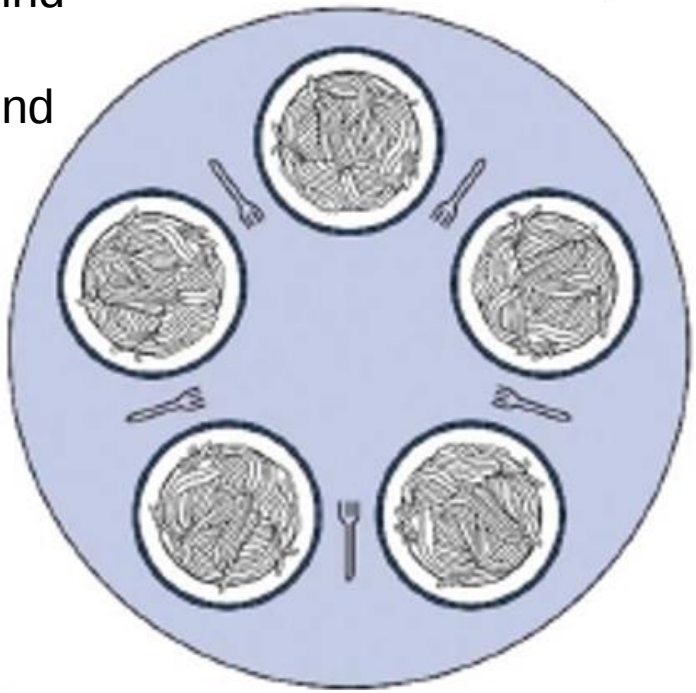
Philosophen-Problem (Synchronisationsprimitive genannt)

- Dijkstra, 1965, formuliert Standard-Aufgabe für Synchronisations-Primitive
  - 5 Philosophen sitzen um einen runden Tisch denken oder essen
  - Jeder mit einem Teller Spaghetti vor sich
  - Jeder Philosoph braucht zwei Gabeln, da die Spaghetti zu flutschig sind
  - Zwischen zwei Tellern liegt jeweils eine Gabel
  - Jeder Philosoph denkt und isst abwechselnd
  - Wenn er essen will versucht er die linke und die rechte Gabel zu greifen in beliebiger Reihenfolge
  - Gelingt es ihm, isst er eine Zeit lang, legt die Gabeln zurück und denkt weiter
  - Gibt es ein Programm, das niemals in einen Deadlock endet?
  - Wie sieht ein immer funktionierendes Programm für die Philosophen aus?



Philosophen-Problem

- Ansatz 1: einfache Implementierung

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

- Frage: Kann es zu einer Verklemmung (Deadlock) kommen?
  → JA, Alle Philosophen nehmen gleichzeitig die linke Gabel; damit ist es keinem möglich seine rechte Gabel zu nehmen

Philosophen-Problem

- Ansatz 1: einfache Implementierung **(Abänderung)**
  - Prüfung, ob rechte Gabel frei, nachdem die linke Gabel genommen wurde
  - Falls nicht, legt Philosoph seine linke Gabel zurück, wartet eine Weile und widerholt den gesamten Vorgang

Einfache Implementierung

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                                /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat();                                  /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

→ Es kommt wieder zu einer Verklemmung; alle Philosophen könnten gleichzeitig ihre linke Gabel aufnehmen und feststellen, dass die rechte Gabel nicht verfügbar ist. Sie legen die Gabeln wieder hin, warten, nehmen die linken Gabeln wieder gleichzeitig auf, stellen fest… und so für immer weiter.

→ Alle Programme laufen unendlich weiter, machen aber keine Fortschritte

→ Verhungern (**Starvation**)

- Idee: Philosophen warten nicht gleiche Zeit sondern zufällige Zeitspanne

  → Ja, funktioniert; ABER es gibt auch Anwendungen, die nicht aufgrund von Probieren (nicht einschätzbare Anzahl von Zufallszahlen), funktionieren müssen.

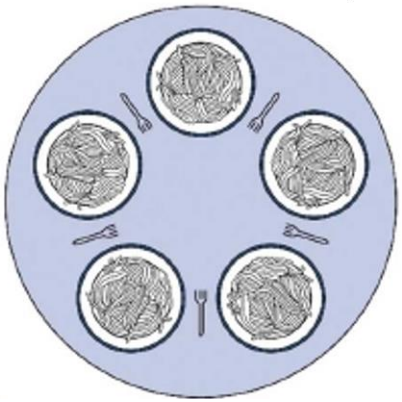
Philosophen-Problem

- Ansatz 2: Binäres Semaphor
  - Anweisungen werden durch ein binäres Semaphor geschützt

```
#define N 5
semaphore mutex

void philosoph()
{
   while(TRUE) {
         think();

         mutex.down();
         take_fork(i); // links
         take_fork((i+1)%N);
         eat();
         put_fork(i);
         put_fork((i+1)%N);
         mutex.up();
   }
}
```



- Frage: Kann es zu einer Verklemmung (Deadlock) kommen?
  → NEIN! Bevor ein Philosoph eine Gabel nimmt, macht er ein down auf mutex und nach dem Zurücklegen wieder ein up.
  → Performanzproblem; Es kann nur ein Philosoph essen (es sollte möglich sein, daß bei 5 verfügbaren Gabeln 2 Philosophen gleichzeitig essen können)

