Klausur WS06/07 – Nebenläufigkeit Lösung

Aufgabe 3 Nebenläufigkeit **(15 Punkte)**

a) semWartebereich(1);semZapfsaeule(3), semEngstelle(1);

b) siehe Algorithmus

Algorithm 1 Prozess: tankendes Auto

```
1:  down(semWartebereich);
2:  fahreInWartebreiche();
3:  down(semZapfsaeule);
4:  fahreAnZapfsaeule();
5:  up(semWartebereich);
6:  tanke();
7:
8:  bezahle();
9:  down(semEngstelle);
10: fahreInEngstelle();
11: up(semZapfsaeule);
12: verlasseEngstelle();
13: up(semEngstelle);
```

c) Ja. Bevor der Semaphor semWartebereich angefordert wird, muss noch der Semaphor für die Zapfsäule angefordert werden. Ansonsten kann es zur Verklemmung kommen.

Nebenläufigkeit

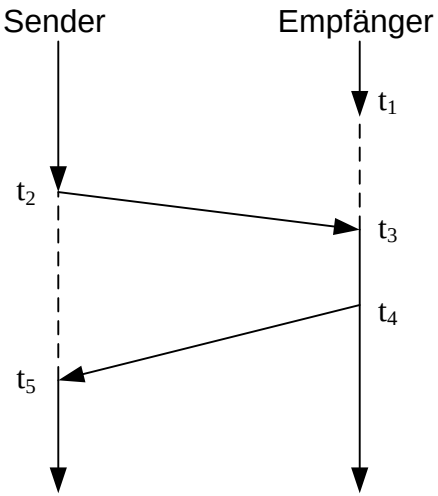
Interprozesskommunikation (IPC)

Interprozesskommunikation

- Notwendigkeit der Interprozesskommunikation
  - Prozesse arbeiten in unterschiedlichen Prozessräumen oder sogar auf unterschiedlichen Prozessoren.
  - Prozesse benötigen evtl. Ergebnisse von anderen Prozessen.
  - Zur Realisierung von wechselseitigen Ausschlüssen werden Mechanismen zur Signalisierung benötigt.
- Klassifikation der Kommunikation
  - synchrone vs. asynchrone Kommunikation
  - pure Ereignisse vs. wertbehaftete Nachrichten

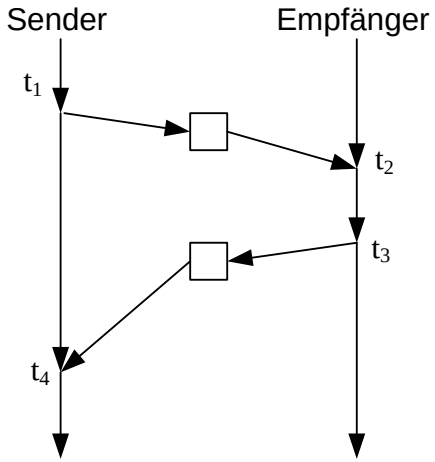
Synchron vs. Asynchron

Synchrone Kommunikation



t_1 : Empfänger wartet auf Nachricht
t_2 : Sender schickt Nachricht und blockiert
t_3 : Empfänger bekommt Nachricht, die Verarbeitung startet
t_4 : Verarbeitung beendet, Antwort wird gesendet
t_5 : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



t_1 : Sender schickt Nachricht an Zwischenspeicher und arbeitet weiter
t_2 : Empfänger liest Nachricht
t_3 : Empfänger schreibt Ergebnis in Zwischenspeicher
t_4 : Sender liest Ergebnis aus Zwischenspeicher

(Nicht eingezeichnet: zusätzliche Abfragen des Zwischenspeichers und evtl. Warten)

IPC-Mechanismen

- Übermittlung von Datenströmen:
  - direkter Datenaustausch
  - Pipes
  - Nachrichtenwarteschlangen (Message Queues)
- Signalisierung von Ereignissen:
  - Signale
  - Semaphore
- Synchrone Kommunikation
  - Barrieren/Rendezvous
  - Kanäle wie z.B. Occam
- Funktionsaufrufe:
  - RPC
  - Corba

Nebenläufigkeit

IPC: Kommunikation durch Datenströme

Direkter Datenaustausch

- Mit Semaphoren und Monitoren geschützte Datenstrukturen eignen sich sehr gut für den Austausch von Daten:
  - schnelle Kommunikation, da auf den Speicher direkt zugegriffen werden kann.
- Allerdings kann die Kommunikation nur lokal erfolgen und zudem müssen die Prozesse eng miteinander verknüpft sein.
- Programmiersprachen, Betriebssysteme, sowie Middlewareansätze bieten komfortablere Methoden zum Datenaustausch.
- Grundsätzlich erfolgt der Austausch über das Ausführen von Funktionen `send(receiver address, &message)` und `receive(sender address, &message)`.

