

Klausur zur Vorlesung "Grundlagen Betriebssysteme und Systemsoftware"

(Prof. Dr. J. Schlichter, Dr. G. Groh, SS 2008)

- Datum: Donnerstag, 3.4.2008, 14-16 Uhr
- Ort: Hörsaal 1 (Gebäude Mathematik-Informatik, Garching)
- Bearbeitungszeit: 90 Minuten
- Zugelassene Hilfsmittel: Keine
- Bitte beschriften Sie ZUERST dieses Deckblatt mit Vorname, Nachname, Studiengang und Matrikelnummer!
- DANN beschriften Sie bitte das karierte Papier!
- Bitte kennzeichnen Sie auf dem karierten Papier vor der Abgabe eindeutig, was bewertet werden soll und wo es sich um freie Notizen handelt!
- Viel Erfolg!

Vorname	Nachname
Matrikelnummer	Studiengang

Mitteilungen vom Studierenden an die Korrigierenden:
--

A1	A2	A3	A4	A5	Σ		Unterschrift Erstkorrektor
A1	A2	A3	A4	A5	Σ	Note	Unterschrift Zweitkorrektor

1 Aufgabe (Petri-Netze) (8 P)

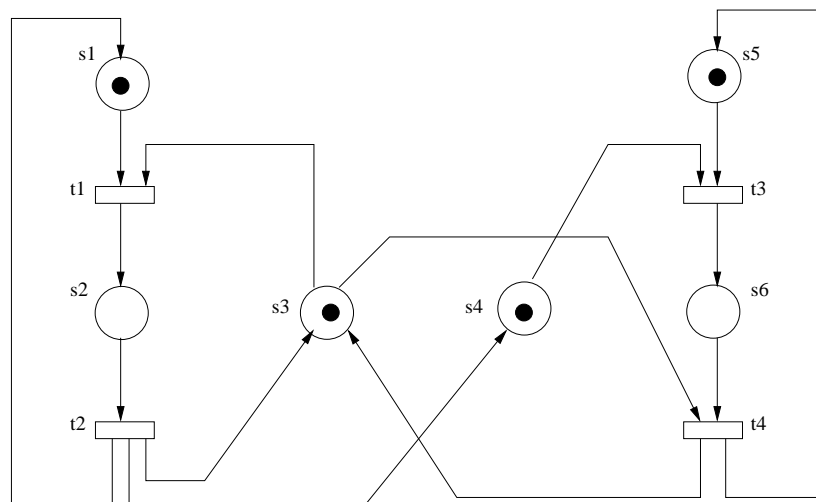
1.1 Teilaufgabe (6 P)

Horst lädt Werner zu sich zum Abendessen ein. Unglücklicherweise gibt es in Horsts Wohnung nur einen Löffel. So kann immer nur eine Person zur selben Zeit von der Vorspeisensuppe essen. Nun sind Horst und Werner mehr am Gespräch interessiert als am Essen und nicht in Eile. Deswegen wird der Löffel immer, wenn einer der beiden etwas Suppe gegessen hat, zurück auf den Tisch gelegt, und die beiden schwatzen ein wenig. Irgendwann später nimmt wieder einer der beiden den Löffel und isst etwas und so weiter.

1. Modellieren Sie die Situation mit einem Petri-Netz! Benennen Sie Stellen und Transitionen aussagekräftig! (3 P)
2. Geben Sie den Erreichbarkeitsgraphen zu Ihrem Petri-Netz an und argumentieren Sie mit dessen Hilfe, dass in Ihrem Petrinetz immer höchstens eine der Personen den Löffel halten kann! (3 P)

1.2 Teilaufgabe (2 P)

Gegeben sei das folgende boolsche Petri-Netz:

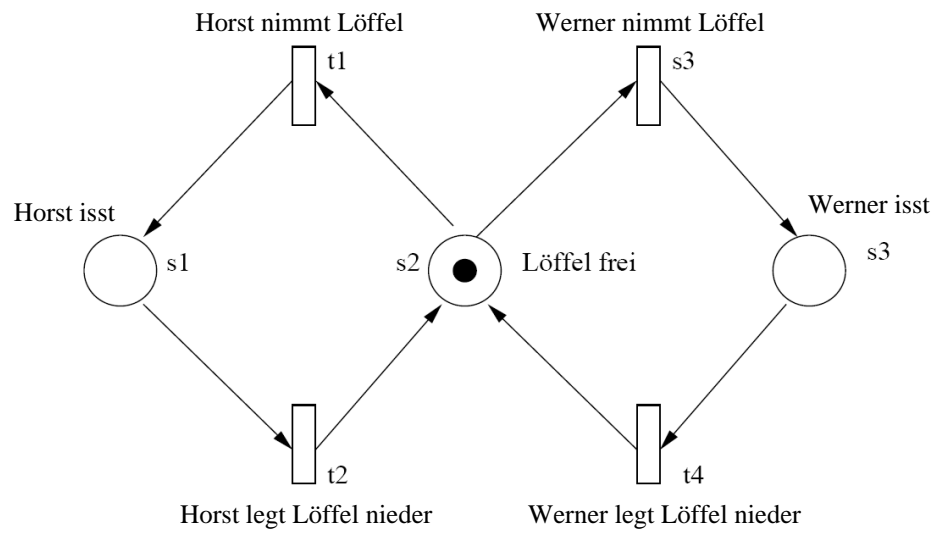


Die Stellen s_2 und s_4 sollen einen kritischen Bereich darstellen. Realisieren Sie einen wechselseitigen Ausschluss zwischen diesen beiden Stellen s_2 und s_4 , indem Sie das gegebene Petri-Netz erweitern! Abläufe, die den kritischen Bereich nicht betreffen, sollen möglichst unverändert bleiben.

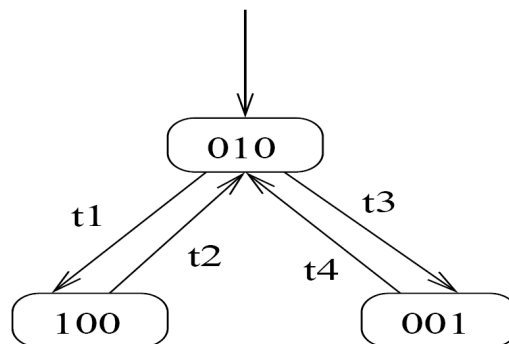
Lösungsvorschläge:

Zu Teilaufgabe 1.1:

(1)

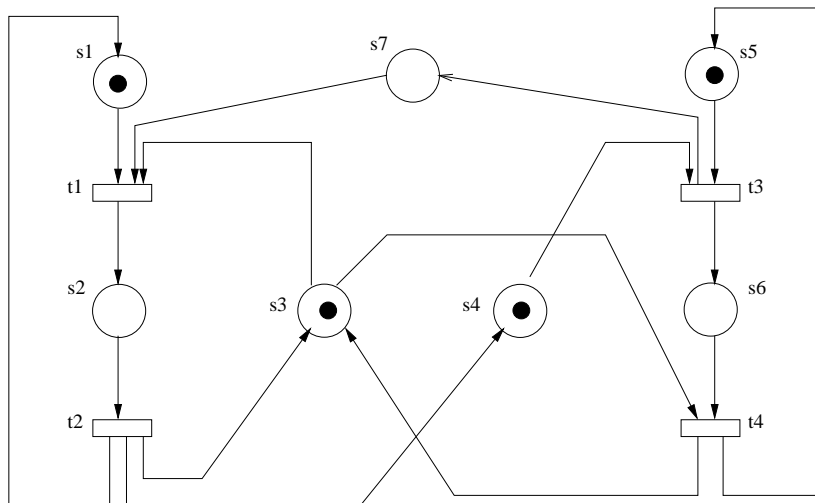


(2)



In diesem Erreichbarkeitsgraphen gibt es keinen Zustand in dem sowohl Horst als auch Werner essen.