Philosophen-Problem

- Ansatz 2: Binäres Semaphor

```
#define N              5                    /* number of philosophers */
#define LEFT           (i+N–1)%N            /* number of i's left neighbor */
#define RIGHT          (i+1)%N              /* number of i's right neighbor */
#define THINKING       0                    /* philosopher is thinking */
#define HUNGRY         1                    /* philosopher is trying to get forks */
#define EATING         2                    /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
     while (TRUE) {                         /* repeat forever */
          think( );                         /* philosopher is thinking */
          take_forks(i);                    /* acquire two forks or block */
          eat( );                           /* yum-yum, spaghetti */
          put_forks(i);                     /* put both forks back on table */
     }
}
```

```
}
/* put both forks back on table */

/* i: philosopher number, from 0 to N–1 */

/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */

/* i: philosopher number, from 0 to N–1 */
     }
}
```

```
/* i: philosopher number, from 0 to N–1 */

/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */

om 0 to N–1 */

[LEFT] != EATING && state[RIGHT] != EATING) {
```

- ein Semaphor pro Philosoph und ein Status-Feld für jeden Philosoph
- Ein Philosoph kann nur in den essenden Zustand kommen, wenn keiner seiner Nachbarn isst.
- Statusfeld gibt an, ob ein Philosoph gerade isst, denkt oder hungrig ist Semaphoren werden benutzt, sodass hungrige Philosophen blockieren, falls benötigte Gabeln in Gebrauch sind.
- Philosophen sind so freundlich, ihre Nachbarn zu informieren, wenn sie mit Essen aufhören

Philosophen-Problem

- Ansatz 2: Binäres Semaphor und Statusfeld (ein Semaphor pro Philosoph und ein Status-Feld für jeden Philosoph)

1

```
#define N              5                    /* number of philosophers */
#define LEFT           (i+N–1)%N            /* number of i's left neighbor */
#define RIGHT          (i+1)%N              /* number of i's right neighbor */
#define THINKING       0                    /* philosopher is thinking */
#define HUNGRY         1                    /* philosopher is trying to get forks */
#define EATING         2                    /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```

2

```
    put_forks(i);                           /* put both forks back on table */
    }
}

void take_forks(int i)                      /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = HUNGRY;                      /* record fact that philosopher i is hungry */
    test(i);                                /* try to acquire 2 forks */
    up(&mutex);                             /* exit critical region */
    down(&s[i]);                            /* block if forks were not acquired */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N–1 */
```

3

```
}

void put_forks(i)                           /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = THINKING;                    /* philosopher has finished eating */
    test(LEFT);                             /* see if left neighbor can now eat */
    test(RIGHT);                            /* see if right neighbor can now eat */
    up(&mutex);                             /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Leser-Schreiber-Problem

- Philosophenproblem: Veranschaulichung des Konkurrierens um den exklusiven Zugriff auf eine begrenzte Anzahl von Ressourcen (z.B. Ein-/Ausgabe-Geräte)
- **Ein weiteres berühmtes Problem: Leser-Schreiber-Problem** (Readers and Writers Problem)
- Das Leser-Schreiber-Problem modelliert den Zugriff auf eine Datenbank (Courtois et al. 1971)
- Gemeinsamer Datenbereich (Datenbank)
  - Zwei Benutzergruppen/Prozessgruppen
    - Leser: verändern die Daten nie
    - Schreiber: lesen und verändern die Daten
  - Randbedingungen:
    - Leser dürfen gleichzeitig mit anderen Lesern zugreifen
    - Schreiber stehen unter wechselseitigem Ausschluss, auch mit Lesern

- Beispiel Leser-Schreiber-Problem: Reservierungssystem einer Fluggesellschaft mit vielen konkurrierenden Prozessen

Leser-Schreiber-Problem

- Realisierung
  - Leser
    - Warte bis kein Schreiber da ist
    - Greife auf die Datenbank zu
    - Abmelden – wartende Schreiber wecken
  - Schreiber
    - Warte bis kein Leser oder Schreiber da ist
    - Greife auf die Datenbank zu
    - Abmelden – wecke wartende Leser oder Schreiber

Leser-Schreiber-Problem

- Realisierung

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}
```

1

2

```
        use_data_read( );               /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

```
void writer(void)
```

- Erster Leser, der Zugriff auf die Datenbank bekommt, führt down auf Semaphor db aus
- Nachfolgende Leser erhöhen den Zähler rc
- Wenn Leser fertig, vermindern des Zählers und up auf Semaphor
- Falls Schreiber wartet, Zugriff auf Datenbank

Aufgabe/Frage

Annahme

- Ein weiterer Leser greift zu, während ein anderer Leser die Datenbank benutzt (kein Problem, da Leser immer zugelassen werden)
- Ein neu hinzukommender Schreiber darf nicht auf die Datenbank zugreifen; er benötigt exklusiven Zugriff; Schreiber wird schlafen gelegt
- Solange ein Leser aktiv ist, werden Leser zugelassen
- Der/die Schreiber werden schlafend bleiben, bis kein Leser mehr da ist
  → es kann passieren, daß ein Schreiber nie mehr an die Reihe kommt.

Frage:
Überlegen Sie sich eine Lösung, bei der der Schreiber auch an die Reihe kommt.

Bedingung:
→ im Team?
→ Zeit: 5 min

Zusammenfassung

- Interprozesskommunikation
  - Wechselseitiger Ausschluss
    - Mutex
      - „Wettstreit“
  - Reihenfolgesynchronisation
    - „Kooperation“
  - Erzeuger-Verbraucher-Problem
  - Lösungsansätze
    - Busy-Wait: Sperrvariable, Peterson, …
    - Semaphoren
      - down(), up()
        - Atomare Funktionen
      - Verallgemeinerung von sleep() und wakeup()

Aufgabe/Frage

- Wir haben ein Modell für Prozesse/Threads!
- Wir können Prozesse/Threads synchronisieren!

