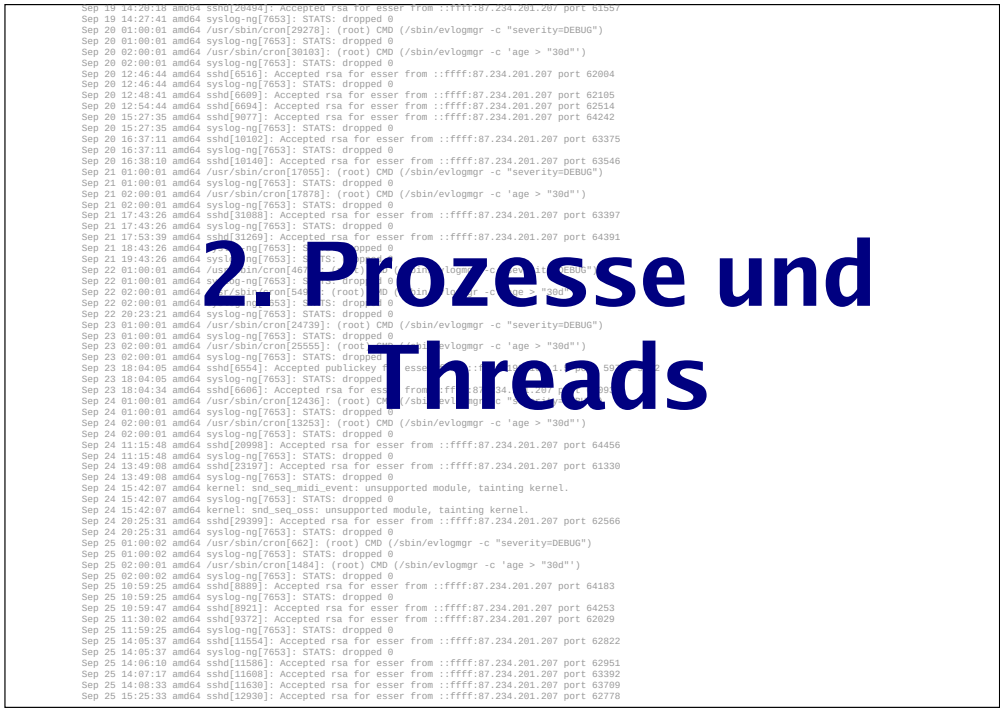
Betriebssysteme I

FH München – WS 2006/07
Hans-Georg Eßer

Zusammenfassung (1/2)

/home/esser/Daten/Dozent/Folien/bs-esser-23.odp



Gliederung

2. Prozesse und Threads
3. Interrupts
4. Scheduler
} **heute**

5. Synchronisation
6. Interprozess-Kommunikation (IPC)
7. Deadlocks
} Mittwoch

BS II: 8. Speicherverwaltung — Dienstag

Prozesse und Threads

Prozess: Programm, das in den Speicher geladen wurde und ausgeführt wird / werden soll
Mehr als nur der Programmcode:

- Eigener Adressraum
- Stack, Stack-Pointer
- Programmzähler
- Umgebung
- Hardware-Register

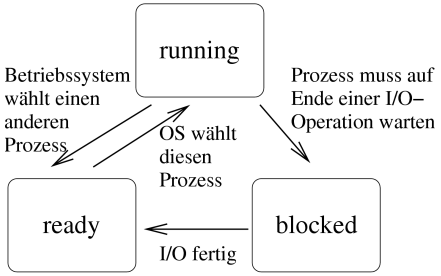
- **Process Control Block** (**PCB**):
  - Identifier (PID)
  - Registerwerte inkl. Befehlszähler
  - Speicherbereich des Prozess
  - Liste offener Dateien und Sockets
  - Informationen wie Vater-PID, letzte Aktivität, Gesamtlaufzeit, Priorität, ...

