Tutorübung Grundlagen: Betriebssysteme und Systemsoftware

Moritz Beckel

München, 11. November 2022

Freitag 10:15-12:00 Uhr

Raum 00.11.038

https://zulip.in.tum.de/#narrow/stream/1295-GBS-Fr-1000-A

Bei Fragen könnt ihr mich hier kontaktieren:

moritz.beckel@tum.de

Organisation

- Midterm-Klausur am Freitag, den 16.12.2022
- Uhrzeit: 18:45 bis 19:30 Uhr
- Findet in Präsenz statt
- Anmeldezeitraum ist vom 14.11.2022 bis 9.12.2022 (beginnt nächste Woche)

Zusammenfassung

Prozess- und Prozessorverwaltung

1. Prozesse
2. Threads
3. Dispatching
4. Scheduling

Zusammenfassung

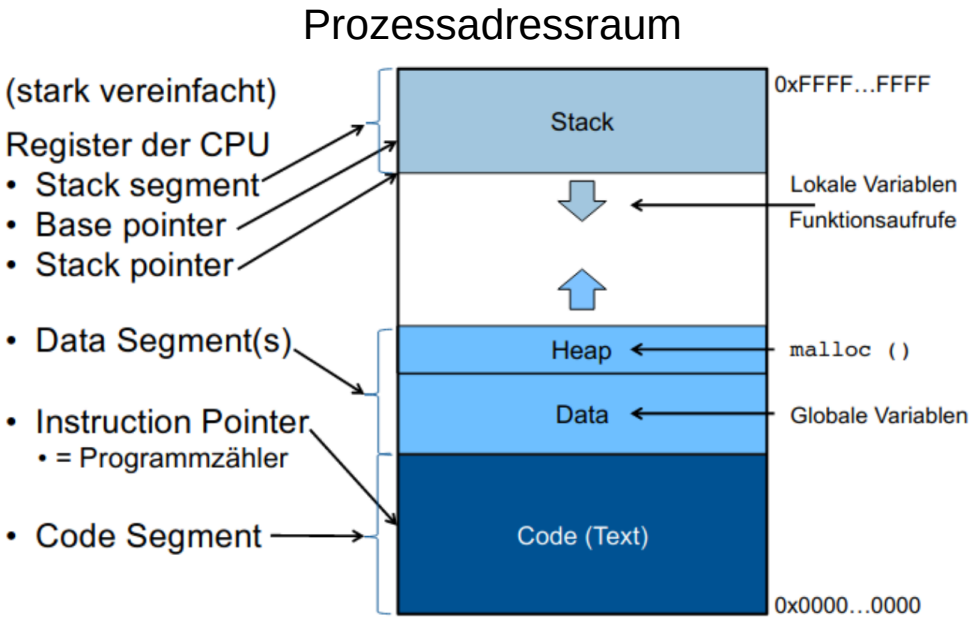
Multiprogramming

1. Programme werden sehr schnell abgewechselt
2. Quasi-parallele Verarbeitung

Multithreading

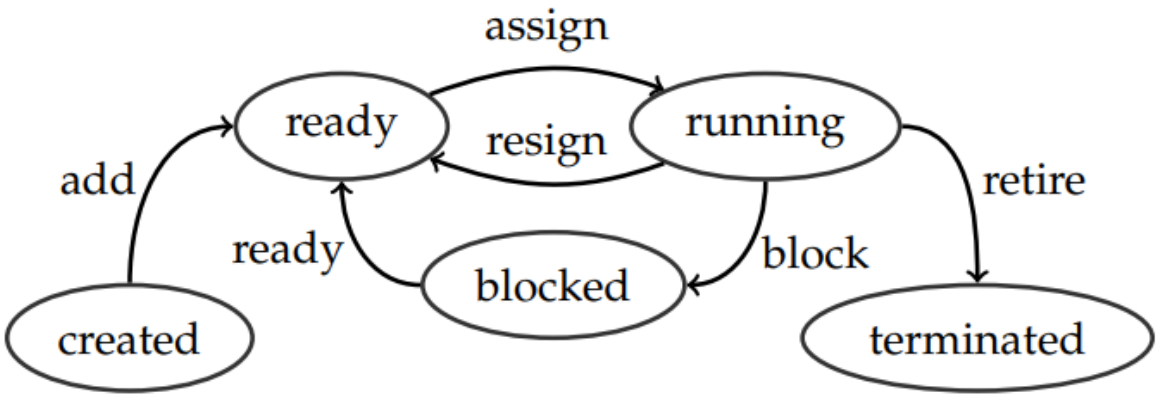
1. Mehrere parallele Kontrollflüsse innerhalb eines Prozesses
2. Erhalten eigene Ressourcen (Stack, Register)

Zusammenfassung



Zusammenfassung

Prozesszustände

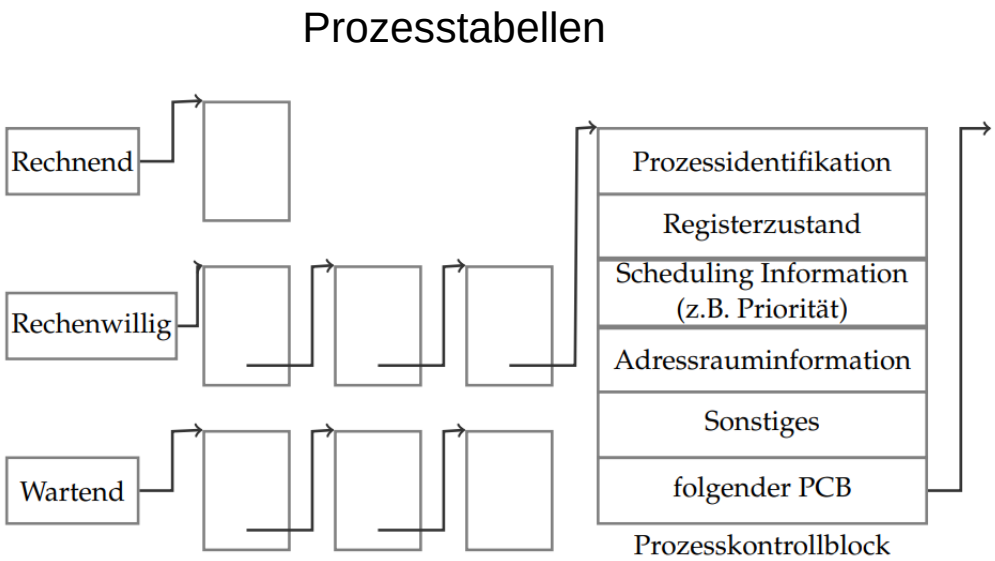


Zusammenfassung

PCB (Process Control Block)

