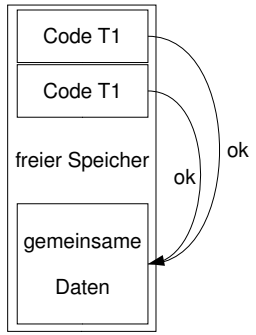
- Parallelverarbeitung wahlweise mit mehreren Prozessen / Threads
- Unterschiede:
  - Aufwand zur Prozesserzeugung vergleichsweise hoch
  - Austausch / Kommunikation untereinander:
    - Prozesse:
      - kein gemeinsamer Speicher
      - Austausch z.B. über Nachrichten, IPC, Dateizugriffe
      - Shared-Memory (muss extra angelegt und verwaltet werden)
    - Threads:
      - gemeinsamer Speicher
      - Austausch z.B. durch direkte Zugriffe auf gemeinsame Variablen

Unterschiede Prozesse / Threads (2)

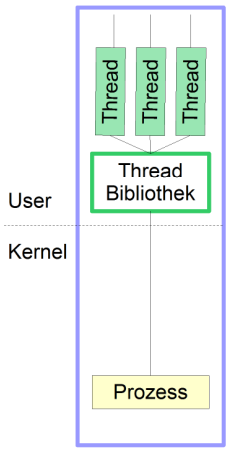
Zwei Prozesse:



Zwei Threads:

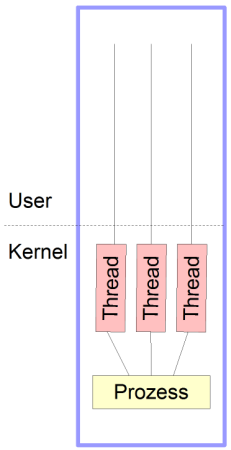


User-Level Threads



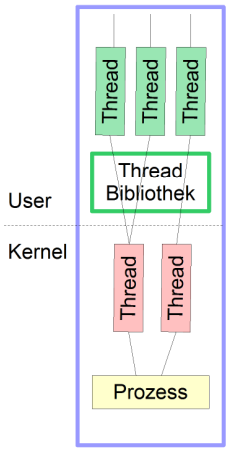
- für BS, die kein Thread-Konzept kennen, BS verwaltet nur Prozesse
- Programm bindet Thread-Bibliothek ein, zuständig für
  - Erzeugen, Auflösen
  - Scheduling vom Anwender beeinflußbar
- sehr effizient:
  - keine Systemaufrufe:
    - keine User-Mode / Kerne-Mode - Wechsel
- Nachteile:
  - wenn ein Thread wegen I/O wartet, dann der ganze Prozess --> blockierende System-Calls zu vermeiden
- keine Nutzung mehrerer CPUs über BS möglich

Kernel-Level Threads



- BS kennt und verwaltet Threads
  - Erzeugen
  - Auflösen
  - Scheduling
- I/O eines Threads blockiert nicht die übrigen
- aufwendig:
  - Kontext-Wechsel zwischen Threads ähnlich komplex wie für Prozesse

Gemischte / kombinierte Thread-Typen



- beide Ansätze kombiniert:
  - KL-Threads und UL-Threads
- Thread-Bibliothek verteilt ULTs auf KLTs
  - z.B. I/O-Anteile auf einem KLT
- Vorteile beider Konzepte:
  - I/O blockiert nur einen KLT
  - Wechsel zwischen ULTs effizient
- SMP (Symmetric Multi-Prozessing):
  - mehrere CPUs effektiv parallel nutzen

Vorteile UL-Thread-Bibliotheken

Welche Vorteile bieten ULT noch ?

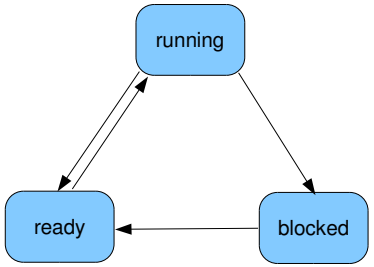
- weit verbreitet und standardisiert, z.B.
  - Posix-Threads (libpthread.a)
  - Thread-Klasse im neuen C++-Standard
- vereinheitlichtes API für multithreaded-Anwendungen (<-> Portierbarkeit)
- einfache Variation von Scheduling-Parameter:
  - Kontext zur Verteilung der CPU-Zeit: Prozess- oder System-weit
  - Thread-Priorität
  - Scheduling-Politik: prioritäts- oder auch zeitgesteuert
- Nachteile:
  - zeitgesteuertes Scheduling effizient nur mittels Systemaufrufe