Thread: ähnlich wie Prozess, aber:

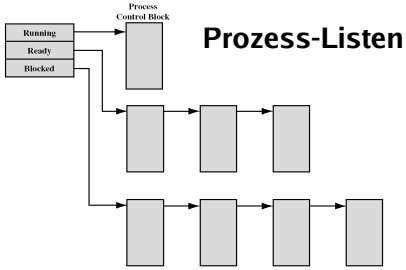
- mehrere Threads greifen auf gleichen Speicher zu
- Thread-Verwaltung nicht unbedingt im Kernel (➞ weniger Verwaltungs-Overhead)
- User level / Kernel level Threads, Mischmodell

Prozess-Zustände

- **laufend / running:** gerade aktiv
- **bereit / ready:** würde gerne laufen
- **blockiert / blocked / waiting:** wartet auf I/O



- **suspendiert:** vom Anwender unterbrochen
- **schlafend / sleeping:** wartet auf Signal (IPC)
- **ausgelagert / swapped:** Daten nicht im RAM



Prozess-Listen

Prozess-Hierarchien / fork/exec

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

Prozesse unter Linux/Unix:

fork (): dupliziert aktiven Prozess
- Vater / Sohn (parent / child)
- Vater erhält bei fork() die PID des Sohnes zurück
- Sohn erhält bei fork() den Wert 0 zurück

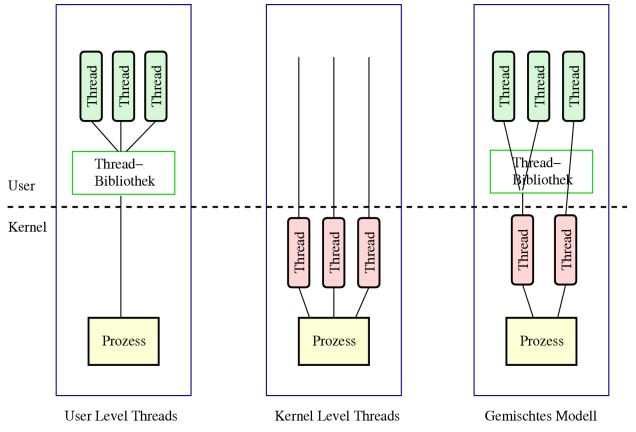
exec (): fremdes Programm in laufenden Prozess laden
- überschreibt aktuellen Code
- exec () kehrt darum nie zurück

wait (): auf einen Sohnprozess warten

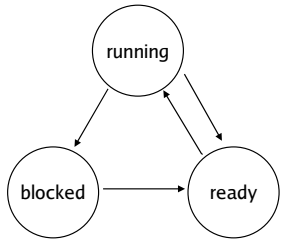
Threads

- einer von mehreren Aktivitätsstrangen in einem Prozess
- Gemeinsamer Zugriff auf Daten des Prozess
- aber: Stack, Befehlszähler, Stack, Stack Pointer, Hardware-Register separat pro Thread
- Prozess-Scheduler verwaltet Threads – oder nicht (Kernel- oder User-level-Threads)
- Multi-Prozessor-System: Mehrere Threads echt gleichzeitig aktiv
- Ist ein Thread durch I/O blockiert, arbeiten die anderen weiter (nur bei Kernel Level Threads)
- Besteht Programm logisch aus parallelen Abläufen, ist die Programmierung mit Threads einfacher

Threads



Thread-Zustände



Prozess-Zustände suspended, sleeping, swapped etc. nicht auf Threads übertragbar

POSIX-Threads

1. Thread-Funktion definieren:

```
void *thread_funktion(void *arg) { ... }
```

2. Thread erzeugen (und sofort starten):

```
pthread_t thread;
pthread_create( &thread, NULL, thread_funktion, NULL) );
```

3. auf Thread warten:

```
pthread_join( thread, NULL);
```

Keine Verwandtschaft (Vater/Sohn)

