

Vorlesung
**Moderne
Realzeitsysteme**

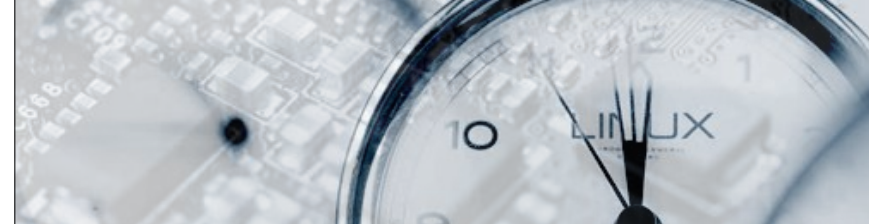
Realzeitprogrammierung

Professor Dr. Michael Mächtel

Realzeitprogrammierung

Realzeitprogrammierung

1. Allgemeines
2. Programmtechnischer Umgang mit Tasks
3. Schutz kritischer Abschnitte
4. Umgang mit Zeiten
5. Kommunikation und Signalisierung



Professor Dr. Michael Mächtel 2

Realzeitprogrammierung

Realzeitprogrammierung

1. Allgemeines
2. Programmtechnischer Umgang mit Tasks
3. Schutz kritischer Abschnitte
4. Umgang mit Zeiten
5. Kommunikation und Signalisierung



Professor Dr. Michael Mächtel 3

Realzeitprogrammierung

Allgemeines

- Realzeitapplikationen greifen sehr häufig auf Peripherie zu und verarbeiten Zeitinformationen.
- Auch wenn Realzeit nicht zwangsläufig »Schnelligkeit« bedeutet, sind Realzeitapplikationen typischerweise besonders effizient realisiert.
 - Realzeitprogrammierer verwenden nur in Ausnahmefällen Floating- Point-Variablen und -Operationen
 - Alternativ die Berechnungen mit Integeroperationen durchführen.
 - Dazu wird beispielsweise bei einer Division der Divident mit $10^{\text{Anzahl gewünschter Nachkommastellen}}$ multipliziert.
 - Die Rechnung $2/3$, die normalerweise im Integerbereich 1 ergibt, ergibt damit bei drei Nachkommastellen $(2 \times 1000 / 3) = 666$.

Professor Dr. Michael Mächtel 4

POSIX

- Der Standard ist sehr umfangreich und eine vollständige Implementierung ist für ressourcenbeschränkte, eingebettete Systeme nicht sinnvoll.
- Daher sind unter der Nummer POSIX 1003.13 POSIX-Profile für den Bereich der Realzeitsysteme definiert worden:
 - PSE51 (Minimal)
 - PSE52 (Controller)
 - PSE53 (Dedicated)
 - PSE54 (Multi-Purpose)

POSIX-Profil PSE51 (Minimal)

- Dieses Profil ist für kleine, eingebettete Systeme gedacht, deren Hardware ohne MMU (Memory Management Unit), Festplatte und typischerweise auch ohne Bildschirm auskommt.
- Softwareseitig handelt es sich um ein Singletasking-System.
- Beispiel
 - Steuerung eines Toasters

POSIX-Profil PSE52 (Controller)

- Dieses Profil wird für Steuerungen verwendet, die keinen Speicherschutz realisieren und kein Multitasking unterstützen, wohl aber neben den Basisfunktionen auf einen Hintergrundspeicher (Festplatte) zugreifen können.
- Es unterstützt einfache Dateisysteme.
- Beispiel
 - Geräte zur Langzeitdatenerfassung

POSIX-Profil PSE53 (Dedicated)

- Dieses Profil deckt eingebettete Systeme ab, deren CPU dank eingebauter MMU Speicherschutz realisieren kann und denen ein Hintergrundspeicher (Flash oder Festplatte) zur Verfügung steht, auf dem Daten über ein Dateisystem organisiert abgelegt werden.
- Es handelt sich um Multitasking-Systeme.
- Das Profil definiert Interfaces für die Kommunikation und für die asynchrone Ein-/Ausgabe.
- Beispiel
 - Basisstationen eines Mobilfunknetzes

POSIX-Profil PSE54 (Multi-Purpose)

- Möglichst generell einsetzbare Systeme mit Realzeitanforderungen
- Systeme mit diesem Profil bieten Speicherschutz, Netzwerk, Dateisysteme und Multiuser-Fähigkeiten.
- Shells und Systemprogramme sind definiert.
- Außerdem haben die Systeme Tastatur und Bildschirm, durchaus auch ein grafisches Benutzerinterface (GUI).
- Beispiel
 - System zur Flugverkehrsüberwachung

Realzeitprogrammierung

1. Allgemeines
2. **Programmtechnischer Umgang mit Tasks**
3. Schutz kritischer Abschnitte
4. Umgang mit Zeiten
5. Kommunikation und Signalisierung



Tasks erzeugen

- Prozesse
 - `fork()`, `execve()`
- Threads
 - `pthread_create()`

Tasks beenden

- Prozesse
 - Eigenes Beenden: `exit()`
 - Fremdes Beenden: `kill()`
- Threads
 - Eigenes Beenden: `pthread_exit()`
 - Fremdes Beenden:
 - `pthread_kill(child_thread, SIGINT);`
 - `pthread_join(child_thread, NULL);`

Tasks parametrieren

- Prozesse
 - Scheduling-Verfahren setzen
 - `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)`
 - Mögliche Scheduling-Verfahren
 - SCHED_RR: Round Robin
 - SCHED_FIFO: First Come First Serve
 - SCHED_OTHER: Default-Verfahren für Jobs ohne (Realzeit-)Priorität
 - SCHED_DEADLINE: Earliest Deadline First
 - Abfrage des Task Schedulingverfahren
 - `sched_getparam(pid_t pid, struct sched_param *param)`

Threads parametrieren (Attribute) (1)

- `pthread_attr_init()`, `pthread_attr_destroy()`
 - Objekt initialisieren bzw. deinitialisieren
- `pthread_attr_setstacksize()`, `pthread_attr_getstacksize()`
 - Diese Methoden geben Zugriff auf die Stackgröße.
- `pthread_attr_setstackaddr()`, `pthread_attr_getstackaddr()`
 - Diese Methoden geben Zugriff auf die Stackadresse.