Frage:
Was wird im System noch benötigt, um die Aufgaben (Verwaltung, Abstraktion der CPU) zu realisieren?

Bedingung:

→ im Team?

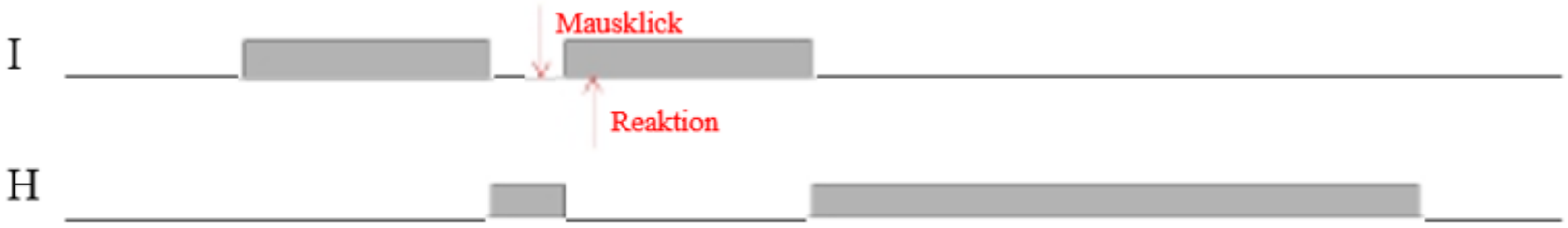
→ Zeit: 5 min

Scheduling-Szenarien

- Batch-System vs. Interaktives System
  - Hintergrundprozess (H) und Interaktiver Prozess (I)

  - Lange Zeit zwischen Mausklick und der Reaktion

  - BESSER



  - Kurze CPU-Belastung

Verschiedene Einsatzgebiete erfordern verschiedene Schedulingstrategien

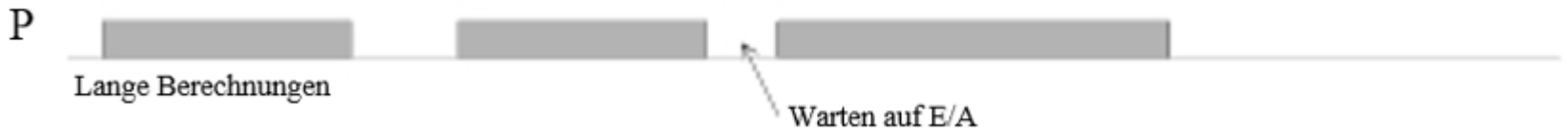
- **Stapelverarbeitungssystem-System**
  - Viele nicht interaktive Aufträge
  - häufig bei Großrechnern
  - Es gibt keine ungeduldigen Benutzer, die auf eine schnelle Antowrt warten
  - Anzahl der Prozesswechsel wird verringert → Verbesserung der Performanz

- **Interaktiver Betrieb**
  - Typisch für Arbeitsplatzrechner
  - Server
  - Unterbrechen notwendig; kein Prozess soll die CPU an sich reißen und anderen Prozessen Dienste verweigern

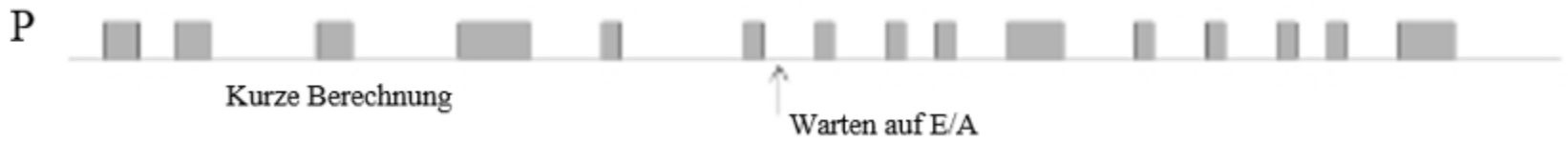
- **Echtzeitbetrieb**
  - Steueraufgaben
  - Multimedia-Anwendungen
  - Prozesse wissen, daß sie nicht lange Zeit arbeiten
  - Es laufen nur solche Programme ab, die die vorliegende Anwendung unterstützt

Kriterien von Schedulingstrategien

- **Prozess-Charakteristik**
  - CPU-lastiger Prozess, rechenintensiv (compute intensive)



  - E/A-lastiger Prozess, datenintensiv (data intensive)



Anmerkung:
- CPU-Performance nimmt schneller zu als Platten-Performance !!!!
- Programme werden immer mehr E/A-lastig

Aufgabe/Frage

- Der Scheduler ist verantwortlich für den Wechsel von Prozessen.

Frage:
1. Wann soll der Scheduler im Betriebssystem aktiviert werden?

2. Überlegen Sie sich Kriterien für Scheduler.
Hilfe: Welche Kriterien gibt es aus Benutzersicht?
Welche Kriterien gibt es aus Systemsicht?

