Betriebssysteme I

Sequentielle und eng gekoppelte parallele Systeme

Kapitel 9: Realzeitsysteme

Stand: WS 10/11 (25.01.11)

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

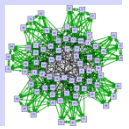
Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

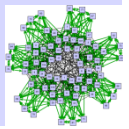
Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

SR



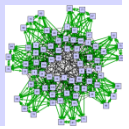
Realzeitsysteme

- Betrachteter Bereich:
  - Betriebssysteme für Anwendungen, die in Echtzeit reagieren und kommunizieren müssen.
  - Beschränkung auf Funktionalität, die in dem POSIX.4 Standard spezifiziert ist und eine UNIX Erweiterung darstellt.
- Echt-zeit = recht-zeitig
  - nicht unbedingt schnell, aber: zuverlässig und termingerecht
  - „harte“ Realzeit: verspätetes Resultat ist völlig nutzlos
    - 100% der Antworten bis zur deadline benötigt
  - „weiche“ Realzeit (*soft realtime*) : verspätetes Resultat ist weniger wert als rechtzeitiges Resultat
    - Antwortzeiten statistisch um deadline verteilt



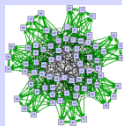
Realzeitsysteme

- Realzeitprozess
  - Prozess, dessen Ergebnis zu einer bestimmten, realen (system-unabhängigen) Zeit t vorliegen muss, um nützlich zu sein.
  - Üblicherweise: Prozess, der direkt ein Gerät steuert bzw. über Eingaben und Ausgaben direkt mit der Umwelt kommuniziert.
- Beispiele:
  - Kein RT-Prozess: Wetterprogramm, auch wenn dieses früh morgens fertig sein muss.
  - Weicher RT-Prozess: Steuerung eines Bankautomaten
  - Harter RT-Prozess: integrierte Motorsteuerung, Raketenabwehr
- Der RT-Bereich umfasst also alle Gerätesteuerungen von Flugzeugen (fly-by-wire), Bremssystemen (ABS, ESP), Industrierobotern, bis hin zu Bankautomaten und Multi-Media Spielgeräten.



Realzeitsysteme

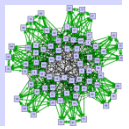
- Klassisches Szenario:
  - Microcontroller oder Spezialhardware mit speziellen, proprietären, maßgeschneiderten Betriebssystemen (klein wegen beschränktem Speicher und schnellem Prozesswechsel)
  - VxWorks, QNX, RT-Linux, …
  - OSEK (Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen)
- Heute: Ausdehnung auf Workstations
  - Prozessoren schneller und Speicher größer → Standard- statt Spezialsysteme
  - Multimedia erfordert auf Arbeitsplatzrechner auch RT