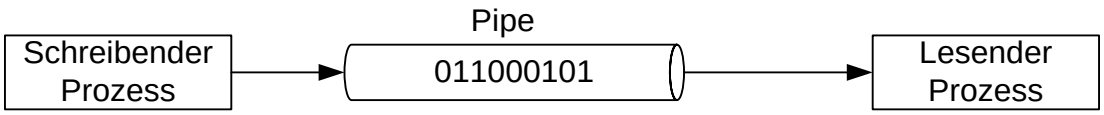
Fragestellungen beim Datenaustausch

- Nachrichtenbasiert oder Datenstrom?
- Lokale oder verteilte Kommunikation?
- Kommunikationsparameter:
  - mit/ohne Bestätigung
  - Nachrichtenverluste
  - Zeitintervalle
  - Reihenfolge der Nachrichten
- Adressierung
- Authentifizierung
- Performance
- Sicherheit (Verschlüsselung)

In diesem Kapitel vor allem lokale Kommunikation, echtzeitfähige Protokolle zur Kommunikation in eigenem Kapitel

Pipes

- Die Pipe bezeichnet eine gepufferte, unidirektionale Datenverbindung zwischen zwei Prozessen nach dem **First-In-First-Out-** (**FIFO**-)Prinzip.
- Über den Namen der Pipe (ähnlich einem Dateinamen) können Prozesse unterschiedlichen Ursprungs auf eine Pipe lesend oder schreibend zugreifen. Zur Kommunikation zwischen Prozessen gleichen Ursprungs (z.B. Vater-, Kindprozess) können auch anonyme Pipes verwendet werden. Die Kommunikation erfolgt immer asynchron.



Pipes in Posix

- POSIX (Portable Operating System Interface) versucht durch Standardisierung der Systemaufrufe die Portierung von Programmen zwischen verschiedenen Betriebssystemen zu erleichtern.
- POSIX.1 definiert folgende Funktionen für Pipes:

```
int mkfifo(char *name, int mode);        /*Erzeugen einer benannten Pipe*/
int unlink ( char *name );               /*Loeschen einer benannten Pipe*/
int open ( char *name, int flags);       /*Oeffnen einer benannten Pipe*/
int close ( int fd );                    /*Schliessen des Lese- oder Schreibendes einer
                                           Pipe*/
int read ( int fd, char *outbuf, unsigned bytes ); /*Lesen von einer Pipe*/
int write ( int fd, char *outbuf,unsigned bytes ); /*Schreiben an eine Pipe*/
int pipe ( int fd[2] );                  /*Erzeugen eine unbenannte Pipe*/
```

Nachteile von Pipes

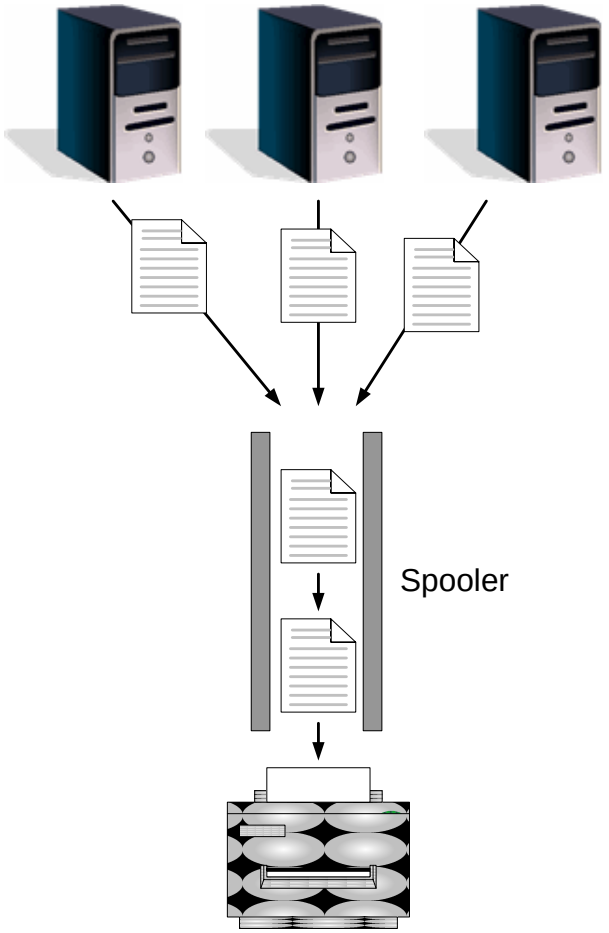
- Pipes bringen einige Nachteile mit sich:
  - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
  - Daten sind nicht priorisierbar.
  - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung:
  - Es können keine Daten aufgehoben werden.
  - Beim Öffnen blockiert der Funktionsaufruf, bis auch das zweite Zugriffsende geöffnet wird (Verhinderung durch O_NDELAY Flag).
- Lösung: Nachrichtenwartschlangen

Nachrichtenwarteschlangen (message queues)

- Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenwarteschlangen, wie in POSIX 1003.1b (Echtzeiterweiterung von POSIX) definiert, betrachtet.
- Eigenschaften der POSIX MessageQueues:
  - Beim Anlegen einer MessageQueue wird der benötigte Speicher reserviert.) Speicher muss nicht erst beim Schreibzugriff angelegt werden.
  - Die Kommunikation erfolgt nachrichtenorientiert. Die Anzahl der vorhandenen Nachrichten kann dadurch abgefragt werden.
  - Nachrichten sind priorisierbar → Es können leichter Zeitgarantien gegeben werden.