Prozess(-or)verwaltung	**Speicherverwaltung**	**Dateiverwaltung**
Registerinhalte	Pointer auf...	Dateideskriptoren
Program Counter	• Stack-Segment	User ID (UID)
Stack Pointer	• Code-Segment	Group ID (GID)
Statusregister	• Data-Segment	
Prozesszustand (ready, etc.)	Größe der Segmente	
Prioriät		
Process ID (PID)		
Parent PID (PPID)		
Process Group ID (PGID)		

Zusammenfassung



Zusammenfassung

Prozesserzeugung

1. Prozess wird von Elternprozess erzeugt
2. PCB wird „vererbt“ (copy on write)
3. Mittels fork() system call realisiert

Prozessterminierung

1. Normale Beendigung exit(0)
2. Vorzeitige Beendigung im Fehlerfall exit(1)
3. Terminierung durch das Betriebssystem (Segmentation Fault)
4. Terminierung durch einen anderen Prozess kill(pid, signal)

Zusammenfassung

Prozesshierarchie

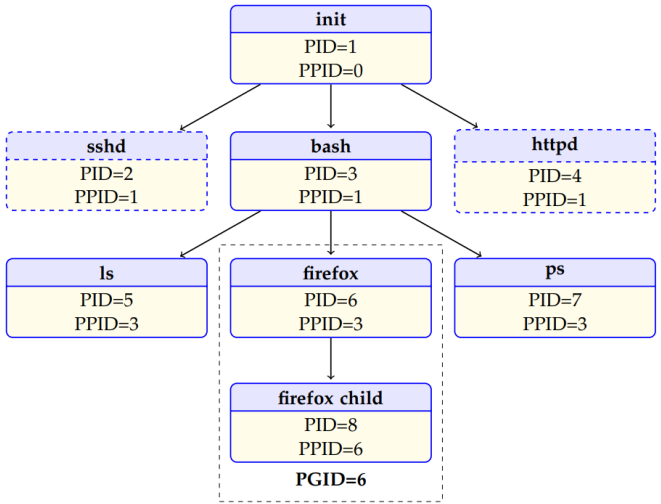
1. PID (Process ID)
2. PPID (Parent PID)
3. PGID (Process Group ID) (Wert entspricht Wurzel des Teilbaums)

Waisen (= Kindprozesse, dessen Elternprozess vor ihnen terminiert, vom init-Prozess adoptiert, werden Hintergrundprozesse)

Zombies (= Kindprozesse, welche terminieren und PCB in Prozesstabelle bleibt, da Elternprozess nicht mit wait() Zustand erfragt oder selbst terminiert)

Zusammenfassung

Prozesshierarchie



Zusammenfassung

Prozesse vs. Threads

1. Threads teilen Adressraum eines Prozesses
2. Eigener Stack
3. Threads haben ebenfalls eigene Threadzustände + Threadkontext (reduziert)

Arten von Threads

1. User-Level Threads
2. Kernel-Level Threads
3. Hybride Threads

Zusammenfassung

Dispatching

1. P0 wird gerade ausgeführt.
2. Durch einen Interrupt (bzw. Systemaufruf) wird in den Kernel Mode gewechselt. 3.
3. Der Scheduler bestimmt, dass P1 fortgesetzt werden soll. 4.
4. Der Dispatcher ändert je nach Situation den Zustand von P0 zu «ready», «blocked» oder «terminated».
5. Der Dispatcher sichert den Kontext von P0 (Registerwerte, Programmzähler, usw.) im zugehörigen Process Control Block. 6.
6. Der Dispatcher lädt den Kontext von P1 (Registerwerte, Programmzähler, usw.) in die entsprechenden CPU-Register. 7.
7. Der Dispatcher setzt den Zustand von P1 auf «running».
8. Es wird in den User Mode gewechselt und die Kontrolle an P1 abgegeben.
9. P1 wird fortgesetzt.

Zusammenfassung

Scheduling

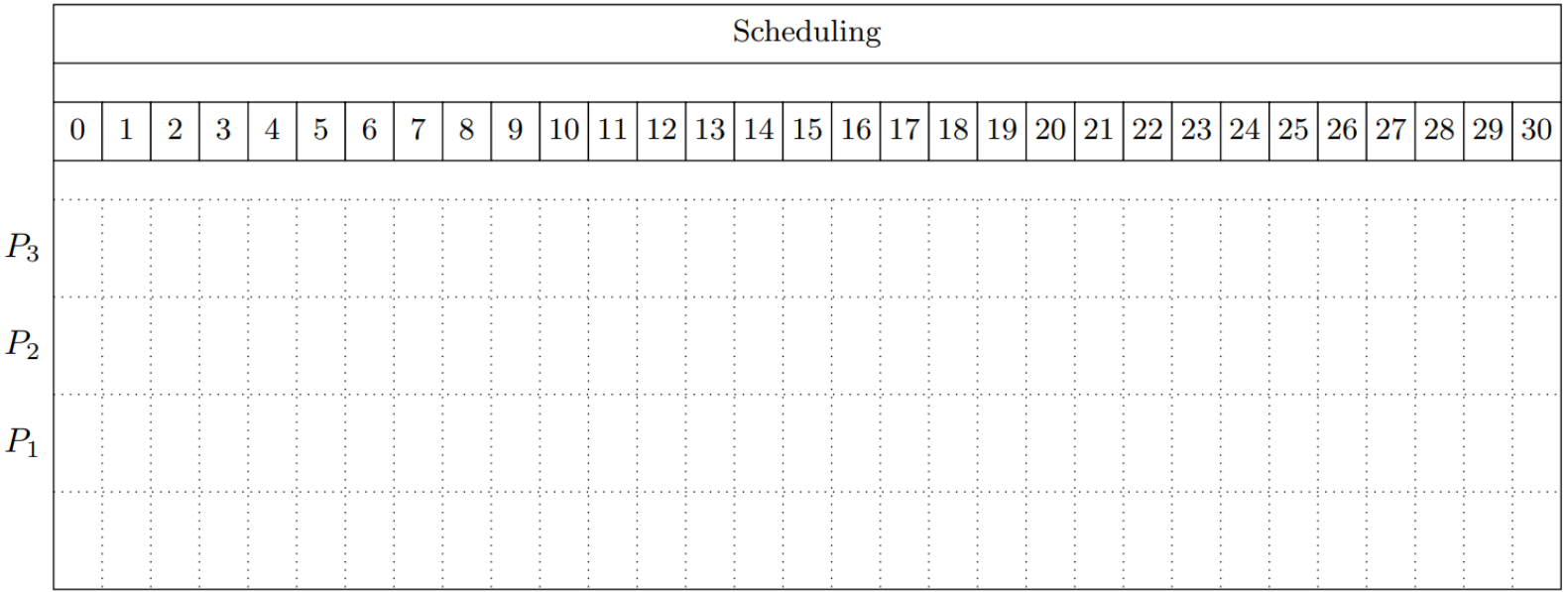
1. Preemptive (unterbrechend) vs. non-preemptive (nicht unterbrechend)
2. CPU-Bound (CPU-lastig) vs. I/O-Bound (I/O-lastig)
3. Allgemeine Optimierungsziele:
    1. Fairness
    2. Balance der Systemauslastung (I/O und CPU)

