Technische Universität München
Fakultät für Informatik
Prof. Dr. A. Knoll

WS 2010/2011
08. Februar 2011

Lösungsvorschläge der Klausur zu Echtzeitsysteme

Aufgabe 1 Wissensfragen (Lösungsvorschlag) (20 Punkte)

a) Aussagen über Echtzeitsysteme: **(8 P)**

(i) Aussage: Harte Echtzeitsysteme sind Systeme, die besonders schnell sind, also besonders kurze Berechnungszeiten haben. **Falsch**. Beispiel: eine Anfrage in einer Suchmaschine wird zwar sehr schnell beantwortet, allerdings ist es kein hartes Echtzeitsystem.

(ii) Aussage: Harte Echtzeitsysteme sind Systeme, bei denen ein bestimmtes Zeitverhalten garantiert werden kann, wobei das Zeitverhalten aber langsam sein kann. **Richtig**. Beispiel: Ampelsteuerung. Das Zeitverhalten ist im Sekunden- bzw. Minutenbereich, allerdings muss sichergestellt sein, dass alle Ampeln gleichzeitig umschalten.

b) Synchronitätshypothese: **(6 P)**

(i) Hypothese: Das zugrunde liegende System ist unendlich schnell, so dass die Reaktion auf ein Ereignis augenblicklich erfolgt.

(ii) Vorteil: die Reaktion auf zwei Ereignisse kann sich nicht überlappen, das System verhält sich deterministisch.

(iii) Umsetzung: Während der Berechnung der Reaktion auf ein Ereignis werden alle anderen Ereignisse blockiert. Die Reaktion auf diese blockierten Ereignisse erfolgt erst in der nächsten Runde.

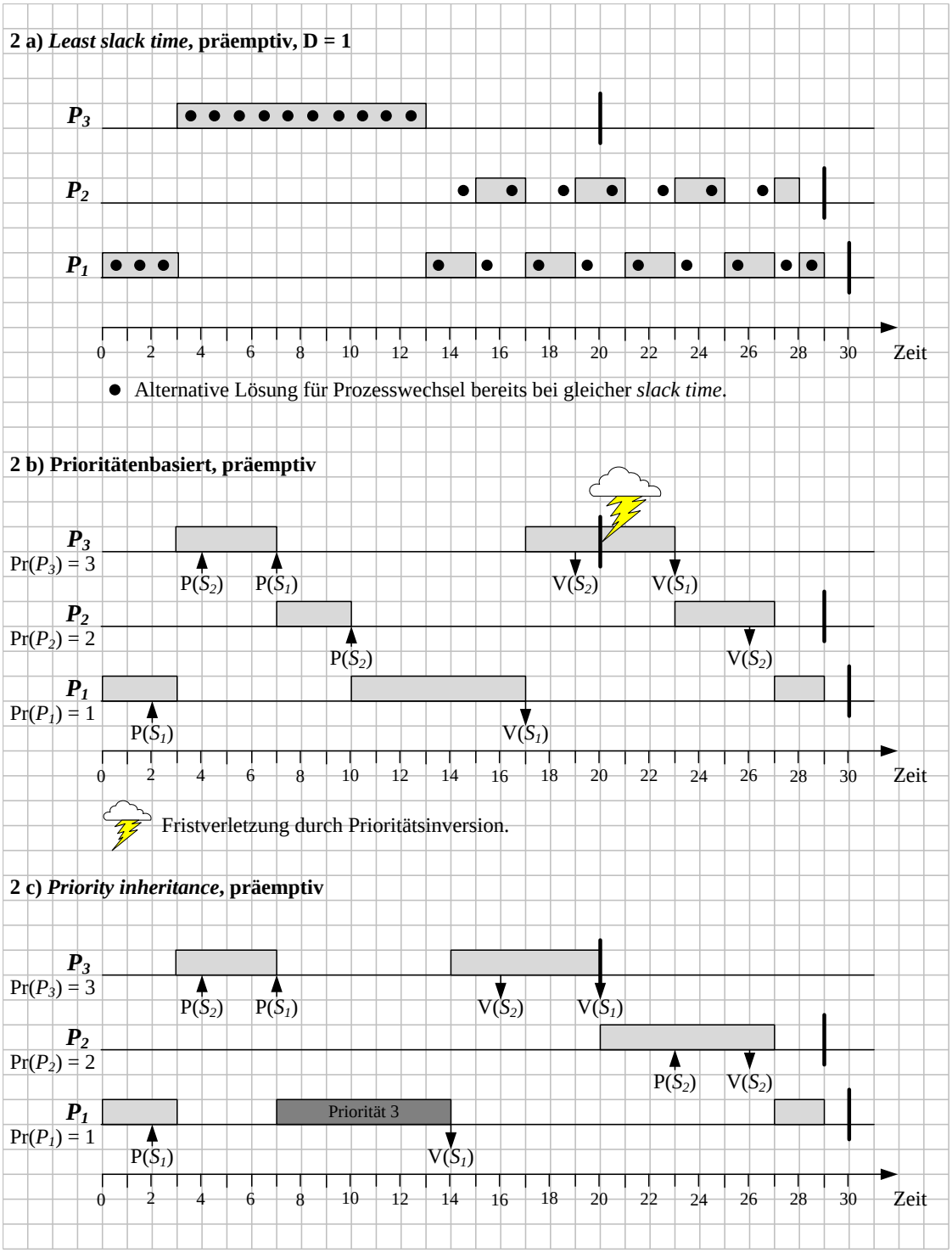
c) Zuordnung Ausführungsmodelle - Anwendungen: **(6 P)**