Nachrichtenwarteschlangen

- Schreibzugriff in Standardsystemen: Der schreibende/sendende Prozess wird nur dann blockiert werden, falls der Speicher der Datenstruktur bereits voll ist. **Alternative in Echtzeitsystemen: Fehlermeldung ohne Blockade**.
- Lesezugriff in Standardsystemen: Beim lesenden/empfangenden Zugriff auf einen leeren Nachrichtenspeicher wird der aufrufende Prozess blockiert bis eine neue Nachricht eintrifft. **Alternative: Fehlermeldung ohne Blockade**.
- Ein anschauliches Beispiel für den Einsatzbereich ist der Spooler eines Druckers: dieser nimmt die Druckaufträge der verschiedenen Prozesse an und leitet diese der Reihe nach an den Drucker weiter.



Message Queues in POSIX

- POSIX definiert folgende Funktionen für Nachrichtenwartschlangen:

```
mqd_t mq_open(const char *name, int oflag, ...);   /*Oeffnen einer Message Queue*/
int mq_close(mqd_t mqdes);                         /*Schliessen einer Message Queue*/
int mq_unlink(const char *name);                   /*Loeschen einer
                                                     Nachrichtenwarteschlange*/

int mq_send(mqd_t mqdes, const char *msg_ptr,
    size_t msg_len, unsigned int msg_prio);        /*Senden einer Nachricht*/
size_t mq_receive(mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned int *msg_prio);       /*Empfangen einer Nachricht*/
int mq_setattr(mqd_t mqdes, const struct
    mq_attr *mqstat, struct mq_attr *mqstat);      /*Aendern der Attribute*/
int mq_getattr(mqd_t mqdes,
    struct mq_attr *mqstat);                       /*Abrufen der aktuellen
                                                     Eigenschaften*/

int mq_notify(mqd_t mqdes,
    const struct sigevent *notification);          /*Anforderung eines Signals bei
                                                     Nachrichtenankunft*/
```

Nebenläufigkeit

IPC: Kommunikation durch Ereignisse

Signale

- **Signale** werden in Betriebssystemen typischerweise zur Signalisierung von Ereignissen an Prozessen verwendet.
- Signale können verschiedene Ursachen haben:
  - Ausnahmen, z.B. Division durch Null (SIGFPE) oder ein Speicherzugriffsfehler (SIGSEGV)
  - Reaktion auf Benutzereingaben (z.B. Ctrl / C)
  - Signal von anderem Prozess zur Kommunikation
  - Signalisierung von Ereignissen durch das Betriebssystem, z.B. Ablauf einer Uhr, Beendigung einer asynchronen I/O-Funktion, Nachrichtankunft an leerer Nachrichtenwarteschlange (siehe mq_notify())

Prozessreaktionen auf Signale

- Der Prozess hat drei Möglichkeiten auf Signale zu reagieren:
  1. Ignorierung der Signale
  2. Ausführen einer Signalbehandlungsfunktion
  3. Verzögerung des Signals, bis Prozess bereit für Reaktion ist
- Zudem besteht die Möglichkeit mit der Standardreaktion auf das bestimmte Signal zu reagieren. Da aber typischerweise die Reaktion auf Signale die Beendigung des Empfängerprozesses ist, sollte ein Programm über eine vernünftige Signalbehandlung verfügen, sobald ein Auftreten von Signalen wahrscheinlich wird.

Semaphore zur Vermittlung von Ereignissen

- Semaphore können neben der Anwendung des wechselseitigen Ausschlusses auch zur Signalisierung von Ereignissen verwendet werden.
- Es ist zulässig, dass Prozesse (Erzeuger) Semaphore andauernd freigeben und andere Prozesse (Verbraucher) Semaphore dauern konsumieren.
- Es können auch benannte Semaphoren erzeugt werden, die dann über Prozessgrenzen hinweg verwendet werden können.
- Notwendige Funktionen sind dann:
  - sem_open(): zum Erzeugen und / oder Öffnen eines benannten Semaphors
  - sem_unlink(): zum Löschen eines benannten Semaphors

Signalisierung durch Semaphore: Beispiel

- Beispiel: ein Prozeß **Worker** wartet auf einen Auftrag (abgespeichert z.B. in einem char-Array job) durch einen Prozess **Contractor**, bearbeitet diesen und wartet im Anschluß auf den nächsten Auftrag:

Worker*:

```
while(true)
{
        down(sem); /*wait for
                   next job*/
        execute(job);
}
```

Contractor*:

```
...
job=...  /*create new job and save
         address in global variable*/
up(sem); /*signal new job*/
...
```

** sehr stark vereinfachte Lösung, da zu einem Zeitpunkt nur ein Job verfügbar sein darf*

Probleme

- Problematisch an der Implementierung des Beispiels auf der letzten Folie ist, dass der Zeiger auf den Auftrag `job` nicht geschützt ist und es so zu fehlerhaften Ausführungen kommen kann.
- Durch Verwendung eines zusätzlichen Semaphors kann dieses Problem behoben werden.
- Ist die Zeit zwischen zwei Aufträgen zu kurz um die rechtzeitige Bearbeitung sicherzustellen, so kann es zu weiteren Problemen kommen:
  - Problem 1: Der Prozess **Contractor** muss warten, weil der Prozeß **Worker** den letzten Auftrag noch bearbeitet.
  - Problem 2: Der letzte Auftrag wird überschrieben, falls dieser noch gar nicht bearbeitet wurde. Abhängig von der Implementierung des Semaphors könnte dann der neue Auftrag zudem zweifach ausgeführt werden.