Multilevel-Scheduling

1. Short-Term-Scheduler (Auswahl der rechenwilligen Prozesse)
2. Medium-Term-Scheduler (Ein-/Auslagerung von Prozessen in Arbeitsspeicher, Swapping)
3. Long-Term-Scheduler (Verwaltet Übergang erzeugte -> rechenwillige Prozesse)

Betriebsart	Schedulingstrategien	Optimierungsziele
Batch-Systeme	• First-Come-First-Served non-preemptive • Shortest Job First (SJF) non-preemxptive • Shortest Remaining Time Next (SRTN) preemptive	• Durchsatz (Anzahl Aufträge max.) • CPU-Belegung (max.) • Ausführungszeit (Wartezeit + Rechenzeit min.)
Interaktive Systeme	• Round Robin (RR) preemptive • Priority Scheduling preemptive	• Antwortzeit (min.) • Proportionalität (Schneller Mausclick vs. langsamer Download)
Echtzeit Systeme	• Earliest Deadline First (EDF) (non)-preemptive • Rate-Monotonic Scheduling (RMS) preemptive	• Deadlines einhalten • Vorhersagbarkeit der Rechendauer • Kein Verhungern

Aufgabe 1 – Gantt Diagramm



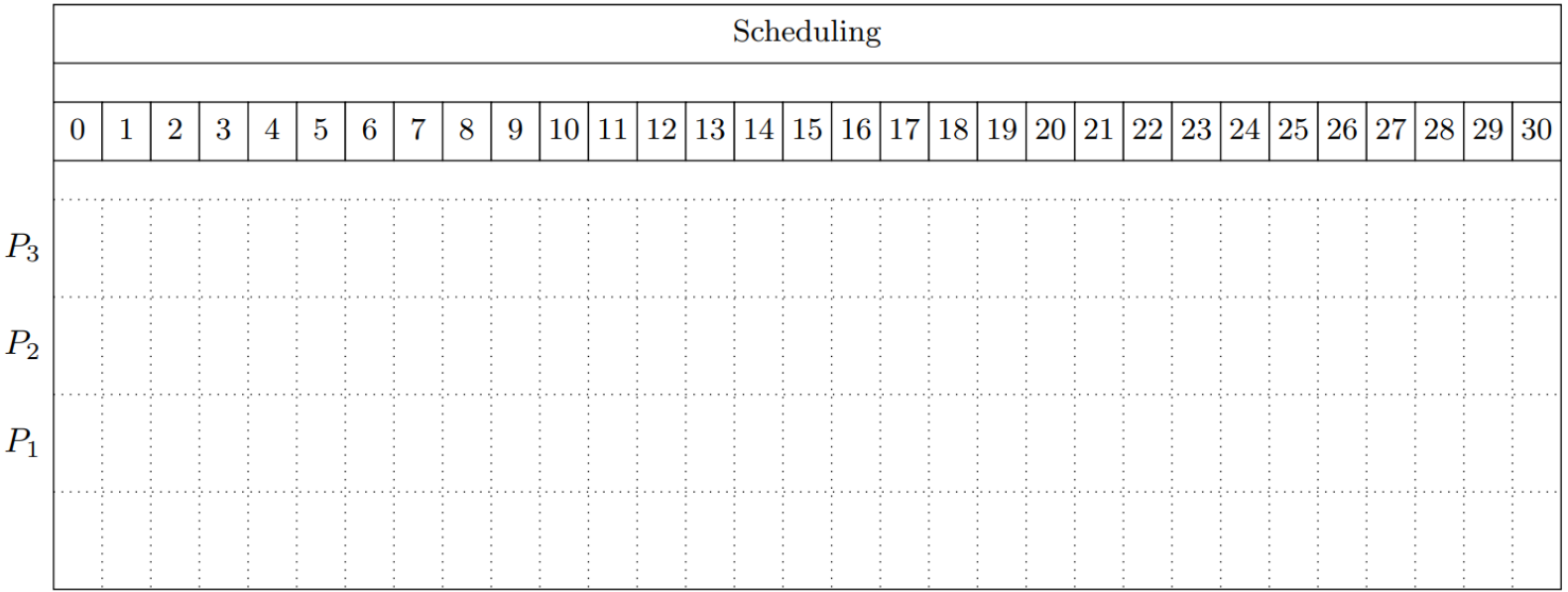
Aufgabe 1

Es seien 3 Prozesse gegeben. Der Vektor ihrer Ankunftszeiten am Scheduler beträgt $\vec{a} = (0, 5, 2)$. Ihre Rechenzeiten sind durch $\vec{r} = (7, 3, 4)$ gegeben. Nehmen Sie an, dass ein Kontextwechsel eine Zeiteinheit benötigt.

- Modellieren Sie den Scheduler/Dispatcher als einen eigenständigen Prozess.
- Wartezeit mit einem - (beginnend mit der Ankunftszeit des Prozesses), und die Rechenzeit mit einem X
- Vernachlässigen Sie den initialen Kontextwechsel. Beginnen Sie im ersten Zeitslot mit dem ersten rechnenden Prozess.

