Theoretische Aufgabe 2

Aufgabe 1)

a) Wie verhält sich eine Applikation, die aus mehreren Prozessen bzw. aus mehreren UserspaceThreads besteht bei einem blockierenden Diskzugriff?

Bei blockierendem Zugriff wird der gesamte Prozess und dessen Threads angehalten. Eine Applikation aus mehreren Prozessen hat also nicht zwingend alle Prozesse blockiert, während die Applikation mit mehreren Threads komplett blockiert.

b) Welche der folgenden Ressourcen werden von allen Threads eines Prozesses geteilt und welche bestehen pro Thread? Program-Counter, Heap-Speicher, globale Variablen, Stack, CPU-Register, geöffnete Dateien, Accounting- und Benutzer-Informationen.

Geteilt: Heap-Speicher Globale Variablen Geöffnete Dateien Accounting- und Benutzer-Informationen

Pro Thread: Program Counter Stack CPU-Register

Aufgabe 2)

a) Langezeit- / Kurzzeitscheduling

Das Langzeit-Scheduling übernimmt die Zuteilung von Zeit und Speicher zu einer Task. Wird eine Task gerade bearbeitet übernimmt das Kurzzeit-Scheduling und teilt der CPU die im Speicher befindlichen Prozesse zu.

b) Warum I/O- / CPU-bound unterscheiden

CPU-bound Prozesse können selbstständig auf der CPU ablaufen, während I/O- bound Prozesse von dem Daten Eingang aus einer anderen Quelle angewiesen sind. Die Prozesse müssen darauf warten, und können erst nach dem Empfang der Daten weiterverarbeitet werden. D.h., damit die Prozesse effizient geschedulet werden können, muss man die Wartezeiten bei I/O-bound Prozessen beachten und diese demnach anders behandeln, als CPU-bound Prozesse.

c) Pros & Cons von premptives/ non-preemptives Multitasking

"Preemptiv" heißt, dass ein Task während der Durchführung zu gunsten eines anderen Tasks unterbrochen werden darf.

Preemptiv

- Pro: Jeder neue Task zu einer Neuzuteilung der CPU führen -> Optimale Laufzeit
- Contra: Prozess kann jeder Zeit die CPU verlieren

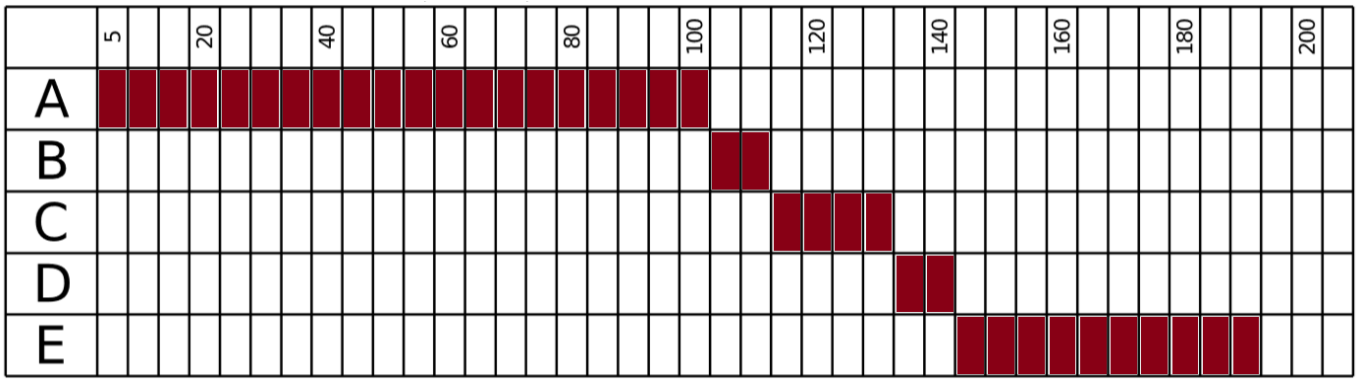
Non-Preemptiv

- Pro: Unterbrechungen von Task kann durch System Calls geregelt werden
- Contra: keine automatische Optimerung

d)