→ mit Semaphoren sind nur einfache Signalisierungsprobleme (ohne Datentransfer) zu lösen, ansonsten sollten Warteschlangen verwendet werden

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Vorherige Lösung:

Reader:

```
...
        down(semWriter);
        down(semCounter);
        rcounter++;
        up(semCounter);
        up(semWriter);

        read();

        down(semCounter);
        rcounter--;
        up(semCounter);
...
```

Writer:

```
...
        down(semWriter);

        while(true)
        {
                down(semCounter);
                if(rcounter==0)
                        break;
                up(semCounter);
        }
        up(semCounter);

        write();

        up(semWriter);
...
```

Problem: Busy Waiting

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Lösung mit Signalisierung:

Reader:

```
...
            down(semWriter);
            down(semCounter);
            rcounter++;
            if(rcounter==1)
                        down(semReader);
            up(semCounter);
            up(semWriter);

            read();

            down(semCounter);
            rcounter--;
            if(rcounter==0)
                        up(semReader);
            up(semCounter);
...
```

Writer:

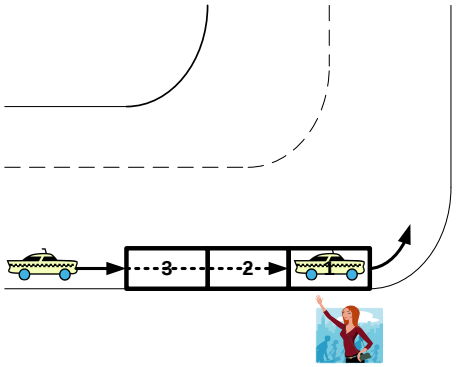
```
...
            down(semWriter);
            down(semReader);
            up(semReader);

            write();


            up(semWriter);
...
```

Klausur WS07/08 – Nebenläufigkeit (20 Punkte = 20min)

- Gegeben Sie folgendes Szenario: am Münchner Odeonsplatz gibt es eine Wartebucht für Taxis. Zur Vereinfachung gehen wir davon aus, dass die Wartebucht aus drei Plätzen besteht und immer nur ein Passagier gleichzeitig auf ein Taxi wartet. Passagiere steigen an der ersten Wartebucht ein, die Taxis rücken nach, sobald das Taxi vor ihnen losgefahren ist. Implementieren Sie nun schrittweise eine Prozesssynchronisation, so dass es zu keinen Auffahrunfällen kommt, die Taxis in der Ankunftsreihenfolge auch wieder losfahren, Taxis nur mit Passagier losfahren, Passagiere nicht aus Versehen ein nicht-existentes Taxi betreten und es zu keinen Verklemmungen kommt.



a) Notieren Sie die wichtigen Programmabschnitte des Taxiprozesses und des Passagierprozesses. Lassen Sie genügend Platz für spätere Synchronisationsoperationen.
Beispiel: `fahreInErsteWartebucht();`

b) Geben Sie die zur Synchronisation der Taxis und Passagiere benötigten Semaphore, sowie der Initialwerte an. Gehen Sie dabei davon aus, dass zu Beginn kein Taxi in der Wartebucht und keine wartenden Passagiere vorhanden sind.
Beispiel: `semTaxi(1)` würde bedeuten, Sie verwenden einen Semaphor semTaxi, der mit 1 initialisiert ist.
`int i=0;` wenn sie eine ganzzahlige Variable mit Initialisierungswert 1 benutzen wollen.

c) Ergänzen Sie den Taxiprozess und Passagierprozess mit passenden up() und down()-Methoden, um die Aufgabenstellung zu erfüllen.
Beispiel: `down(semTaxi);` bedeutet das Anfordern des Semaphors semTaxi
Beispiel: `up(semTaxi);` bedeutet das Freigeben des Semaphors semTaxi

d) Der Wartebereich am Odeonsplatz ist begrenzt. Stellen Sie sicher, dass maximal 3 Taxis auf Fahrgäste warten und kein Rückstau entsteht. Die Überprüfung ob der Wartebereich belegt ist, soll dabei so schnell wie möglich erfolgen um den Straßenverkehr nicht zu behindern. Andererseits, sollen die Taxifahrer auf jeden Fall in den letzten Warteplatz fahren, falls dieser frei ist.

