# Übungen Betriebssysteme (BS)

## U2 – Thread-Synchronisation

https://moodle.tu-dortmund.de/course/view.php?id=34604

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

https://sys.cs.tu-dortmund.de/EN/People/ulbrich/

technische universität
dortmund

arbeitsgruppe
systemsoftware

# Theoriefrage 3

**Vorteil: *Virtual Round Robin* ⇔ *Round Robin*?**

# Theoriefrage 3

**Vorteil: *Virtual Round Robin ⇔ Round Robin*?**

- ***Round Robin:***
  Auch wenn Zeitscheibe *nicht komplett aufgebraucht* wurde,
  → Einsortierung am Ende der Warteliste
  (obwohl sie nur wenig CPU-Zeit in Anspruch genommen haben)

# Theoriefrage 3

**Vorteil: *Virtual Round Robin* ⇔ *Round Robin*?**

- ***Round Robin:***
  Auch wenn Zeitscheibe *nicht komplett aufgebraucht* wurde,
  → Einsortierung am Ende der Warteliste
  (obwohl sie nur wenig CPU-Zeit in Anspruch genommen haben)
- ***Virtual Round Robin:***
  - Wenn Zeitscheibe noch nicht aufgebraucht wurde,
    → Einsortierung auf der **Vorzugsliste**.
  - Prozesse aus der Vorzugsliste dürfen ***vor*** der Warteschlange ihre **Restlaufzeit** aufbrauchen

# Theoriefrage 3

## Vorteil: *Virtual Round Robin* ⇔ *Round Robin*?

- ***Round Robin:***
  Auch wenn Zeitscheibe *nicht komplett aufgebraucht* wurde,
  → Einsortierung am Ende der Warteliste
  (obwohl sie nur wenig CPU-Zeit in Anspruch genommen haben)
- ***Virtual Round Robin:***
  - Wenn Zeitscheibe noch nicht aufgebraucht wurde,
    → Einsortierung auf der **Vorzugsliste**.
  - Prozesse aus der Vorzugsliste dürfen ***vor*** der Warteschlange ihre **Restlaufzeit** aufbrauchen

## Implementierungsunterschiede:

# Theoriefrage 3

## Vorteil: *Virtual Round Robin* ⇔ *Round Robin*?

- ***Round Robin:***
  Auch wenn Zeitscheibe *nicht komplett aufgebraucht* wurde,
  → Einsortierung am Ende der Warteliste
  (obwohl sie nur wenig CPU-Zeit in Anspruch genommen haben)
- ***Virtual Round Robin:***
  - Wenn Zeitscheibe noch nicht aufgebraucht wurde,
    → Einsortierung auf der **Vorzugsliste**.
  - Prozesse aus der Vorzugsliste dürfen ***vor*** der Warteschlange ihre **Restlaufzeit** aufbrauchen

## Implementierungsunterschiede:

- ***Struktur:*** → **Vorzugsliste**

# Theoriefrage 3

## Vorteil: *Virtual Round Robin* ⇔ *Round Robin*?

- ***Round Robin:***
  Auch wenn Zeitscheibe *nicht komplett aufgebraucht* wurde,
  → Einsortierung am Ende der Warteliste
  (obwohl sie nur wenig CPU-Zeit in Anspruch genommen haben)
- ***Virtual Round Robin:***
  - Wenn Zeitscheibe noch nicht aufgebraucht wurde,
    → Einsortierung auf der **Vorzugsliste**.
  - Prozesse aus der Vorzugsliste dürfen ***vor*** der Warteschlange ihre **Restlaufzeit** aufbrauchen

## Implementierungsunterschiede:

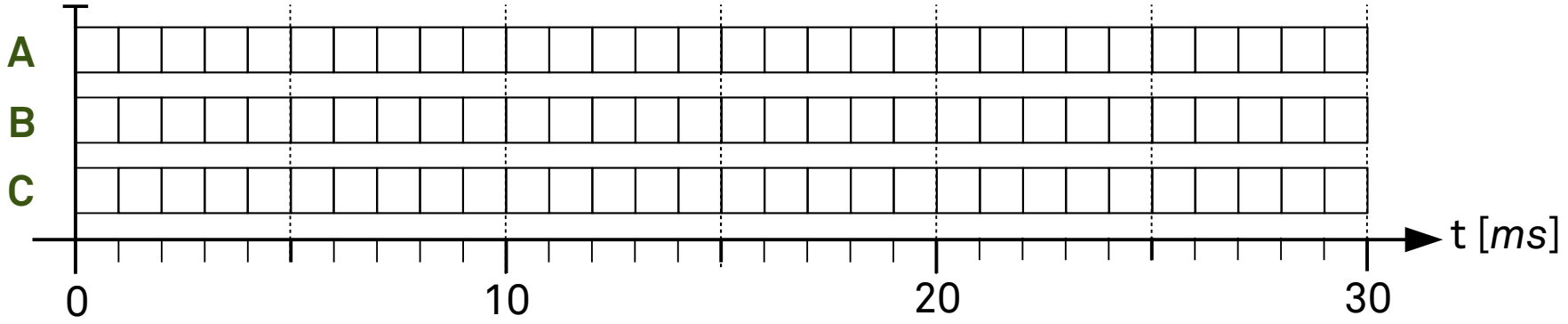
- ***Struktur:*** → **Vorzugsliste**
- ***Verhalten:***
  - Bei Restlaufzeit → Vorzugsliste
  - Aufbrauchen der Restlaufzeit → Warteliste