Bedingung:
→ im Team?
→ Zeit: 5 min

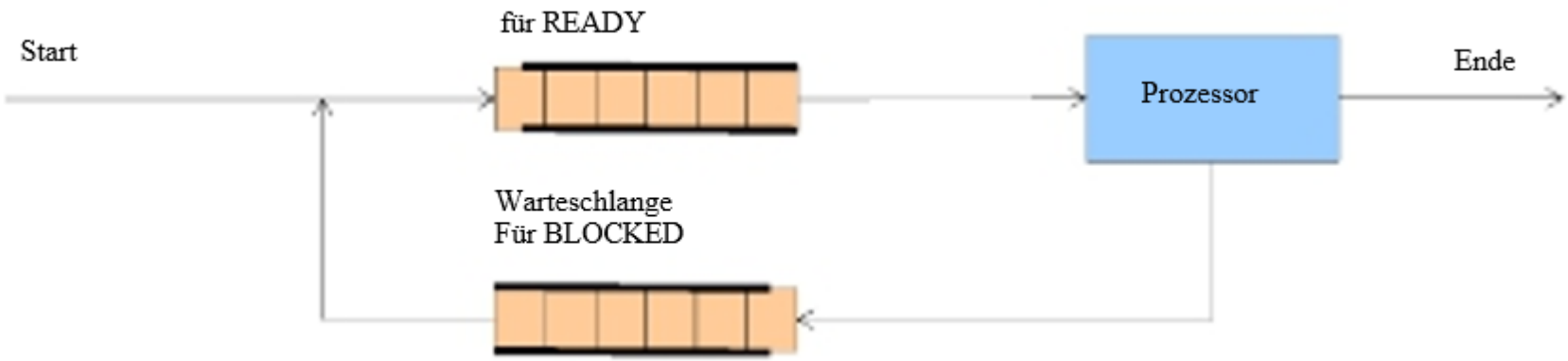
Scheduling-Strategien

- **Nicht-unterbrechend (nonpreemptive)**
  - der ausgewählte Prozess läuft bis er freiwillig die CPU aufgibt oder blockiert
  - Einsatz in Stapelverarbeitungs- und Echtzeitsystemen

- **Unterbrechend (preemptive)**
  - Der aktive Prozess wird blockiert und damit die CPU entzogen
    - durch ein Timer-Interrupt
      - ➢ Voraussetzung für unterbrechendes Scheduling
    - durch einen Prozess mit höherer Priorität
  - Einsatz in interaktiven und Echtzeit-Systemen

Scheduling-Strategien

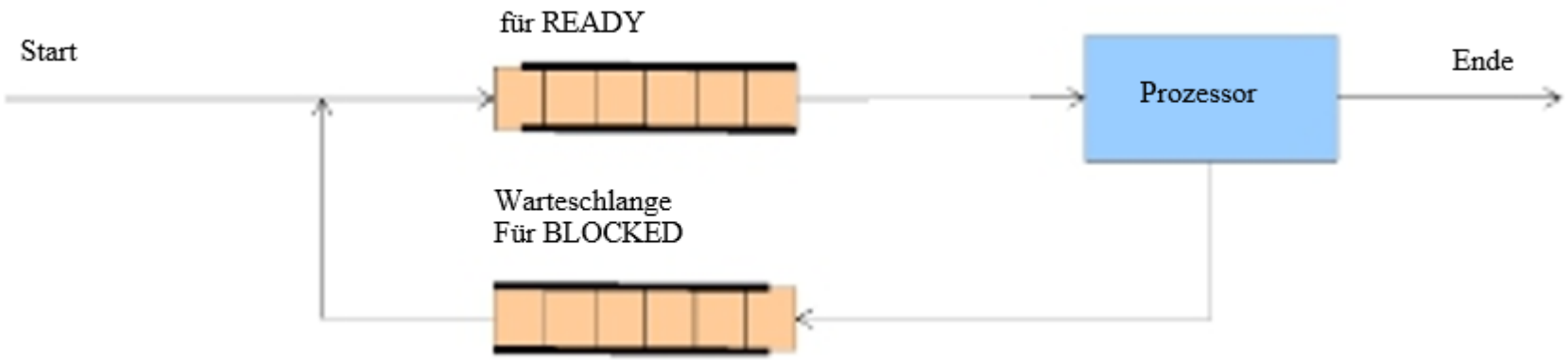
- **First Come First Served (FCFS)**
  - Einfachste Scheduling-Strategie
  - Prozessen wird die CPU in der Reihenfolge zugewiesen, in der sie diese anfordern
  - Alle Prozesse sind gleichzeitig da; der am längsten wartende Prozess mit Status READY darf als nächstes rechnen
    → **Nicht unterbrechendes Verfahren**



Alle Prozesse sind gleichzeitig da

Der am längsten wartende Prozess mit Status READY darf als nächstes rechnen

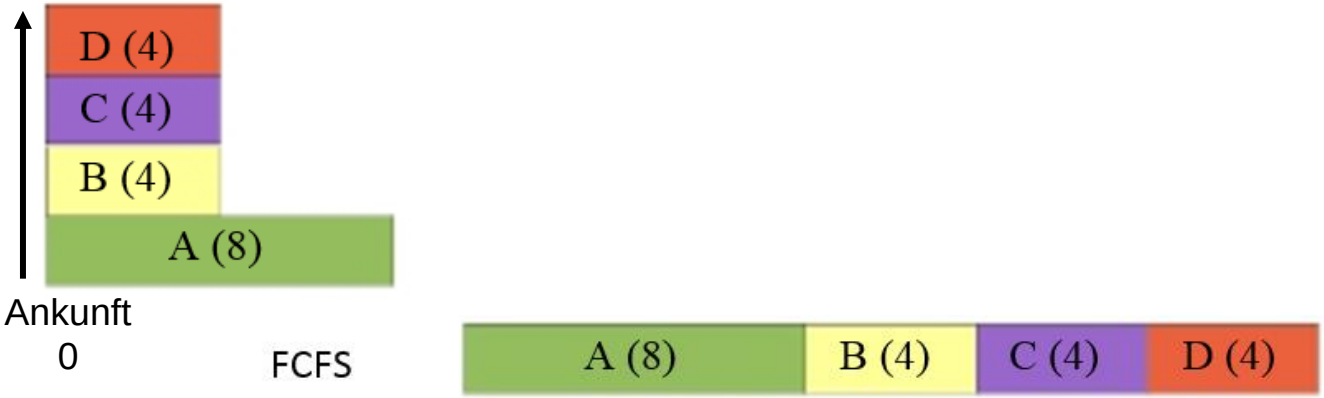
→ Nicht unterbrechendes Verfahren



Scheduling-Strategien

- **Shortest Job First (SJF)**
  - Der am kürzesten rechnende Prozess mit Status READY darf als nächstes rechnen
    - **Nicht unterbrechendes Verfahren**
    - Optimales Verfahren, wenn alle Jobs gleichzeitig vorliegen
  - Voraussetzung: Laufzeit im Voraus bekannt
    - Nicht ungewöhnlich in der Stapelverarbeitung