FCFS

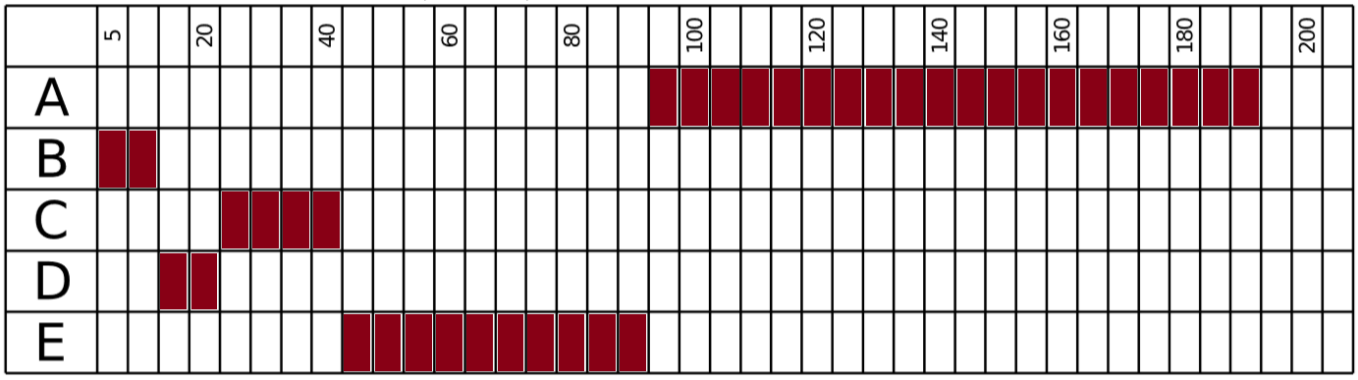
Die Prozesse werden nacheinander, so wie sie reinkommen ausgeführt. Da alle bei t = 0 reinkommen, fangen wir einfach bei A an.



WT = (100 + 110 + 130 + 140)/5 = 96
TT = (100 + 110 + 130 + 140 + 190)/5 = 134

Short Jobs First

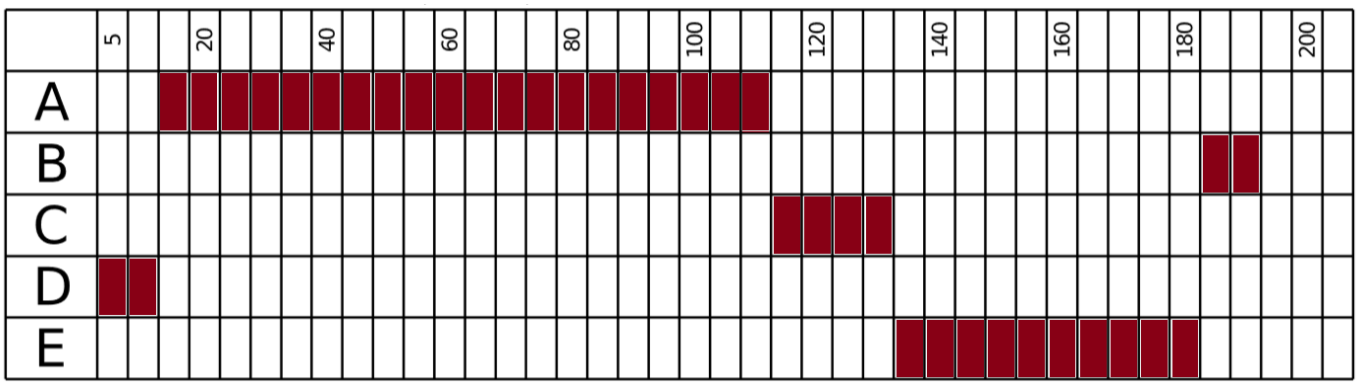
Der Kürzeste Job fängt an.