# Theoriefrage 1

**Wie sieht die Prozesszuteilung nach dem Verfahren *Virtual Round Robin* aus?**

**Zeitscheibe: 3 ms**

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |

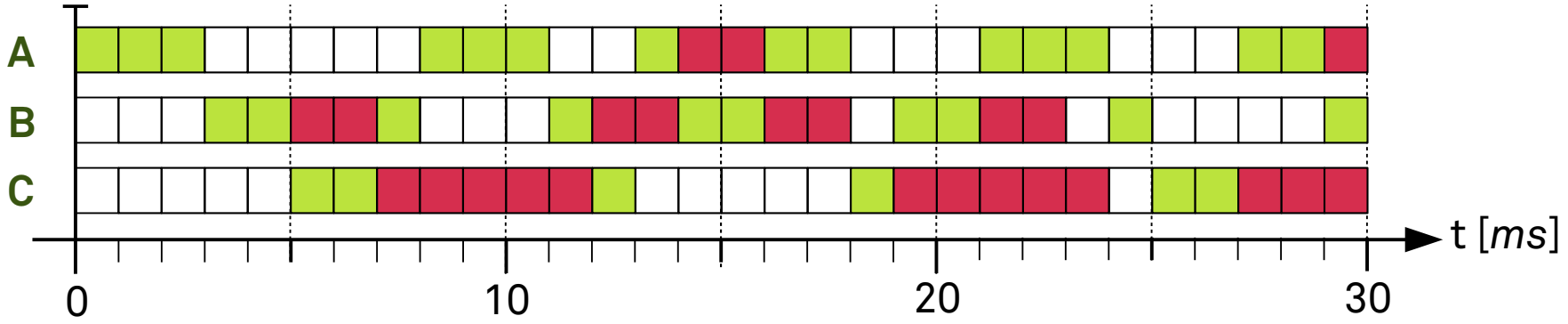


# Theoriefrage 1

**Wie sieht die Prozesszuteilung nach dem Verfahren *Virtual Round Robin* aus?**

**Zeitscheibe: 3 ms**

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |

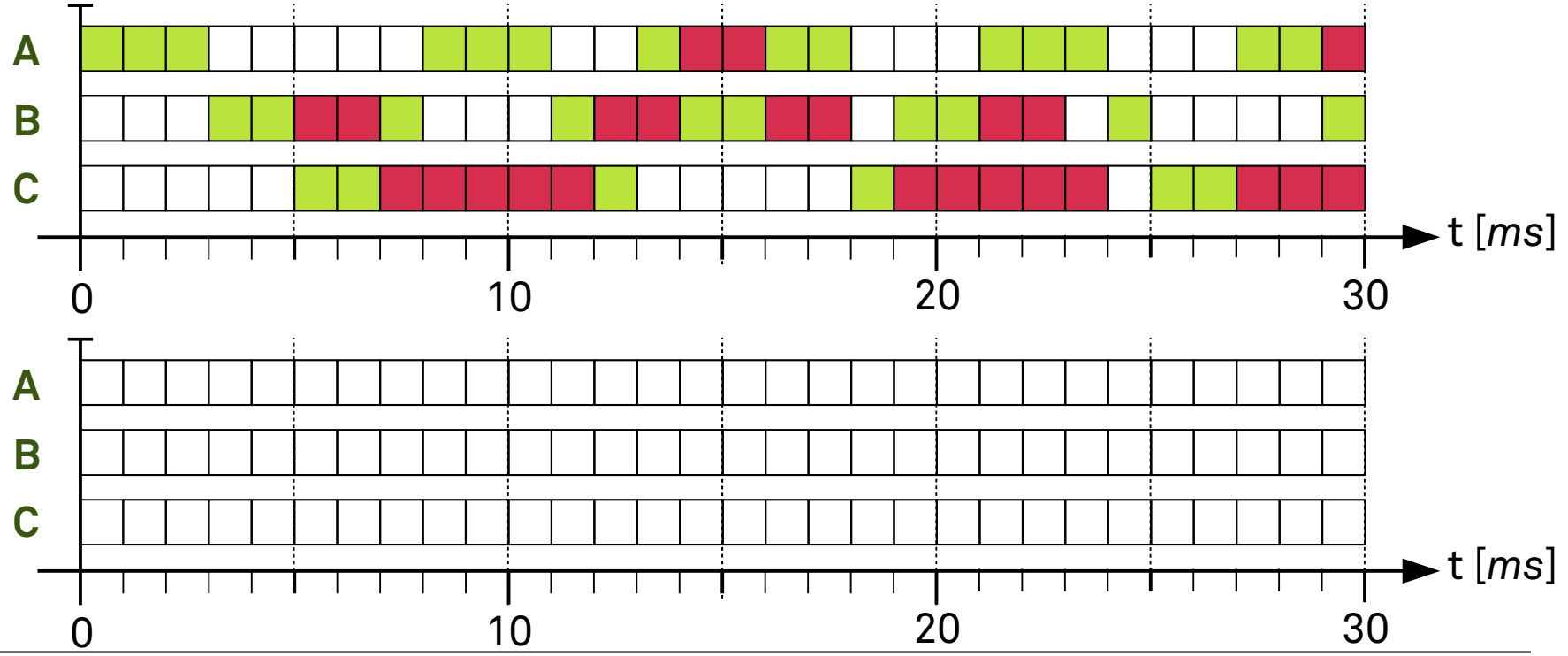


