Script **generated by TTT**

Title: Grundlagen_Betriebssysteme (23.11.2012)

Date: Fri Nov 23 08:30:54 CET 2012

Duration: 91:04 min

Pages: 22

Prozesserzeugung

Prozesse erzeugen andere Prozesse: parent und child. Vater

- kann auf Beendigung von Kind warten, oder
- parallel weiterlaufen.

Prozesserzeugung: 2 Vorgehensweisen

- Windows NT: Vaterprozess veranlasst eine Reihe von Systemaufrufen, die das Kind entstehen lassen.
- Unix: Vater klont sich mit Systemaufruf fork(); Kind ändert selbst sein Aufgabe.

Unix Prozesserzeugung

Linux unterstützt den Systemaufruf `clone()` zur Erzeugung neuer Thread-Kopien.

Aufruf von fork() führt zu einem fast exakten Duplikat des Vaterprozesses. Unterschiede sind

- unterschiedlicher process identifier (PID)
- der Ergebniswert von fork()
  - Vater: PID des Kindprozesses
  - Kind: 0

Beispielprogramm

Kind hat vieles mit dem Vater gemeinsam:

- liest dieselben Dateien, gleicher Benutzername, benutzt dieselben Daten

Unix Kind fängt mit dem Code des Vaters an und ändert sich dann

- Systemaufruf exec(): ersetzt das Programmbild des Vaters mit einem anderen.

Beispielprogramm für exec

Prinzipablauf


```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char *myname = argv[1];
    int cpid = fork();
    if cpid == 0 {
       printf("The child of %s is %d\n", myname, getpid());
       .....
       return (0);
    } else {
       printf("My child is %d\n", cpid);
       /* wird vom Vaterprozess durchlaufen */
       .....
       return(0);
    }
}
```

Unix Prozesserzeugung

Aufruf von fork() führt zu einem fast exakten Duplikat des Vaterprozesses. Unterschiede sind

- unterschiedlicher process identifier (PID)
- der Ergebniswert von fork()
  - Vater: PID des Kindprozesses
  - Kind: 0

Beispielprogramm

- Kind hat vieles mit dem Vater gemeinsam:
  - liest dieselben Dateien, gleicher Benutzername, benutzt dieselben Daten
- Unix Kind fängt mit dem Code des Vaters an und ändert sich dann
  - Systemaufruf exec(): ersetzt das Programmbild des Vaters mit einem anderen.

Beispielprogramm für exec

Prinzipablauf


```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char *myname = argv[1];
    int cpid = fork();
    if cpid == 0 {
       int rc;
       rc = execl("/bin/ls", "ls", "-l", (char *) 0);
       printf("Fehler bei execl Aufruf: %d\n", rc);
       exit(1)
    } else {
       /* wird vom Vaterprozess durchlaufen */
    }
}
```


Aufruf von fork() führt zu einem fast exakten Duplikat des Vaterprozesses. Unterschiede sind

- unterschiedlicher process identifier (PID)
- der Ergebniswert von fork()
  - Vater: PID des Kindprozesses
  - Kind: 0

Beispielprogramm

- Kind hat vieles mit dem Vater gemeinsam:
  - liest dieselben Dateien, gleicher Benutzername, benutzt dieselben Daten
- Unix Kind fängt mit dem Code des Vaters an und ändert sich dann
  - Systemaufruf exec(): ersetzt das Programmbild des Vaters mit einem anderen.

Beispielprogramm für exec

Prinzipablauf

Generated by Targeteam

Aufruf von fork() führt zu einem fast exakten Duplikat des Vaterprozesses. Unterschiede sind

- unterschiedlicher process identifier (PID)
- der Ergebniswert von fork()
  - Vater: PID des Kindprozesses
  - Kind: 0

Beispielprogramm

- Kind hat vieles mit dem Vater gemeinsam:
  - liest dieselben Dateien, gleicher Benutzername, benutzt dieselben Daten
- Unix Kind fängt mit dem Code des Vaters an und ändert sich dann
  - Systemaufruf exec(): ersetzt das Programmbild des Vaters mit einem anderen.