Zu Teilaufgabe 1.2:



2 Aufgabe (Quiz) (8 P)

1. Gegeben sei ein Prozess $P = (E, \leq, \alpha)$ mit $\alpha = \{(e_1, a_1), (e_2, a_2), \dots, (e_5, a_5)\}$, $E = \{e_1, e_2, \dots, e_5\}$ und $\leq = (\{(e_1, e_2), (e_2, e_3), (e_1, e_5), (e_2, e_5), (e_4, e_5)\})^*$ wobei $()^*$ die Bildung der reflexiven transitiven Hülle bedeute. Geben Sie alle Paare (genauer: zwei-elementige Mengen) von Ereignissen an, die gemäß dieser Modellierung nebenläufig sind! (2 P)
2. Nennen Sie 2 Beispiele für Informationen bzw. Datenstruktur-Instanzen die für jeden (User-)Thread spezifisch sind und nennen Sie 2 Beispiele für Informationen bzw. Datenstruktur-Instanzen die für jeden Prozess spezifisch sind (die also (User-)Threads gemeinsam nutzen müssen)! (1 P)
Erläutern Sie kurz, warum der Aufwand für das Erzeugen eines (User-)Threads i.A. geringer ist als für das Erzeugen eines Prozesses! (1 P)
3. Folgender Ausschnitt aus einem C Programm sei gegeben. Ersetzen sie die Platzhalter $\langle expr1 \rangle$, $\langle expr2 \rangle$, $\langle expr3 \rangle$ und $\langle expr4 \rangle$ so, dass das Programm eine doppelt verkettete Liste von 10 gleichartigen Personalakten erzeugt! (2 P)

```
struct personalRecord{
    int salary;
    struct personalRecord* previousRecord;
    struct personalRecord* nextRecord;
};

int main(){
    int i;
    struct personalRecord firstRecord;
    struct personalRecord *theNewRecord;
    struct personalRecord *theRecordBeforeTheNewRecord;
    firstRecord.previousRecord = NULL;
    firstRecord.nextRecord = NULL;
    firstRecord.salary = 1000;
    theRecordBeforeTheNewRecord =  $\langle expr1 \rangle$  ;
    for(i=0; i<9; i++){
        theNewRecord = newRecord();
        theNewRecord->salary = 1000;
         $\langle expr2 \rangle$  = theRecordBeforeTheNewRecord;
         $\langle expr3 \rangle$  = theNewRecord;
        theRecordBeforeTheNewRecord = theNewRecord;
    }
    return 0;
}

struct personalRecord* newRecord(){
    struct personalRecord *newPersonalRecord;
    newPersonalRecord = (struct personalRecord *) malloc(  $\langle expr4 \rangle$  );
    return newPersonalRecord;
}
```

4. Memory-Management: Seitenadressierung: Wahl einer vernünftigen Seitengröße: Erläutern Sie 2 Vorteile kleiner Seiten und erläutern sie 2 Vorteile großer Seiten! (2 P)

Lösungsvorschläge:

1. $(\{(e_1, e_2), (e_2, e_3), (e_1, e_5), (e_2, e_5), (e_4, e_5)\})^* = \{(e_1, e_2), (e_2, e_3), (e_1, e_5), (e_2, e_5), (e_4, e_5), (e_1, e_3), \forall i(e_i, e_i)\}$.
Alle "Paare" $(\overset{E}{2}) \setminus \leq$, die nicht in Relation \leq stehen, sind nebenläufig: $\{\{e_1, e_4\}, \{e_2, e_4\}, \{e_3, e_4\}, \{e_3, e_5\}\}$

2. Thread-spezifisch sind bspw.: Befehlszähler, aktuelle Registerwerte, Keller (Stack), Ablaufzustand (running, waiting, ready, terminated).
Prozess-spezifisch sind bspw.: Adressraum, globale Variable, offene Dateien, Kindprozesse, eingetretene Alarmer bzw. Interrupts, Verwaltungsinformationen.
(User-)Threads haben keinen eigenen Adressraum. Beim Erzeugen eines Prozesses muss dieser erst erzeugt werden.
3. Zu ersetzen sind:
 - <expr1> mit `&firstRecord`;
 - <expr2> mit `theNewRecord->previousRecord`
 - <expr3> mit `theRecordBeforeTheNewRecord->nextRecord`
 - <expr4> mit `sizeof(struct personalRecord)`
4.
 - Je kleiner die Seite, desto rascher sind i.A. die Transfers zwischen Arbeitsspeicher und Permanent-Speicher (bspw. Platte).
 - Je kleiner die Seite, desto geringer ist i.A. der Verschchnitt (interne Fragmentierung) durch nicht voll ausgenützte Seiten.
 - Je größer die Seite, desto geringer ist i.A. der Overhead für Transport zwischen Arbeitsspeicher und Platte pro Byte (Arm positionieren, Warten bis Spur unter Lese-Schreib-Kopf etc.).
 - Je größer die Seite, desto seltener sind Transfers i.A. erforderlich.