# Theoriefrage 2

**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |

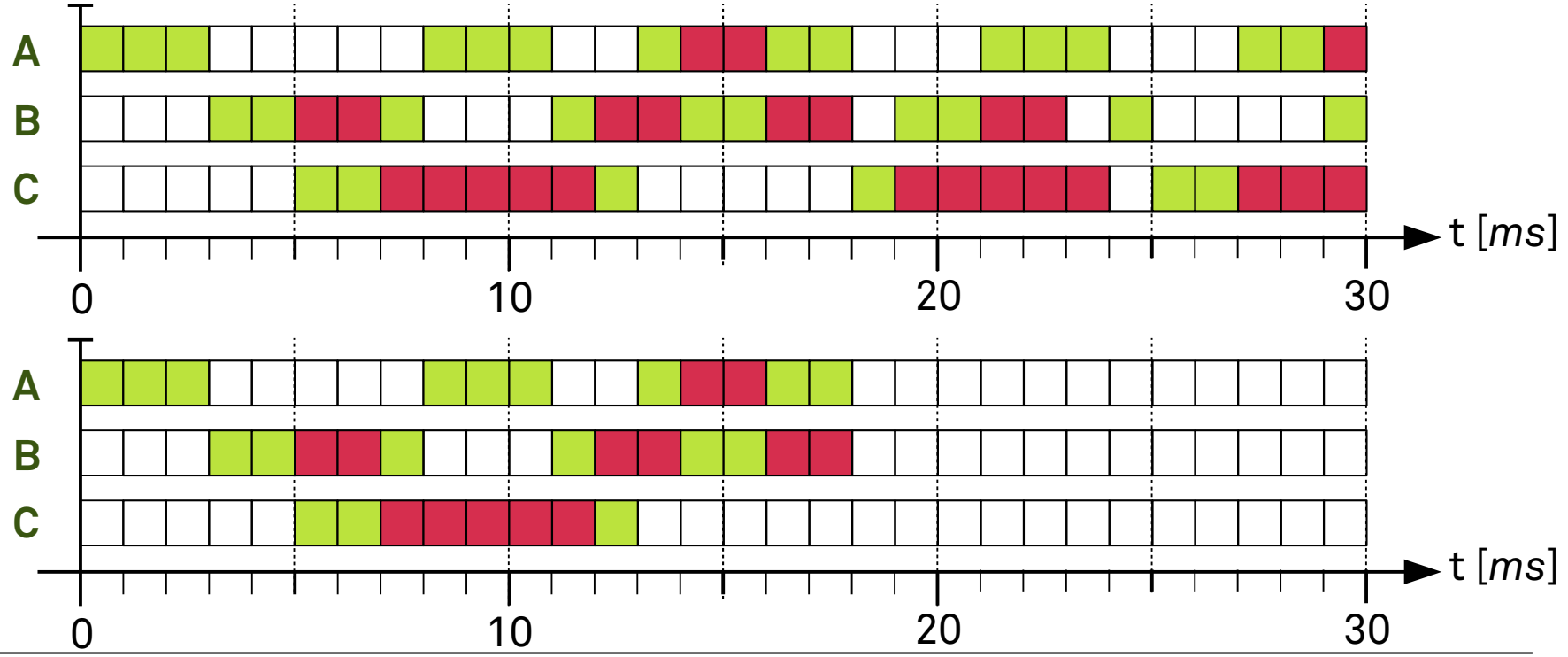


# Theoriefrage 2

**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |

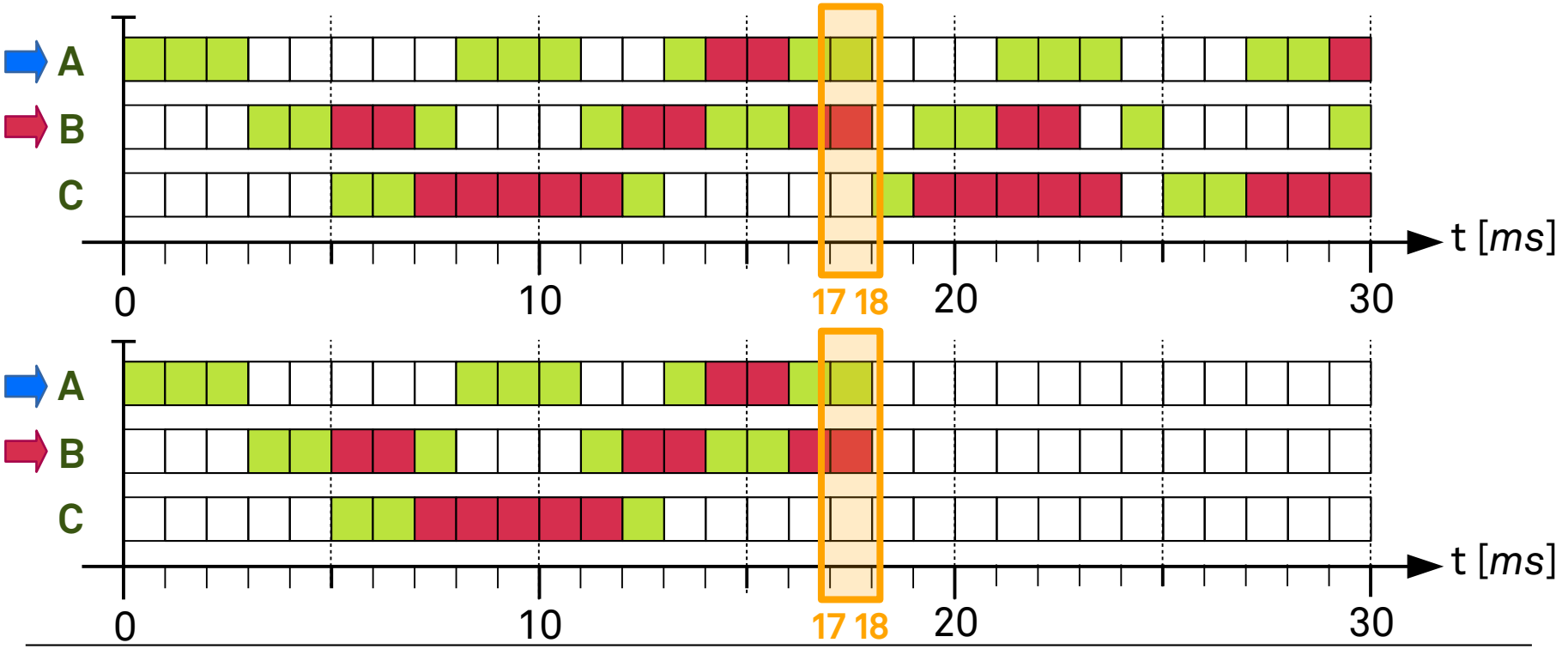


# Theoriefrage 2

**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

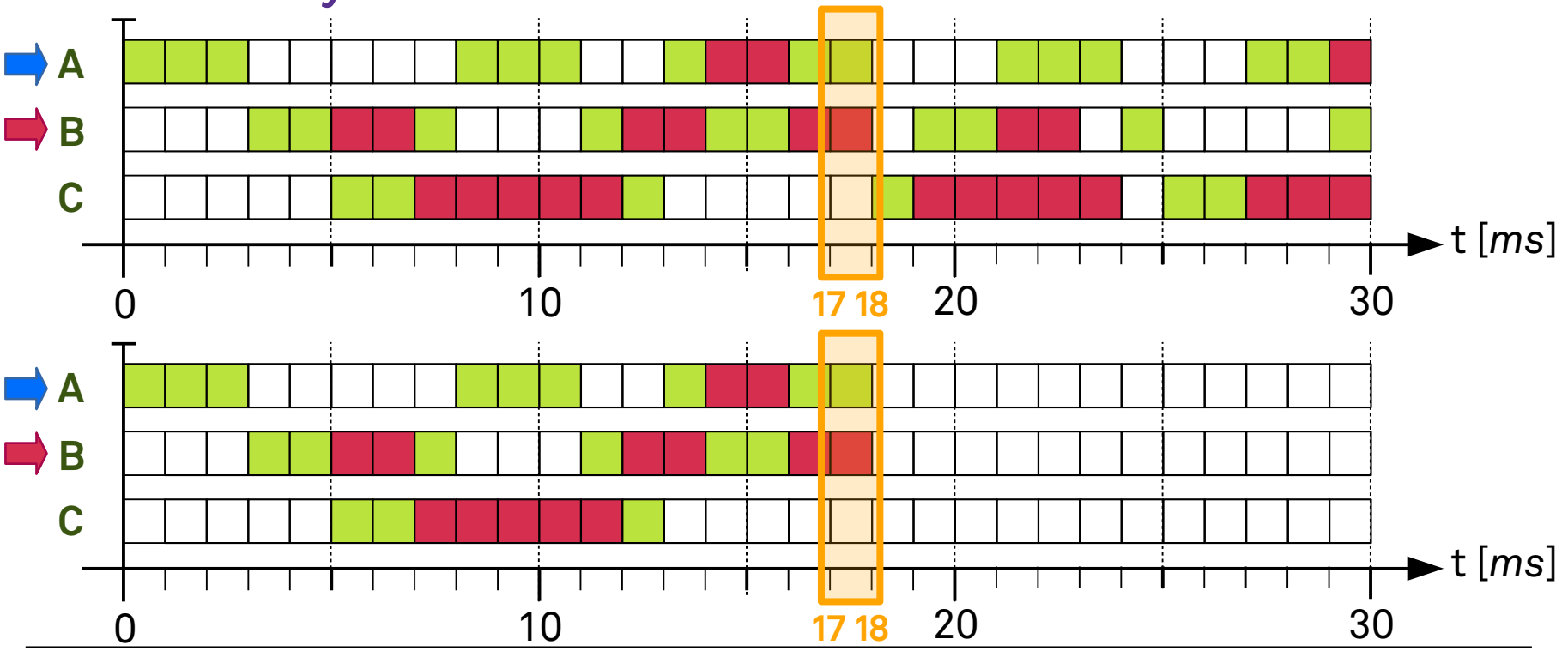
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