WT = (10 + 20 + 40 + 90)/5 = 32
TT = (10 + 20 + 40 + 90 + 190)/5 = 70

Priority Scheduling

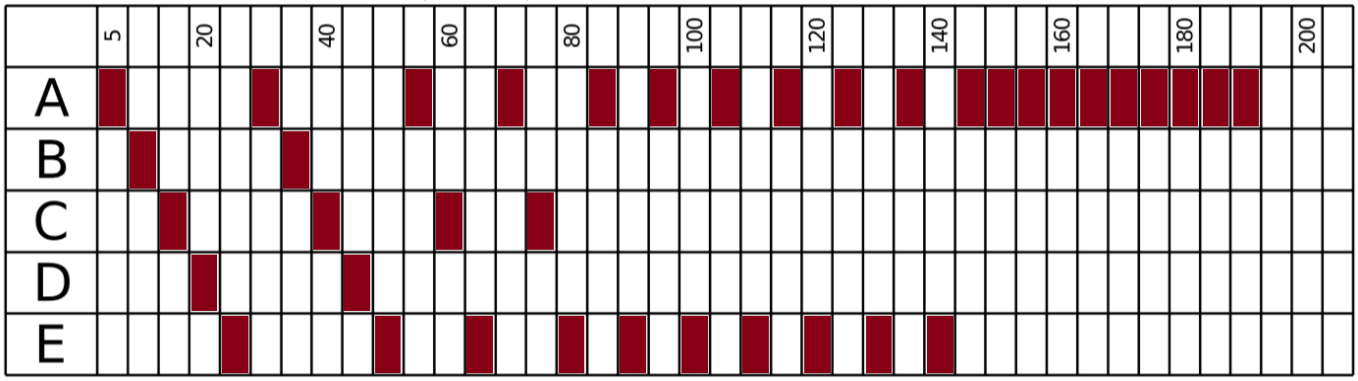
Die höchste Priorität fängt an.



WT = (10 + 110 + 130 + 180)/ 5 = 86
TT = (10 + 110 + 130 + 180 + 190)/ 5 = 124

Round Robin mit einer Zeitscheibe von 5

Pseudo - Parallelität: nach 5 Zeiteinheiten wird der nächste Prozess fortgesetzt.



WT(A) = 20 + 20 + 10 + 10 + 5 + 5 + 5 + 5 + 5 + 5 = 90
WT(B) = 5 + 20 = 25
WT(C) = 10 + 20 + 15 + 10 = 55
WT(D) = 15 + 20 = 35
WT(E) = 20 + 20 + 10 + 10 + 5 + 5 + 5 + 5 + 5 + 5 = 80

WT = 50
TT(A) = 190
TT(B) = 35
TT(C) = 75
TT(D) = 45
TT(E) = 140
TT = 97

Aufgabe 3)

a) Race Conditions, Deadlock, Starvation und Lösungen

```
global const buf_size := 10
global var buf[buf_size]
global var num := 0

function boolean push(a)
    if (num < buf_size)
        buf[num] := a
        num := num + 1
        return true
    else
        return false
    end
end

function pop()
    if  (num > 0)
        num := num - 1
        return buf[num]
    else
        return null
    end
end
```

Die Raceconditions entstehen an den Stellen, da mehrer Threads gleichzeitig auf die globalen Variablen zu greifen und diese auch global ändern. Wenn Thread 1 in die Section geht und num um 1 erhöht, ist Thread 2 evtl. auch schon in der If-Bedingung drin, wenn es aber bei `buf[num]` ist hat Thread 1 num schon hochgezählt. Dadurch kann es schnell zu Fehlern kommen. Das gleich kann auch bei `function pop()` passieren.

Raceconditions kann man leicht mit Semaphores lösen. Bspw. kann man atomic Variablen verwenden, auf die nur ein Thread gleichzeitig zugreifen kann. Man könnte auch mit verschiedenen Bool Variablen arbeiten, um den Zugang für andere Threads zu blockieren, bis der Aktuelle fertig ist.

b) Pseudocode zu TestAndSet, Swap, FetchAndAdd

```
atomar function testAndSet(a,b,c)
    if(compare(a,b))
```

```
        then set(c)
        and  return true
    else return false
    end
end

atomar function swap(other)
    set(tmp, other)
    set(other, this)
    set(this, tmp)
end

atomar function FetchAndAdd(Value)
    old = get(this)
    set(this, value)
    return old
end
```

c) binäre Semaphore

```
typedef struct semaphor{
  int a = 0;
  int b = 0;
    semaphor(){}
    wait(){
        while(!TestAndAdd(a, b, -1)){}
    }
    signal(){
        TestAndAdd(a,b, 1)
    }
}

atomar function TestAndAdd(int &a,int &b, c)
    if (a == b)
        a := a + c
        return true
    else
        return false
    end
end
```