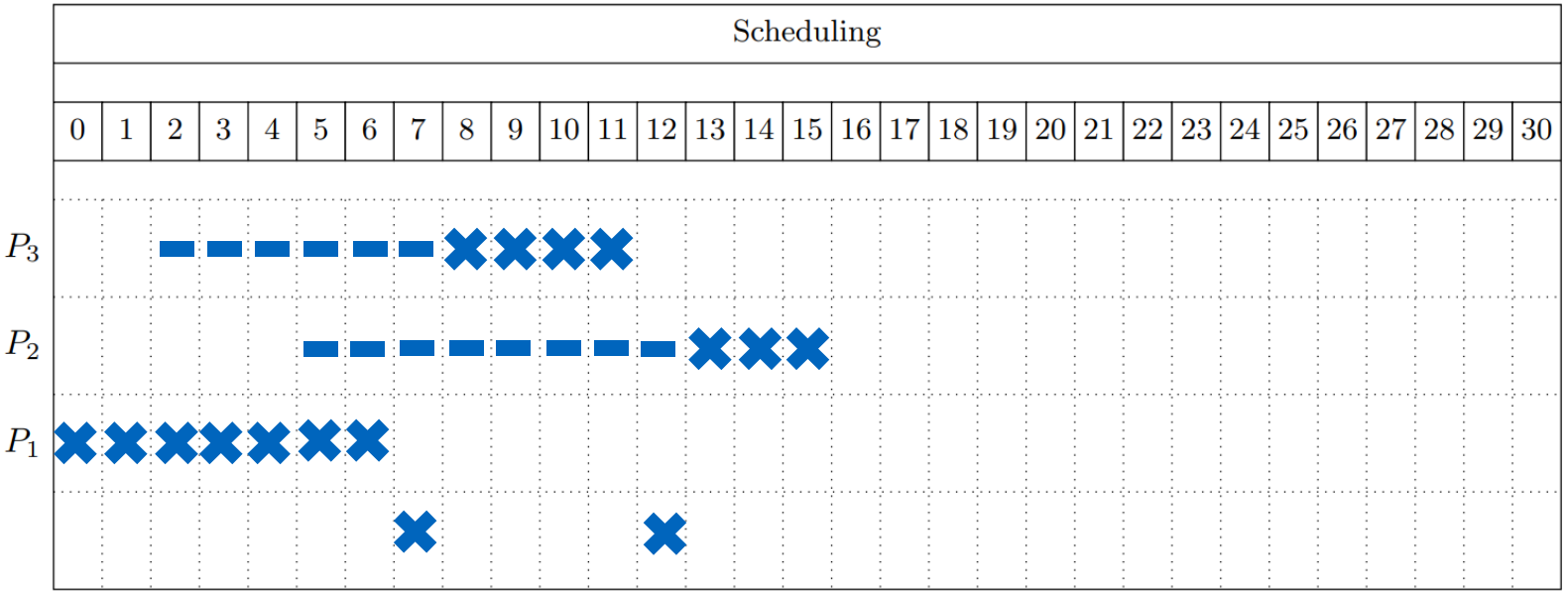
a = (0, 5, 2) r = (7, 3, 4)

Aufgabe 1 a) First-Come-First-Served (FCFS): Non-preemptive



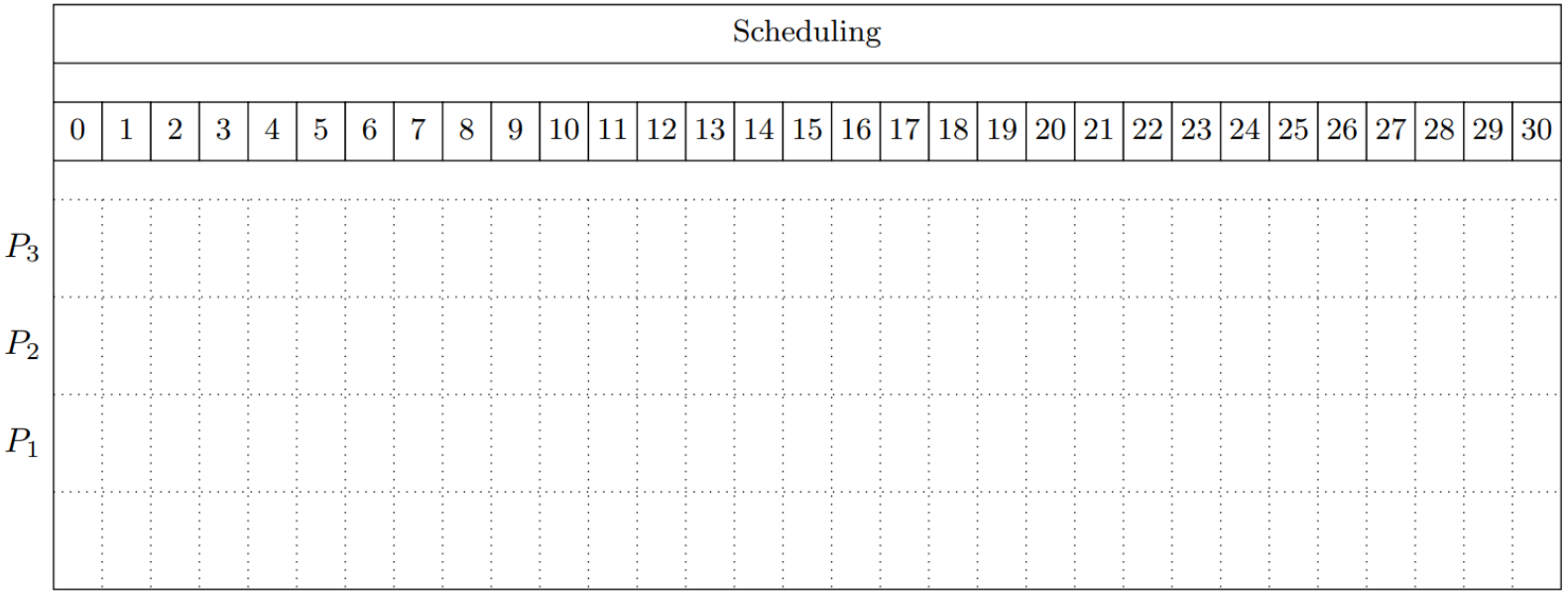
a = (0, 5, 2) r = (7, 3, 4)