Threads parametrieren (Attribute) (2)

- `pthread_attr_setdetachstate()`, `pthread_attr_getdetachstate()`
 - Threads können »detached« (PTHREAD_CREATE_DETACHED) oder »joinable« (PTHREAD_CREATE_JOINABLE) sein.
 - Im Zustand »detached« dürfen die zugehörigen Thread-Ressourcen, beispielsweise die Thread-Id oder der TCB, direkt neu genutzt werden.
 - Im Zustand »joinable« ist das erst möglich, nachdem `pthread_join()` aufgerufen worden ist.

Threads parametrieren (Attribute) (3)

- `pthread_attr_setscope()`, `pthread_attr_getscope()`
 - Der sogenannte Contention Scope definiert, in welchem Kontext das Scheduling eines Threads stattfindet.
 - Threads mit dem Scope PTHREAD_SCOPE_SYSTEM werden vom normalen Scheduler verwaltet.
 - Threads mit dem Scope PTHREAD_SCOPE_PROCESS werden von einem POSIX-internen Scheduler verarbeitet.
 - Dieser kommt dann zum Einsatz, wenn der System-Scheduler die Threadgruppe ausgewählt hat.

Threads parametrieren (Attribute) (4)

- `pthread_attr_setinheritsched()`, `pthread_attr_getinheritsched()`
 - Dieses Attribut legt fest, ob die Attribute eines erzeugten Threads identisch mit denen des erzeugenden Threads (PTHREAD_INHERIT_SCHED) sind oder ob diese geändert werden können (PTHREAD_EXPLICIT_SCHED).

Threads parametrieren (Attribute) (5)

- `pthread_attr_setschedpolicy()`, `pthread_attr_getschedpolicy()`
 - Diese Methoden geben Zugriff auf die Scheduling-Policy.
- `pthread_attr_setschedparam()`, `pthread_attr_getschedparam()`
 - Diese Methoden geben Zugriff auf die Scheduling-Parameter.

Realzeitprogrammierung

1. Allgemeines
2. Programmtechnischer Umgang mit Tasks
3. Schutz kritischer Abschnitte
4. Umgang mit Zeiten
5. Kommunikation und Signalisierung



Semaphore

- Race Conditions lassen sich vermeiden, indem nie mehr als ein Prozess in einen kritischen Abschnitt eintritt (Mutual Exclusion, gegenseitiger Ausschluss). Dies lässt sich mithilfe von Semaphoren sicherstellen.
- Ein Semaphore wird zur Synchronisation, insbesondere bei gegenseitigem Ausschluss, eingesetzt.
- P-Operation
 - `s = s - 1;`
 - `if (s < 0) {`
 - `sleep_until_semaphore_is_free();`
 - `}`
- V-Operation
 - `s = s + 1;`
 - `if (s <= 0) {`
 - `wake_up_sleeping_process_with_highest_priority();`
 - `}`

Mutex

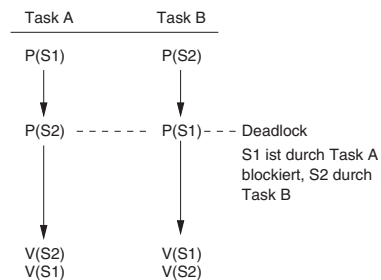
- Ein Mutex ist ein Semaphore mit dem Maximalwert $N = 1$ (binäres Semaphore).
- `pthread_mutex_init()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`

Behandlung Prioritätsinversion

- Der Umgang mit Prioritätsinversion wird bei Initialisierung eines Mutex über die Funktion `pthread_mutexattr_setprotocol()` gewählt
- Dabei hat der Programmierer die Möglichkeit
 - mit dem Define `PTHREAD_PRIO_NONE` ganz auf die Behandlung einer Prioritätsinversion zu verzichten,
 - per `PTHREAD_PRIO_INHERIT` die klassische Prioritätsvererbung (Priority Inheritance) zu benutzen, oder
 - per `PTHREAD_PRIO_PROTECT` das Protokoll **Priority Ceiling** zu aktivieren.
 - Wird Priority Ceiling aktiviert, kann über `pthread_mutexattr_getprioceiling()` und `pthread_mutexattr_setprioceiling()` die obere Schranke (Ceiling) gelesen beziehungsweise auch gesetzt werden.

Deadlock

- Durch Critical Sections kann es leicht zu Verklemmungen, den sogenannten Deadlocks, kommen.
 - Muss eine Task beispielsweise zwei Datenstrukturen manipulieren, die durch zwei unabhängige Semaphore geschützt sind, und eine zweite Task muss das Gleiche tun, nur dass sie die Semaphore in umgekehrter Reihenfolge alloziert, ist eine Verklemmung die Folge



Deadlock Wahrscheinlichkeit

- In der Praxis kommen derartige Konstellationen häufig vor.
- Dabei sind sie nur sehr schwer zu entdecken, da das Nehmen und Freigeben des Semaphors nicht selten in Funktionen gekapselt ist.
- Die Erkennung, Vermeidung und Behandlung von Deadlocks gehört zu den schwierigsten Aufgaben bei der Programmierung moderner Realzeitanwendungen.
- Beim Software-Entwurf können Deadlocks unter Umständen durch die Modellierung und Analyse des zugehörigen Petrinetzes entdeckt werden (siehe „Formale Beschreibungsmethoden“).

Deadlocks vermeiden

- Damit in einem System Deadlocks auftreten, müssen folgende vier Bedingungen gleichzeitig erfüllt sein:
 - Ressourcen stehen nur exklusiv zur Verfügung, mehrere Tasks können somit nicht gleichzeitig auf die Ressourcen zugreifen.
 - Eine Task arbeitet mit mehreren Ressourcen gleichzeitig, hält also eine Ressource, während sie auf eine andere wartet.
 - Die Ressource kann einer Task nicht einfach entzogen werden.
 - Bei der Benutzung der Ressourcen der Tasks untereinander entsteht eine zyklische Kette (siehe vorheriges Beispiel).

Schreib- / Lese-Lock

- Da das parallele Lesen unkritisch ist, bieten viele Systeme sogenannte Schreib- / Lese-Locks an.
- Dabei handelt es sich um Semaphore, die parallele Lesezugriffe erlauben, aber einem schreibenden Job einen exklusiven Zugriff ermöglichen.
- Bei der Anforderung des Mutex muss der Rechenprozess mitteilen, ob er den kritischen Abschnitt nur zum Lesen oder auch zum Schreiben betreten möchte.
- Eine Lesezugriff einer Task wird blockiert wenn:
 - Eine andere Task schreibend zugreift oder
 - Eine andere Task wartet, um einen Schreibzugriff zu starten
 - Würde eine Task, die lesen möchte, nicht blockiert, könnte die Task, die schreiben möchte, »verhungern«.