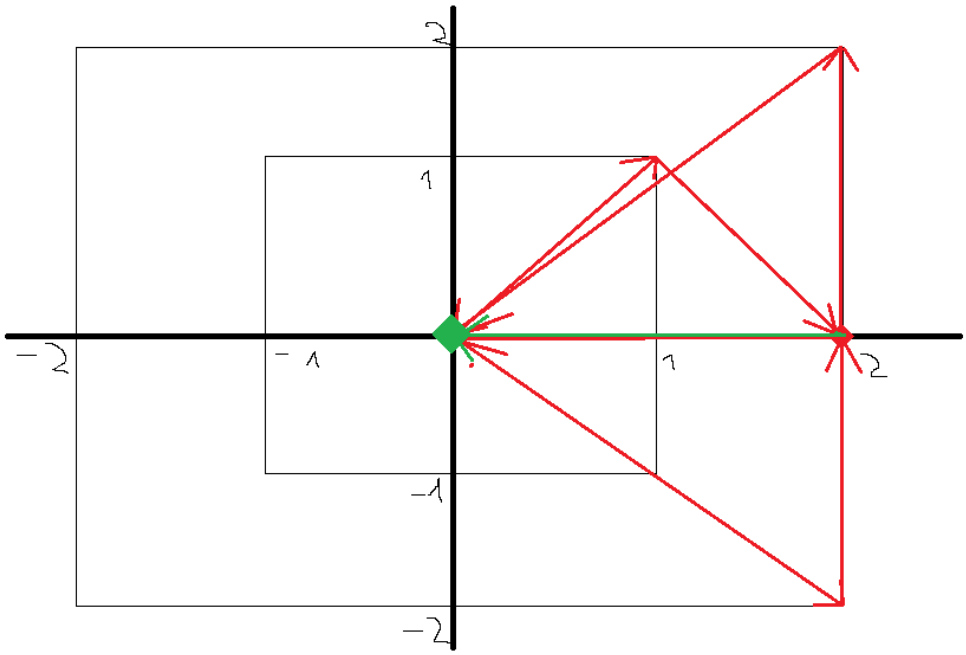
Um eine binäre Semaphore mit TestAndAdd zu erstellen, muss man sie mithilfe von 2 Werten aufbauen, die durch die TestAndAdd verändert werden.
Wird in einem Thread nun die wait() Funktion aufgerufen, erfüllt dieser zunächst den Test und ändert den Wert a (c = -1). Dadurch bestehen alle anderen Threads den Test nicht, und bleiben in der while-Schleife hängen. Ist der erste Thread fertig und ruft podt() auf, wird die Änderung von a wieder Rückgängig gemacht (c = 1) und der nächste Thread besteht wieder den Test und ändert a.

d) Dining-Philosophers-Problem: Ist die Lösung Deadlockfrei?

Circular wait wird ausgeschlossen.
Es ist nicht starvation free, da ein Philosoph nachdem er fertig ist mit Essen und die Gabeln wieder hinlegt, er sie sofort wieder aufnehmen kann bevor der andere der auf ihn wartet sie sich nehmen kann.

Aufgabe 4)

1.



Unter der Annahme, dass B1 und A1-6 jeweils eine Sekunde dauern, und der Rest keine Zeit benötigt sieht das Bild nach 9s wie oben skizziert aus. Läuft das Programm in einer Endlosschleife ist nach weiteren 8s ist ein um 2 verschobenes Dreieck der selben Form gezeichnet. Nach 3 weiteren Sekunden, also bei Sekunde 20 befinden sich der Zeichnkopf demnach bei
(-2, -2).

2.

```
101 Semaphore S1=0
102 Semaphore S2=-1
103 Semaphore S3=-2
104 start(P1,P2,P3)

200 down(S1)        //Start: P1 startet: S1 = -1
205 (A1)
210 (A4)
212 up(S2)          //S2 = 0, P2 Startet
213 down(S1)        //S1 = -2, P1 Wartet
215 (A3)
216 up(S2)        //S2 = -1
217 up(S2)        //S2 = 0, P2 Startet
218 down(S1)        //S1 = -1, P1 Wartet
```

```
300 down(S2)        //P2 startet: S2 = -1
305 (A1)
307 up(S3)        //S3 = -1
308 up(S3)        //S3 = 0, P3 Startet
309 down(S2)        //S2 = -2, P2 Wartet
310 (A2)
311 up(S1)        //S1 = 0, P1 Startet
312 down(S2)        //S2 = -1, P2 Wartet  Ende: S1 = 0, S2 = -1, S3 = -2

400 down(S3)        //P3 Startet: S3 = -1
405 (A5)
410 (A6)
415 (A2)
416 up(S1)        //S1 = -1
417 up(S1)        //S1 = 0, P1 Startet
418 down(S3)        //S3 = -2, P3 Wartet
```