Klausur WS07/08 – Nebenläufigkeit - Lösung

Algorithm 1 Lösung Nebenläufigkeit: Benötigte Semaphoren

```
1: semBucht3(1);
2: semBucht2(1);
3: semBucht1(1);
4: semTaxi(0);
5: semPassagier(0);
6: semTest(1);
7: int frei=1;
```

Algorithm 2 Lösung Nebenläufigkeit: Taxiprozess

```
1: down(semBucht3);
2: fahreInBucht3();
3: down(semBucht2);
4: fahreInBucht2();
5: up(semBucht3);
6: down(semBucht1);
7: fahreInBucht1();
8: up(semBucht2);
9: up(semTaxi);
10: down(semPassagier);
11: fahreLos();
12: up(semBucht1);
```

Algorithm 3 Lösung Nebenläufigkeit: Passagierprozess

```
1: down(semTaxi);
2: steigeEin();
3: up(semPassagier);
```

Algorithm 4 Lösung Nebenläufigkeit: Überprüfung

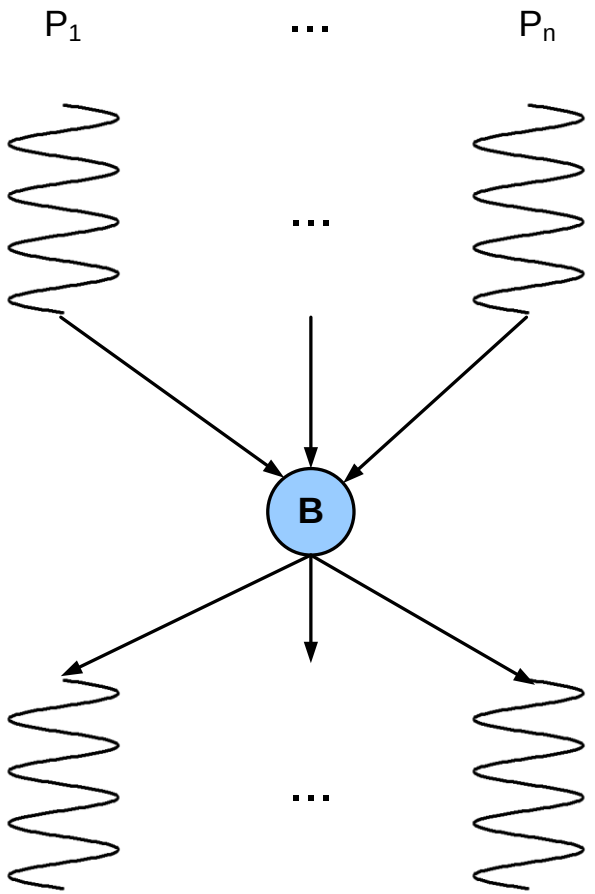
```
1: down(semTest);
2: if frei==1 then
3:     frei=0;
4:     up(semTest);
5:     fahreInBucht3();
6:     down(semBucht2);
7:     fahreInBucht2();
8:     down(semTest);
9:     frei=1;
10:    up(semTest);
11:    ...
12: else
13:    up(semTest);
14:    fahreWeiter();
15: end if
```

Nebenläufigkeit

Synchrone Kommunikation: Barrieren

Synchrone Kommunikation: Barrieren

- **Definition**: Eine Barriere für eine Menge *M* von Prozessen ist ein Punkt, den alle Prozesse $P_i 2 M$ erreichen müssen, bevor irgendein Prozess aus *M* die Berechnung über diesen Punkt hinaus fortfahren kann.
- Der Spezialfall für $|M|=2$ wird als Rendezvous, siehe auch Ada, bezeichnet.
- Barrieren können mit Hilfe von Semaphoren implementiert werden (→ Hausaufgabe).



Erfolgskontrolle: Was Sie aus dem Kapitel mitgenommen haben?

- Definition und Gründe für Nebenläufigkeit
- Arten der Umsetzung von Nebenläufigkeit (Prozesse, Threads, Interrupts), Technische Umsetzung, Unterschiede und Anwendungsgebiete
- Probleme (Race Conditions, Verklemmungen, Starvation) , die durch Nebenläufigkeit entstehen und passende Lösungen
  - Insbesondere Anwendung von Semaphoren
- Mechanismen zur Interprozesskommunikation

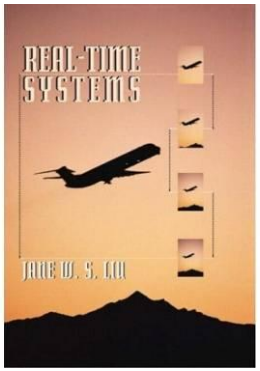
Kapitel 5

Scheduling

Inhalt

- Definitionen
- Kriterien zur Auswahl des Scheduling-Verfahrens
- Scheduling-Verfahren
- Prioritätsinversion
- Exkurs: Worst Case Execution Times