Realzeitsysteme

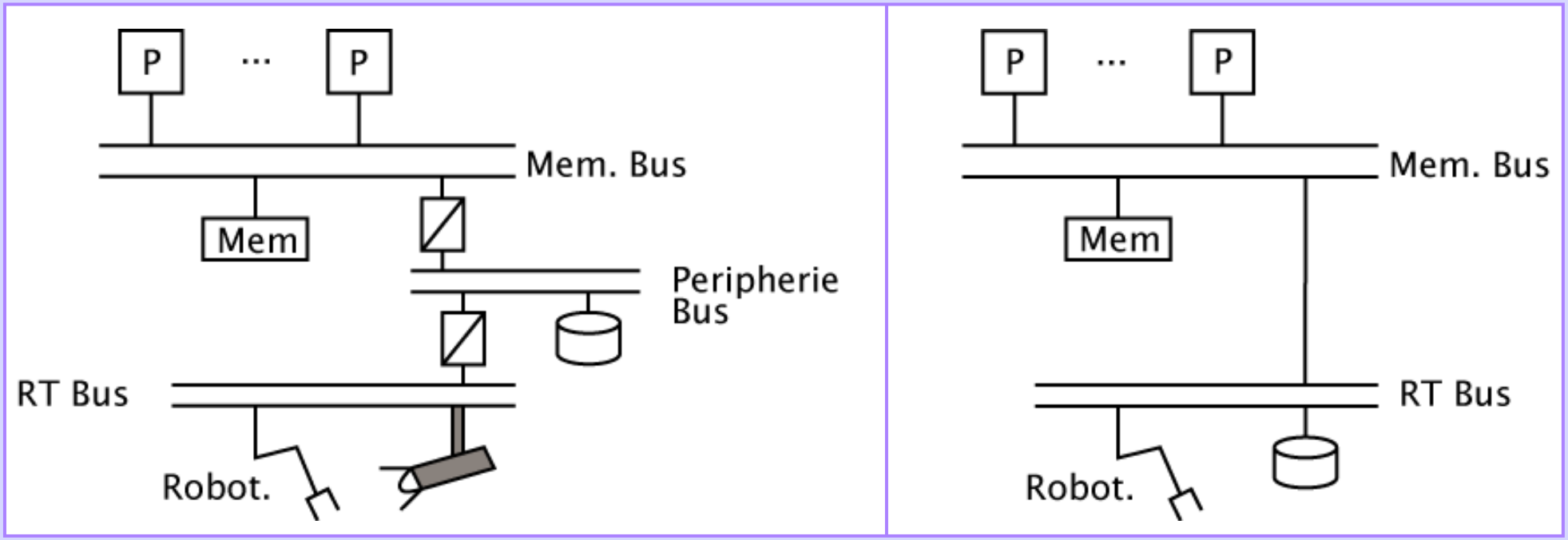
Typisch für den RT-Bereich sind

- Parallelität
  - da oft mehrere Geräte gleichzeitig zu steuern sind
  - (z.B. Ein- und Ausgaben, mehrere Motoren in einem Roboter etc.)
- Prioritäten
  - die vom Benutzer in einer Hierarchie selbst definiert werden, bis hin zur höchsten Systempriorität.
  - Prioritäten in *allen* Warteschlangen im System.
- Paralleles Betriebssystem (*reentrant code*)
  - mit möglichst kurzen kritischen Abschnitten
- Feingranulare Interprozesskommunikation auf Benutzerebene.
- Leichte Prozesse
  - um Prozessumschaltzeiten möglichst kurz zu halten
- Asynchrone I/O
  - damit ein Prozess nicht unvorhersehbar lange in I/O aufgehalten werden kann



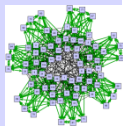
Realzeitsysteme

- Ein für Multiprozessoren geeignetes BS (mit leichten Prozessen und evtl. Mikrokern) erfüllt bereits die meisten Anforderungen an ein RT-System.
- Annahme im Folgenden: Standardsystem (Workstation), bei dem die RT-Geräte durch einen zweiten Bus angesteuert werden.
- 2 Varianten:



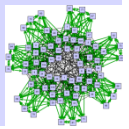
Realzeit IPC – POSIX

- Kommunikation in einem RT-System
  - muss sicher und schnell sein
  - portabler Standard muss auch einfach sein
- POSIX *(Portable OS Interface for UNIX based systems)*
  - verschiedene Funktionalitäten werden spezifiziert, die Implementierung liegt beim BS
- POSIX.4 (Realzeit-Erweiterungen).
  - Erweitert die UNIX Signale, definiert Messages, Shared Memory und Semaphore neu, die in System V zu kompliziert und langsam sind. Führt die RT Scheduling Klasse ein und definiert FIFO und RR-Scheduling.
  - siehe Gallmeister: POSIX.4. O 'Reilly 1995.
- Überblick:
  - POSIX.1 Grundlegende BS-Schnittstellen (neu 1003.1 – 1990)
  - POSIX.2 Kommando-Sprache (sh etc.)
  - POSIX.4 Realzeit-Erweiterungen (1003.1b – 1993)
  - POSIX.4a Threads-Erweiterungen (1003.1c – 1994)
  - POSIX.4b Weitere Realzeit-Erweiterungen (1003.1d)