Aufgabe 1 a) First-Come-First-Served (FCFS): Non-preemptive



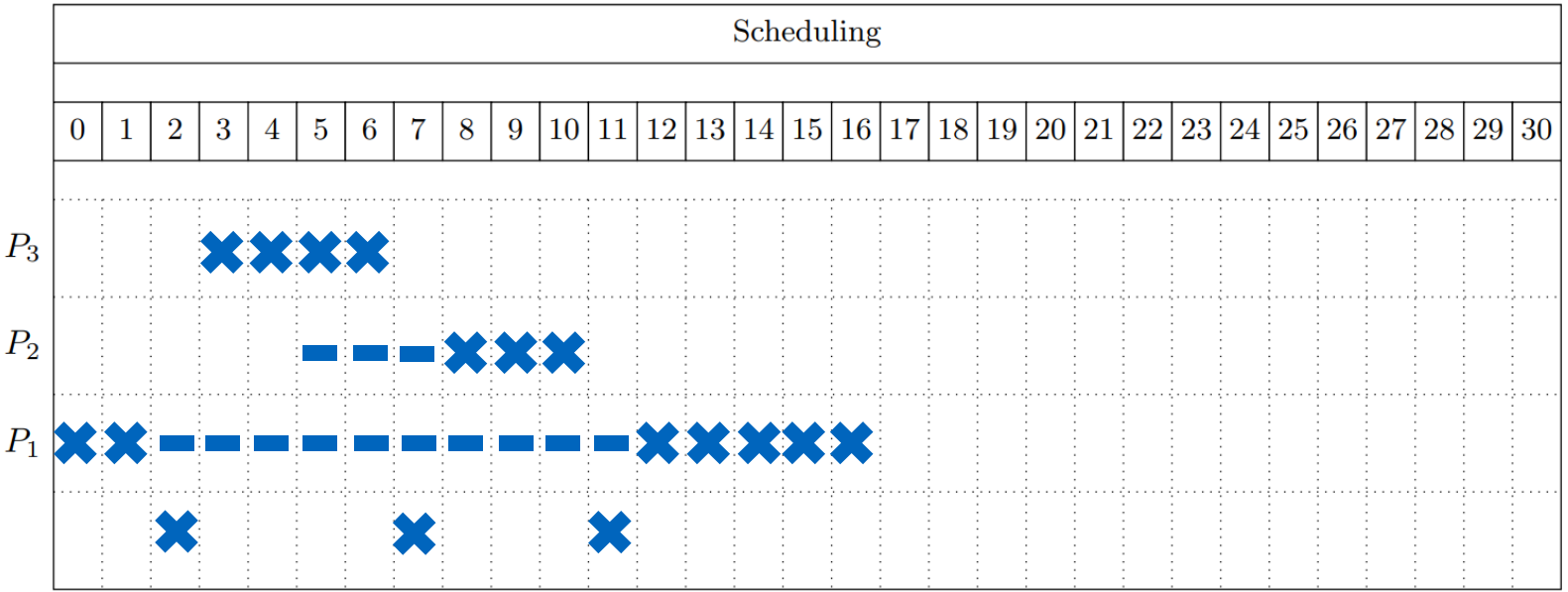
a = (0, 5, 2) r = (7, 3, 4)

Aufgabe 1 b) Shortest Remaining Time Next (SRTN): Preemptive



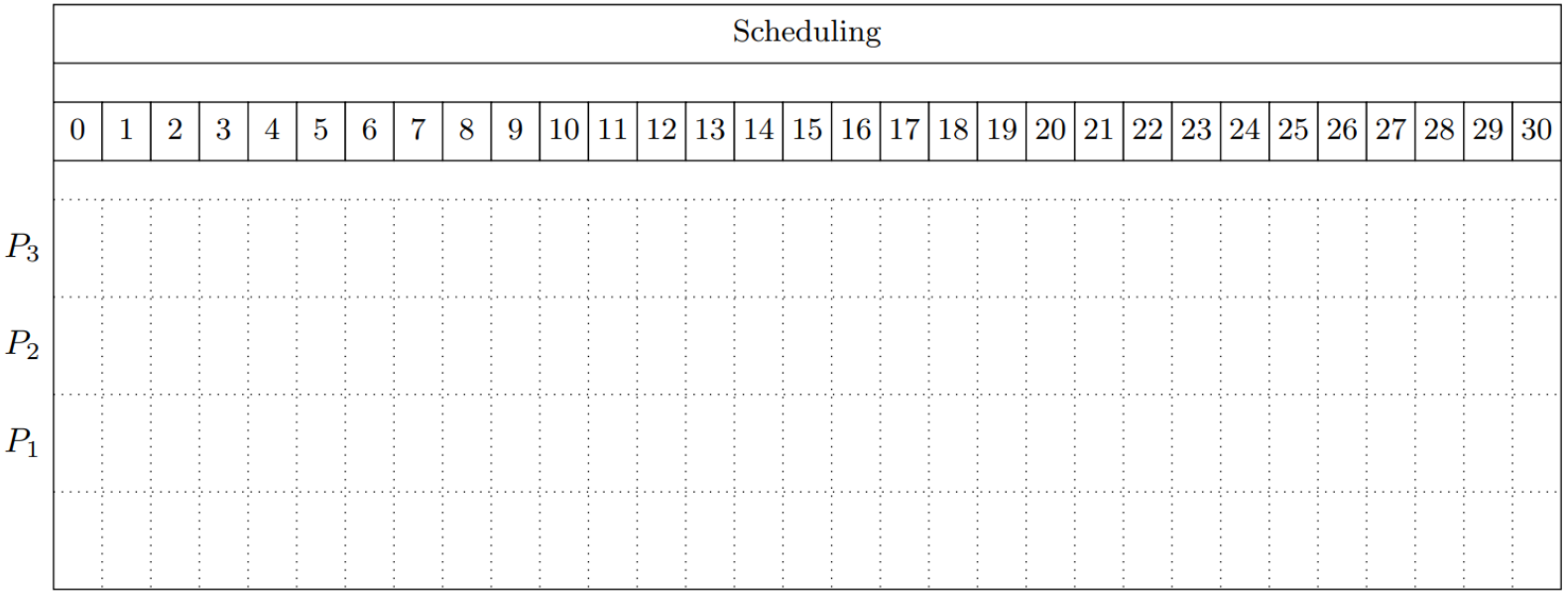
a = (0, 5, 2) r = (7, 3, 4)

Aufgabe 1 b) Shortest Remaining Time Next (SRTN): Preemptive



a = (0, 5, 2) r = (7, 3, 4)