Thread-Zustände

Thread-Zustände:

- laufend, bereit, blockiert wie bei Prozessen
- Prozess-Zustände
  - suspended,
  - sleeping,
  - swapped etc

  nicht auf Threads übertragbar
- darum nur drei Thread-Zustämde



Prozesse und Threads

Keine „Vater-“ oder Kind-Threads“

- POSIX-Threads kennen keine Verwandtschaft
- zum Warten auf Thread ist Thread-Variable notwendig

Unterschiedliche Semantik

- Prozess erzeugen mit fork():
  - erzeugt zwei (fast) identische Prozesse
  - beide setzen Ausführung an gleicher Stelle fort (nach fork())
- Thread erzeugen mit pthread_create(..., fkt, ...):
  - Thread springt in angegebene Funktion
  - euzeugender Prozess fährt nach pthread_create() fort

Posix- vs. Kernel-Thread:

- clone() erzeugt einen Kernel-Thread. Dieser ist nicht gleich einem Posix-Thread
- Posix-Bibliothek muss gewünschtes Standard-Verhalten über Linux-Kernel-Threads implementieren

Andere Thread-APIs (6)

Prozess mit mehreren Threads

- OMP-Threads:

```
#pragma omp parallel for num_threads(NCPU) schedule(static,chunkSize)
  for ( long i = 0; i < elems; ++i ) {
        ...
  }
```

- Thread-Klasse im C++0x-Standard
  - Abbildung auf pthreads oder andere betriebssystemspezifische Threads
  - --> einheitliches API
- Java-Threads

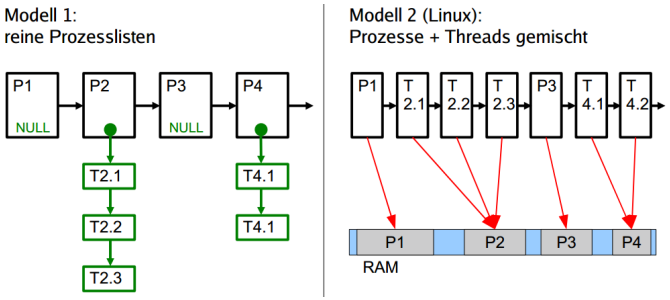
Spezialitäten: Prozesserzeugung

Wichtigste Datei in Kernelquellen: kernel/fork.c (enthält u.a. copy_process())

- Aufrufkette: `fork()` -> `clone()` -> `do_fork()` -> `copy_process()`
- `task_struct` enthält Prozessliste
- `copy_process()`:
  - `dup_task_struct()`:
    - neuer Kernel-Stack
    - thread_info Struktur
    - `task_struct`-Eintrag
  - Kind-Status auf TASK_UNINTERRUPTIBLE
  - `copy_flags()`: PF_FORKNOEXEC
  - `get_pid()`: neue PID für Kind vergeben
  - je nach clone()-Parametern
    - offene Dateien, Signal-Handler, Prozess-Speicherbereiche etc. kopieren oder gemeinsam nutzen
  - verbleibende Rechenzeit aufteilen (-> Scheduler)
- danach: aufwecken, starten (Kind kommt vor Vater dran)

Spezialitäten: Threads im Kernel (1)

- Linux verwendet gleiche Verwaltungsstrukturen (task list) für Prozesse und Threads
- Thread: Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt:, z.B.
  - virtueller Speicher
  - offene Dateien
- Jeder Thread hat `task_struct` und sieht für Kernel wie ein Prozess aus



Spezialitäten: Threads im Kernel (2)

- Thread-Erzeugung: auch über clone()
- einfach andere Aufrufparameter:
  - Prozess:
    - `clone( SIGCHLD, 0 );`
  - Thread:
    - `clone( CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0 );`
      - vm: virtual memory
      - fs: z.B. Arbeitsverzeichnis, umask, root-Verzeichnis des Prozesses
      - files: offene Dateien
      - sighand: Signal Handler

Praxisbeispiele

Praxis: Linux pthreads (1)

pthread-Bibliothek (POSIX-Threads/Standard):

	Thread	Prozess
Erzeugen	pthread_create()	fork()
Auf Ende warten	pthread_join()	wait()

- Neuer Thread:

  `pthread_create()`
  erhält als Argument eine Funktion, die eigenständig in neuem Thread läuft
- Auf Thread-Ende warten:

  `pthread_join()`
  wartet auf bestimmten Thread
- Deklarationen einfügen:
  `#include <pthread.h>`
- Kompilieren und einbinden:
  `g++ -lpthread -o prog prog.cc`
- Anm.: Portierungen auch für Windows

Praxis: Linux pthreads (2)

pthread-Bibliothek: pthread_create():

1. **Thread-Funktion definieren:**

```
void * thread_fkt( void * arg ) {
        ...
        return ...;
}
```

2. **Thread erzeugen und starten:**

```
pthread_t thread;

if ( pthread_create( &thread, NULL, thread_fkt, NULL ) ) {
        printf( "Fehler bei Thread_Erzeugung\n" );
        abort()
}
```


Praxis: Linux pthreads (3)

pthread-Bibliothek: pthread_join():

1. **Thread-Funktionen definieren:**

```
void * thread_fkt_1( void * arg ) { ... }
void * thread_fkt_2( void * arg ) { ... }
```

2. **Threads erzeugen und starten:**

```
pthread_t tid1, tid2;

pthread_create( &tid1, NULL, thread_fkt_1, NULL ) );
pthread_create( &tid2, NULL, thread_fkt_2, NULL ) );
```

3. Auf Ende der Threads warten:

```
pthread_join( tid1, NULL );          //thread_1 beendet
pthread_join( tid2, NULL );          //thread_2 beendet
```

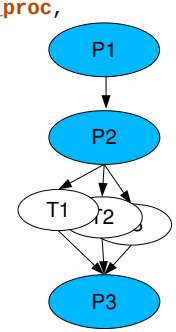

Praxis: Linux pthreads (4)

Beispiel zur Verkehrsanalyse:

```
while (...) {
     Object * ptr = new Object();
     // Receive data from another process and store it in *ptr

     while ( (rc = pthread_create( &thread_id, NULL, thr_proc,
                                   (void *)ptr )) ) {
          // error treatment
     } /*while*/
     pthread_detach( thread_id );
} /*while*/

void * thr_proc( void * ptr ) {
     // process data in *ptr
     // transmit result to next process
     delete (Object *)ptr;
     return  NULL;
}
```



Praxis: Linux pthreads (5a)

Was gehört nicht zum Thread-Kontext ?

```
int thread_cnt = 0;
while (...) {
     Object * ptr = new Object[100];
     while ( thread_cnt > 20 )  sleep( 1 );
     pthread_create( &thread_id, NULL, thr_proc, (void *)ptr );
     pthread_detach( thread_id );
} /*while*/

void * thr_proc( void * ptr ) {
     static int i;
     thread_cnt++;                              //ist zu synchronisieren
     for ( i=0; i < 100; ++i )  ((Object *)ptr)[i].init();
     ...
     delete [] (Object *)ptr;
        thread_cnt--;                           //ist zu synchronisieren
     return  NULL;
}
```

Praxis: Linux pthreads (5b)

Was gehört nicht zum Thread-Kontext ?

```
int thread_cnt = 0;
while (...) {
     Object * ptr = new Object[100];
     while ( thread_cnt > 20 )  sleep( 1 );
     pthread_create( &thread_id, NULL, thr_proc, (void *)ptr );
     pthread_detach( thread_id );
} /*while*/

void * thr_proc( void * ptr ) {
     static int i;
     thread_cnt++;                          //ist zu synchronisieren
     for ( i=0; i < 100; ++i )  ((Object *)ptr)[i].init();
     ...
     delete [] (Object *)ptr;
        thread_cnt--;                       //ist zu synchronisieren
     return  NULL;
}
```

ptr verweist auf Heap (Prozess-Kontext), Nutzung per Programmlogik ausschließlich in jeweils einem Thread

static Variablen nicht auf Stack

thread_cnt und ptr global definiert

Praxis: Linux pthreads (6)

Prozess mit mehreren Threads

- nur ein Eintrag in normaler Prozessliste
- Status: “I“, multi-threaded
- über 'ps -eLF' Thread-Informationen
  - NLWP: Number of light weight processes
  - LWP: Thread ID

```
> ps auxw | grep thread
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    12022  0.0  0.0  17976   436 pts/15   Sl+  22:58   0:00 ./thread

> ps -eLf | grep thread
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
esser    12166  4031 12166  0    3 23:01 pts/15   00:00:00 ./thread1
esser    12166  4031 12167  0    3 23:01 pts/15   00:00:00 ./thread1
esser    12166  4031 12177  0    3 23:01 pts/15   00:00:00 ./thread1
```