3 Aufgabe (Java-Monitore und Semaphore) (8 P)

3.1 Teilaufgabe Java-Monitore (5 P)

Gegeben sei folgende Java-Klasse `Lager`, die ein Paket verwaltet und als Thread realisiert ist. Zum Zugriff auf das Paket stehen die beiden wechselseitig ausgeschlossenen Methoden `anliefern` und `abholen` zur Verfügung.

```
public class Lager extends Thread
{
    private boolean vorhanden = false;
    private int paket;

    public synchronized int abholen ()
    {
        if (vorhanden == true)
        {
            vorhanden = false;
            return paket;
        }
    }

    public synchronized void anliefern (int wert)
    {
        if (vorhanden == false)
        {
            vorhanden = true;
            paket = wert;
        }
    }
}
```

Bei dieser Implementierung von `Lager` gibt es das Problem, dass die Operation `abholen` keinen Rückgabewert liefert, wenn kein Paket vorhanden ist, und deshalb nicht übersetzt werden kann. Notwendig wäre, dass `abholen` wartet, bis ein Paket angeliefert wird. Genauso schreibt `anliefern` nur dann den übergebenen Wert in `paket`, wenn kein Paket vorhanden ist.

Geben Sie modifizierte Versionen von `abholen` und `anliefern` an, so dass die Operationen nunmehr warten, bis sie durchgeführt werden können, keine Verklemmung auftreten kann und der Zugriff auf das Paket wechselseitig ausgeschlossen ist.

3.2 Teilaufgabe Semaphore (3 P)

Gegeben sei das aus der Vorlesung bekannte Erzeuger / Verbraucher-Problem mit dem Puffer `W` (Kapazität: `n` Elemente). Um wechselseitigen Ausschluss zu erreichen, sei folgender Lösungsversuch mit der Semaphore `wa` gegeben (Pseudocode):

Deklaration:

```
wa(1);
```

Erzeuger E:

```
while(true){  
    produziere Element;  
    wa.P();  
    schreibe Element nach W;  
    wa.V();  
}
```

Verbraucher V:

```
while(true){  
    wa.P();  
    entnimm Element aus W falls Element vorhanden, sonst warte;  
    wa.V();  
    verarbeite Element;  
}
```

1. Welches Problem kann dabei auftreten? (1 P)
2. Geben Sie eine verbesserte Version an, in der keine Probleme mehr auftreten, indem Sie zwei weitere Semaphoren geeignet deklarieren und geeignete Aufrufe von P und V einfügen! (2 P)

Lösungsvorschlag:

Zu Teilaufgabe 3.1:

```
public class Lager extends Thread  
{  
    private boolean vorhanden = false;  
    private int paket;  
  
    public synchronized int abholen ()  
    {  
        int paket_tmp;  
        while (vorhanden == false)  
        {  
            // Schleife, bis Paket vorhanden  
            try  
            {  
                wait(); // Warte  
            }  
            catch (InterruptedException e)  
            {  
            }  
        }  
        vorhanden = false;  
        paket_tmp = paket; // Zwischenspeicher Paket  
        notify(); // Wecke wartenden Thread  
        return paket_tmp;  
    }  
}
```

```

public synchronized void anliefern (int wert)
{
    while (vorhanden == true)
    {
        // Schleife, bis Paket frei
        try
        {
            wait(); // Warte
        }
        catch (InterruptedException e)
        {
        }
    }
    vorhanden = true;
    paket = wert;
    notify(); // Wecke wartenden Thread
}
}

```

Zu Teilaufgabe 3.2:

(1) Problem: Es kann eine Verklemmung auftreten, wenn der Verbraucher wa.P() ausführt und warten muss, weil der Puffer kein Element enthält. Andererseits kann der Erzeuger den Puffer nicht betreten, da bereits der Verbraucher den Puffer exklusiv belegt hat.