Literatur



Jane W. S. Liu, Real-Time Systems, 2000

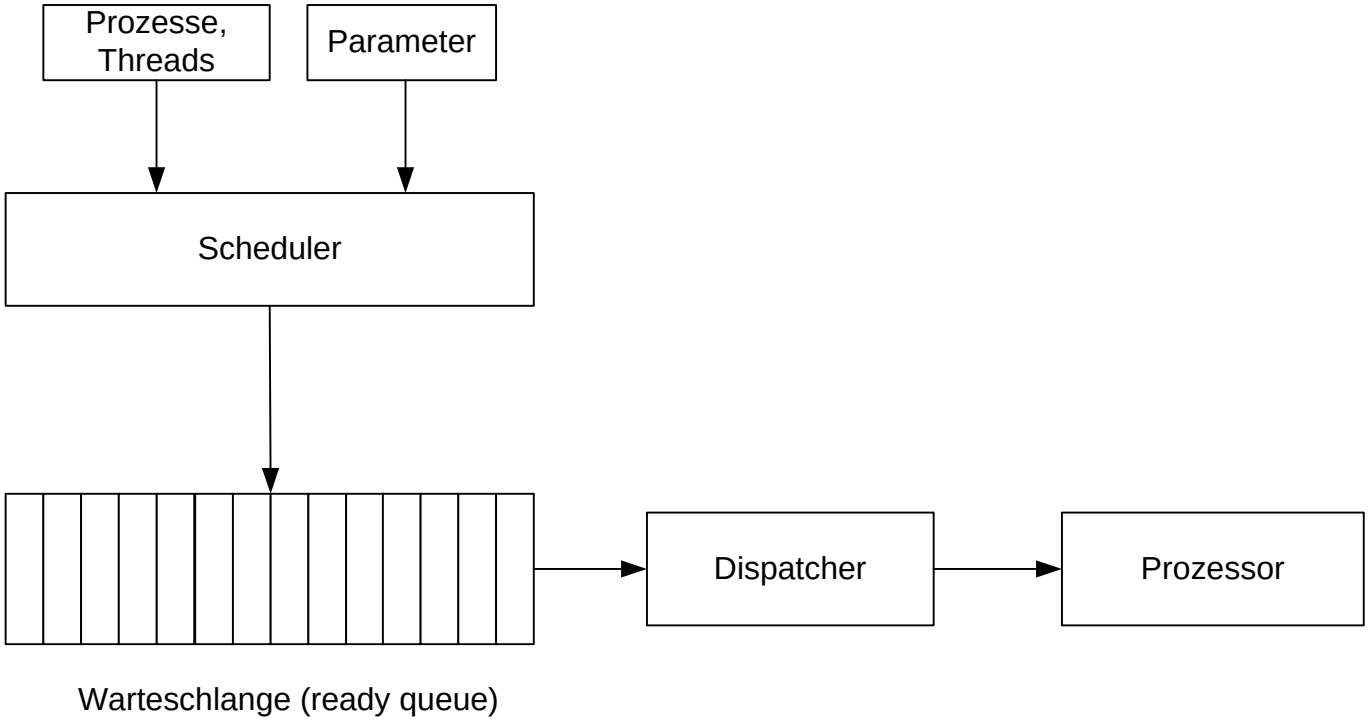
Fridolin Hofmann: Betriebssysteme - Grundkonzepte und Modellvorstellungen, 1991

- Journals:
  - John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo: Implications of classical scheduling results for real-time systems. IEEE Computer, Special Issue on Scheduling and Real-Time Systems, 28(6):16–25, June 2005.
  - Giorgio C. Buttazzo: Rate Monotonic vs. EDF: Judgement Day (http://www.cas.mcmaster.ca/~downd/rtsj05-rmedf.pdf)
  - Puschner, Peter; Burns, Alan: A review of Worst-Case Execution-Time Analysis, Journal of Real-Time Systems 18 (2000), S.115-128

Scheduling

Definitionen

Scheduler und Dispatcher

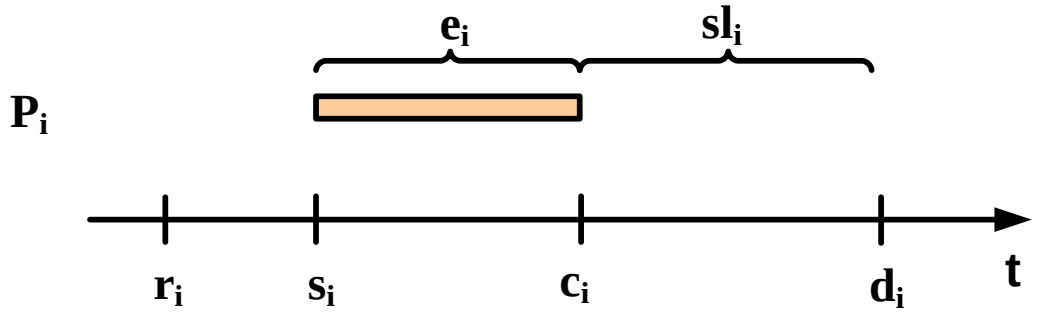


Scheduler und Dispatcher

- **Scheduler**: Modul eines Betriebssystems, das die Rechenzeit an die unterschiedlichen Prozesse verteilt. Der ausgeführte Algorithmus wird als Scheduling-Algorithmus bezeichnet. Aufgabe des Schedulers ist also die langfristige Planung (Vergleich: Erstellung eines Zugfahrplans).
- **Dispatcher**: Übersetzung: Einsatzleiter, Koordinator, Zuteiler (v.a. im Bereich der Bahn gebräuchlich). Im Rahmen der Prozessverwaltung eines Betriebssystems dient der Dispatcher dazu, bei einem Prozesswechsel dem derzeit aktiven Prozess die CPU zu entziehen und anschließend dem nächsten Prozess die CPU zuzuteilen. Die Entscheidung, welcher Prozess der nächste ist, wird vom Scheduler im Rahmen der Warteschlangenorganisation getroffen.