Weitere Schutzmechanismen

- Unterbrechungssperre
 - Hierbei werden die Interrupts oder auch nur die Taskwechsel auf einem System für die Zeit des Zugriffs auf den kritischen Abschnitt gesperrt.
- Spinlock
 - Bei Spinlocks entscheidet – wie schon beim Semaphore – eine Variable darüber, ob ein kritischer Abschnitt betreten werden darf oder nicht. Ist der kritische Abschnitt bereits besetzt, wartet die zugreifende Einheit **aktiv** so lange, bis der kritische Abschnitt wieder freigegeben worden ist

Welchen Schutzmechanismus? (1)

- Die **Unterbrechungssperre** steht normalen Applikationen nicht zur Verfügung. Bei Einprozessorsystemen (Uni-Processor, UP) wird sie im Betriebssystemkern eingesetzt
- **Spinlocks** werden ebenfalls im Kernel eingesetzt.
 - Sie realisieren ein aktives Warten und lassen sich aus diesem Grund im Kernel nur in einer SMP-Umgebung verwenden.
 - Auf Applikationsebene funktionieren sie auch in Einprozessorumgebungen.
 - Spinlocks werden einem Mutex oder Semaphore dann vorgezogen, wenn die Bearbeitungszeit des kritischen Abschnitts deutlich kürzer ist als die Zeit, die für einen Kontextwechsel benötigt wird.

Welchen Schutzmechanismus? (2)

- Das **Semaphor** schließlich ist für den gegenseitigen Ausschluss auf Applikationsebene gedacht.
 - Es ist sowohl für UP- als auch für SMP-Umgebungen verwendbar.
 - Der Einsatz im Betriebssystemkern ist möglich, aber nur dann, wenn der kritische Abschnitt nicht von einer ISR betreten werden soll.
 - In diesem Fall müsste man nämlich die ISR schlafen legen, was nicht möglich ist.

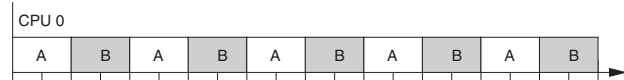
Unterbrechungsmodell allgemein

- Das Unterbrechungsmodell ist verantwortlich für den Schutz kritischer Abschnitte und damit auch für Latenzzeiten im System.
- Es teilt das Betriebssystem in mehrere Ebenen ein.
 - die Ebenen sind unterschiedlich priorisiert
 - durch Zuordnung von Funktionen zu Ebenen wird das Realzeitverhalten generell beeinflusst.
- Systemarchitekt und Programmierer muss das Modell kennen, da nur so kritische Abschnitte identifizieren und effizient geschützt werden können.

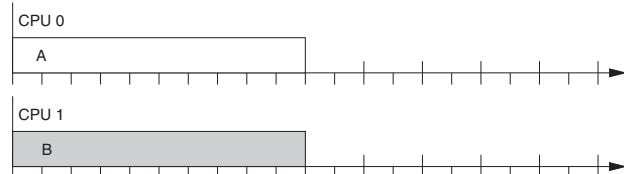
Unterbrechungsmodell: Verarbeitung

Grundsätzlich muss man die quasi-parallelen von der realparallelen Verarbeitung unterscheiden.

quasi-parallele Verarbeitung

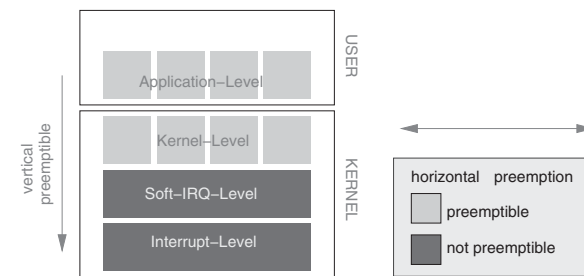


real-parallele Verarbeitung



Unterbrechungsmodell: Ebenen

1. Applikationsebene (Userland)
2. Kernel-Ebene
3. Die Soft-IRQ-Ebene
4. Die Interrupt-Ebene



Applikationsebene

- Auf der Applikationsebene, dem Userland, laufen die Userprogramme, die beispielsweise über Prioritäten gescheduled werden.
- Kritische Abschnitte, wie die Zugriffe auf globale Variablen, werden über Semaphore oder Spinlocks geschützt.

Kernel-Ebene

- Rufen Applikationen Systemcalls auf (beispielsweise *open()*, *read()* oder *clock_nanosleep()*), werden diese auf der Kernel-Ebene im User-Kontext abgearbeitet.
- Der Systemcall hat dabei die Priorität des aufrufenden Jobs.
- Auf gleicher Ebene laufen Kernel-Threads ab, die sich dadurch von User-Threads unterscheiden, indem sie keine Ressourcen (z.B. Speicher) im Userland belegen und damit im Kernel-Kontext arbeiten.
- Kernel-Threads haben eine Priorität und konkurrieren mit User-Threads. Anders als User-Threads werden die Kernel-Threads jedoch nicht unterbrochen.
- Aktuelle Linux-Kernel bieten die Möglichkeit von sogenannten Threaded Interrupts. Hierbei laufen Interrupts als Kernel-Threads, also im Kernel-Kontext ab.

Soft-IRQ-Ebene

- Sind anstehende Hardware ISRs abgearbeitet, gibt der Kernel Interrupts frei und startet Soft-IRQs, die im 'Interruptkontext' abgearbeitet werden.
- Soft-IRQs können nicht unterbrochen werden, können aber auf einer Mehrkernmaschine real-parallel abgearbeitet werden.
- Hier gilt die Einschränkung, dass ein und derselbe Soft-IRQ nicht mehrfach gleichzeitig ablaufen kann.

Interrupt-Ebene

- Höchste Priorität in der Abarbeitungsreihenfolge haben Interrupt- Service-Routinen, wobei Linux standardmäßig alle Interrupts gleich priorisiert (Hardware-Priorität).
- Sobald ein Interrupt auftritt und Interrupts freigegeben sind, werden sie auf dem lokalen Kern gesperrt und die zugehörige ISR wird abgearbeitet.
- Eine quasi-parallele Verarbeitung findet damit nicht statt. Auf einer Mehrkernmaschine können real jedoch mehrere ISRs parallel verarbeitet werden, ein und dieselbe typischerweise jedoch nicht.
- Eine Datenstruktur, die von nur einer ISR verwendet wird, muss daher nicht gesondert geschützt werden.
- Da Funktionen, die im Interrupt-Kontext ablaufen, können nicht schlafen gelegt werden können (keine Semaphore!)