Realzeit IPC – Probleme mit POSIX.1 Signalen

- Zu wenig Signale
  - in POSIX.1 nur SIGUSR1 und SIGUSR2
  - In POSIX.4 gibt es RTSIG_MAX (mind. =8) Realzeitsignale mit Signalnummern zwischen SIGRTMIN und SIGRTMAX.
- keine Priorität.
  - Werden mehrere Signale deblockiert, ist undefiniert, welches zuerst ausgeliefert wird.
  - POSIX.4: Signale haben eine über die Signalnummer definierte Priorität von SIGRTMAX (niederste) bis SIGRTMIN (höchste).

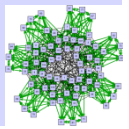


Realzeit IPC – Probleme mit POSIX.1 Signalen

- Signale können verloren gehen
  - ... falls ein zweites Signal eintrifft, bevor das erste Signal bearbeitet werden konnte
  - Es kann nur ein Informationsgehalt von einem Bit direkt übertragen werden, da pro Signal nur ein Bit gesetzt wird.
    - z.B. Temperatur steigt – aber nicht um wie viel
  - POSIX.4 RT-Signale werden aufgereiht (*queued*) und können zusätzlich einen Wert in Wortgröße übertragen.
    - Eine Handler-Funktion hat dann den Prototyp

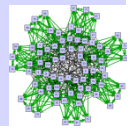
```
void handler(int signum, siginfo_t* data)
```

Entsprechend nimmt die Funktion sigqueue, die kill ersetzt, einen siginfo_t* als entsprechendes Argument.



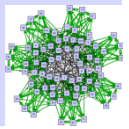
Realzeit IPC –Signale

- UNIX Signale sind ein asynchrones Kommunikationsmittel.
  - Prozess kann ein Signal empfangen, ohne dass er explizit darauf gewartet hat.
  - Probleme durch Asynchronität: Wenn man z.B. zwischen Signal Handler und Haupt-Thread synchronisieren muss, um race-conditions zu vermeiden.
  - Asynchronität verursacht Kosten, denn Umschaltung zum Handler durch das BS ist teurer als ein normaler Funktionsaufruf.



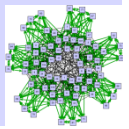
Realzeit IPC –Signale

- Geringerer Aufwand möglich, falls Signale nur Mittel zur Synchronisation zwischen Prozessen sein sollen
  - sigwaitinfo() blockiert, bis eines der Signale aus einer gegebenen Menge ansteht und liefert seine Nummer ab. Es wird kein Signalhandler gerufen, sondern dies bleibt dem rufenden Prozess überlassen, der ja ohnehin gewartet hatte.
  - Man vergleiche dazu sigsuspend(), das blockiert, bis ein unmaskiertes Signal eintrift. Erst ruft das BS den Signalhandler, und nachdem dieser fertig ist, wird der Prozess deblockiert.
  - Variante zu sigwaitinfo(): sigtimedwait() mit timeout.
  - Vermeiden von race-conditions mit Signal-Handlern möglich, da POSIX.4-Signale gespeichert werden: Signal wird während eines KA maskiert und danach wird mit sigtimedwait() nachgesehen, ob es in der Zwischenzeit eingetroffen ist (falls nein, vermeidet man Warten durch sofortigen timeout).



Realzeit IPC – Messages

- POSIX.4: Nachrichtenwarteschlangen (*message queues*) mit Prioritäten der Nachrichten.
  - Grundfunktionalität ähnlich System V, aber Schnittstelle einfacher zu benutzen, da analog zu file-handling aufgebaut
  - Warteschlangen sind einfacher zu implementieren, da z.B. pro WS max. Anzahl und max. Größe der Nachrichten spezifiziert wird.