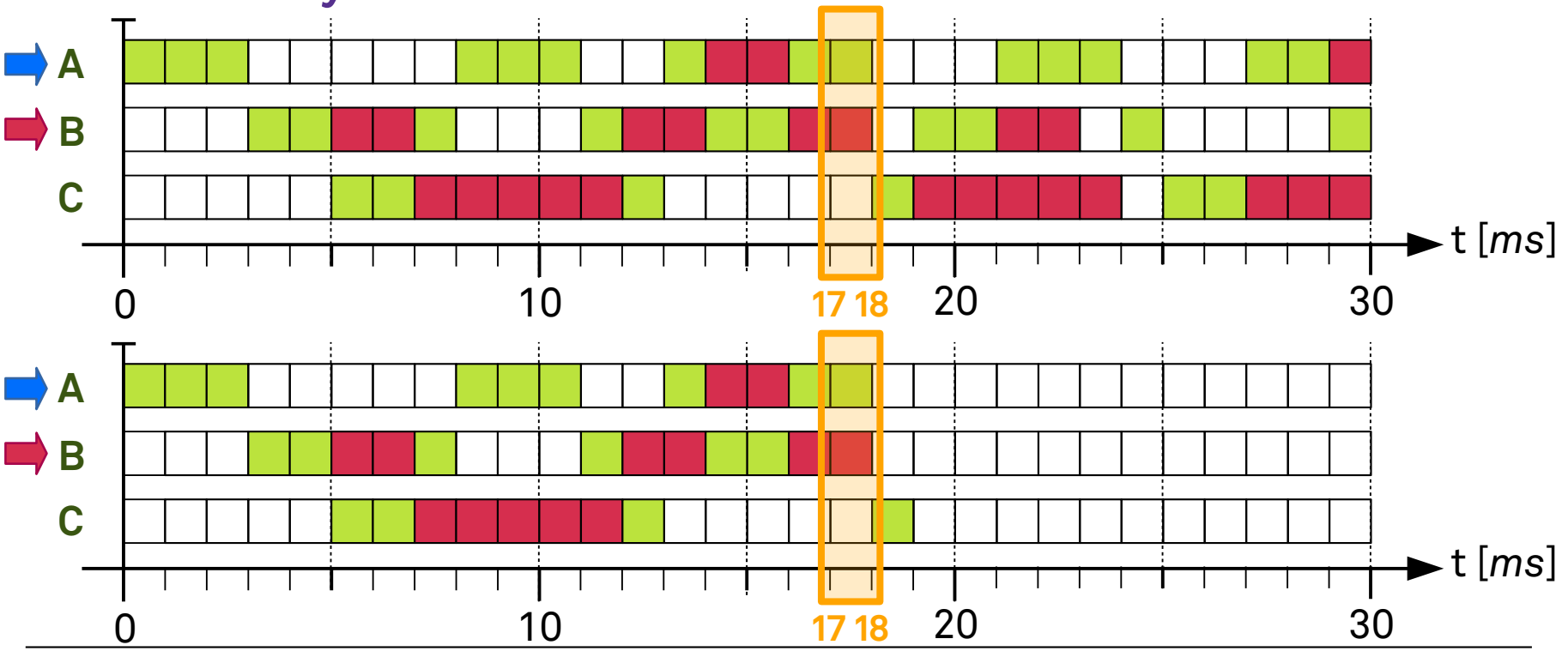
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

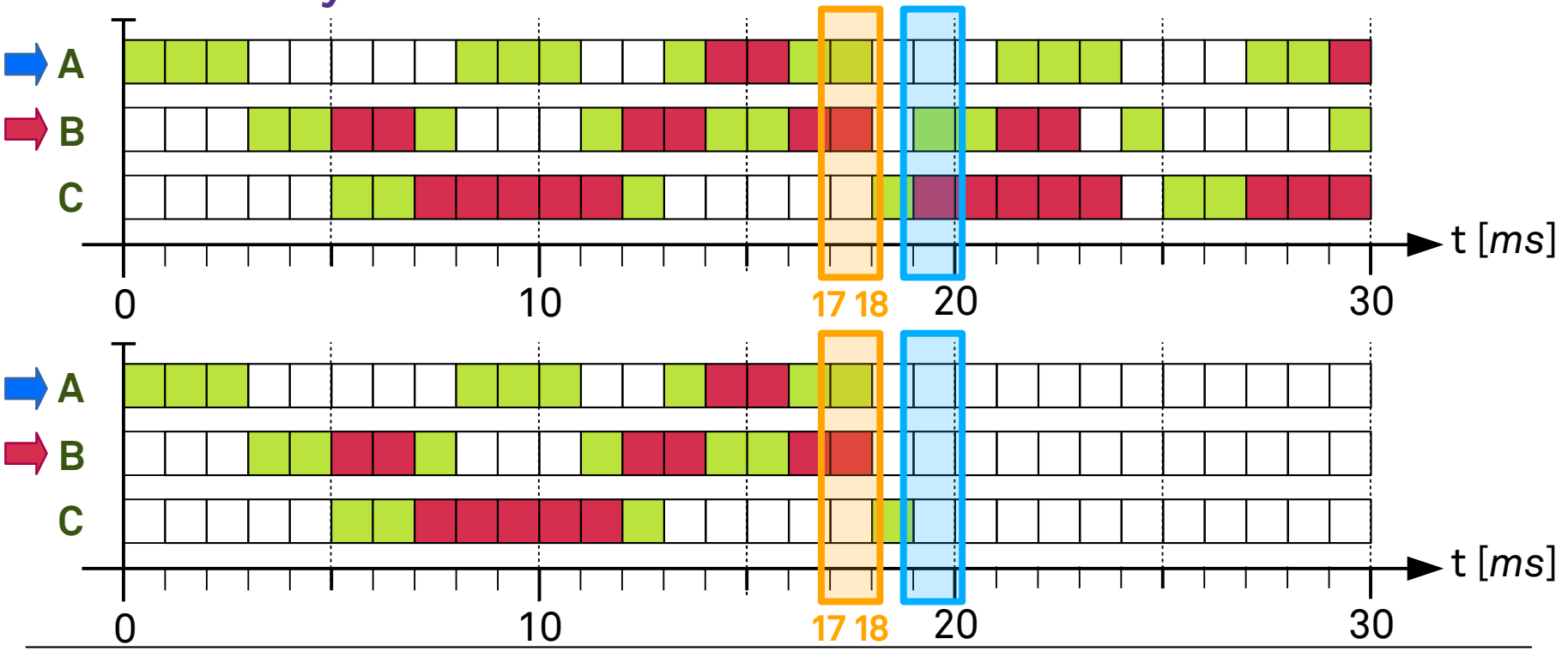
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