Realzeitprogrammierung

1. Allgemeines
2. Programmtechnischer Umgang mit Tasks
3. Schutz kritischer Abschnitte
- 4. Umgang mit Zeiten**
5. Kommunikation und Signalisierung



Umgang mit Zeiten

Eigenschaften von Zeitgebern

- Genauigkeit
 - Taktrate
 - Schwankungen
 - Zeitsprünge
- Zuverlässigkeit
 - Verhalten bei (bewussten) Schwankungen der Taktung und bei Zeitsprüngen
- Bezugspunkt
 - Intern („Start des Systems“, „Start des Jobs“)
 - Extern („Start der Epoche“ - seit der Geburt Christi)
- Darstellung
 - Datentypen
 - Maximaler Zeitbereich
 - ergibt sich durch die Auflösung des Zeitgebers und die Bitbreite der Variablen

Umgang mit Zeiten

Datentypen

- `time_t`
 - Zeit in Sekundenauflösung
- `clock_t`
 - Timerticks
 - Anzahl Ticks pro Sekunde kann über den Systemcall `_sysconf(...)` ausgelesen werden
- `struct tms`
 - Auflösung Timerticks
 - Accounting, Verarbeitungs- und Reaktionszeiten
- `struct tm`
 - Absolute Zeitangabe (seit Christi Geburt)
 - Auflösung Sekunden

Umgang mit Zeiten

Datentypen: `struct timeval` / `struct timespec`

- | | |
|---|--|
| <ul style="list-style-type: none"> ■ <code>struct timeval</code> <ul style="list-style-type: none"> ■ Mikrosekundenauflösung. ■ Normalisierte Darstellung: Der Mikrosekundenanteil muss kleiner als 1 Millionen sein (kleiner 1 Sekunde). ■ Wertebereich: <ul style="list-style-type: none"> - 32-Bit: etwa 4 Milliarden Sekunden - 64-Bit: über 500 Milliarden Jahre | <ul style="list-style-type: none"> ■ <code>struct timespec</code> <ul style="list-style-type: none"> ■ Nanosekundenauflösung. ■ Normalisierte Darstellung: Der Nanosekundenanteil muss kleiner als 1 Milliarde sein (kleiner 1 Sekunde). ■ Wertebereich: <ul style="list-style-type: none"> - 32-Bit: etwa 4 Milliarden Sekunden - 64-Bit: über 500 Milliarden Jahre |
|---|--|

Umgang mit Zeiten

```

struct timeval {
    time_t      tv_sec;   /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
struct timespec {
    time_t      tv_sec;   /* seconds */
    long        tv_nsec; /* nanoseconds */
};
struct tms {
    clock_t      tms_utime; /* user time */
    clock_t      tms_stime; /* system time */
    clock_t      tms_cutime; /* user time of children */
    clock_t      tms_cstime; /* system time of children */
};
struct tm {
    int tm_sec; /* seconds */
    int tm_min; /* minutes */
    int tm_hour; /* hours */
    int tm_mday; /* day of the month */
    int tm_mon; /* month */
    int tm_year; /* year */
    int tm_wday; /* day of the week */
    int tm_yday; /* day in the year */
    int tm_isdst; /* daylight saving time */
};

```

Umgang mit Zeiten

Wertebereich

- Die Darstellungs- beziehungsweise Repräsentationsform reflektiert auch den darstellbaren Wertebereich.
- Da bei den dargestellten Datenstrukturen für den Typ `time_t` (Sekunden) ein `long` eingesetzt wird, lassen sich
 - auf einer 32-Bit-Maschine rund 4 Milliarden Sekunden (~136 Jahre) zählen,
 - auf einer 64-Bit-Maschine 264 (mehr als 500 Milliarden Jahre).

Umgang mit Zeiten

Verhalten bei Schwankungen und Zeitsprüngen:

- `CLOCK_REALTIME`
 - Systemweite, aktuelle Zeit
 - Reagiert auf Zeitsprünge
 - Reagiert auf schwankende Taktungen (Stichwort NTP)
- `CLOCK_MONOTONIC`
 - Reagiert nicht auf Zeitsprünge
 - Läuft nur vorwärts
 - Reagiert auf schwankende Taktungen (NTP)
- `CLOCK_MONOTONIC_RAW`
 - Linuxspezifisch
 - Reagiert nicht auf Zeitsprünge
 - Reagiert nicht auf geänderte Taktungen (NTP)

Umgang mit Zeiten

| Funktion | Beschreibung |
|----------------------------|--|
| <code>time</code> | Gibt die Anzahl Sekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind. |
| <code>gettimeofday</code> | Gibt die Anzahl Sekunden und Mikrosekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind. |
| <code>clock_gettime</code> | Sekunden und Nanosekunden, die seit dem 1.1.1970 (UTC) vergangen sind. Die Genauigkeit kann per <code>clock_getres()</code> ausgelesen werden. |
| <code>times</code> | Liefert die aktuelle Zeit als Timerticks (Bezugszeitpunkt ist nicht genau definiert); zusätzlich auch die Verarbeitungszeit, die im Kernel und die im Userland angefallen ist. |
| <code>TSC</code> | Der TSC ist ein mit der Taktfrequenz der CPU getakteter, sehr genauer Zähler, auf den per Maschinenbefehle zugegriffen werden kann. Vorsicht: Die Taktfrequenz der CPU kann schwanken. |

Umgang mit Zeiten

Zeit lesen

- `clock_gettime(clockid_t clk_id, struct timespec *tp)`
 - Sekunden und Nanosekunden seit dem 1.1.1970 (Unix-Epoche)
 - Zeitzone: UTC !
 - 32-Bit-Systeme: Überlauf am 19. Januar 2038
 - `Clock_id`: `CLOCK_REALTIME`, `CLOCK_MONOTONIC`
 - `clock_gettime()`: Auslesen der Genauigkeit
 - Die Funktion ist nicht in der Standard-C-Bibliothek zu finden
 - `librt`