Scheduling-Strategien – Beispiel FCFS



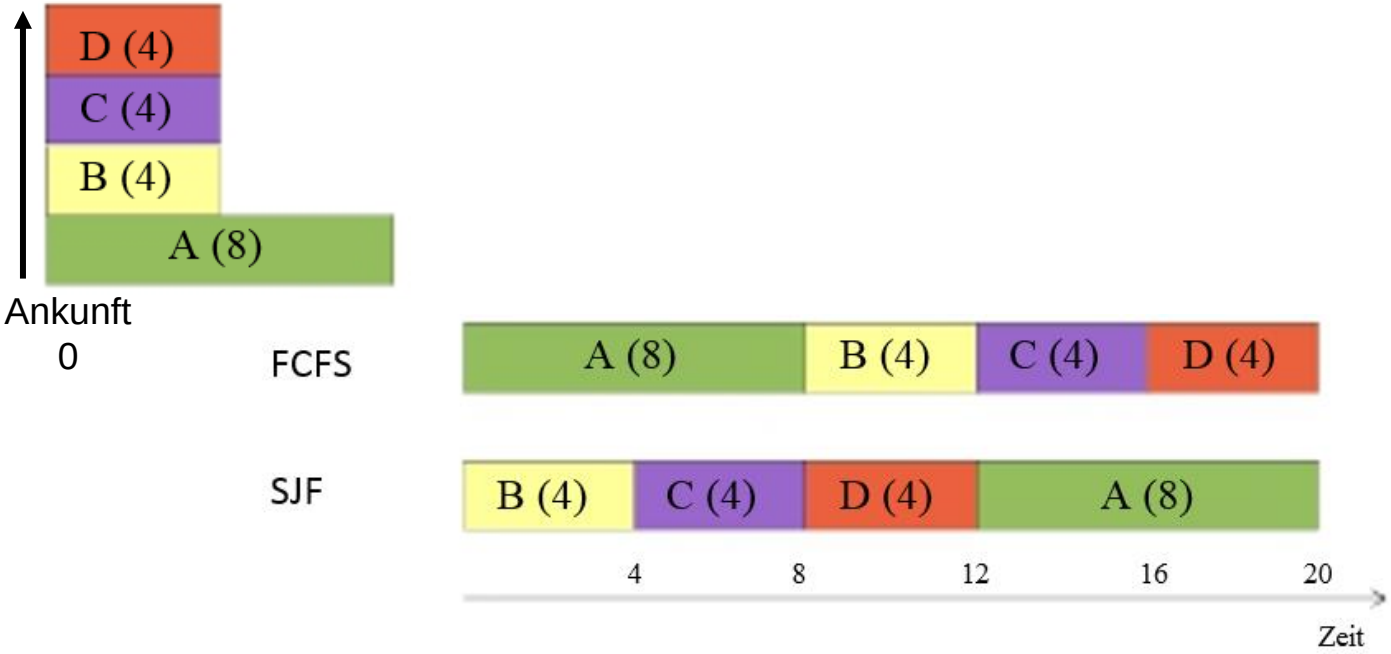
Alle Jobs gleichzeitig da
A hat Laufzeit 8, B hat Laufzeit 4, usw.

Durchlaufzeit FCFS

A=8	=>8
B=8+4	=>12
C=8+4+4	=>16
D=8+4+4+4	=>20

Mittlere Durchlaufzeit: (8+(8+4)+(8+4+4)+(8+4+4+4)/4=14

Scheduling-Strategien – Beispiele SJF



Alle Jobs gleichzeitig da
A hat Laufzeit 8, B hat Laufzeit 4, usw.

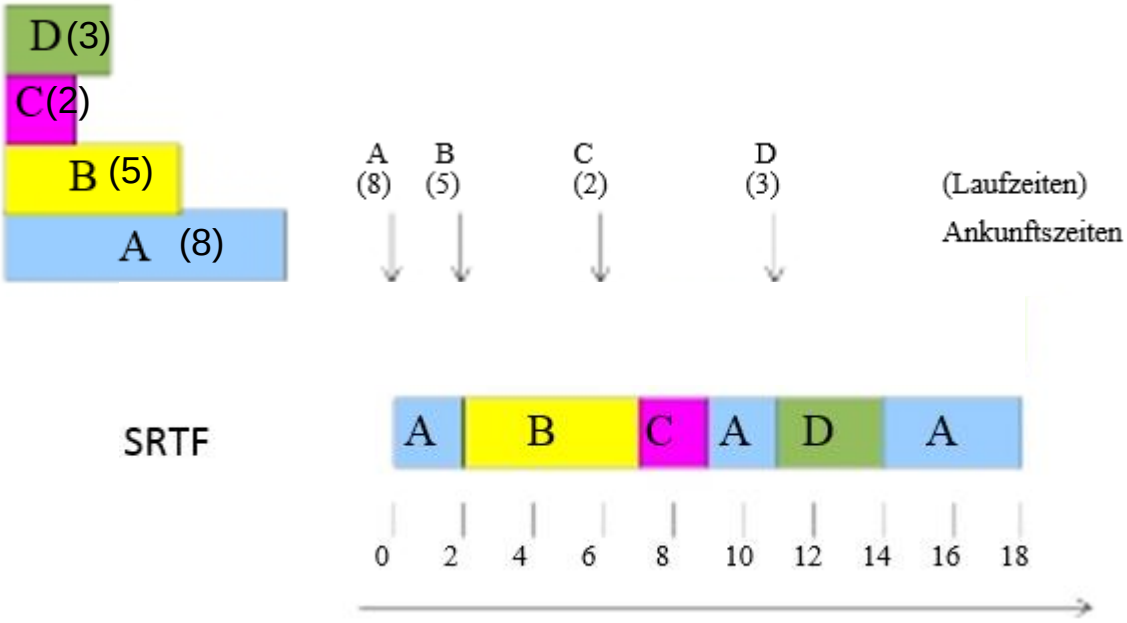
B ist shortest job und vor C im Alphabet
C ist shortest job und vor C im Alphabet
D ist shortest job
Am Ende ist nur noch A da