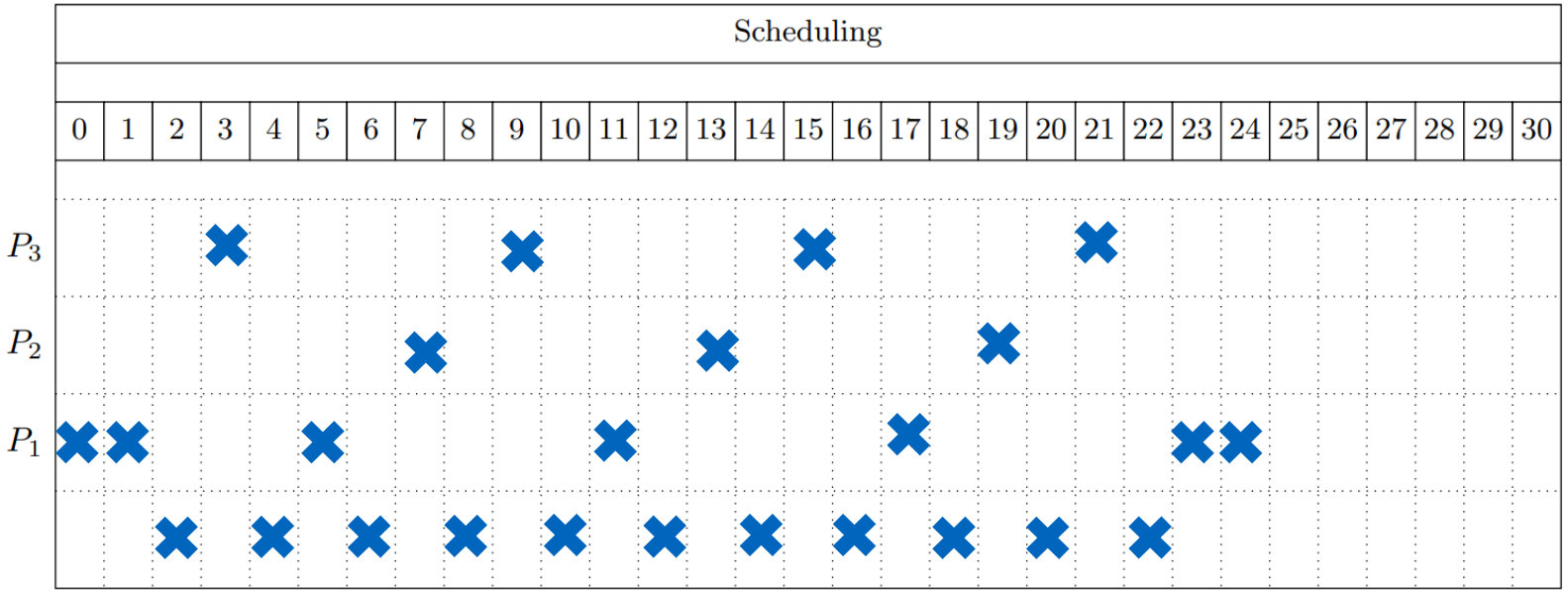
Aufgabe 1 c) Round-Robin mit einem Zeitquantum von einer Zeiteinheit und zyklischer Abarbeitung der Prozesse (Sortierung nach der PID)



a = (0, 5, 2) r = (7, 3, 4)

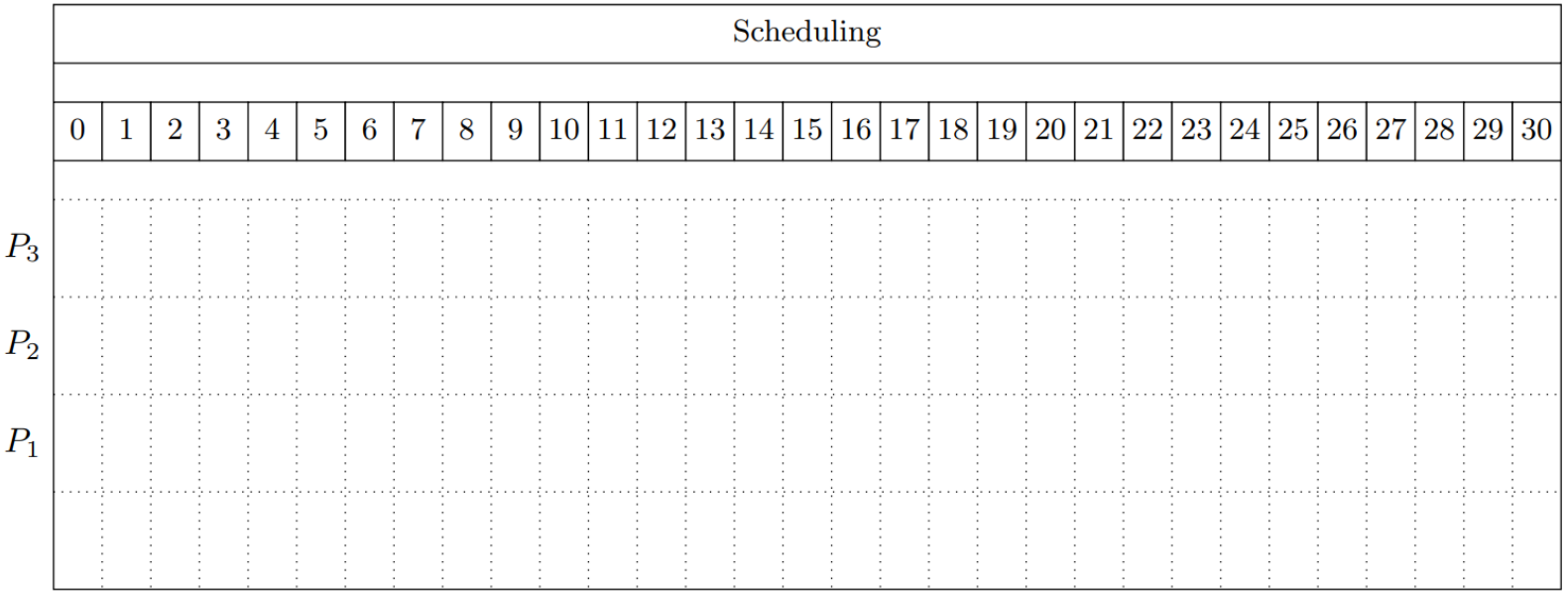
Aufgabe 1 c)

Round-Robin mit einem Zeitquantum von einer Zeiteinheit und zyklischer Abarbeitung der Prozesse (Sortierung nach der PID)



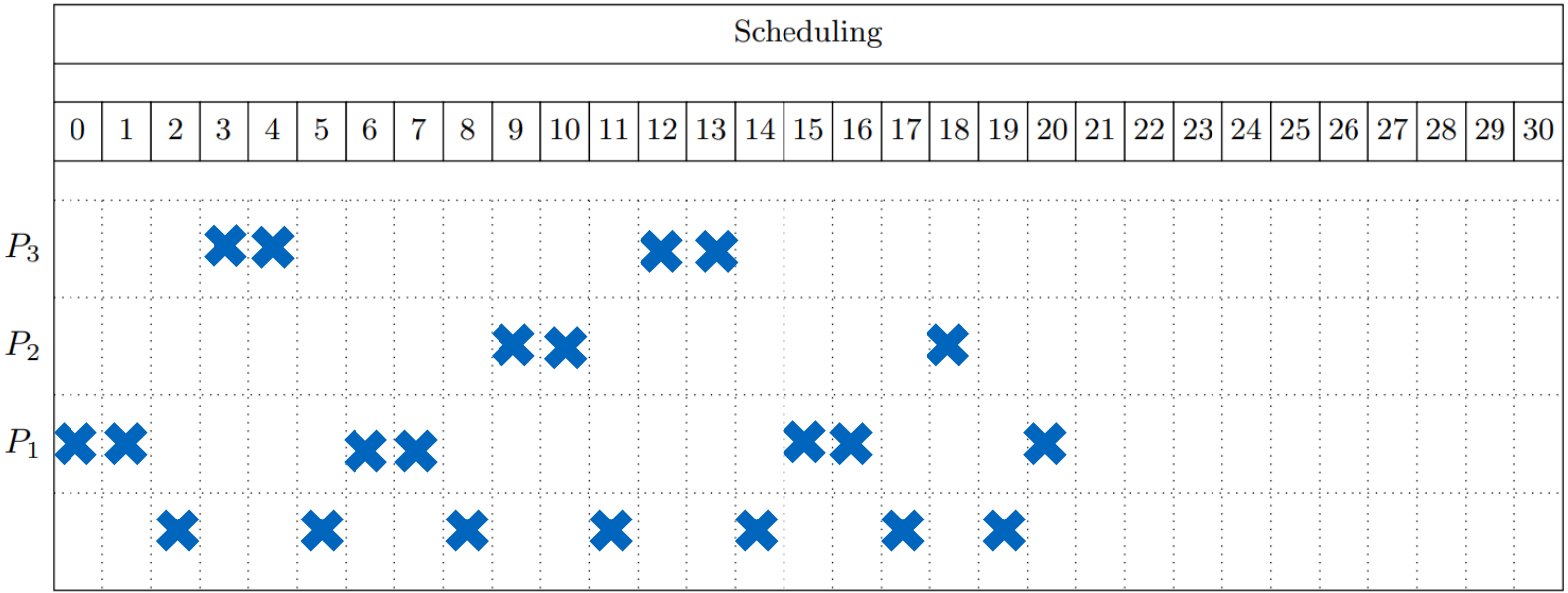
a = (0, 5, 2) r = (7, 3, 4)