Umgang mit Zeiten

Beispiel `clock_gettime()`

```
struct timespec timestamp;
...
if (clock_gettime(CLOCK_MONOTONIC, &timestamp)) {
    perror("timestamp");
    return -1;
}
printf("seconds: %ld, nanoseconds: %ld\n",
       timestamp.tv_sec, timestamp.tv_nsec);
```

Umgang mit Zeiten

Beispiel `time()`

- Sekunden seit dem 1.1.1970 UTC (Unix-Zeit)
- Umrechnung in Absolutzeit per
 - `struct tm *localtime_r(time_t *time_p, struct tm *result);`
 - `struct tm *gmtime_r(time_t *time_p, struct tm *result);`
- Umwandlung von `struct tm` nach `time_t`: `mktime()`;

```
time_t now;
...
now = time(NULL);
```

Umgang mit Zeiten

Zeit lesen

- `gettimeofday(struct timeval *tv, struct timezone *tz)`
 - Sekunden und Mikrosekunden seit dem 1.1.1970
 - `tz` ist ungenutzt (NULL)
 - Umrechnung in Absolut-Zeit (`struct tm`):
 - `struct tm *localtime_r(time_t *seconds, struct tm *result);`
 - `struct tm *gmtime_r(time_t *seconds, struct tm *result);`
- Unterschied zu `clock_gettime()`:
 - Schlechtere Auflösung (Mikro- statt Nanosekunden)
 - Keine Auswahl der Zeitgeber (`CLOCK_MONOTONIC`, `CLOCK_REALTIME`) möglich

Umgang mit Zeiten

Zeit lesen

- `clock_t times(struct tms *buf)`
 - Verarbeitungszeit im Kernel
 - Verarbeitungszeit im Userland
 - Verarbeitungszeiten von Kindtasks
- Reaktionszeit
- Bezugspunkt ist nicht festgelegt (oft „Start des Systems“)
 - Nur für Differenzzeitmessungen geeignet

Umgang mit Zeiten

Zeit lesen

- `sysconf(_SC_CLK_TCK)`
 - kann die Anzahl der Timerticks pro Sekunde ausgelesen werden
 - der Kehrwert gibt dann die Zeitdauer eines Timerticks an

```
ticks_per_second = sysconf(_SC_CLK_TCK);
tickduration_in_ms = 1000/ticks_per_second;

act_time = times( &exec_time );
printf("actual time (in ms): %ld\n",
act_time*tickduration_in_ms);
printf("execution time (in ms): %ld\n", (exec_time.tms_utime
+exec_time.tms_stime)*tickduration_in_ms);
```

Umgang mit Zeiten

Zeit lesen

- Time Stamp Counter (TSC)
 - Zeitgeber, der mit der Taktfrequenz der CPU getaktet wird
 - Sehr genau
 - Nicht von allen Prozessoren unterstützt
 - Wechselnde Taktfrequenzen sind zu beachten (Stichwort Speedstep)

Umgang mit Zeiten

Zeitvergleiche

- Sollen zwei Zeitstempel verglichen werden, die über eine `struct timeval` repräsentiert sind:
 - kann das Makro `int timercmp(struct timeval *a, struct timeval *b, CMP)` eingesetzt werden
 - Für CMP ist der Vergleichsoperator »<«, »>«, »==«, »<=« oder »>=« einzusetzen
- die Vergleiche »<=«, »==« und »>=« sind nicht immer korrekt implementiert
 - für eine portierbare Realzeitapplikation die negierten Varianten verwenden

Umgang mit Zeiten

Zeitvergleiche

- Vergleiche von zwei Absolutzeitstempeln:
 - Über die Repräsentation in Sekunden (Funktion *mktime()*)
- Vergleiche von Relativzeitstempeln, die gesichert nicht weiter als die Hälfte des messbaren Zeitbereiches auseinanderliegen:
 - Vergleich per Makro:
 - *time_after(a,b)*
 - *time_befor(a,b)*
 - Datentypen der Zeitstempel: unsigned long

```
#define time_after(a,b) \
  (typecheck(unsigned long, a) && \
   typecheck(unsigned long, b) && \
   ((long)(b) - (long)(a) < 0))
```

Umgang mit Zeiten

Zeitvergleiche

- Relativzeitstempel, die weit auseinanderliegen
 - Kein Vergleich möglich
 - Ausweichen auf andere Repräsentierungsformen (64-Bit)

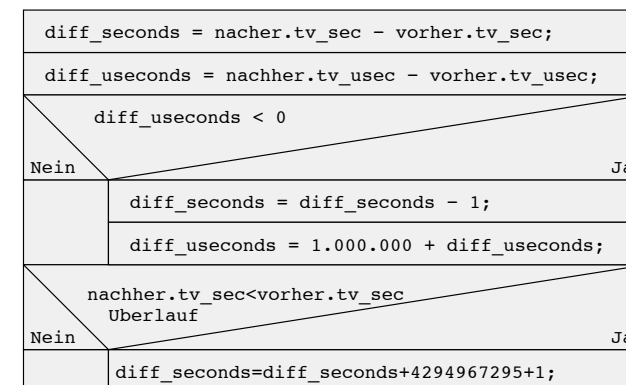
Umgang mit Zeiten

Differenzzeitmessung

- In Realzeitapplikationen ist es häufig notwendig, eine Zeitdauer zu messen, Zeitpunkte zu erfassen oder eine definierte Zeit verstreichen zu lassen.
- Dabei sind folgende Aspekte zu beachten:
 - Die Genauigkeit der eingesetzten Zeitgeber,
 - die maximalen Zeitdifferenzen,
 - Schwankungen der Zeitbasis, beispielsweise durch Schlafzustände,
 - Modifikationen an der Zeitbasis des eingesetzten Rechnersystems (Zeitsprünge) und
 - die Ortsabhängigkeit absoluter Zeitpunkte.