(2) Folgende modifizierte Variante führt zum Ziel:

Deklaration:

```

wa(1);
voll(0);
leer(n);

```

Erzeuger E:

```

while(true){
    produziere Element;
    leer.P();
    wa.P();
    schreibe Element nach W;
    wa.V();
    voll.V();
}

```

Verbraucher V:

```

while(true){
    voll.P();
    wa.P();
    entnimm Element aus W;
    wa.V();
    leer.V();
    verarbeite Element;
}

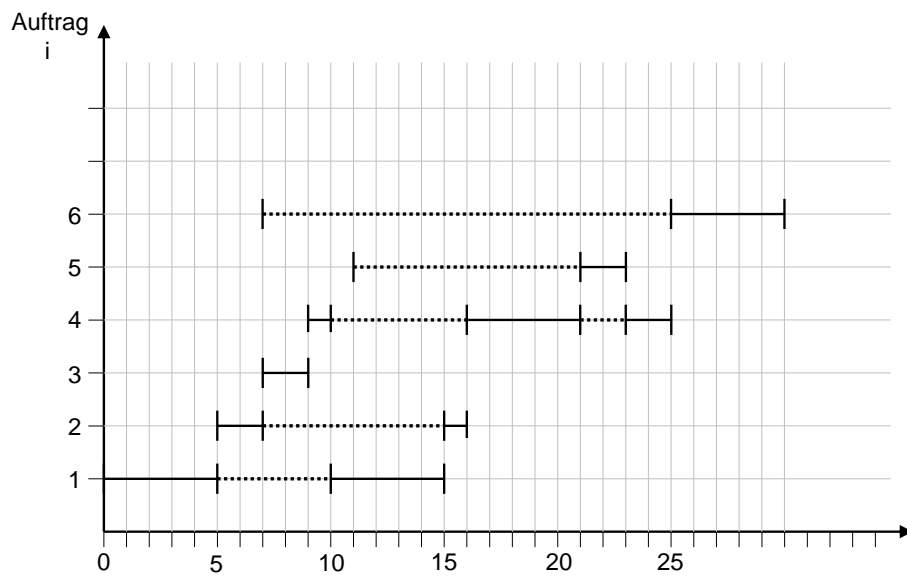
```

4 Aufgabe (Process-Scheduling) (8 P)

Wir betrachten Aufträge mit Bedienzeiten (Rechenzeiten) sowie Ankunftszeiten (der Zeitpunkt, ab dem der Auftrag im System vorhanden ist und berechnet werden kann) und einen Prozessor, auf dem jeweils immer nur einer der Aufträge bearbeitet werden kann. Wir nehmen vereinfachend an, dass ein Prozesswechsel keine Zeit kostet.

Es existieren sechs Aufträge A_1, A_2, \dots, A_6 mit den Bedienzeiten $b = (5, 2, 2, 4, 2, 8)$. Der Vektor der Ankunftszeiten sei durch $a = (0, 2, 3, 4, 6, 7)$ gegeben. Vor dem Zeitpunkt $t = 0$ sei das Bedienungs-System leer.

Zwei Process-Scheduling Strategien sollen in einem Diagramm folgender Struktur visualisiert werden: Auf der x-Achse seien die Zeiteinheiten und auf der y-Achse seien die Nummer der verschiedenen Aufträge angetragen. Dabei ist jeder Auftrag durch eine Linie von seinem Ankunftszeitpunkt bis zu seiner abgeschlossenen Berechnung eingetragen, wobei die Linie gestrichelt ist solange er wartet und durchgezogen, solange ihm der Prozessor zugeteilt ist. Die folgende Abbildung zeigt zur Erläuterung ein fiktives Beispiel-Diagramm:



1. Visualisieren Sie die Strategien SJF (shortest job first) und SRPT (shortest remaining processing time) für die gegebenen sechs Aufträge in der geschilderten Art und Weise! Sie können dazu die leeren Diagramme benutzen! (6 P)