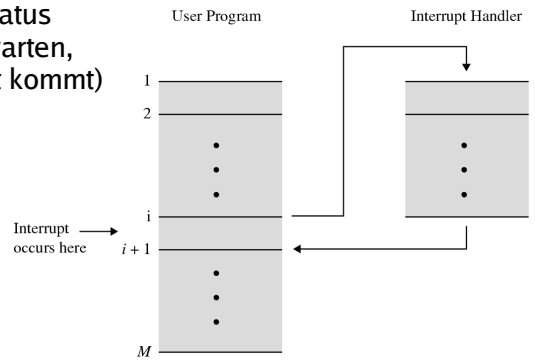
3. Interrupts

Prozesse / Threads unter Linux

- Prozesse und Threads tauchen in gleicher **Task-Liste** auf (beides sind Tasks mit untersch. Eigenschaften, z.B. Speicher)
- anders gesagt: Thread ist Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt
- Linux sortiert Prozesse in **Prozessgruppen** und **Sessions**
  - möglich, Signale an alle Mitglieder einer Proz.gr. / Session zu schicken
- In normaler Prozessliste tauchen Threads nicht auf (aber das ist willkürlich; über geeignete *ps*-Parameter sieht man sie)

Interrupts

- **Effizienz** (I/O-Zugriff sehr langsam ➞ sehr lange Wartezeiten, wenn Prozesse warten, bis I/O abgeschlossen ist)
- **Programmierlogik** (Nicht immer wieder Gerätestatus abfragen, sondern abwarten, bis passender Interrupt kommt)
- **Kein Polling** (BS fragt regelmäßig bei allen Geräten nach, ob ein Ereignis stattgefunden hat)



Interrupts

Was tun bei Mehrfach-Interrupts?

- Während Abarbeitung eines Interrupts alle weiteren ausschließen (DI, disable interrupts) ➞ Interrupt-Warteschlange
- Während Abarbeitung andere Interrupts zulassen
- Interrupt-Prioritäten: Nur Interrupts mit höherer Priorität unterbrechen solche mit niedrigerer

Multitasking und Interrupts

- Multitasking verbessert CPU-Nutzung:
  - I/O-lastiger Prozess wartet auf I/O-Events,
  - CPU-lastiger Prozess rechnet weiter
- Prozess stößt I/O-Operation an und legt sich schlafen (wartet auf Signal)

System Calls

- Argumente über Register übergeben
- Maschinenbefehl `int 0x80` ausführen ➞ Trap (Software Interrupt), Wechsel in Kernel Mode
- Funktion `system_call` in *arch/i386/kernel/entry.S* ausführen
- System Calls in Syscall-Tabelle definiert
- z. B. open(), read(), write(), close() für Dateizugriff

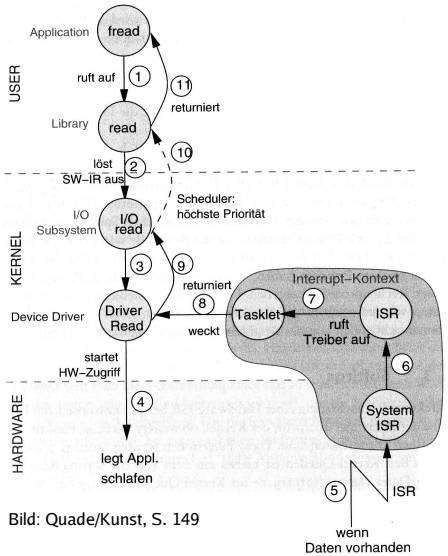


Bild: Quade/Kunst, S. 149

Linux-Interrupt-Handler

Für jedes Gerät:

- Interrupt Request (IRQ) Line
- Interrupt Handler (Interrupt Service Routine, ISR) ➞ Teil des Gerätetreibers
- läuft in speziellem Context (Interrupt Context)

top half

- Interrupt Handler startet sofort, erledigt zeitkritische Dinge
- bestätigt (der Hardware) den Erhalt des Interrupts, setzt Gerät zurück etc.