Umgang mit Zeiten

Differenzzeitmessung



Umgang mit Zeiten

Differenzzeitmessung

- **Aufgabe**
 - Messung der Dauer eines zeitlichen Ablaufs
- **Verfahren**
 - 1. Zeitstempel wird zu Beginn genommen
 - 2. Zeitstempel wird zum Abschluss genommen
 - Zeitdauer = 2. Zeitstempel – 1. Zeitstempel
- **Problem**
 - Zeitstempel liegen als `struct timespec` oder `struct timeval` vor
- **Lösung**
 - Sekunden und Mikrosekunden werden getrennt subtrahiert
 - Ist der Mikrosekundenanteil negativ erfolgt eine Korrektur

Umgang mit Zeiten

Beispiel Differenzzeitmessung

```
#define MICROSECONDS_PER_SECOND 1000000

struct timespec * diff_time( struct timeval before,
                             struct timeval *result )
{
    if (result==NULL)
        return NULL;
    result->tv_sec = after.tv_sec - before.tv_sec;
    result->tv_nsec = after.tv_nsec - before.tv_nsec;

    if (result->tv_nsec<0) {
        result->tv_sec--;
        /* result->tv_nsec is negative, therefore we use "+" */
        result->tv_nsec = MICROSECONDS_PER_SECOND+result->tv_nsec;
    }
    return result;
}
```

Umgang mit Zeiten

Differenzzeitmessung: „The poor man's solution“

- Zeitstempel werden in Mikro- oder Nanosekunden umgerechnet, die sich einfach subtrahieren lassen.
- Nachteil:
 - Kommt nicht mit Zählerüberläufen zurecht
 - Zeitstempel müssen nah beieinander liegen
- Lösung ist nicht professionell!

```
timestamp2 = (ts_end.tv_sec*1000000)+ts_end.tv_nsec;
timestamp1 = (ts_start.tv_sec*1000000)+ts_start.tv_nsec;

duration = timestamp2 - timestamp1;
```

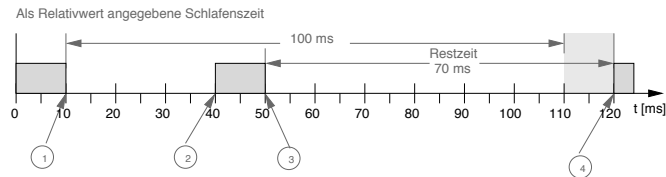
Umgang mit Zeiten

Schlafen

- Jobs werden in den Zustand „schlafend“ versetzt.
- Gründe:
 - Haushalten mit Rechenzeit
 - Zeitgesteuerte Systeme (in Abgrenzung zu ereignisgesteuerten Systemen)
- Zeitangaben zum Schlafen:
 - Relativ („schlafe für 100ms“)
 - Absolut („schlafe bis 5:12 Uhr“)
- Ob relativ oder absolut geschlafen wird, hat durchaus Relevanz!

Umgang mit Zeiten

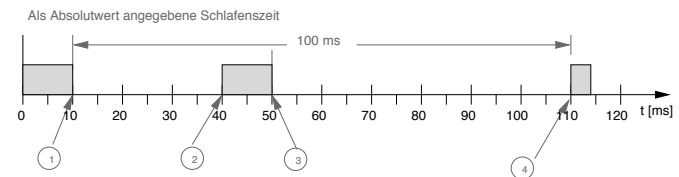
Schlafen mit Relativwert



- 1 Der Job legt sich für 100 ms schlafen.
- 2 Der Job wird unplanmäßig durch ein Signal aufgeweckt. Die Rest-Schlafenszeit beträgt 70 ms.
- 3 Der Job legt sich für die Restzeit (70 ms) schlafen.
- 4 Der Job wacht auf, allerdings um $t_{E, \text{Unterbrechung}}$ später als geplant.

Umgang mit Zeiten

Schlafen mit Absolutwert



- 1 Der Job legt sich bis zum Zeitpunkt $t = 110$ ms schlafen.
- 2 Der Job wird unplanmäßig durch ein Signal aufgeweckt.
- 3 Der Job legt sich wieder bis zum Zeitpunkt $t = 110$ ms schlafen.
- 4 Der Job wacht pünktlich auf.

Umgang mit Zeiten

Schlafen: Auswertung

- „Absolut“-Schlafen hat Vorteile:
 - Schlafende Rechenprozesse werden unter Umständen zwischendurch aufgeweckt (weil beispielsweise ein Signal eintrifft).
 - Anhand des Rückgabewertes der Funktion zum Schlafenlegen ist erkennbar, dass die anvisierte Zeit noch nicht vollständig abgelaufen ist.
 - Der Job wird also auf die Restzeit erneut schlafen gelegt.
 - Das Auswerten und erneute Schlafenlegen kostet Rechenzeit, die sich als Fehler auf die Gesamtschlafenszeit aufaddiert.
 - Beim absoluten Schlafen gibt es diesen Fehler nicht.

Umgang mit Zeiten

„Schlaf“-Funktionen

- `sleep()` – Genauigkeit Sekunden
- `usleep()` – Genauigkeit Mikrosekunden
- `nanosleep()` – Genauigkeit Nanosekunden

```
struct timespec req, rem;
int error;
...
req.tv_sec = 60;
req.tv_nsec = 1000;
while ((error=nanosleep(&req,&rem))!=-1) {
    if (errno==EINTR) {
        req = rem;
    } else {
        perror("nanosleep");
        break;
    }
}
```


Umgang mit Zeiten

Schlafen: Professionelle Lösung

- `clock_nanosleep()`
 - Genauigkeit Nanosekunden
 - Auswahl von Zeitgebern möglich (CLOCK_REALTIME, CLOCK_MONOTONIC)
 - Auswahl "relativ" oder "absolut" schlafen
 - Nicht in der Standardbibliothek, nur in der "librt"

Umgang mit Zeiten

Schlafen: Beispiel für professionelle Lösung

```
void sigint_handler(int signum)
{
    printf("SIGINT (%d)\n", signum);
}

int main( int argc, char **argv, char **envp )
{
    struct timespec sleeptime;
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigint_handler;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror( "sigaction" );
        return -1;
    }
    printf("id going to sleep for 10 seconds...\n", getpid());
    clock_gettime( CLOCK_MONOTONIC, &sleeptime );
    sleeptime.tv_sec += 10;
    sleeptime.tv_nsec += 0;
    while (clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
        &sleeptime, NULL) == EINTR) {
        printf("interrupted...\n");
    }
    printf("woke up...\n");
    return 0;
}
```

Umgang mit Zeiten

Weckruf per Timer

- Funktionen werden periodisch durch das Betriebssystem aktiviert.
- Funktion ist realisiert als:
 - Signal-Handler (SIGEV_SIGNAL) oder
 - Thread (SIGEV_THREAD_ID)
- Die aufzurufende Funktion wird über **struct sigevent** definiert.
- Unterstützt die Zeitgeber CLOCK_MONOTONIC und CLOCK_REALTIME.
- Auflösung: Nanosekunden
- Bereich: >4 Milliarden Sekunden