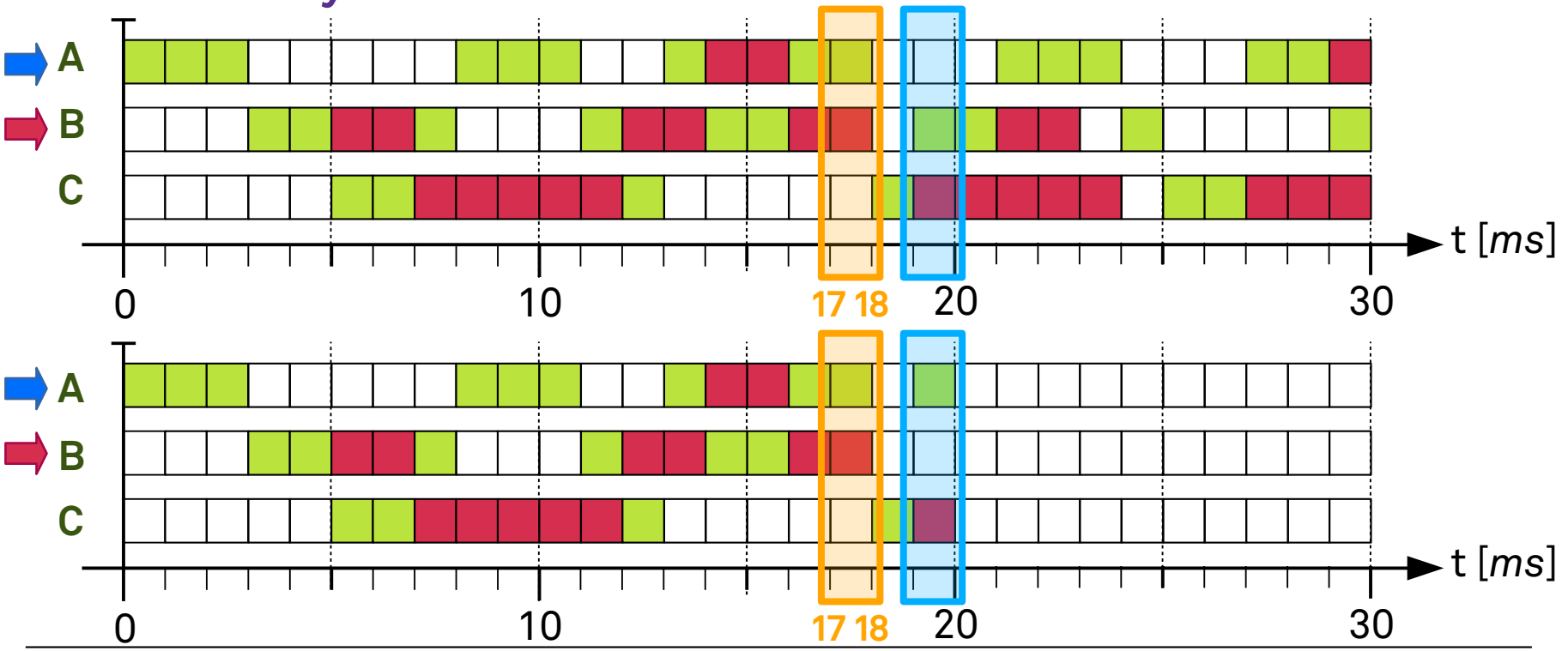
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

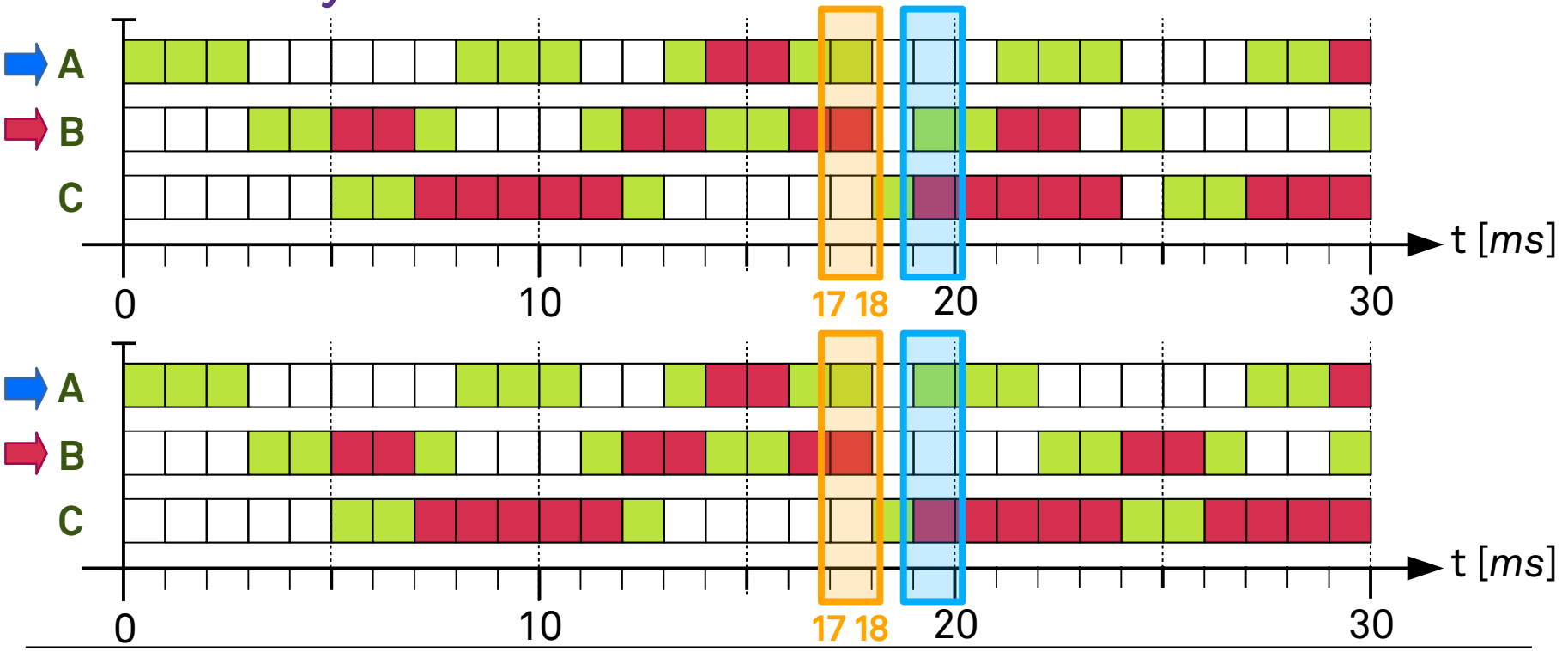
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