bottom half / Tasklet

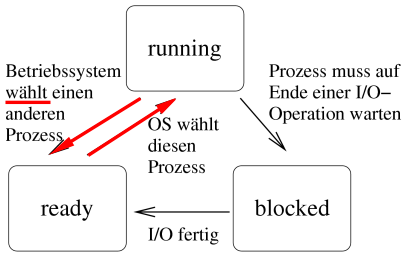
- startet später, macht die eigentliche Arbeit
- kein Prozess (struct tasklet_struct), läuft direkt im Kernel; im Interrupt-Context
- Zwei Prioritäten: *tasklet_hi_schedule*, *tasklet_schedule*

4. Scheduling

```
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[29278]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 /usr/sbin/cron[30103]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[6516]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:46:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:48:41 amd64 sshd[6609]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd[6694]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:35 amd64 sshd[9077]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64242
Sep 20 15:27:35 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:37:11 amd64 sshd[10102]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:37:11 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:38:10 amd64 sshd[10148]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63546
Sep 21 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 21 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 02:00:01 amd64 /usr/sbin/cron[17878]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 21 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:43:26 amd64 sshd[31088]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63397
Sep 21 17:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:53:39 amd64 sshd[31269]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64391
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 19:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 01:00:01 amd64 /usr/sbin/cron[4674]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 02:00:01 amd64 /usr/sbin/cron[5499]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:21 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 01:00:01 amd64 /usr/sbin/cron[24739]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 23 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 /usr/sbin/cron[25765]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:05 amd64 sshd[6554]: Accepted publickey for esser from ::ffff:192.168.1.5 port 59771 ssh2
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[6606]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62093
Sep 24 01:00:01 amd64 /usr/sbin/cron[12436]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[13253]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[20998]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[23197]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_midi_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[29399]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[662]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1484]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 25 02:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[8889]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[8921]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[9372]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11586]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:33 amd64 sshd[11630]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778
```

Scheduler

- Rechenzeit (CPU) an Prozesse verteilen
- Scheduling-Prinzipien: präemptiv, kooperativ
- Scheduling-Verfahren: Round Robin S., Priority S., Shortest Job First S., etc.
- Was passiert beim Prozess-wechsel?



Scheduler

- **Kooperatives Scheduling:**
  - Prozess rechnet solange, wie er will; bis zum nächsten I/O-Aufruf oder bis `exit()`
  - Scheduler wird nur bei Prozess-Blockieren oder freiwilliger CPU-Aufgabe aktiv
- **Präemptives (unterbrechendes) Scheduling:**
  - Timer aktiviert regelmäßig Scheduler, der neu entscheiden kann, „wo es weiter geht“
- **I/O-lastig:** Prozess hat zwischen I/O-Phasen nur kurze Berechnungsphasen (CPU)
- **CPU-lastig:** Prozess hat zwischen I/O-Phasen lange Berechnungsphasen

Scheduling-Ziele

- **[A1] Ausführdauer:** Wie lange läuft der Prozess insgesamt?
- **[A2] Reaktionszeit:** Wie schnell reagiert der Prozess auf Benutzerinteraktion?
- **[A3] Deadlines** einhalten
- **[A4] Vorhersehbarkeit:** Gleichartige Prozesse sollten sich auch gleichartig verhalten, was obige Punkte angeht
- **[A5] Proportionalität:** „Einfaches“ geht schnell
- **[S1] Durchsatz:** Anzahl der Prozesse, die pro Zeit fertig werden
- **[S2] Prozessorauslastung:** Zeit, die der Prozessor aktiv war
- **[S3] Fairness:** Prozesse gleich behandeln, keiner darf „verhungern“
- **[S4] Prioritäten** beachten
- **[S5] Ressourcen** gleichmäßig einsetzen

Anforderungen an Betriebssystem

Stapelverarbeitung

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- S1 Durchsatz
- A1 Ausführdauer
- S2 Prozessor-Auslastung

Interaktives System

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- A2 Reaktionszeit
- A5 Proportionalität

Echtzeitsystem

- S3 Fairness
- S4 Prioritäteneinsatz
- S5 Ressourcen-Balance
- A3 Deadlines
- A4 Vorhersehbarkeit