Umgang mit Zeiten

Weckruf per Timer

- Timer-Objekt **struct itimerspec**:
 - Zeitpunkt des ersten Auftretens (in Sekunden und Nanosekunden)
 - Periode (in Sekunden und Nanosekunden)
- Die Zeitangabe für das erste Auftreten kann
 - absolut (TIMER_ABS) oder
 - relativ (0) sein

```
struct itimerspec {
    struct timespec it_interval; /* Timer interval */
    struct timespec it_value;    /* Initial expiration */
};
```

Umgang mit Zeiten

Weckruf per Timer

```
void timer_function( union sigval parameter )
{
    printf("timer with id %d active\n", getpid());
    return;
}

int main( int argc, char **argv, char **envp )
{
    timer_t itimer;
    struct sigevent sev;
    struct itimerspec interval;

    sev.sigev_notify      = SIGEV_THREAD;
    sev.sigev_notify_function = timer_function;
    sev.sigev_value.sival_int = 99;
    sev.sigev_notify_attributes = NULL;
    if (timer_create(CLOCK_MONOTONIC, &sev, &itimer) == -1) {
        perror("timer_create");
        return -1;
    }
    interval.it_interval.tv_sec = 1;
    interval.it_interval.tv_nsec = 0;
    interval.it_value.tv_sec = 1;
    interval.it_value.tv_nsec = 0;
    timer_settime( itimer, 0, &interval, NULL ); /* activate timer */
    sleep( 5 );
    return 0;
}
```

Realzeitprogrammierung

1. Allgemeines
2. Programmtechnischer Umgang mit Tasks
3. Schutz kritischer Abschnitte
4. Umgang mit Zeiten
5. Kommunikation und Signalisierung



Kommunikation und Synchronisierung

1. Inter-Prozess-Kommunikation
2. Signalisierung
3. Peripheriezugriff
4. Memory Management



Kommunikation und Synchronisierung

1. Inter-Prozess-Kommunikation
2. Signalisierung
3. Peripheriezugriff
4. Memory Management



Inter-Prozess-Kommunikation

- In vielen Realzeitsysteme und in Unix-Systemen ist ein Mailbox-Mechanismus realisiert über
 - Pipes (FIFO) und über
 - Messages
- Shared Memory
- Sockets

Pipes

- Systemcall *pipe()* erstellt zwei Deskriptoren
 - Über 1. Deskriptor lesen per *read()*
 - Über 2. Deskriptor schreiben per *write()*
- Typischerweise wird per *fork()* oder *pthread_create()* eine neue Task erzeugt, die die Deskriptoren erbt.
- Kommunikation zur neuen Task über die beiden Deskriptoren
- Da die Kommunikation über Pipes unidirektional ist, gibt jede der Tasks den Deskriptor wieder frei, der nicht benötigt wird.

Messages

- POSIX sowie System V stellen Messages zur Verfügung
- POSIX Messages
 - einfacheres Interface
 - weniger fehleranfällig
- POSIX Message Interface API stellt synchronen als auch asynchronen Datenaustausch zur Verfügung
 - Name der POSIX Message Queue
 - muss mit einem Slash '/' beginnen, aber keine weiteren Slashes
 - max 255 Zeichen
 - Prototypen in Header `mqueue.h`
 - Funktionen in Bibliothek `librt`

Shared Memory (1)

- Shared Memory ist ein gemeinsamer Speicherbereich zwischen einer oder mehrerer Tasks
 - schneller als Messages, da keine Daten kopiert werden müssen
- Realisierung im Falle von Threads trivial
- Wollen mehrere Prozesse Shared Memory nutzen, müssen sie diesen vom Betriebssystem anfordern
- Speicheradresse des Shared Memory kann dabei bei jedem Prozess unterschiedlich sein
 - Pointer innerhalb von Dateistrukturen daher relativ zum Anfang (oder Ende) des Speicherbereichs angeben
- Globale Variablen: Potenzieller kritischer Bereich!

Shared Memory (2)

- POSIX sowie System V stellen Shared Memory zur Verfügung
- POSIX Shared Memory
 - einfacheres Interface
 - weniger fehleranfällig
- Handhabung durch symbolische Namensgebung wie bei Messages
 - Name des POSIX Shared Memory
 - muss mit einem Slash '/' beginnen, aber keine weiteren Slashes
 - max 255 Zeichen
 - Prototypen in Header `sys/mman.h`
 - Funktionen in Bibliothek `librt`

Sockets (1)

- Die wichtigste Schnittstelle für Inter-Prozess-Kommunikation
- Datenaustausch zwischen Prozessen auf unterschiedlichen Rechnern (verteiltes System)
- Die Socket-Schnittstelle bietet Zugriff zur TCP/IP- und zur UDP- Kommunikation
- Datenaustausch nach Verbindungsaufbau über
 - `read()`
 - `write()`

Sockets (2)

- Die im IP-Protokoll festgelegten Datenstrukturen gehen von einem »Big Endian«-Ablageformat der Variablen (z.B. Integer) aus.
- Das bedeutet, dass eine Applikation, die auf einem Rechner abläuft, der ein »Little Endian«-Datenablageformat verwendet, die Inhalte der Datenstrukturen erst konvertieren muss.
- Dieser Vorgang wird durch die Funktionen `ntohX()` (net to host) und `htonX()` (host to net) unterstützt (X steht hier für »s« oder »l«, also für short oder long).
- Auf einem Rechner mit »Big Endian«-Ablageformat sind die Funktionen (Makros) leer, auf einem Rechner mit »Little Endian«-Ablageformat wird dagegen eine Konvertierung durchgeführt.

Kommunikation und Synchronisierung

1. Inter-Prozess-Kommunikation
2. Signalisierung
3. Peripheriezugriff
4. Memory Management



Signalisierung

- Synchronisierung des Programmablaufs über
 - Condition Variable
 - Signale

Condition Variable

- Mittels eine Condition-Variable kann ein Rechenprozess so lange den Prozessor freigeben, bis eine bestimmte Bedingung erfüllt ist.
- Die Basisoperationen auf Condition-Variablen sind:
 - **Schlafen**, bis die Condition-Variable den Zustand ändert
 - **Setzen** der Condition-Variablen.
- Das Setzen der Condition-Variablen, wenn kein anderer Job darauf schläft, bleibt wirkungslos; das Ereignis wird nicht zwischengespeichert.
- Um die daraus resultierende Deadlock-Situation zu vermeiden, kombiniert man die Condition-Variable mit einem Mutex.