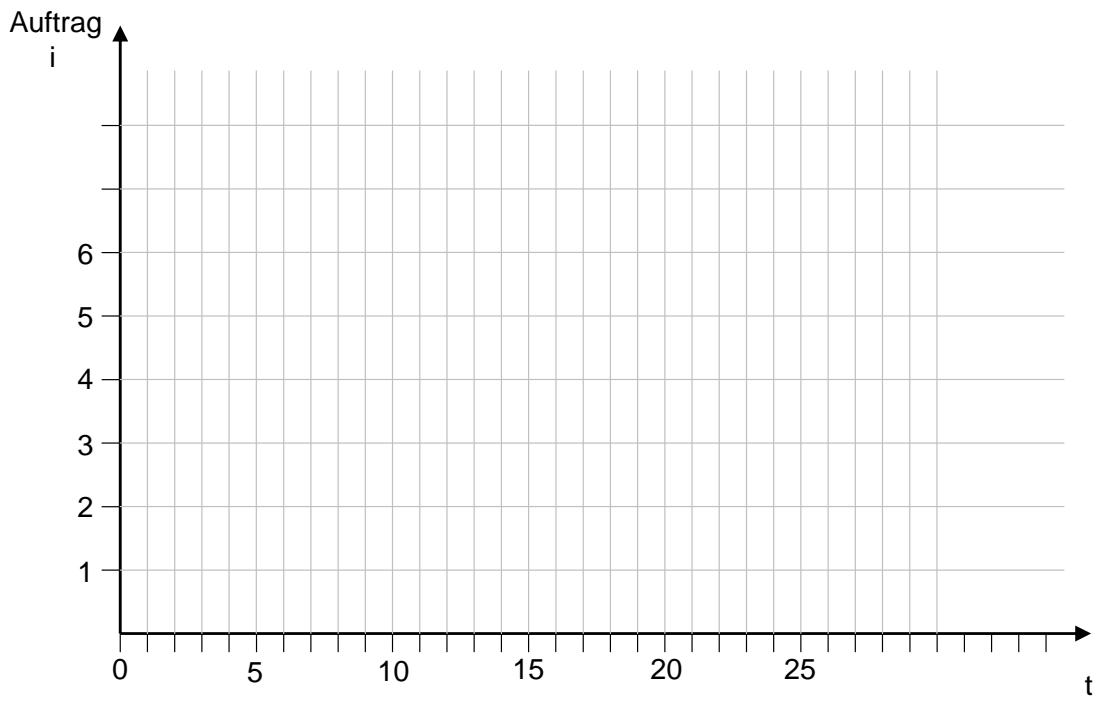
SRPT (die präemptive Variante von Shortest Job First) bedeutet, dass in jedem Zeitintervall einer der Aufträge mit kürzester Restbedienzeit ausgeführt wird. Auftragsunterbrechungen sind höchstens zu Ankunftszeitpunkten von neuen Aufträgen möglich).

Annahme für SJF: Im Fall gleicher Bedienzeiten wird der Prozess ausgeführt, der am längsten wartet.

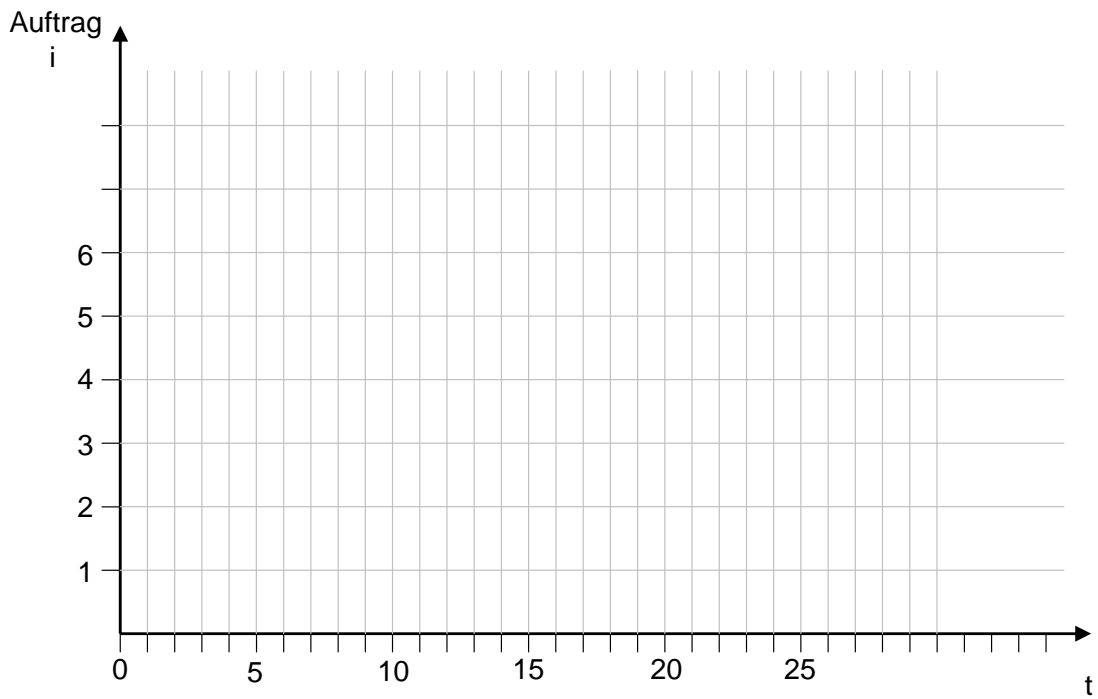
2. Berechnen Sie jeweils die mittleren Wartezeiten! (2 P)