Durchlaufzeit SJF
A=20
B=4
C=4+4
D=4+4+4

Mittlere Durchlaufzeit: (20+4+(4+4)+(4+4+4)/4)= 11

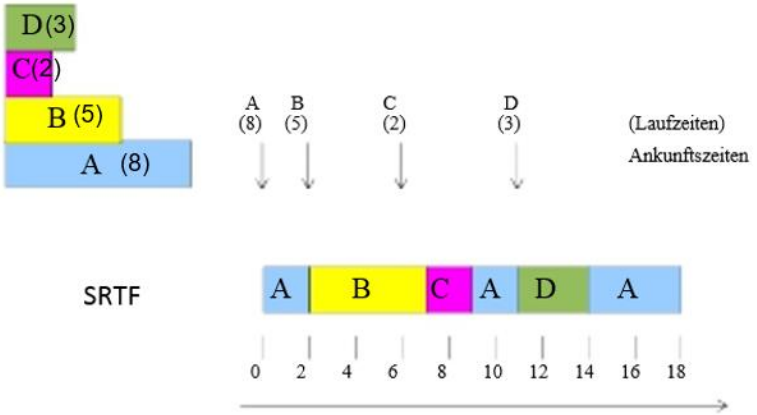
Scheduling-Strategien

- **Shortest Remaining Time First (SRTF)**
  - der Prozess mit der kürzesten verbleibenden Zeit und Status READY darf als nächstes rechnen
    - Für neue kurze Jobs sehr günstig
  - **Unterbrechende Variante** von Shortest Job First
    - Auch hier muss Laufzeit im Voraus bekannt sein



Scheduling-Strategien – Beispiel SRTF

Durchlaufzeit A =8
Durchlaufzeit B =5
Durchlaufzeit C=2
Durchlaufzeit D=3
Pfeile zeigen die Ankunftszeiten

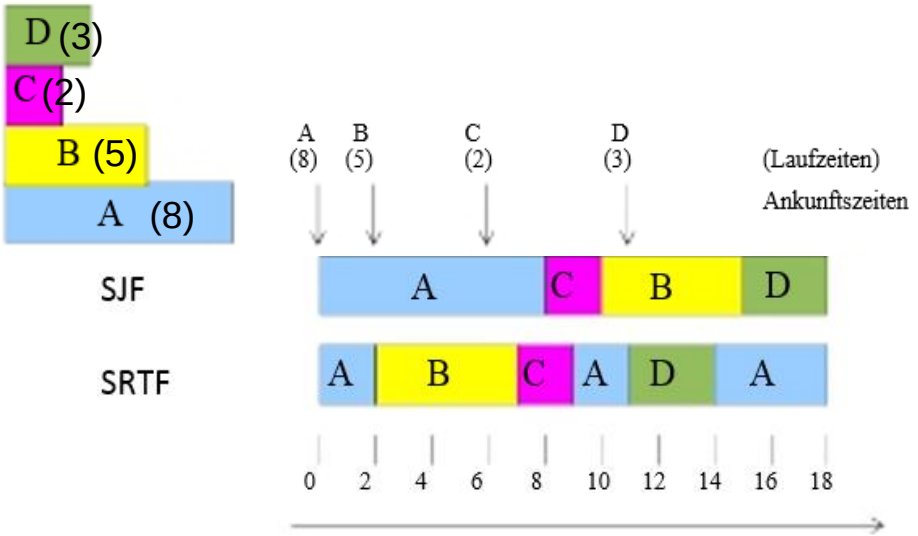


1. A beginnt
2. B kommt an und hat geringere Laufzeit als A, daher kommt B zum Zuge
3. B hat noch ne Laufzeit von 2, wenn C ankommt auch mit 2, B läuft weiter
4. D ist noch nicht da, A wartet, C wartet, C ist kleiner als A, C kommt dran
5. D ist noch nicht da, wenn C fertig, A wartet, A kommt dran
6. A läuft kurz an, D taucht auf, D hat nur ne Laufzeit von 3, A hat noch 4, D kommt dran
7. Wenn D fertig, kommt A dran

Mittlere Durchlaufzeit von SRTF: (18+(7-2)+(9-6)+(14-11))/4=7,25

Scheduling-Strategien – Beispiel SJF

Durchlaufzeit A =8
Durchlaufzeit B =5
Durchlaufzeit C=2
Durchlaufzeit D=3
Pfeile zeigen die Ankunftszeiten