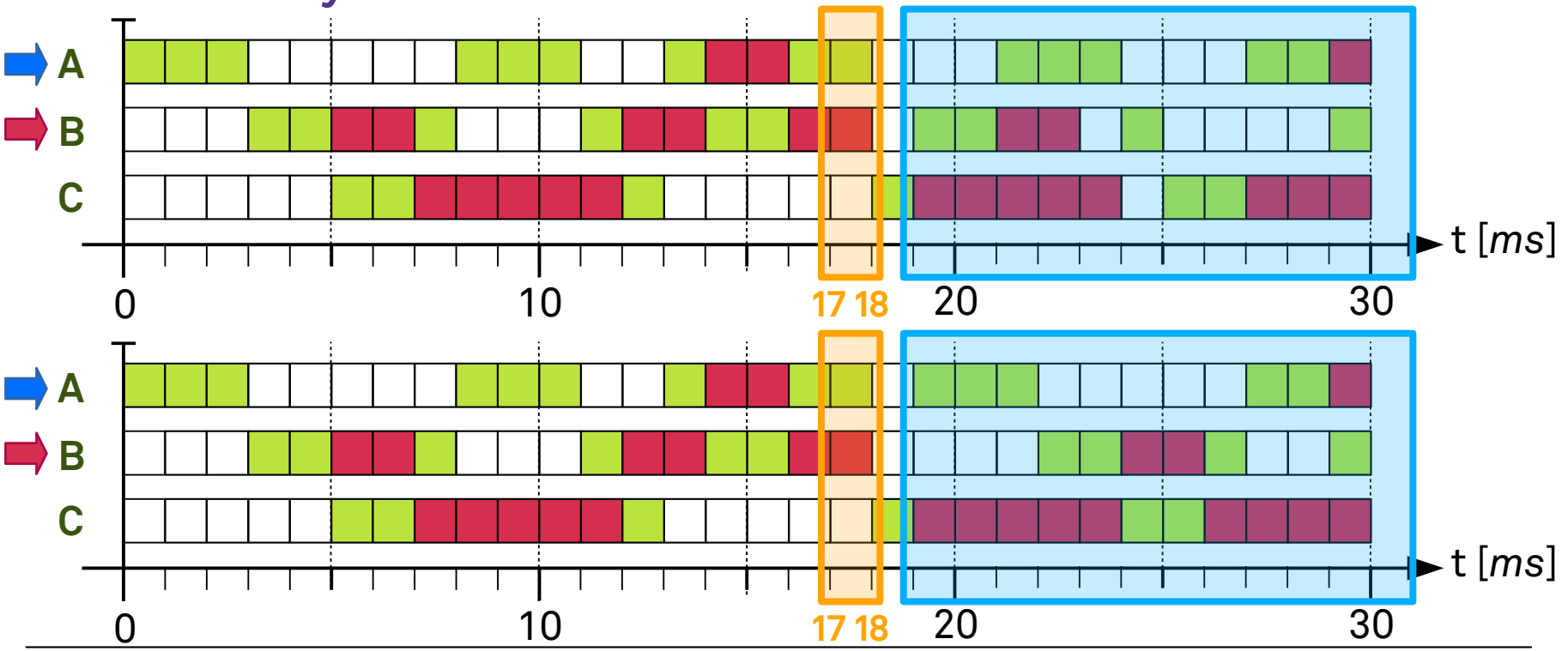
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 2