Zeitliche Bedingungen

- Folgende Größen sind charakteristisch für die Ausführung von Prozessen:
  1. P_i bezeichnet den i. **Prozess** (bzw. Thread)
  2. r_i: **Bereitzeit** (**ready time**) des Prozesses P_i und damit der früheste Zeitpunkt an dem der Prozess dem Prozessor zugeteilt werden kann.
  3. s_i: **Startzeit**: der Prozessor beginnt P_i auszuführen.
  4. e_i: **Ausführungszeit** (**execution time**): Zeit die der Prozess P_i zur reinen Ausführung auf dem Prozessor benötigt.
  5. c_i: **Abschlußzeit** (**completion time**): Zeitpunkt zu dem die Ausführung des Prozesses P_i beendet wird.
  6. d_i: **Frist** (**deadline**): Zeitpunkt zu dem die Ausführung des Prozesses P_i in jeden Fall beendet sein muss.
  7. sl_i: **Slack**: Deadline-(aktuelle Zeit + verbleibende Berechnungszeit)



Spielraum (slack time)

- Mit dem Spielraum (slack time) sl_i eines Prozesses P_i wird Zeitraum bezeichnet, um den ein Prozess noch maximal verzögert werden darf:
  - Die Differenz zwischen der verbleibenden Zeit bis zum Ablauf der Frist und der noch benötigten Ausführungszeit zur Beendigung des Prozesses Pi .
- Der Spielraum eines Prozesses, der aktuell durch den Prozessor ausgeführt wird, bleibt konstant, während sich die Spielräume aller nicht ausgeführten Prozesse verringern.

Faktoren bei der Planung

- Für die Planung des Schedulings müssen folgende Faktoren berücksichtigt werden:
  - Art der Prozesse (periodisch, nicht periodisch, sporadisch)
    - Periodisch: Prozesse sind regelmäßig mit fixer Frequenz startbereit
    - Nicht-Periodisch: Prozessbereitzeiten können mit gewissen Schranken vorhergesagt werden
    - Sporadisch: Es können keine/kaum Aussagen darüber getroffen werden, wann und wie oft Prozesse startbereit sind
  - Gemeinsame Nutzung von Ressourcen (**shared resources**)
  - Fristen
  - Vorrangrelationen (**precedence constraints**: Prozess P_i muss vor P_j ausgeführt werden)

Arten der Planung

- Es kann zwischen unterschiedlichen Arten zum Planen unterschieden werden:
  - offline vs. online Planung
  - statische vs. dynamische Planung
  - präemptives vs. nicht-präemptives Scheduling

Offline Planung

- Mit der offline Planung wird die Erstellung eines Ausführungsplanes zur Übersetzungszeit bezeichnet. Zur Ausführungszeit arbeitet der Dispatcher den Ausführungsplan dann ab.
- **Vorteile**:
  - deterministisches Verhalten des Systems
  - wechselseitiger Ausschluss in kritischen Bereichen wird direkt im Scheduling realisiert
- **Nachteile**:
  - Bereitzeiten, Ausführungszeiten und Abhängigkeit der einzelnen Prozesse müssen schon im Voraus bekannt sein.
  - Die Suche nach einem Ausführungsplan ist im Allgemeinen ein NP-hartes Problem. Es werden jedoch keine optimalen Pläne gesucht, vielmehr ist ein gute Lösung (Einhaltung aller Fristen) ausreichend.

Online Scheduling

- Alle Schedulingentscheidungen werden online, d.h. auf der Basis der Menge der aktuell lauffähigen Prozesse und ihrer Parameter getroffen.
- Im Gegensatz zur offline Planung muss wechselseitiger Ausschluss nun über den expliziten Ausschluss (z.B. Semaphoren) erfolgen.
- Vorteile:
  - Flexibilität
  - Bessere Auslastung der Ressourcen
- Nachteile:
  - Es müssen zur Laufzeit Berechungen zum Scheduling durchgeführt werden → Rechenzeit geht verloren.
  - Garantien zur Einhaltung von Fristen sind schwieriger zu geben.
  - Problematik von Race Conditions

Statische vs. dynamische Planung

- Bei der statischen Planung basieren alle Entscheidungen auf Parametern, die vor der Laufzeit festgelegt werden.
- Zur statischen Planung wird Wissen über:
  - die Prozessmenge
  - ihre Prioritäten
  - das Ausführungsverhalten

  benötigt.
- Bei der dynamischen Planung können sich die Scheduling-Parameter (z.B. die Prioritäten) zur Laufzeit ändern.
- **Wichtig**: Statische Planung und Online-Planung schließen sich nicht aus: z.B. Scheduling mit festen Prioritäten.