Aufgabe 1 d) Round-Robin mit einem Zeitquantum von 2 Zeiteinheiten und zyklischer Abarbeitung der Prozesse (Sortierung nach der PID)



a = (0, 5, 2) r = (7, 3, 4)

Aufgabe 1 d) Round-Robin mit einem Zeitquantum von 2 Zeiteinheiten und zyklischer Abarbeitung der Prozesse (Sortierung nach der PID)



Aufgabe 2

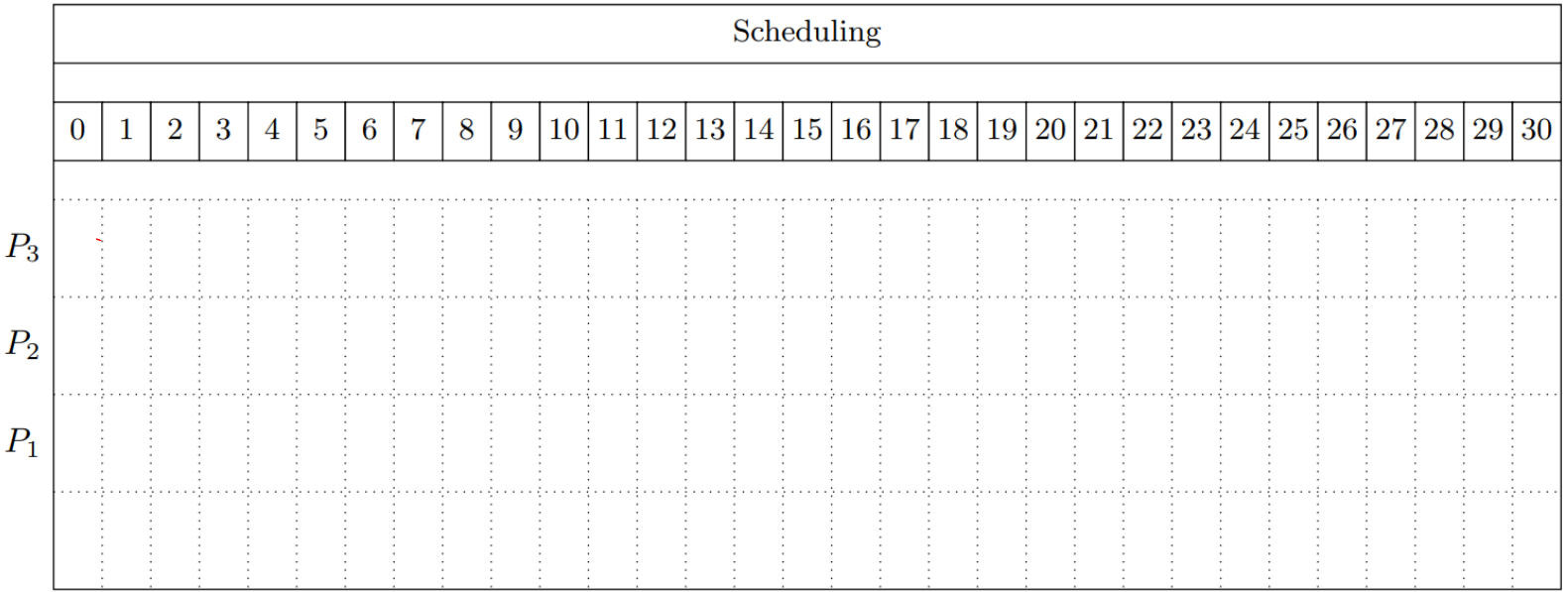
Priority Scheduling (priorisiertes Round Robin Scheduling Verfahren mit dynamischen Prioritäten)

- Quantum 2 Zeiteinheiten..
- Im rechnenden Zustand wird die Priorität des Prozesses je nach 1 Zeiteinheit um 2 erniedrigt.
- Im rechenwilligen Zustand wird die Priorität des Prozesses alle 2 Zeiteinheiten um 1 erhöht.
- Prioritäten reichen von 0 bis 20
- In jedem Zeitquantum wird der Prozess mit der höchsten Priorität ausgewählt.
- Vernachlässigen Sie dabei die Zeit, die durch den Scheduler und Dispatcher verbraucht wird.

Initialprioritäten I = (10, 9, 14); Ankunftszeiten a = (0, 2, 0); Rechenzeiten r = (6, 6, 8)