Beispielprogramm für exec

Prinzipablauf

Generated by Targeteam

Bei der Verwaltung von Vater-/Kindprozess sind eine Reihe von Entscheidungen zu treffen:

Ausführung

Vaterprozess und Kind werden gleichzeitig ausgeführt, oder

der Vaterprozess wartet darauf, dass das Kind beendet wird

Ressourcen

Vater und Kind teilen sich alle Ressourcen.

Vater und Kind teilen sich einen Teil der Ressourcen.

Vater und Kind haben keine Ressourcen gemeinsam.

Adressraum

Das Kind ist ein Duplikat des Vaters.

Das Kind führt durch automatisches Laden ein neues Programm aus (exec-Systemaufruf).

Threads

Das Kind dupliziert alle Threads des Vaters.

Das Kind dupliziert nur den Thread des Vaters, der die fork-Operation ausgelöst hat.

Dispatcher

Aufgabe des Dispatchers: Realisieren der Zustandsübergänge zwischen ***rechnend*** und ***rechenwillig*** : Prozessor binden und entbinden. Dazu ist ein Kontextwechsel erforderlich.

Kontextwechsel

CPU wird entzogen und einer anderen Aktivität zugeteilt; ein Kontextwechsel ist erforderlich, falls der rechnende Prozess P1 in den Zustand ***wartend*** oder z.B. durch Prozessorentzug in den Zustand ***rechenwillig*** übergeführt wird.

Problem

aktueller Ausführungskontext des Prozesses muss gesichert werden und Kontext des nächsten rechenbereiten Prozesses muss geladen werden.

Achtung : je umfangreicher ein PCB ist, desto "teurer" sind Prozesswechsel, d.h. das Umschalten der CPU zwischen den Prozessen.

Threads

Threads haben einen sehr viel kleineren Kontext ⇒ Umschalten zwischen Threads innerhalb eines Prozesses sehr schnell, da Adressraum und andere Ressourcen (z.B. Dateien) gemeinsam genutzt werden.

Beispiel: Kontext-Wechsel in Unix

Beispiel: Kontext-Wechsel in Unix

Kontextwechsel z.B. durch den Aufruf der Systemoperation sleep durch einen Prozess.

1. Maskieren von Interrupts;
2. Lokalisieren der benötigten Warteschlange;
3. Neuberechnung der Priorität des Prozesses;
4. Einfügen des Prozesses in die Warteschlange;
5. Aufruf der Operation zum Kontextwechsel.

Arbeitsmodi

Ziel für den Betrieb von Rechensystemen: kontrollierter Zugriff auf Hardwarekomponenten nur durch BS.

Lösung: alle Zugriffe auf Hardware nur über ***privilegierte*** Befehle zulässig;

Frage: wer darf privilegierte Befehle ausführen ⇒ Antwort: Prozesse in einem privilegierten Modus.

Herkömmlich unterscheidet man zwischen dem Benutzer- (engl. user mode) und dem Systemmodus (engl. kernel mode).

Benutzermodus

nur nicht privilegierten Befehle; Zugriff auf Prozessadressraum und unkritische Register (Befehlszähler, Indexregister); Benutzerprozesse im Benutzermodus.

Systemmodus

Es sind auch die privilegierten Befehle verfügbar (z.B. Anhalten der Maschine, Verändern der Ablaufpriorität). Die Dienste des Betriebssystemkerns werden im Systemmodus ausgeführt.

Nutzung der Hardware-Komponenten nur über Dienste des BS: Aufruf eines BS-Dienstes über spezielle Befehle: den ***Systemaufruf*** .

Ein Systemaufruf ist eine Funktion, die von einem Benutzerprozess aufgerufen wird, um einen BS-Kerndienst aufzuführen.

1. Der Systemaufruf überprüft die übergebenen Parameter und bildet daraus eine Datenstruktur, um die Daten an den BS-Kern weiterzureichen.

2. Danach wird eine spezielle Instruktion, ein Software Interrupt (Trap), ausgeführt. Diese Instruktion identifiziert über einen Operanden den gewünschten Systemdienst.

3. Bei Ausführung der Trap-Instruktion wird der Zustand des Benutzerprozesses gerettet und es findet ein Wechsel in den Systemmodus statt.