Signale

- Bei Signalen handelt es sich um Software-Interrupts auf Applikationsebene
 - Ein Signal führt zu einer Unterbrechung des Programmablaufs innerhalb der Applikation.
 - Das Programm wird entweder abgebrochen oder reagiert mit einem vom Programm zur Verfügung gestellten Signal-Handler (ähnlich einer Interrupt-Service- Routine).
- Signale können durch eine Applikation ausgelöst werden (Systemcall kill())
- Signale können aber auch durch Ereignisse innerhalb des Betriebssystems selbst ausgelöst werden.
 - Speicherzugriff: Segmentation-Fault

Signale versus Condition Variable

| Signal | Condition-Variable |
|---|---|
| Die Signalisierung kommt asynchron zum Programmablauf und wird asynchron verarbeitet. | Die Signalisierung kommt synchron zum Programmablauf und wird synchron verarbeitet. |
| Charakter einer Interrupt-Service-Routine (Software-Interrupt). | Rendezvous-Charakter |

- Ein Signal führt – wenn nicht anders konfiguriert – zum sofortigen Abbruch eines gerade aktiven Systemcalls.
- Werden im Rahmen einer Applikation Signale verwendet bzw. abgefangen, muss jeder Systemcall daraufhin überprüft werden, ob selbiger durch ein Signal unterbrochen wurde, und, falls dieses zutrifft, muss der Systemcall neu aufgesetzt werden!

Kommunikation und Synchronisierung

1. Inter-Prozess-Kommunikation
2. Signalisierung
3. **Peripheriezugriff**
4. Memory Management



Peripheriezugriff der Anwendung

- Der applikationsseitige Zugriff auf Peripherie erfolgt in mehreren Schritten.
 1. Die Applikation teilt dem OS mit, auf welche Peripherie sie in welcher Art (lesend / schreibend) zugreifen möchte. Ist der Zugriff erlaubt, vergibt das OS einen Deskriptor.
 2. Die Applikation greift mithilfe des Deskriptors auf die Peripherie so oft und so lange wie notwendig zu.
 3. Wenn keine Zugriffe mehr notwendig sind, wird der Deskriptor wieder freigegeben.

Direct-IO versus Buffered-IO

- Wenn Ein- / Ausgabe (*read()* / *write()*) direkt ohne Verzögerung umgesetzt spricht man von **Direct-IO**.
- Funktionen wie beispielsweise *fprintf()*, *fread()*, *fwrite()* oder *fscanf()* gehören zur sogenannten **Buffered-IO**.
 - Bei dieser puffert das Betriebssystem (genauer die Bibliothek) die Daten aus Gründen der Effizienz zwischen.
 - Erst wenn es sinnvoll erscheint, werden die zwischengespeicherten Daten per Systemcall *read()* oder *write()* transferiert.
 - Dies ist beispielsweise der Fall, wenn ein »\n« ausgegeben wird oder wenn 512 Byte Daten gepuffert sind.
- Da nur die Direct-IO-Funktionen die volle Kontrolle über die Ein- und Ausgabe geben, werden in Realzeitapplikationen nur diese für den Datentransfer mit der Peripherie eingesetzt.

Splice Systemfunktion

- Klassische Ein / Ausgabe:
 - Applikation reserviert Buffer und liest per *read()* Daten von der Eingabequelle in diesen Buffer
 - Kernel kopiert hierzu die Daten über den Gerätetreiber zunächst in den Kernel-Speicher und von dort in den von der Applikation bereitgestellten Buffer
 - Applikation sendet dann die Daten per *write()* dem Kernel
 - Kernel speichert noch einmal zwischen und aktiviert dann den Gerätetreiber, damit dieser den eigentlichen Schreibzugriff durchführt
- ***splice()*** kopiert ohne obigen Umweg über das Userland direkt die Daten von Eingabequelle an Ausgabequelle
 - Ein- und Ausgabequellen sind dabei wie üblich als Filedeskriptoren referenziert.

Kommunikation und Synchronisierung

1. Inter-Prozess-Kommunikation
2. Signalisierung
3. Peripheriezugriff
4. Memory Management



3 Aspekte bei Memory Management

- Anwendung muss das **Auslagern** von Speicherseiten (Swapping) verhindern
- Anwendung beugt Verzögerungen vor, die entstehen können, wenn die Realzeitapplikation **Daten auf dem Stack** ablegt (lokale Variablen).
 - Benötigt der Betriebssystemkern hierfür nämlich eine neue Page (Page- Fault) und sind sämtliche Pages belegt, muss er die Seite eines gerade nicht aktiven Jobs auslagern und die frei gewordene Seite der Realzeitapplikation zur Verfügung stellen.
- Anwendung beugt Verzögerungen vor, die entstehen können, wenn die Realzeitapplikation **Daten auf dem Heap** ablegt
 - Sind sämtliche Speicherseiten belegt, kommt es zum Page-Fault und der Kernel lagert die Page eines gerade nicht aktiven Jobs aus.

Schutz vor Auslagern

- Mit `mlock()` werden die Speicherseiten, die zum übergebenen Adressbereich gehören markiert, sodass sie von der Speicherverwaltung des Kernels nicht mehr ausgelagert werden.
- Die Freigabe erfolgt später über `munlock()`
- In Realzeitapplikationen wird jedoch vorwiegend die Variante `mlockall()` verwendet, die je nach Flag entweder
 - die aktuell vom Prozess verwendeten Speicherseiten (MCL_CURRENT) oder
 - auch die zukünftigen Speicherseiten (MCL_FUTURE) markiert
- In den meisten Fällen dürfte die Kombination MCL_CURRENT|MCL_FUTURE Anwendung finden.

Prefault

- Mit Prefault wird die Technik umschrieben, mit der ein Betriebssystemkern veranlasst wird, später benötigte Speicherseiten vorzeitig in den Adressraum der Realzeitapplikation einzubinden.
- Der Trick besteht darin, direkt zu Beginn der RT-Applikation sämtlichen Speicher zu reservieren, der später benötigt wird.
 - Das betrifft sowohl den Stack
 - als auch den Heap
- Beispiele siehe Buch
- Wichtig ist, den Speicher nicht nur anzufordern, sondern auch einmal auf den Speicher zuzugreifen.



Vielen Dank

Fragen?

Professor Dr. Michael Mächtel