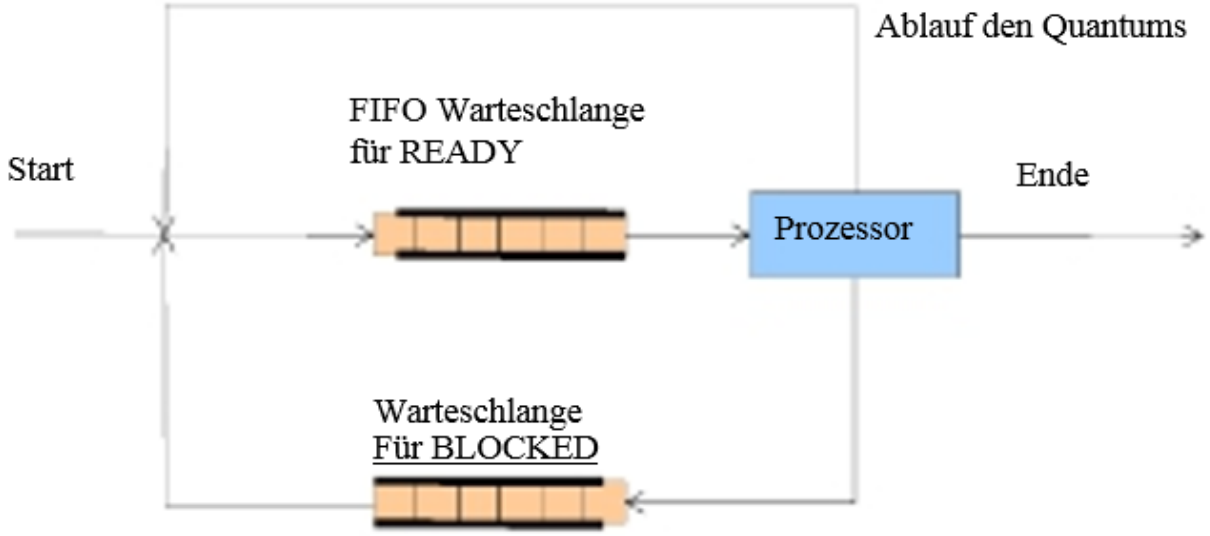


1. A beginnt
2. B kommt an, C kommt an
3. Da pre-emptive: A läuft fertig
4. Nach 8 wird geschaut, welcher Prozess am kürzesten ist. Das ist C (2), C kommt zum Zuge
5. Da C fertig ist, bevor D ankommt, und B wartet, kommt B dran
6. Nach B kommt D dran

Mittlere Durchlaufzeit von SJF: (8+(15-2)+(10-6)+(18-11)/4=8

Scheduling-Strategien

- **Round Robin (RR) / Zeitscheibenverfahren**
  - Einfachste Strategie für interaktive Systeme
    - **Unterbrechende Variante** von FCFS
    - Jeder laufende Prozess
      - wird spätestens nach einer bestimmten Zeit (Quantum) unterbro chen und von einem anderen Prozess abgelöst oder
      - er wird davor blockiert oder
      - er beendet sich selbst



Scheduling-Strategien

- **Round Robin (RR) / Zeitscheibenverfahren**
  - Länge des Quantum
    - Kurzes Quantum ergibt kurze Antwortzeiten
      - ➢ schlechte CPU Nutzung durch häufige Prozesswechsel
    - Langes Quantum ergibt lange Antwortzeiten
      - ➢ bessere CPU Nutzung durch weniger Prozesswechsel
    - Praxis-Wert: 20-50 ms
  - Round Robin ist nicht fair
    - E/A-lastige Prozesse werden benachteiligt
      - ➢ Sie geben durch E/A die CPU oft freiwillig ab

Aufgabe/Frage

Sie kennen jetzt mehrere Scheduling-Strategien. Eine davon ist Round Robin.
Dazu eine Aufgabe (Klausur-nah).

- Drei Prozesse treffen zu den in der Tabelle angegebenen Zeiten in der Bereitliste eines Schedulers ein. Von jedem Prozess ist die Bedienzeit (benötigte Rechenzeit) bekannt. Zusätzlich hat jeder Prozess eine Priorität (0 stellt die höchste Priorität dar).

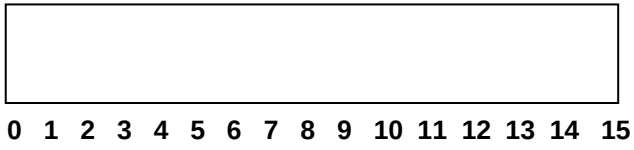
Prozess	Ankunftszeit	Bedienzeit	Priorität
A	0	7	2
B	2	5	1
C	5	3	0

- Die Prozesse benutzen nur die CPU und werden nie durch Eingabe/Ausgabe oder sonstige Gründe blockiert. Der Scheduler entscheidet online, d.h. nur aufgrund der zum Scheduling-Zeitpunkt bereits vorliegenden Prozesse.