(i) Synchronous Data Flow $\leftrightarrow$ Regelungsanwendung: Synchronous Data Flow eignet sich vor allem für periodischen Berechnungen, die auf einem Rechner ausgeführt werden (da die Synchronitätshypothese so leichter umgesetzt werden kann).

(ii) Synchronous Reactive $\leftrightarrow$ Transaktionssystem: Synchronous Reactive eignet sich vor allem für die Modellierung und Entwicklung von ereignisbasierten Systemen, wie der Ampelsteuerung.

(iii) Time-Triggered Execution $\leftrightarrow$ Fahrzeugsteuerung: Zeitgesteuerte Ausführung ist ideal um periodische Prozesse im verteilten System auszuführen.

Aufgabe 2 Scheduling (Lösungsvorschlag) (20 Punkte)



a) *Least slack time*, präemptiv, $D = 1$ **(6 P)**

b) Prioritätenbasiert, präemptiv **(8 P)**

c) *Priority inheritance*, präemptiv **(6 P)**

Aufgabe 3 Nebenläufigkeit (Lösungsvorschlag) **(30 Punkte)**

Programmieraufgaben:

Eine Implementierung, die alle Anforderungen der Teilaufgaben a)-d) erfüllt, ist in den Algorithmen 1, 2 und 3 angegeben.

a) Binärer Semaphor / Mutex: **(4 P)**
Die zentrale Türe wird durch den Semaphor `semTür` geschützt, der mit `1` initialisiert ist.

b) Zählender Semaphor: **(4 P)**
Die Platzbeschränkung wird durch den Semaphor `semAusgabe` realisiert (initaler Wert: `50`). Dabei müssen auch Studenten, die die Mensa verlassen, ohne etwas zu essen, `semAusgabe` wieder freigeben.

c) Nachrichten-Warteschlange (*message queue*): **(10 P)**
Die Einhaltung der Reihenfolge der an der Kasse wartenden Studentenprozesse wird durch die Nachrichten-Warteschlange `qKasse` realisiert. Die ankommenden Studenten reihen ihre `ID` in `qKasse` ein, und fordern dann ihren persönlichen Semaphor an. Da dieser mit `0` initialisiert ist, blockieren sie bei diesem Aufruf. Der Kassierer entnimmt jeweils eine Kennung aus `qKasse`, löst den entsprechenden Semaphor auf (`getSemaphoreById()`) und gibt diesen frei, so dass der entsprechene Studentenprozess aufgeweckt und entblockiert wird.

d) Barriere: **(6 P)**
Die Synchronization der drei Prozesse, für die das Prädikat `is3T() true` zurückliefert erfolgt durch Warten auf die mit `3` initialiserte Barriere `barTTT`.