Präemption

- Präemptives (bevorrechtigt, entziehend) Scheduling: Bei jedem Auftreten eines relevanten Ereignisses wird die aktuelle Ausführung eines Prozesses unterbrochen und eine neue Schedulingentscheidung getroffen.
- Präemptives (unterbrechbares) Abarbeiten:
  - Aktionen (Prozesse) werden nach bestimmten Kriterien geordnet (z.B. Prioritäten, Frist,…).
  - Diese Kriterien sind statisch festgelegt oder werden dynamisch berechnet.
  - Ausführung einer Aktion wird sofort unterbrochen, sobald Aktion mit höherer Priorität eintrifft.
  - Die unterbrochene Aktion wird an der Unterbrechungsstelle fortgesetzt, sobald keine Aktion höherer Priorität ansteht.
  - Typisch für Echtzeitaufgaben (mit Ausnahme von Programmteilen, die zur Sicherung der Datenkonsistenz nicht unterbrochen werden dürfen).
  - Nachteil: häufiges Umschalten reduziert Leistung.

Ununterbrechbares Scheduling

- Ein Prozess, der den Prozessor zugewiesen bekommt, wird solange ausgeführt, bis der Prozess beendet wird oder er aber den Prozess freigibt.
- Scheduling-Entscheidungen werden nur nach der Prozessbeendigung oder dem Übergang des ausgeführten Prozesses in den blockierten Zustand vorgenommen.
- Eine begonnene Aktion wird beendet, selbst wenn während der Ausführung Aktionen höherer Dringlichkeit eintreffen
  → Nachteil: evtl. Versagen (zu lange Reaktionszeit) des Systems beim Eintreffen unvorhergesehener Anforderungen
- Anmerkung: Betriebssysteme unterstützen allgemein präemptives Scheduling solange ein Prozess im Userspace ausgeführt, Kernelprozesse werden häufig nicht oder selten unterbrochen.

  → Echtzeitbetriebssysteme zeichnen sich in Bezug auf das Scheduling dadurch aus, dass nur wenige Prozesse nicht unterbrechbar sind und diese wiederum sehr kurze Berechnungszeiten haben.

Schedulingkriterien

- Kriterien in Standardsystemen sind:
  - Fairness: gerechte Verteilung der Prozessorzeit
  - Effizienz: vollständige Auslastung der CPU
  - Antwortzeit: interaktive Prozesse sollen schnell reagieren
  - Verweilzeit: Aufgaben im Batchbetrieb (sequentielle Abarbeitung von Aufträgen) sollen möglichst schnell ein Ergebnis liefern
  - Durchsatz: Maximierung der Anzahl der Aufträge, die innerhalb einer bestimmten Zeitspanne ausgeführt werden
- In Echtzeitsystemen:
  - Einhaltung der Fristen: d.h. für alle Prozesse P_i gilt $c_i<d_i$ unter Berücksichtigung von Kausalzusammenhängen (Synchronisation, Vorranggraphen, Präzedenzsystemen)
  - Determinismus des Verfahrens
  - Zusätzliche Kriterien können anwendungsabhängig hinzugenommen werden, solange sie der Einhaltung der Fristen untergeordnet sind.

Scheduling

Verfahren

Allgemeines Verfahren

- Gesucht: Plan mit aktueller Start und Endzeit für jeden Prozess P_i.
- Darstellung zum Beispiel als nach der Zeit geordnete Liste von Tupeln (P_i, s_i, c_i)
- Falls Prozesse unterbrochen werden können, so kann jedem Prozess P_i auch eine Menge von Tupeln zugeordnet werden.
- Phasen der Planung:
  - Test auf Einplanbarkeit (feasibility check)
  - Planberechnung (schedule construction)
  - Umsetzung auf Zuteilung im Betriebssystem (dispatching)
- Bei Online-Verfahren können die einzelnen Phasen überlappend zur Laufzeit ausgeführt werden.
- Zum Vergleich von Scheduling-Verfahren können einzelne Szenarien durchgespielt werden.

Definitionen

- **Zulässiger Plan**: Ein Plan ist zulässig, falls alle Prozesse einer Prozessmenge eingeplant sind und dabei keine Präzedenzrestriktionen und keine Zeitanforderungen verletzt werden.
- **Optimales Planungsverfahren**: Ein Verfahren ist optimal, falls es für jede Prozessmenge unter gegebenen Randbedingung einen zulässigen Plan findet, falls ein solcher existiert.

Test auf Einplanbarkeit

- Zum Test auf Einplanbarkeit können zwei Bedingungen angegeben werden, die für die Existenz eines zulässigen Plans notwendig sind (Achtung: häufig nicht ausreichend):
  1. $r_i + e_i < d_i$, d.h. jeder Prozess muss in dem Intervall zwischen Bereitzeit und Frist ausgeführt werden können.
  2. Für jeden Zeitraum $[t_i, t_j]$ muss die Summe der Ausführungszeiten e_x der Prozesse P_x mit $r_x > t_i \wedge d_x < t_j$ kleiner als der Zeitraum sein (bei Multicoreprozessoren mit n gleichen Prozessoren kleiner dem n-fachen des Zeitraums).
- Durch weitere Rahmenbedingungen (z.B. Abhängigkeiten der einzelnen Prozesse) können weitere Bedingungen hinzukommen.

Schedulingverfahren

- Planen aperiodischer Prozesse
  - Planen durch Suchen
  - Planen nach Fristen
  - Planen nach Spielräumen
- Planen periodischer Prozesse
  - Planen nach Fristen
  - Planen nach Raten
- Planen abhängiger Prozesse