Frage:

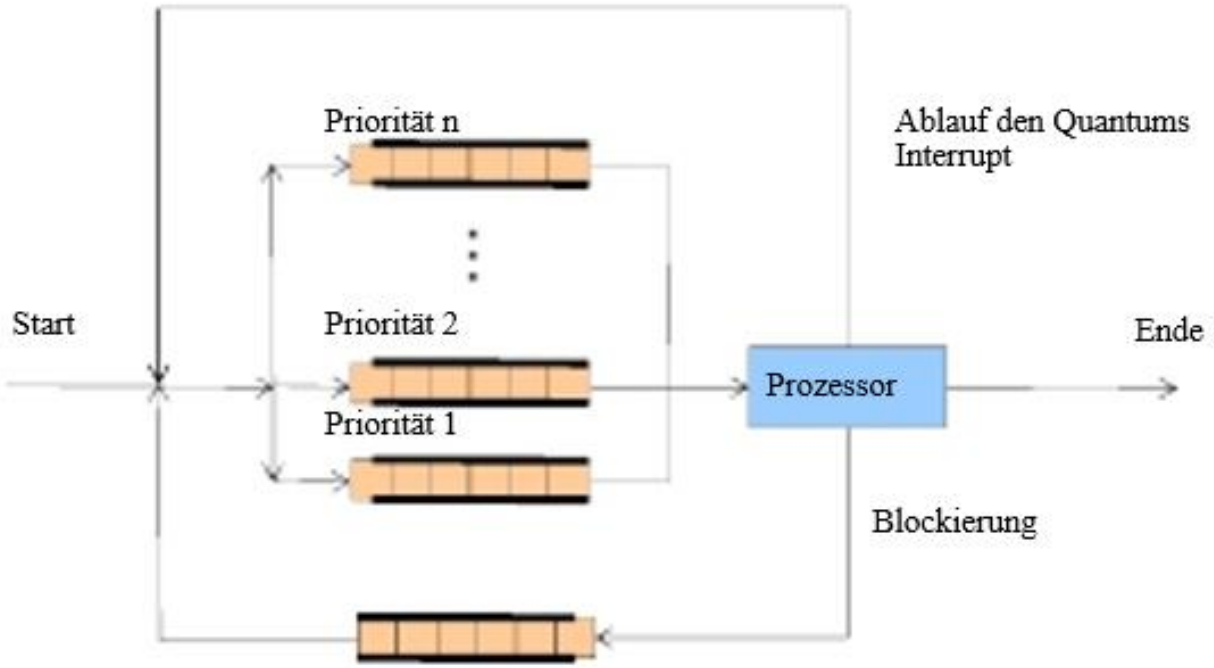
1. Zeichnen Sie den zeitlichen Ablauf der Prozessausführung für das Verfahren Round Robin (RR) mit einem Quantum (Zeitscheibenlänge) von 3 ohne Berücksichtigung der Priorität?
2. Wie sieht das Ergebnis mit Berücksichtigung der Priorität aus? Wie ist die mittlere Durchlaufzeit?

10-15 min



Scheduling-Strategien

- Realisierung von Prioritätenbasiertem Scheduling
  - Zusammenfassung von Prozessen in Prioritätsklassen
    - Eine Warteschlange zwischen den Prioritätsklassen
  - Prioritäten-Scheduling zwischen den Klassen
    - Innerhalb der Klassen Round Robin



Scheduling-Strategien

- Realisierung von Prioritätenbasiertem Scheduling
  - Beispiel von Prioritäten-basiertem Scheduling
    - Statisches Multilevel Scheduling
      - Verschiedene Betriebsarten im System
        - Abbildung auf Prioritätenklasse
      - Aber: unterschiedliche Scheduling-Strategien innerhalb der Warteschlangen

Priorität	Prozessklasse	SchedulingStrategie
1	Echtzeitprozesse (Multimedia)	Prioritäten
2	Interaktive Prozesse	RR
3	E/A-Prozesse	RR
4	Rechenintensive Stapelprozesse	FCFS

Scheduling in Echtzeitsystemen

- Zeit spielt eine entscheidende Rolle
  - Richtige, aber verspätete Antwort genauso schlecht wie eine falsche Antwort
- Harte Echtzeitsysteme
  - Absolute Deadlines, die strikt eingehalten werden müssen
- Weiche Echtzeitsysteme
  - Die Verletzung von Deadline ist unerwünscht, aber tolerierbar
- Verhalten (Laufzeit Ci) der Prozesse sind bekannt
- Ereignisse treten periodisch oder aperiodisch auf
- Statische und dynamische Scheduling-Strategien verwendet
  - Statische Strategien erfordern die genaue Kenntnis der Prozesse
  - Dynamische Strategien sind diesbezüglich flexibler

Thread-Scheduling

- Thread-Realisierung im Benutzer-Modus
  - Kosten für Thread-Wechsel sind geringer
  - Scheduler des Laufzeitsystems läuft im Benutzermodus
    - Angepasste Scheduling-Algorithmen möglich
    - Timer-Interrupts/Zeitscheiben i.d.R. nicht möglich!

- Thread-Realisierung im Kernmodus
  - Das Umschalten zwischen Threads kann das Umschalten zwischen Prozessen bedeuten
    - Höhere Kosten
    - Scheduler kann das evtl. berücksichtigen

Zusammenfassung

- Scheduling
  - Entscheidung welcher Prozess wie lange rechnen darf
    - Verwaltung der Ressource Prozessor
  - Unterschiedliche Anforderungen
    - Betriebsart und Sichtweise
  - Nicht-unterbrechendes und unterbrechendes Scheduling
    - Unterbrechend: BS kann einem Prozess die CPU entziehen

- Verschiedene Scheduling-Algorithmen
  - FCFS – First Come First Served
    - Einfaches, nicht unterbrechendes Verfahren
  - SJF – Shortest Job First
    - Optimiert die Durchlaufzeit von Jobs
  - Round Robin (RR)
    - Präemptive Version von FCFS
    - Prozesse dürfen nur bestimmte Zeit rechnen
  - Prioritätenbasiertes Scheduling
    - Prozess mit höchster Priorität bekommt CPU
  - Multilevel-Scheduling
    - Mehrere Warteschlangen mit eigener Auswahlstrategie