Allgemeine Fragen (**keine** Implementierung nötig!):

e) Bedingungen für Deadlock: **(4 P)**

- *Wechselseitiger Ausschluss*: Es gibt eine Menge von exklusiven Ressourcen $\mathcal{R}_{exkl}$, die entweder frei sind oder genau einem Prozess zugeordnet sind.
- *Hold-and-wait-Bedingung*: Prozesse, die bereits im Besitz von Ressourcen aus $\mathcal{R}_{exkl}$ sind, fordern weitere Ressourcen aus $\mathcal{R}_{exkl}$ an.
- *Ununterbrechbarkeit*: Die Ressourcen $\mathcal{R}_{exkl}$ können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
- *Zyklische Wartebedingung*: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.

f) Probleme bei nebenläufiger Programmierung: **(2 P)**

- *Race Conditions:* Situationen, in denen zwei oder mehrere Threads/Prozesse, die gleichen geteilten Daten lesen oder schreiben und das Resultat davon abhängt, wann genau welcher Prozess ausgeführt wurde, werden Race Conditions genannt.
  Lösung: Wechselseitiger Ausschluss.
- *Priority Inversion* (Prioritätsinversion): Wichtige Prozesse können durch unwichtigere Prozesse, die Betriebsmittel belegt haben verzögert werden.
  Lösung: Priority inheritance, priority ceiling.

Algorithmus 1 Codegerüst für Deklaration globaler Variablen

```
1:  // Deklaration von globalen Variablen, Semaphoren, etc.
2:
3:  Tür tür;
4:
5:  // a)
6:  Semaphore semTür(1);
7:
8:  // b)
9:  Semaphore semAusgabe(50);
10:
11: // c)
12: MessageQueue qKasse;
13:
14: // d)
15: Barrier barTTT(3);
```

Algorithmus 2 Codegerüst für Kassiererprozess

```
1:  // Deklaration von lokalen Variablen, Semaphoren, etc. für Kassiererprozess, c)
2:  int kennungAktuellerStudent; // c)
3:  Semaphore semaphoreAktuellerStudent; // c)
4:
5:  // Code für Kassiererprozess
6:
7:  while (!feierabend()) do
8:      // c)
9:      kennungAktuellerStudent = recvMsg(qKasse);
10:     semaphoreAktuellerStudent = getSemaphoreById(kennungAktuellerStudent);
11:     up(semaphoreAktuellerStudent);
12:
13:     kassiere();
14:
15:     schaueAufDieUhr();
16:
17: end while
18:
```

Algorithmus 3 Codegerüst für Studentenprozess

```
1:  // Deklaration von lokalen Variablen, Semaphoren, etc. für Studentenprozess
2:  Essen essen;
3:  int ID; // Eindeutige Kennung für jeden Studentenprozess
4:  Semaphore mySemaphore; // c)
5:
6:  // Codegerüst für Studentenprozess
7:  down(semAusgabe); // b)
8:
9:  down(semTür); // a)
10: betreteMensa(tür);
11: up(semTür);
12:
13: essen = liesMenü();
14:
15: if (essen == KeinEssen) then
16:
17:     down(semTür); // a)
18:     verlasseMensa(tür);
19:     up(semTür);
20:
21:     up(semAusgabe); // b)
22:     return
23:
24: end if
25:
26: holeEssen();
27: // c)
28: sendMsg(qKasse, ID);
29: mySemaphore = getSemaphoreById(ID);
30: down(mySemaphore);
31:
32: bezahle();
33:
34: up(semAusgabe); // b)
35: // d)
36: if (is3T()) then
37:     wait(barTTT);
38: end if
39: essen();
40:
41: gibTablettZurück();
42:
43: down(semTür); // a)
44: verlasseMensa(tür);
45: up(semTür);
46:
```