Beispiel

Beispiel

Lesen von Daten aus einer Datei und Kopieren in eine andere Datei. Dabei treten die folgenden Systemaufrufe auf:

(1) Schreiben des Prompts auf Bildschirm: Angabe der Dateinamen

(2) Lesen der Tastatureingabe (bzw. Analoges bei Mouse-Eingabe)

(3) Öffnen der zu lesenden Datei (open)

(4) Erzeugen der neuen Datei

(5) ggf. Fehlerbehandlung: Nachricht auf Bildschirm

(6) Schleife: Lesen von Eingabedatei (ein Systemaufruf) und schreiben in zweite Datei (auch Systemaufruf)

(7) Schließen beider Dateien

(8) Ausgabe auf Bildschirm

Durch die Verwendung von Laufzeitroutinen ergibt sich eine höhere Abstraktionsebene ⇒ Aufruf einer Routine, die die notwendigen Systemaufrufe durchführt.

Dieser Abschnitt behandelt das Prozesskonzept, Datenstrukturen zur Beschreibung des aktuellen Prozesszustandes sowie Dienste zum Aufruf von Systemfunktionen.

Prozesse repräsentieren eine Aktivität; sie haben ein Programm, Eingaben, Ausgaben und einen Zustand.

Prozesskonzept

Dispatcher

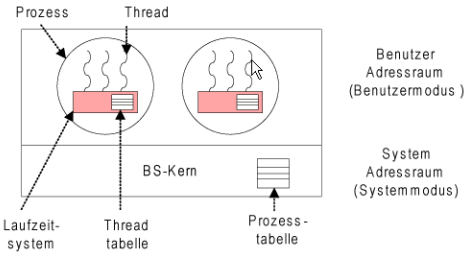
Arbeitsmodi

Systemaufrufe

Realisierung von Threads

im Benutzer-Adressraum

Der BS-Kern verwaltet nur single-threaded Prozesse.



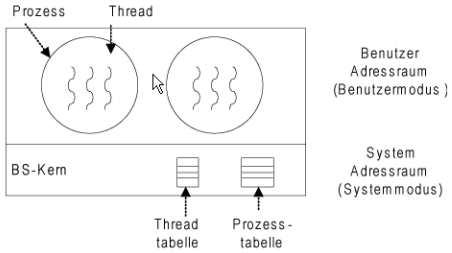
Threads werden durch Laufzeitsystem im Benutzeradressraum verwaltet. Eine Thread-Tabelle speichert Informationen (Register, Zustand, etc.) über Threads pro Prozess.

Prozessorzuteilung im BS-Kern erfolgt an Prozesse. Laufzeitsystem bestimmt, welcher Thread rechnend gesetzt wird.

Problem: Systemaufruf eines Threads blockiert die anderen Threads des Prozesses.

Problem: wie wird einem Thread die CPU entzogen?

Neben den Prozessen werden im BS-Kern auch alle Threads verwaltet.



Thread-Tabelle speichert Informationen (Register, Zustand, etc.) über Threads.

Prozessorzuteilung im BS-Kern erfolgt an Threads.

Der Systemaufruf eines Threads blockiert nicht die anderen Threads des Prozesses.

Realisierung von Threads

Es existieren zwei grundlegende Ansätze, Threads in einem Rechensystem zu realisieren: im ***Benutzer-Adressraum*** (Benutzermodus) oder im ***System-Adressraum*** (Systemmodus).

im Benutzer-Adressraum

im System-Adressraum

Dieser Abschnitt behandelt das Prozesskonzept, Datenstrukturen zur Beschreibung des aktuellen Prozesszustandes sowie Dienste zum Aufruf von Systemfunktionen.

Prozesse repräsentieren eine Aktivität; sie haben ein Programm, Eingaben, Ausgaben und einen Zustand.