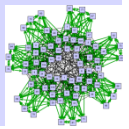


Realzeit IPC – Messages

- Überblick über die Funktionalität:

```
/* Öffnen; die letzten zwei Argumente nur für das Kreieren */
mqd_t mq_open(const char *mq_name, int oflag,
        mode_t create_mode, struct mq_attr *create_attrs);

/* Schließen */
int mq_close(mqd_t mqueue);

/* Operation */
int mq_send(mqd_t mymq, const char* msqbuf, size_t msgsize,
        unsigned int msgprio);
i=mq_receive(mqd_t mymq, const char* msgbuf,
        size_t * msgsize, unsigned int* msgprio);
/* receive oldest msg of highest priority. Blocks unless queue status was
   nonblocking. set attr, get attr manipulieren Status. */
```

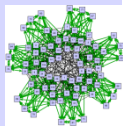
- Mit mq_notify() kann eine Queue beauftragt werden, einem Prozess ein Signal zu schicken, wenn eine Nachricht eintrifft.



POSIX.4 Semaphore

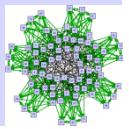
- Zwei Arten von Semaphoren, benannte und namenlose.

```
/* Memory-based (unnamed) semaphores */
int sem_init (sem_t* semaphore_location,
              int pshared,     // 1 for inter-process, else 0.
              unsigned int initial_value);

int sem_destroy (sem_t* semaphore_location);

/* Named semaphores */
sem_t* sem_open(const char* semaphore_name,
                int oflags,                    // O_CREAT or O_EXCL
                mode_t creation_mode,          // like file permission
                unsigned int initial_value);
int sem_close(sem_t* semaphore);
int sem_unlink(const char* semaphore_name);
```



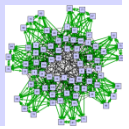
POSIX.4 Semaphore

```
/* Semaphore operations (for both kinds) */
int sem_wait(sem_t* semaphore);

int sem_trywait(sem_t* semaphore);

int sem_post(sem_t* semaphore);

int sem_getvalue(sem_t* semaphore, int* value);
```



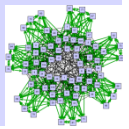
POSIX.4 Semaphore mit Namen

- systemweite Ressource des BS
- existierende Semaphore werden mit Namen vor dem ersten Gebrauch geöffnet.
  - oflag O_CREAT: Semaphor soll neu geschaffen werden, falls es noch nicht existiert;
  - oflag O_EXCL: Aufruf soll mit Fehler (-1) abbrechen, falls das Semaphor schon existiert.
  - creation_mode spezifiziert grundsätzliche Rechte am Semaphor.
    - Makros S_IRWXU, S_IRWXG und S_IRWXO machen Semaphor für user, group und others benutzbar.
- Wait, Post und Try_wait funktionieren wie erwartet
  - down, up, down mit timeout
  - Warten ist priorisiert, d.h. der wichtigste Prozess bekommt das Semaphor als nächstes.
- Getvalue kann eingesetzt werden, um den Zustand des Semaphors auszugeben (→ Fehlersuche)



POSIX.4 speicherbasierte Semaphore

- Ressource des Prozesses
- werden im Speicher angelegt durch `sem_init`.
  - `pshared` == 1 : Gebrauch zwischen Prozessen
    `pshared` == 0 : Gebrauch zwischen Threads innerhalb eines Prozesses.
  - Weiteres Argument: Zeiger auf die Speicherstelle, an der das Semaphor initialisiert werden soll. → Semaphor kann in Datenstrukturen eingegliedert werden.