Aufgabe 4 Kommunikation (Lösungsvorschlag) **(20 Punkte)**

a) Kommunikationsablauf für **CSMA/CA** siehe Abbildung 1. **(8 P)**

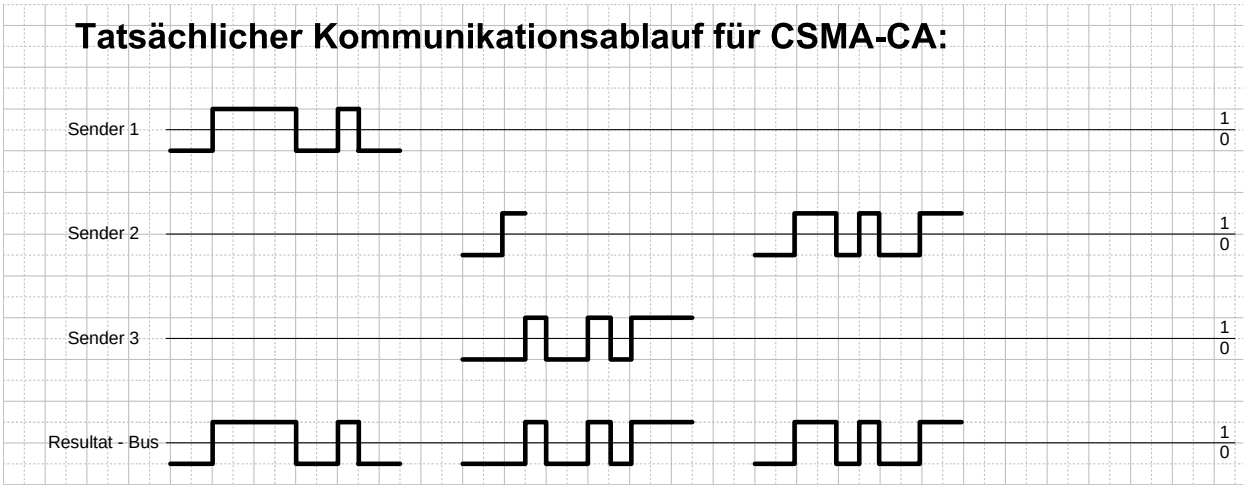


Abbildung 1: Kommunikation / Buszugriff mittels CSMA/CA

b) Kommunikationsablauf für **TTP** siehe Abbildung 2. **(6 P)**

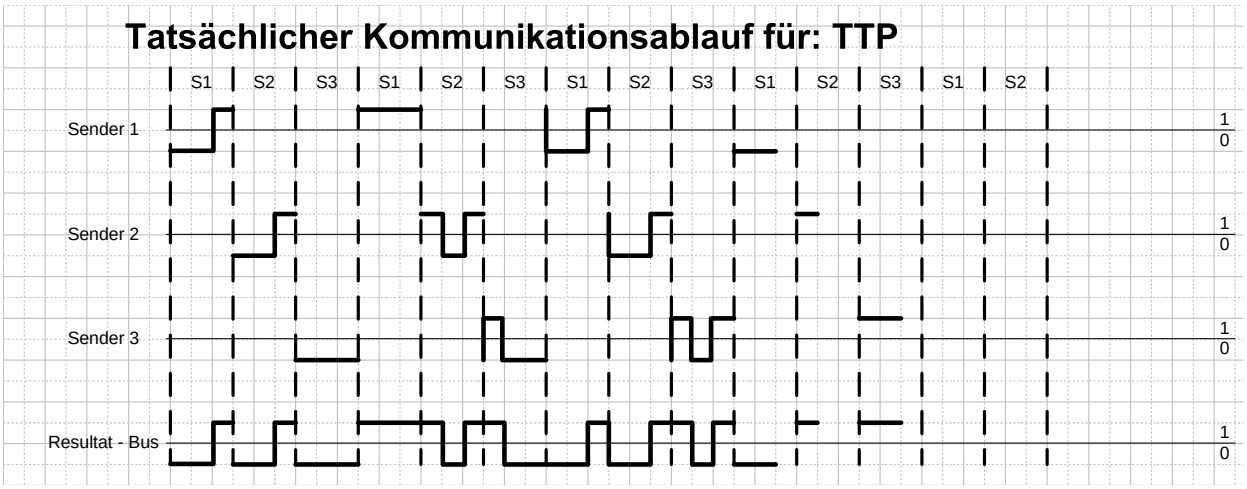


Abbildung 2: Kommunikation / Buszugriff mittels TTP

c) Unterschied beim Buszugriff zwischen TTP, Token Ring und CAN: **(6 P)**

- TTP: Zugriff über TDMA / Zeitschlitze
- CAN: Zugriff über CSMA/CA (lesen vom Bus beim schreiben, Priorisierung)
- Token Ring: schreibender Zugriff auf Bus nur dann, wenn Teilnehmer Token hat.