Prozesskonzept

Dispatcher

Arbeitsmodi

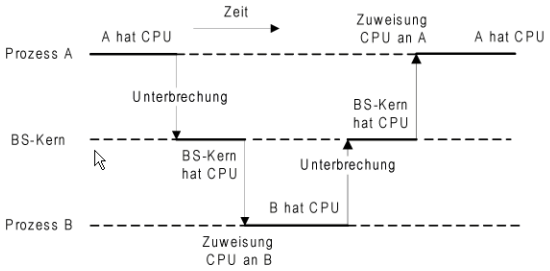
Systemaufrufe

Realisierung von Threads

Eine wesentliche Aufgabe der Prozessorverwaltung besteht darin zu entscheiden, welcher der um den bzw. die Prozessor(en) konkurrierenden Prozesse (bzw. Threads) zu einem Zeitpunkt an den bzw. die Prozessor(en) gebunden wird. Dazu steht die BS-Komponente ***Scheduler*** zur Verfügung.

Prozessablauf besteht aus einer Sequenz von alternierenden CPU- und E/A-Perioden.

Zeitliche Verschränkung der Prozessbearbeitung bei einer CPU.



Kriterien

Scheduling-Strategien

Beispiel Unix Scheduling

Thread Scheduling

Mehrschichtiges Scheduling

Echtzeit Scheduling

Der Scheduler wählt aus der Menge der rechenwilligen Prozesse den nächsten auszuführenden Prozess aus. Es existieren unterschiedliche Verfahren die von der jeweiligen Prozessorverwaltungsstrategie abhängen. Mögliche ***Leistungskriterien*** für ein Schedulingverfahren:

- Fairness.
- Effizienz, Prozessorauslastung.
- Antwortzeit für interaktive Benutzer (Dialogverarbeitung).
- Wartezeit, insbesondere für Batch-Jobs (Stapelverarbeitung).
- Ausführungszeit, d.h. Zeitspanne von Auftragsbeginn bis Auftragsende.
- Abschlusszeit, insbesondere für Realzeitsysteme.
- Durchsatz, Anzahl der Aufträge pro Zeiteinheit.

Kriterien der Betriebsarten

Die Ziele der Schedulingverfahren hängen von der Umgebung und den Betriebsarten des Rechensystems ab.

Alle Systeme

- Fairness - jeder Prozess bekommt Rechenzeit der CPU.
- Balance - alle Teile des Systems sind ausgelastet.
- Policy Enforcement - Scheduling Policy wird nach außen sichtbar durchgeführt.

Stapelbetrieb

- Durchsatz - maximiere nach Aufträge pro Zeiteinheit.
- Ausführungszeit - minimiere die Zeit von Start bis zur Beendigung.
- CPU-Belegung - belege CPU konstant mit Aufträgen.

Dialogbetrieb - interaktive Systeme

- Antwortzeit - antworte schnellstmöglich auf Anfragen.
- Proportionalität - Erwartungshaltung der Nutzer berücksichtigen.

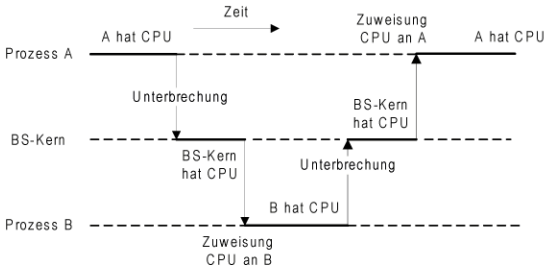
Echtzeitsysteme

- Abschlusszeit - kein Verlust von Daten.
- Vorhersagbarkeit - Qualitätsverlust bei Multimedia vermeiden.

Prozessorverwaltung

Eine wesentliche Aufgabe der Prozessorverwaltung besteht darin zu entscheiden, welcher der um den bzw. die Prozessor(en) konkurrierenden Prozesse (bzw. Threads) zu einem Zeitpunkt an den bzw. die Prozessor(en) gebunden wird. Dazu steht die BS-Komponente ***Scheduler*** zur Verfügung.

- Prozessablauf besteht aus einer Sequenz von alternierenden CPU- und E/A-Perioden.
- Zeitliche Verschränkung der Prozessbearbeitung bei einer CPU.



Kriterien

Scheduling-Strategien

Beispiel Unix Scheduling

Thread Scheduling

Mehrschichtiges Scheduling

Echtzeit Scheduling