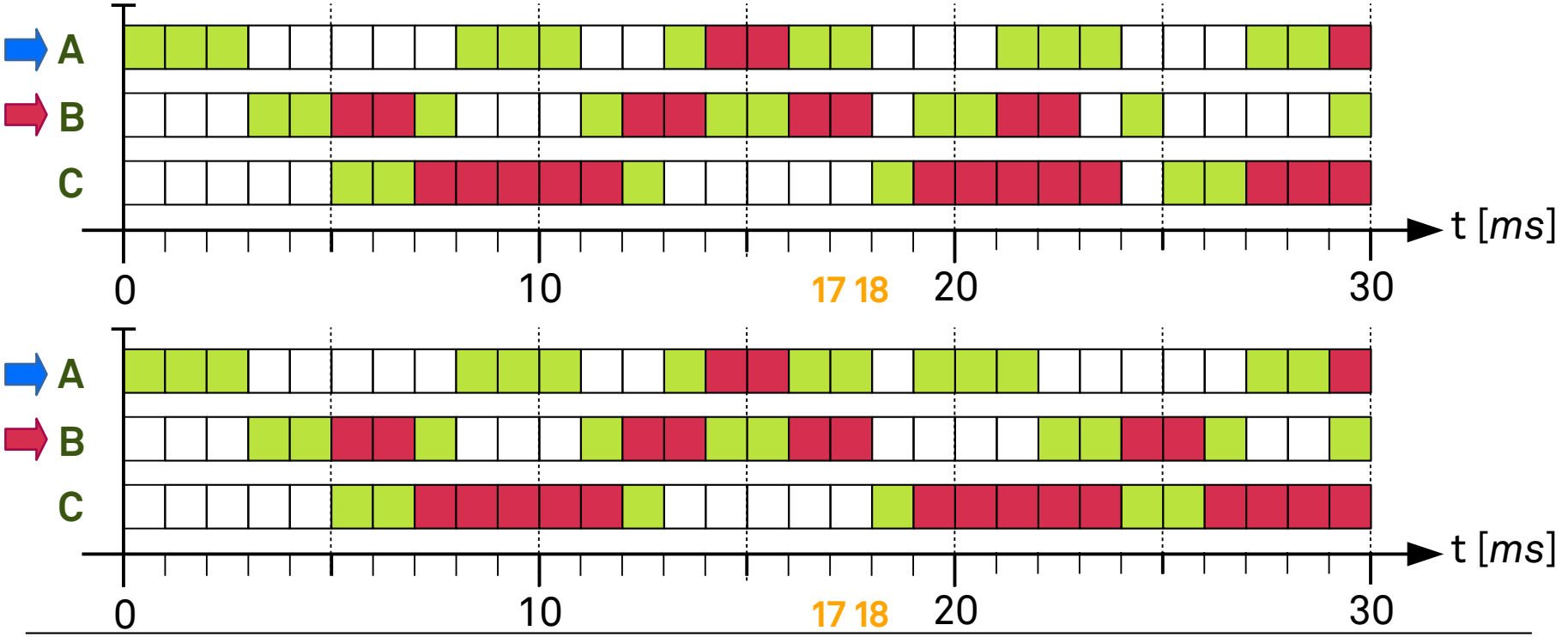
**Schaut euch den Zeitraum von 17-19 ms an? Was für ein Problem kann an dem Zeitpunkt 19 entstehen?**

**Zeitscheibe: 3 ms**

**A:** **Zeitscheibe** *aufgebraucht*
→ *Ready-Liste*

**B:** **I/O-Burst** *beendet*
→ *Ready-Liste*

| *Prozess* | *CPU-Burst* | *I/O-Burst* |
|---|---|---|
| A | 7 | 2 |
| B | 2 | 2 |
| C | 2 | 5 |



# Theoriefrage 4

**Was macht *Shortest Process Next* (*SPN*) anders als *Round Robin* bzw. *Virtual Round Robin*?**

# Theoriefrage 4

**Was macht *Shortest Process Next* (*SPN*) anders als *Round Robin* bzw. *Virtual Round Robin*?**

+ ***Kenntnis*/*Abschätzung*** der Prozesslaufzeit
+ **Keine Verdrängung** von Prozessen
➔ **Geringere Benachteiligung kurzer CPU-Stöße**

# Theoriefrage 4

**Was macht *Shortest Process Next* (*SPN*) anders als *Round Robin* bzw. *Virtual Round Robin*?**

+ ***Kenntnis*/*Abschätzung*** der Prozesslaufzeit
+ **Keine Verdrängung** von Prozessen
➔ **Geringere Benachteiligung kurzer CPU-Stöße**

**Welche *Herausforderungen* bzw. *Problemen* bestehen bei *Shortest Process Next* typischerweise?**

# Theoriefrage 4

**Was macht *Shortest Process Next* (*SPN*) anders als *Round Robin* bzw. *Virtual Round Robin*?**

- \+ ***Kenntnis*/*Abschätzung*** der Prozesslaufzeit
- \+ **Keine Verdrängung** von Prozessen
- ➔ **Geringere Benachteiligung kurzer CPU-Stöße**

**Welche *Herausforderungen* bzw. *Problemen* bestehen bei *Shortest Process Next* typischerweise?**

- ***Wie*** kann die **exakte Laufzeit** ***vorhergesagt*** werden?
- Gefahr der ***Aushungerung*** bei ***langen CPU-Stößen***

# Theoriefrage 5

**Fasst die zwei *wichtigsten Unterschiede* zwischen *Semaphoren* und *Mutexen* zusammen.**

# Theoriefrage 5

**Fasst die zwei *wichtigsten Unterschiede* zwischen *Semaphoren* und *Mutexen* zusammen.**

- **`Mutex`** (*Mut*ual *ex*clusion):

# Theoriefrage 5

**Fasst die zwei *wichtigsten Unterschiede* zwischen *Semaphoren* und *Mutexen* zusammen.**

- **`Mutex`** (*Mut*ual *ex*clusion):
  - ***2** Zustände*: ***LOCKED*/*UNLOCKED*** (*binär*)
  - ***`unlock()`*** kann **NUR** vom ***Besitzer*** aufgerufen werden

# Theoriefrage 5

**Fasst die zwei *wichtigsten Unterschiede* zwischen *Semaphoren* und *Mutexen* zusammen.**

- **`Mutex`** (*Mut*ual *ex*clusion):
  - ***2** Zustände*: ***LOCKED*/*UNLOCKED*** (*binär*)
  - ***`unlock()`*** kann **NUR** vom ***Besitzer*** aufgerufen werden
- **`Semaphor`**:

# Theoriefrage 5

**Fasst die zwei *wichtigsten Unterschiede* zwischen *Semaphoren* und *Mutexen* zusammen.**

- **Mutex** (*Mutual exclusion*):
  - ***2** Zustände*: ***LOCKED*/*UNLOCKED*** (*binär*)
  - ***unlock()*** kann **NUR** vom ***Besitzer*** aufgerufen werden
- **Semaphor**:
  - ***„Nicht-negative ganze Zahl“*** (*zählend*)
  - Operationen können **von allen** aufgerufen werden
  - → **P** (wait):
    - `= 0`: *Blockierung* des *aktuellen* Prozesses
    - `> 0`: `Semaphor-Wert -= 1`
  - → **V** (signal):
    - `= 0`:
      - `> 0` *Blockierungen* → ***DeBlockierung*** von *1 Prozess*
      - `= 0` Blockierungen → `Semaphor-Wert += 1`
    - `> 0`: `Semaphor-Wert += 1`

# Programmieren in C (1) (a)

```c
int main(int argc, char *argv[]) {
  pthread_t servicek[ANZ_SERVICEK];
  struct zu_bedienen arg[ANZ_SERVICEK];

  /// Servicekraft beginnt mit der Arbeit
  for (int i = 0; i < ANZ_SERVICEK; ++i) {
    arg[i] = bedienliste_fuer(i);
    int fehler = pthread_create(&servicek[i], NULL, &bedienen, &arg[i]);
    if (fehler) {
      printf("Konnte Servicekraft %d nicht starten", i);
      exit(EXIT_FAILURE);
    }
  }

  /// Servicekraft macht Feierabend
  for (int i = 0; i < ANZ_SERVICEK; ++i) {
    int fehler = pthread_join(servicek[i], NULL);
    if (fehler) {
      printf("Konnte Servicekraft %d nicht beenden", i);
      exit(EXIT_FAILURE);
    }
  }

  return EXIT_SUCCESS;
}
```