Scheduler für Batch-Betrieb

- Kein interaktiver Betrieb (kein Login etc.)
- Job-Management-Tool nimmt Jobs an
- Long term scheduler entscheidet, wann ein Job gestartet wird – evtl. basierend auf Informationen über Ressourcenverbrauch und erwartete Laufzeit des Programms

Scheduling-Verfahren für Batch-Betrieb

- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time (SRT)
- Priority Scheduling

Scheduler für Batch-Betrieb

First Come, First Served (FCFS)

- Warteschlange, keine Unterbrechung (preemption)
- Blockierende Prozesse stellen sich wieder in Warteschlange an
- gut für lange Prozesse, schlecht für I/O-lastige Prozesse

Shortest Job First (SJF)

- wie FCFS: keine Unterbrechung
- nächster Burst (Rechendauer bis zum Blockieren) muss bekannt sein (woher? Burst-Prognose-Verfahren); Prozess mit dem kürzesten Burst auswählen
- minimiert durchschnittliche Laufzeit über alle Prozesse

Shortest Remaining Job Next (SRJ)

- ähnlich SJF, aber *mit* Unterbrechungen
- Scheduler prüft Reihenfolge regelmäßig und bei neuen Jobs
- Prozess mit kürzerer Restlaufzeit unterbricht den aktuellen

Scheduler für interaktive Systeme

- Typisch: Interaktive und Hintergrund-Prozesse
- Desktop- und Server-PCs
- Eventuell mehrere / zahlreiche Benutzer, die sich die Rechenkapazität teilen
- Scheduler für interaktive Systeme prinzipiell auch für Batch-Systeme brauchbar (aber nicht umgekehrt)

Scheduling-Verfahren für interaktive Systeme

- Round Robin
- Prioritäten-Scheduler
- Lotterie-Scheduler
- Fair-Share-Scheduler

Scheduler für interaktive Systeme

Round Robin / Time-Slicing

- wie FCFS, aber mit Unterbrechung
- jeder Prozess erhält Zeitscheibe (Quantum)
- läuft der Prozess bei Ablauf noch -> unterbrechen
- zum Quantum:
  - groß -> Verzögerungen; klein -> häufige Context Switches
  - oft: etwas größer als typische Zeit, die für Interaktion nötig ist
- bevorzugt CPU-lastige Prozesse (I/O-lastige brauchen immer nur Teil ihres Quantums)

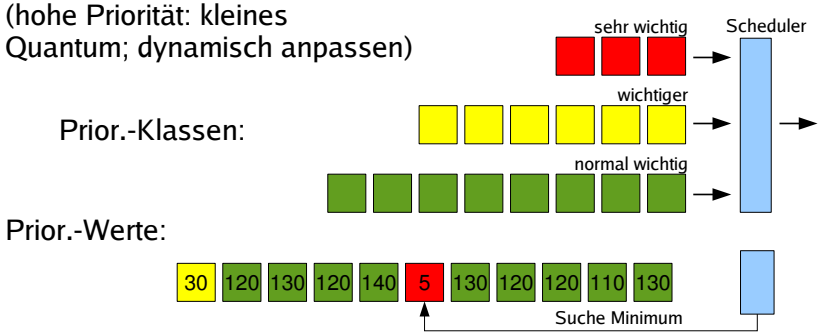
Virtual Round Robin

- Round Robin, aber: I/O-lastigen Pr. helfen -> „Restguthaben“
- hat Prozess Quantum nicht verbraucht? merken und Prozess in Extra-Queue stecken
- Scheduler bevorzugt Prozesse in Extra-Queue und lässt sie ihr Restguthaben aufbrauchen

Scheduler für interaktive Systeme

Prioritäten-Scheduler

- Prioritätsklassen oder individuelle Prioritätswerte
- Scheduler bevorzugt Prozesse mit hoher Priorität
- Prior. statisch vergeben oder dynamisch regelmäßig neu ber.
- Vorsicht: **Prioritätsinversion** (Ausweg: **Aging**)
- Variante: Verschiedene Quantenlängen (hohe Priorität: kleines Quantum; dynamisch anpassen)