Shared Memory

- Gemeinsamer Speicher wird logisch als File angesehen, das mittels mmap in den HSP abgebildet wird.
  - SM-Objekt wird deshalb geöffnet, abgebildet, geschlossen und evtl. zerstört.
  - Offene und abgebildete Speicherobjekte werden bei fork() vererbt und sind dann wie Files beiden Prozessen zugänglich.

Öffnen

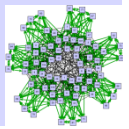
```
int shm_open(const char* name, int oflag, mode_t mode);
```

- gibt File-Deskriptor zurück, der das SM-Objekt bezeichnet.
- Analog zu open:
  - oflag O_CREAT : Objekt wird, wenn nötig, erzeugt
  - mode : spezifiziert Zugriffsrechte wie O_RDWR oder O_RDONLY.

Größe bestimmen

```
int ftruncate(int fd, off_t total_size);
```

- setzt die Größe des Objektes fd fest (für alle Benutzer des Objekts).
- Ein Ergebnis < 0 zeigt Fehler an.



Shared Memory – Aufräumen

```
int close(int fd)
```

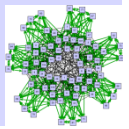
- schließt den Deskriptor eines SM-Objektes.
- Das SM-Objekt selbst wird aber nicht entfernt.

```
int shm_unlink(const char* name);
```

- markiert das SM-Objekt zur Zerstörung vor.
- Die Zerstörung findet statt, wenn jeder beteiligte Prozess die Objekt-Abbildung aufgehoben hat.

```
int munmap(void* begin, size_t length);
```

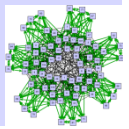
- hebt die Abbildung der Seiten auf, die die Adressen von begin bis begin + length enthalten.
- Sinnvoll: Speicherbereich auf Seitengrenzen beginnen und enden lassen.



Shared Memory – Speicherabbildung

```
void* mmap(void* where,          // Wunsch 0=anywhere
           size_t length,
           int memory_protections, // Zugriffsrechte
           int mapping_flags,      // Shared, private, fixed
           int fd,                 // Objekt
           off_t offset            // im Objekt
);
```

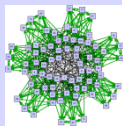
- length Bytes, beginnend bei offset im Objekt fd, werden in den HSP abgebildet, wenn möglich beginnend mit Adresse where.
- Rückgabewert: Die tatsächlich gewählte Adresse
- Zugriffsrechte in der MMU:
  - PROT_READ, PROT_WRITE, PROT_EXEC oder PROT_NONE
  - EXEC erlaubt lediglich das Ausführen von Code,
  - NONE erlaubt nichts (Anlegen von Seiten, die als Brandmauern Speicherbereiche abgrenzen)
- mapping_flags:
  - shared: bestimmt Gebrauch zu shared memory,
  - fixed: where nicht Wunsch, sondern Befehl
  - private: verhindert, dass Änderungen in das SM-Objekt zurückgeschrieben werden.



Shared Memory – Seitenschutz

```
int mprotect(void* begin, size_t length,
             int memory_protections);
```

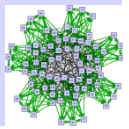
- MMU Schutzbits einzelner Seiten können individuell bestimmt werden.
  - Bsp: Stacksegment abschließen durch eine Seite, auf die nicht zugegriffen werden darf.
  - Art Brandmauer gegen das nächste Stacksegment → läuft ein Pointer über und in die geschützte Seite hinein, wird das Signal SIGSEGV erzeugt.



Shared Memory – Synchronisation

```
int msync(void* begin, size_t length, int flags);
```

- Falls abgebildetes Objekt tatsächlich eine Datei ist, kann es nützlich sein, Datei und HSP Bild zu synchronisieren.
- Flags:
  - MS_SYNC: Aufruf blockiert, bis Datei und Bild synchron sind.
  - MS_ASYNC: Synchronisation wird eingeleitet, aber deren Ende nicht abgewartet.
  - MS_INVALIDATE: Entsprechende Seiten in jedem anderen beteiligten Prozess werden invalidiert und mit dem synchronisierten Bild nachgeladen.