# Programmieren in C (1) (b)

**(1.a) Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms?**

# Programmieren in C (1) (b)

**(1.a) Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms?**

**(1.b) Wie nennt man eine solche Situation?**

**(1.c) Welche Ressource wird geteilt?**

# Programmieren in C (1) (b)

**(1.a) Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms?**

- Die Ausgabe geht durcheinander,
  weil die Küche mehrfach gleichzeitig benutzt wird

**(1.b) Wie nennt man eine solche Situation?**

**(1.c) Welche Ressource wird geteilt?**

# Programmieren in C (1) (b)

**(1.a) Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms?**

- Die Ausgabe geht durcheinander,
  weil die Küche mehrfach gleichzeitig benutzt wird

**(1.b) Wie nennt man eine solche Situation?**

- ***Race Condition***

**(1.c) Welche Ressource wird geteilt?**

# Programmieren in C (1) (b)

**(1.a) Welches Problem beobachtet ihr bei der Ausführung des entwickelten Programms?**

- Die Ausgabe geht durcheinander,
  weil die Küche mehrfach gleichzeitig benutzt wird

**(1.b) Wie nennt man eine solche Situation?**

- ***Race Condition***

**(1.c) Welche Ressource wird geteilt?**

- ***kochen-Funktion***

# Programmieren in C (2) (b)

**(2) Warum muss das *Argument* der Threads den Aufruf von `pthread_create`(3) überleben?**

# Programmieren in C (2) (b)

**(2) Warum muss das *Argument* der Threads den Aufruf von `pthread_create`(3) überleben?**

- Argument wird als ***Call by Reference*** (**Pointer**) übergeben
- **`bedienen`**`(`*`void*`*`)` Funktion der Threads muss zu ihrer ***gesamten Laufzeit*** auf das **Argumente** zugreifen können
- Daher muss das Argument mindestens so lange leben wie alle Threads

# Programmieren in C (3) (b)

**(3) Warum ist die `pthread_self`(3) nicht geeignet, um die zu *bedienenden Gäste* mittels *`bedienliste_fuer`* innerhalb der `bedienen` Funktion zu ermitteln?**

# Programmieren in C (3) (b)

**(3) Warum ist die `pthread_self`(3) nicht geeignet, um die zu *bedienenden Gäste* mittels *`bedienliste_fuer`* innerhalb der `bedienen` Funktion zu ermitteln?**

- `pthread_self`(3) gibt eine Thread-ID zurück
- Diese muss *keiner festen Ordnung* folgen
- Es kann *keine verlässliche Rechenvorschrift* für die zu bedienenden Gäste abgeleitet werden

# Programmieren in C (1) (c)

```
   […]
pthread_mutex_t koch;
   […]

int main(int argc, char *argv[]) {
   pthread_t servicek[ANZ_SERVICEK];
   struct zu_bedienen arg[ANZ_SERVICEK];

   /// Koch initialisieren
   int fehler = pthread_mutex_init(&koch, NULL);
   if (fehler) {
     printf("Konnte Koch nicht anheuern");
     exit(EXIT_FAILURE);
   }

   /// Servicekraft beginnt mit der Arbeit
      […]

   /// Servicekraft macht Feierabend
      […]

   /// Koch macht Feierabend
   fehler = pthread_mutex_destroy(&koch);
   if (fehler) {
     printf("Koch hat Karoshi\n");
     exit(EXIT_FAILURE);
   }
      […]
```

# Programmieren in C (2) (c)

```
   […]
pthread_mutex_t koch;

/// Routine für einen Servicekraft
void *bedienen(void *arg) {
   troedeln();
   struct zu_bedienen *gaeste = (struct zu_bedienen*)(arg);

   for (int i = gaeste->erster_gast; i < gaeste->bis_gast; ++i) {

     int fehler = pthread_mutex_lock(&koch);
     if (fehler) {
       printf("Koch ist abgehauen\n");
       exit(fehler);
     }

     kochen(i);

     fehler = pthread_mutex_unlock(&koch);
     if (fehler) {
       printf("Die Suppe ist ausgelaufen\n");
       exit(fehler);
     }
   }
   pthread_exit(NULL);
}
```

# Programmieren in C (1) (d)

```c
    […]
pthread_mutex_t servicek_mutex;
int bedient = 0;
int letzter_gast = -1;
    […]
int main(int argc, char *argv[]) {
    […]
    /// Koch initialisieren                    /** (c):  Mutex-Init  */
    fehler = pthread_mutex_init(&servicek_mutex, NULL);
    if (fehler) {
        printf("Der Schnapps wurde zu früh getrunken");
        exit(EXIT_FAILURE);
    }
    […]
/// Servicekraft beginnt mit der Arbeit /** (a): pthread-create */
/// Servicekraft macht Feierabend       /** (a): pthread-join   */
/// Koch macht Feierabend               /** (c): Mutex-destroy  */
    […]
    fehler = pthread_mutex_destroy(&servicek_mutex);
    if (fehler) {
        printf("Die Servicekraft liegt betrunken in der Ecke\n");
        exit(EXIT_FAILURE);
    }
    printf("Letzter Gast bekommt Schnapps: %d\n", letzter_gast);
    return EXIT_SUCCESS;
}
```

# Programmieren in C (2) (d)

```
   […]
pthread_mutex_t servicek_mutex;
int bedient = 0;
int letzter_gast = -1;

/// Routine für einen Servicekraft
void *bedienen(void *arg) {
   […]
   /** aus (c): Mutex-Lock ~ Kochen ~ Mutex-Unlock */
   int fehler = pthread_mutex_lock(&servicek_mutex);
   if (fehler) {
     printf("Die Servicekraft ist durcheinander\n");
     exit(fehler);
   }
   /// Berechne wie viele Gäste insgesamt bedient wurden
   bedient = bedient + (gaeste->bis_gast - gaeste->erster_gast);
   if (bedient >= ANZ_GAESTE) {
     letzter_gast = gaeste->bis_gast -1;
   }
   fehler = pthread_mutex_unlock(&servicek_mutex);
   if (fehler) {
     printf("Die Servicekraft ist durcheinander\n");
     exit(fehler);
   }
   […]
```