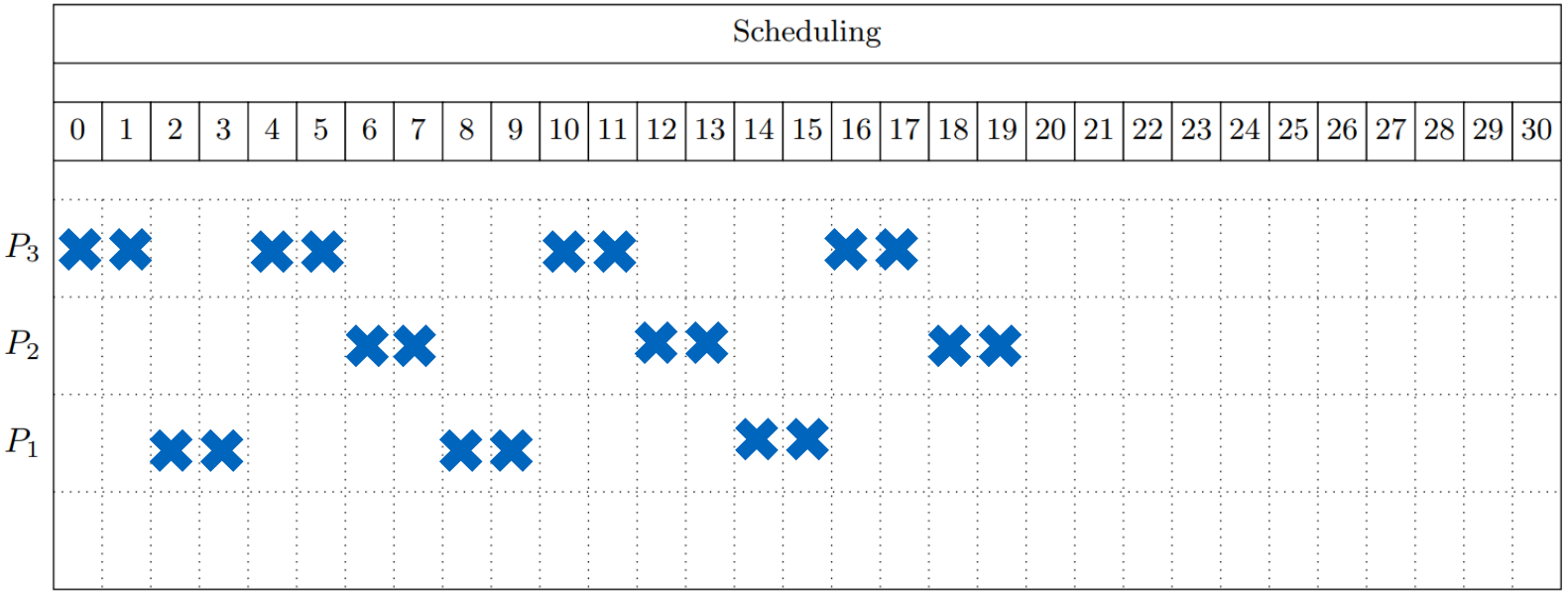
I = (10, 9, 14) a = (0, 2, 0) r = (6, 6, 8)

Aufgabe 2 Priority Scheduling



l = (10, 9, 14) a = (0, 2, 0) r = (6, 6, 8)

Aufgabe 2 Priority Scheduling



Aufgabe 2

b) Berechnen Sie die mittlere Wartezeit $\bar{W} = \frac{\sum_{i=1}^{n} w_i}{n}$ und die mittlere Verweilzeit $\bar{V} = \frac{\sum_{i=1}^{n} v_i}{n}$ für dieses Szenario.

Aufgabe 2

b) Berechnen Sie die mittlere Wartezeit $\bar{W} = \frac{\sum_{i=1}^{n} w_i}{n}$ und die mittlere Verweilzeit $\bar{V} = \frac{\sum_{i=1}^{n} v_i}{n}$ für dieses Szenario.

- V = (16 + 18 + 18)/3 = 52/3
- W = (10 + 12 + 10)/3 = 32/3

Aufgabe 2

c) Was ist der Vorteil von dynamischen Prioritäten gegenüber statischen Prioritäten?

Aufgabe 2

c) Was ist der Vorteil von dynamischen Prioritäten gegenüber statischen Prioritäten?

- Falls ein Prozess schon länger nicht mehr am Rechnen war, dann erhält er irgendwann eine sehr hohe Priorität und darf rechnen, es ist somit garantiert das er irgendwann rechnet
- Bei statischen Prioritäten können Prozesse mit niedriger Priorität kontinuierlich von anderen Prozessen verdrängt werden, somit verhungern diese

Aufgabe 3

a) Betrachten Sie die nachfolgende Implementierung einer Bibliotheksfunktion. Um welche Funktion handelt es sich? Was ist natürlichsprachlich die Abbruchbedingung?

```
void fct(char *s, const char *t) {
    while(*s++ = *t++);
}
```

Aufgabe 3

a) Betrachten Sie die nachfolgende Implementierung einer Bibliotheksfunktion. Um welche Funktion handelt es sich? Was ist natürlichsprachlich die Abbruchbedingung?

```
void fct(char *s, const char *t) {
    while(*s++ = *t++);
}
```

- Die Funktion kopiert Strings. → strcpy, Nullbyte terminiert Schleife

Aufgabe 3

b) Wie unterscheiden sich die folgenden Typdeklarationen? Es gilt: sizeof(void*)==8 und sizeof(short)==2

```
struct v {
    char a;
    short h;
    struct v *o;
}
```

```
struct v {
    struct v *o;
    short h;
    char a;
}
```

Aufgabe 3

b) Wie unterscheiden sich die folgenden Typdeklarationen? Es gilt: sizeof(void*)==8 und sizeof(short)==2

```
struct v {
    char a;
+1  short h;
+4  struct v *o;
}
```

```
struct v {
    struct v *o;
    short h;
    char a;
+5
}
```

- sizeof(struct v) == 16, aufgrund von Alignment gibt es Padding zwischen Elementen

Aufgabe 3

c) Betrachten Sie folgendes C-Programm, welches die n-te harmonische Zahl berechnet. Probleme?

```c
#include <stdio.h>
#include <stdlib.h>

// Returns the first n harmonic numbers
double* harmonic_numbers(unsigned int n) {
    double result[n];
    result [0] = 1.0;

    for (unsigned int i = 1; i < n; i++) {
        result[i] = result[i-1] + (1.0 / (double) (i + 1));
    }
    return result;
}

void print_harmonics(unsigned int n) {
    if (n == 0) return;
    double *result = harmonic_numbers(n);
    for (unsigned int i = 0; i < n; i++) {
        printf("%f\n", result[i]);
    }
}
```

Aufgabe 3

c) Betrachten Sie folgendes C-Programm, welches die n-te harmonische Zahl berechnet. Probleme?

- double result[n]; ist Allokation auf dem Stack (7)
- Adresse von result wird per return von der Methode zurückgegeben (12)
- Code Block endet und gibt alle Ressourcen auf dem Stack frei (13)
- Pointer auf deallokierten Speicher wird dereferenziert (19)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Returns the first n harmonic numbers
5  double* harmonic_numbers(unsigned int n) {
6      double result[n];
7      result [0] = 1.0;
8
9      for (unsigned int i = 1; i < n; i++) {
10         result[i] = result[i-1] + (1.0 / (double) (i + 1));
11     }
12     return result;
13 }
14
15 void print_harmonics(unsigned int n) {
16     if (n == 0) return;
17     double *result = harmonic_numbers(n);
18     for (unsigned int i = 0; i < n; i++) {
19         printf("%f\n", result[i]);
20     }
21 }
```