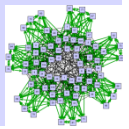


Realer Hauptspeicher

- Seitentauschaktivitäten können ein RT-System empfindlich stören.
  - Seitenfehler verlangsamt Speicherzugriff um einen Faktor 10^5 bis 10^6.
  - Seitentauschverhalten des BS ist nicht vorhersehbar
  - RT-Prozess wird von anderen Prozessen abhängig.
- In POSIX.4 kann man einen Prozess ganz oder in Teilen im HSP festnageln.

```
int mlockall(int flags)
```

  - nagelt den ganzen Prozess fest – Text, Daten, Heap, Stack, shared libs, shared memory.
  - Flags: MCL_CURRENT und MCL_FUTURE
    - Welche mappings sollen mit einbezogen werden – die jetzt vorhandenen oder die zukünftigen?



Realer Hauptspeicher

int munlockall(void)

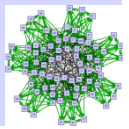
- hebt die Festlegung des Prozesses wieder auf.

int mlock(void* address, size_t length)

- Eine Festlegung von Teilen des Speichers (Bereich von address bis address + length).

int munlock(void* address, size_t length)

- befreit den Bereich von address bis address + length.
- Die Festlegung einer Seite wird in einem Bit gespeichert, so dass ein unlock beliebig viele Festlegungen der Seite aufhebt.

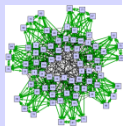


Scheduling – Lösungen in Standard UNIX

- Als Super-User: Mit nice Priorität eines Prozesses erhöhen.
  - Prozess läuft statistisch gesehen besser/schneller.
  - UNIX Scheduler nicht im einzelnen spezifiziert → keine Garantie, dass Prozess immer läuft, wenn er lauffähig ist.
    - nice ist nur einer von mehreren Parametern für die effektive Prozesspriorität
- bessere, aber umständliche Lösung in System V:

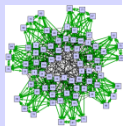
```
long priocntl(idtype_t idtype, id_t id, ind cmd, ... )
```

  - idtype = P_PPID: Priorität aller Kinder des Prozesses id wird gesetzt
  - idtype ≠ P_PPID: Priorität von Prozess id selbst wird gesetzt
    - Weitere Optionen beziehen sich auf Gruppen-, Sitzungs- oder User-Id oder auf jeden Prozess im System.
  - cmd: Informationen über Prozesspriorität bekommen oder Priorität setzen.



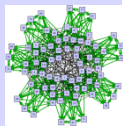
Real-Time Scheduling – Lösungen in Standard UNIX

- Drei Schedulingklassen:
  - Real-Time, System, Time-Sharing.
- Wichtig: RT liegt über System
  - RT-Prozess wird nicht durch System-Aktivitäten unterbrochen, sondern lediglich durch andere RT-Prozesse höherer Priorität



Real-Time Scheduling – POSIX.4 RT Scheduling

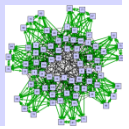
```
#include <unistd.h>
#include <sched.h>
int i, policy;
struct sched_param scheduling_parameters;
pid_t pid;

int sched_scheduler(   pid_t pid,     //process id
                       int policy,    //Algorithmus bzw. Verfahren
                 struct sched_param* scheduling_parameters);

int sched_getscheduler(pid_t pid);

int sched_getparam(pid_t pid,
                 struct sched_param* scheduling_parameters);
```



Real-Time Scheduling – POSIX.4 RT Scheduling

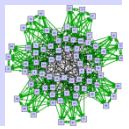
```
int sched_setparam(pid_t pid,
                   struct sched_param* scheduling_parameters);

int sched_yield(void);

int sched_get_priority_min(int policy);

int sched_get_priority_max(int policy);
```

- Scheduling Parameter werden in einer Struktur abgelegt;
- momentan ist in sched_param als einziges Feld
  `int sched_pirority;`
  definiert.
- Später können weitere hinzu kommen.