$i =$	1	2	3	4	5	6
$a_i =$	0	2	3	4	6	7
$b_i =$	5	2	2	4	2	8

SJF:

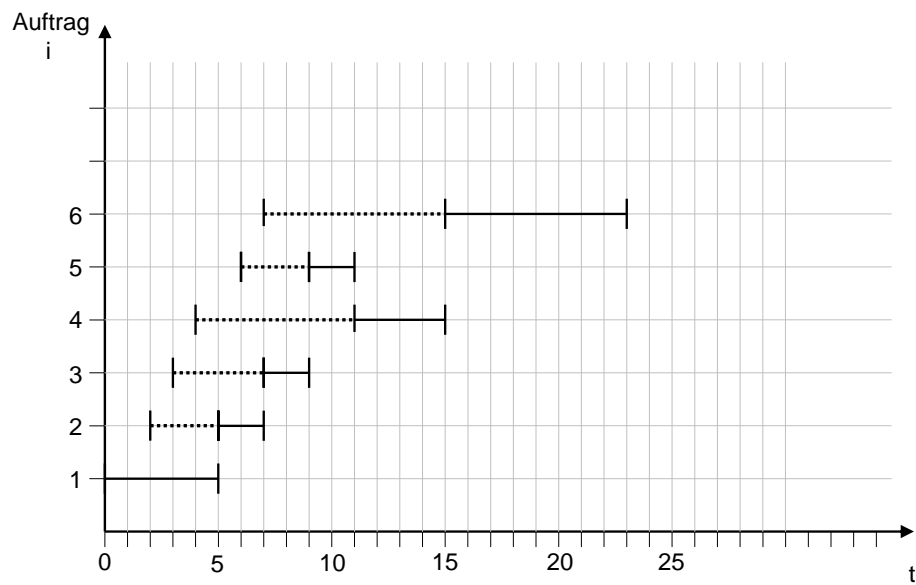


SRPT:



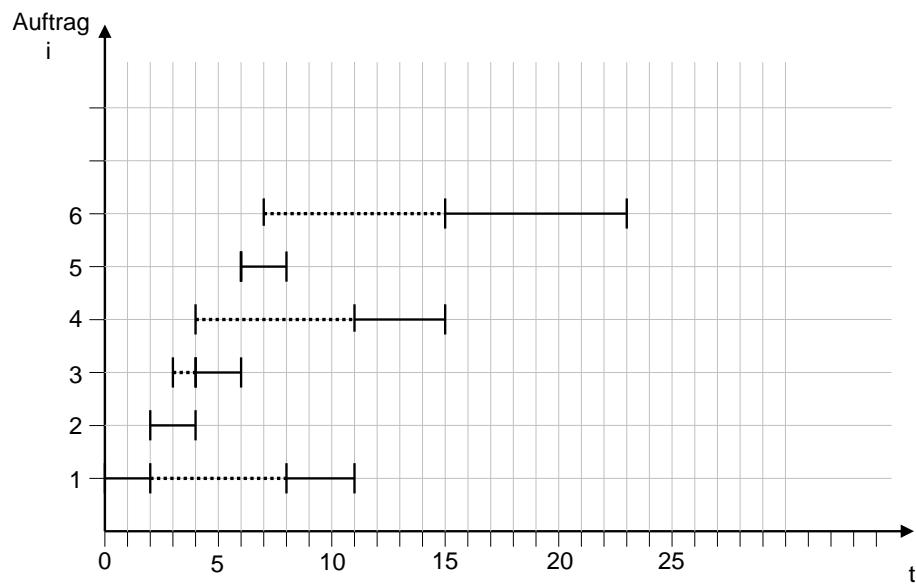
Lösungsvorschläge:

SJF:



$$\bar{W}(SJF) = \frac{1}{6}(0 + 3 + 4 + 7 + 3 + 8) = \frac{25}{6} = 4.1\bar{6}$$

SRPT:



$$\bar{W}(SRPT) = \frac{1}{6}(6 + 0 + 1 + 7 + 0 + 8) = \frac{22}{6} = 3.\bar{6}$$

5 Aufgabe (Memory Management) (8 P)

1. Welche Daten werden vom Betriebssystem typischerweise in einer Halde (Heap) verwaltet und welche Daten in einem Keller (Stack)? Erläutern Sie an Beispielen! (2 P)
2. Worin besteht der Unterschied zwischen externer und interner Fragmentierung von Speicherbereichen? Erläutern Sie! (2 P)
3. Gegeben sei eine Liste freier Speicherbereiche:
100kB - 400kB - 250kB - 200kB - 50kB

Wie verhalten sich jeweils die Strategien "next fit" und "best fit" in Bezug auf die nacheinander eingehenden Speicheranforderungen 30kB, 60kB, 120kB, 20kB, 100kB, 250kB? Halten Sie den Zustand nach jeder Speicheranforderung fest! Benutzen Sie die vorgesehenen leeren Tabellen! (4 P)

next fit:

Anfrage	Liste freier Speicherbereiche				
	100kB	400kB	250kB	200kB	50kB
30kB					
60kB					
120kB					
20kB					
100kB					
250kB					

best fit:

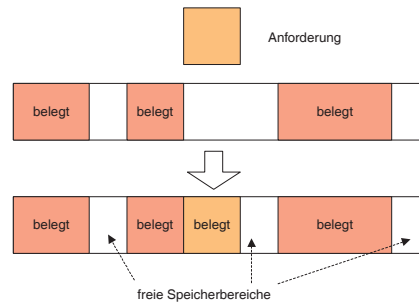
Anfrage	Liste freier Speicherbereiche				
	100kB	400kB	250kB	200kB	50kB
30kB					
60kB					
120kB					
20kB					
100kB					
250kB					

Lösungsvorschlag:

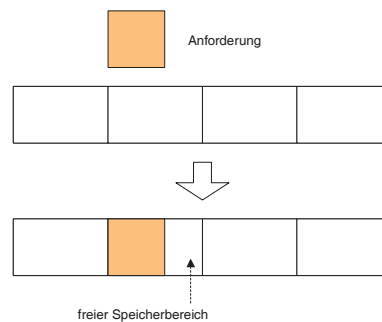
1. Auf der Halde werden dynamisch erzeugte Daten (z.B. Listen, Objekte) verwaltet. Bspw. die in Java mit `new` erzeugten Objekte oder die in C mit `malloc` allozierten Speicherbereiche.

Auf dem Stack werden bspw. die lokalen Variablen von Funktionsaufrufen gespeichert. Der Stack dient somit auch dazu, rekursive Aufrufe von Funktionen aufzulösen.

2. Externe Fragmentierung: Es wechseln sich benutzte und unbenutzte Speicherbereiche innerhalb des Adressraums ab. Speicheranforderungen werden jeweils genau erfüllt.



Interne Fragmentierung: Der Speicher ist in Bereiche fester Größe untergliedert und Speicheranforderungen werden nur in Vielfachen dieser festen Grundgröße befriedigt. Der Verschnitt findet innerhalb dieser Bereiche fester Größe statt.



3. next fit:

Anfrage	Liste freier Speicherbereiche				
	100kB	400kB	250kB	200kB	50kB
30kB	70kB	400kB	250kB	200kB	50kB
60kB	10kB	400kB	250kB	200kB	50kB
120kB	10kB	280kB	250kB	200kB	50kB
20kB	10kB	260kB	250kB	200kB	50kB
100kB	10kB	160kB	250kB	200kB	50kB
250kB	10kB	160kB	0kB	200kB	50kB

best fit:

Anfrage	Liste freier Speicherbereiche				
	100kB	400kB	250kB	200kB	50kB
30kB	100kB	400kB	250kB	200kB	20kB
60kB	40kB	400kB	250kB	200kB	20kB
120kB	40kB	400kB	250kB	80kB	20kB
20kB	40kB	400kB	250kB	80kB	0kB
100kB	40kB	400kB	150kB	80kB	0kB
250kB	40kB	150kB	150kB	80kB	0kB