Scheduler für interaktive Systeme

Lotterie-Scheduler

- Prozesse besitzen Lose, Scheduler zieht ein Los
- Prioritäten: über Anzahl der Lose, die ein Prozess besitzt
- Gruppenbildung / Kooperation: Prozesse können einander Lose überlassen (etwa ein Client einem Server)

Fair Share Scheduler

- CPU-Zeiten nicht auf Prozessbasis, sondern auf Applikations- oder User-Basis aufteilen
- Prozesse / Threads gruppieren
- Algorithmus von **G. Henry**
- Shares als Obergrenzen oder Untergrenzen betrachten

SMP-Scheduler

- **Load Sharing:** Prozesse keiner bestimmten CPU zuordnen; Rechenlast gleichmäßig verteilen, globale Queue
  -> mutual exclusion beim Zugriff; häufiger CPU-Wechsel
- **Gang Scheduling:** Eine Reihe zusammengehöriger Threads erhält vom Scheduler gleichzeitig eine Menge von CPUs zugeteilt
  Scheduling-Gruppen: wie CPU-Zeit aufteilen, wenn unterschiedlich viele Threads in Gruppen?
- **Dedizierte CPU-Zuordnung:** Threads werden einer bestimmten CPU zugeordnet. Verfahren nimmt sogar hin, dass CPUs idle sind. Kein Multitasking auf einzelnen CPUs
- **Dynamisches Scheduling:** Anzahl der Threads kann sich im Verlauf der Programmausführung ändern
  Programm und Scheduler kooperieren (Programm passt Thread-Zahl an Vorgabe durch Scheduler an)

Linux O(1) Scheduler

- O(1): Zeit, die der Scheduler für Auswahl des nächsten Prozesses (für eine CPU) braucht, ist konstant
- CPUs blockieren sich nicht gegenseitig bei gleichzeitigen Schedule-Entscheidungen
- Load-Balancer verteilt Rechenlast auf mehrere CPUs
- Für jede CPU eine separate Warteschlange
- 140 Prioritätslevel, kleiner Wert = hohe Priorität:
  - 1-100: Realtime-Prozesse (MAX_RT_PRIO=100)
  - 101-140: Normale Prozesse (MAX_PRIO=140)
- Normale Tasks: Nice-Wert n ($-19 \leq n \leq 20$), Prio = MAX_RT_PRIO + n + 20, erhalten Zeitquantum
- Echzeit-Tasks: statische Priorität; zwei Klassen: FIFO (ohne Unterbrechungen) und Round Robin (mit Zeitquanten)
- Interaktivitäts-Schätzer (+/- 5 Prior.-Punkte für I/O- / CPU-lastig)

Linux O(1) Scheduler

- Jede CPU muss nur in ihrer privaten Prozessliste suchen
- Bitmap speichert, welche (der 140) Queues leer sind – Suche der Form „1. Bitmap-Feld mit Wert 1“ geht schnell
- Innerhalb der gefundenen Liste den ersten Prozess wählen
- Suchoperation hängt zwar „von 140“ ab, aber nicht von der Anzahl der Prozesse -> O(1)
- Runqueue + Expired Runqueue

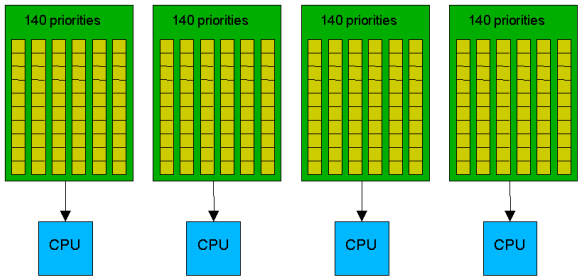


Bild: Linux Journal, http://www.linuxjournal.com/node/7178/

Vorschau

Mittwoch: Teil 2 der Zusammenfassung

5. Synchronisation
6. Interprozess-Kommunikation (IPC)
7. Deadlocks