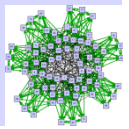
Real-Time Scheduling – POSIX.4 Scheduling Verfahren

- Drei bekannte Verfahren (*policy)*
  1. SCHED_FIFO:
     - *preemptive, priority-based scheduling*
     - Prioritätsbasierte Ablaufplanung mit Prozessorentzug nur durch Prozesse höherer Priorität.
  2. SCHED_RR:
     - *round-robin, preemptive priority-based scheduling with quanta*
     - Prioritätsbasierte Ablaufplanung mit Prozessorentzug sowohl durch Prozesse höherer Priorität als auch durch Prozesse gleicher Priorität nach Ablauf einer Zeitscheibe.
  3. SCHED_OTHER:
     - *implementation defined scheduler*
     - In der Praxis z.B. der übliche UNIX timesharing Scheduler.



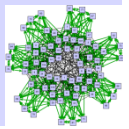
Real-Time Scheduling – POSIX.4 Scheduling Verfahren

- Am wichtigsten für RT: SCHED_FIFO.
  - Prozess kann seine Arbeit zu Ende bringen, wenn er nicht selbst blockiert, einem wichtigeren ein Signal schickt oder ein wichtigerer durch externe HW-Interrupts lauffähig wird.
  - Wichtige Eigenschaft, denn ein RT-Prozess sollte nicht unterbrochen werden, so lange er Geräte, E/A-Kanäle oder andere Ressourcen exklusiv reserviert hat.
- SCHED_RR ist ähnlich zu SCHED_FIFO
  - System V: Zeitscheibe kann durch den Benutzer gesetzt werden.
  - Mit Quantum TQ_INF erhält man das Äquivalent zu SCHED_FIFO.
  - Mit `sched_rr_get_interval(pid_t pid)` kann man das momentane Quantum erfahren.
  - eignet sich gut für Hintergrundprozesse (z.B. logging), die während der RT-Verarbeitung tätig sind, die Maschine aber nicht blockieren dürfen.



Real-Time Scheduling – RT Scheduling Theorie

- Zur Bestimmung der Prioritäten in einem RT System gibt es zwei wichtige Verfahren:
- *earliest deadline first* (EDF):
  - Es wird immer derjenige Prozess als nächstes auf den Prozessor gebracht, der am ehesten fertig sein muss. → *feasible* & optimal.
    - Ein Verfahren ist *feasible*, wenn es immer dann einen Laufplan unter Einhaltung der Beschränkungen generiert, falls dies überhaupt möglich ist.
  - wird von POSIX.4 nicht direkt unterstützt.
- *rate-monotonic scheduling* (RMS):
  - in der Praxis am häufigsten angewendete Verfahren
  - Entspricht einer Näherungslösung zu EDF
  - Dem Prozess mit der höchsten Frequenz an Rechenperioden wird die höchste Priorität gegeben.
    - Priorität kann mit POSIX.4 `sched_setparam` gesetzt werden.



Real-Time Scheduling – RT Scheduling Theorie

- Bestimmen von Prozessprioritäten:
  - bestimmen, wie häufig der Prozess ablaufen muss
    - z.B. muss er 50mal/sec einen Messwert übernehmen
  - Frequenzen absteigend sortieren und Prioritäten in dieser Reihenfolge vergeben
- Idee: Planung derart, dass möglichst wenige Termine verpasst werden.
  - Hochfrequente Prozesse haben die häufigsten Termine, bekommen also die höchste Priorität.
  - Bei hochfrequentem Prozess ist die Wahrscheinlichkeit am größten, dass er bald einen Termin hat, also sowieso nach EDF eingeplant werden müsste.