BKS Übung 3

Leander "Apache" Tolksdorf, Simon Franke

Aufgabe 1

a) RR

Round Robin bedeutet, dass Prozesse nach einer gewissen Zeit unterbrochen werden und der nächste Prozess wird ausgeführt, sofern einer in der Warteschlange liegt. Erst wenn alle Prozesse ausgeführt wurden, wird der erste wieder gestartet. Deshalb ist das Verfahren präemptiv und da jeder Prozess gleich oft und lange ausgeführt wird wird kein Prozess verhungern.

b) HRRN

Das Highest Response Ration Next Verfahren basiert auf der Berechung (time spend waiting + expected service time)/ expected service time. Deshalb werden Prozesse mit erwarteter kürzerer Laufzeit vorgezogen, aber dadurch das auch die bisherige Wartezeit betrachtet wird, wird kein Prozess verhungern, auch wenn es bei langen Prozessen lange dauern kann, bis sie ausgeführt werden. Das Verfahren ist nicht präemtiv.

c) SPN

Der Shortest Process Next Algorithmus ist ein nicht präemtiver Algorithmus, bei dem zunächst alle kurzen Prozesse abgearbeitet werden. Dadurch können lange Prozesse verhungern.

d) SRT

Der präemtive Shortest Remaining Time Algorithmus unterbricht laufende Prozesse, wenn ein Prozess mit kürzerer Laufzeit reinkommt als der laufende Prozess noch benötigt. Dadurch können lange Prozesse verhungern,

e) FCFS

Die First Come First Served Strategie ist non-preemtive, da Prozesse stur nach der Reihenfolge der Eingänge abgearbeitet werden. Dabei verhungern keine Prozesse, aber auch kurze oder hoch priorisierte Prozesse müssen ggf. lange warten.

Scheduling per Hand

Die Kunden teilen sich wie folgt auf die Kassen auf:

K1: 6,10,13,20,4,21
K2: 15,3,15,19,6
K3: 23,7,40
Die Wartetzeit bis ein Kunde an der Reihe ist beträgt im Durchschnitt 23.1429 Zeiteinheiten, die Wartezeit bis ein Kunde fertig ist beträgt im Durchschnitt 37,57 Zeiteinheiten.

Aufgabe 2

```c
#include "scheduler.h"
#include <stdbool.h>
#include <math.h>
#include <stddef.h>

/*
TI3: Betriebs- und Kommunikationssysteme
Bearbeiter: Leander Tolksdorf, Simon Franke (Zusammenarbeit mit Sascha Brüning
und Florian Formanek)
Tutor: Leon Dirmeier
*/

/*
Returns the currently active process by checking all processes' states.
*/
struct process *active(struct process *head) {
    struct process *result = NULL;

    for (struct process *p = head -> next; p != head; p = p -> next) {
        if (p -> state == PS_RUNNING) {
            result = p;
            break;
        }
    }
    return result;
}

/* Round Robin */

/*
Returns the next process with state "READY", starting from "current".
If no ready process after "current", start seaching from the beginning.
If no ready process at all -> return NULL.
*/
struct process *nextReady(struct process *head, struct process *current) {
    for (struct process *c = current -> next; c != head; c = c -> next) {
        if (c -> state == PS_READY) {
            return c;
        }
    }
    for (struct process *c = head -> next; c != current; c = c -> next) {
        if (c -> state == PS_READY) {
            return c;
        }
    }
    return NULL;
}

void rr(struct process *head) {
    // Define pointer to current running process (if none -> NULL)
    struct process *current = active(head);

    // If no running process -> Set first to running and return.
    if (!current) {
```

```
        head -> next -> state = PS_RUNNING;
        return;
    }

    // Define pointer to next process (if none -> NULL)
    struct process *next = nextReady(head, current);

    // If no next process and current process done -> Kill current.
    if (!next) {
        if (current -> cycles_todo == 0) {
            current -> state = PS_DEAD;
        }
    // If next process waiting -> Set current to running or ready
    } else {
        if (current -> cycles_todo == 0) {
            current -> state = PS_DEAD;
        } else {
            current -> state = PS_READY;
        }
        // Set next process to Running.
        next -> state = PS_RUNNING;
    }

}
/* First Come First Serve */
void fcfs(struct process *head) {
    // Get current process (If none -> NULL)
    struct process *current = active(head);

    // If no running process -> set first ready to running and return
    if (!current) {
        nextReady(head, head) -> state = PS_RUNNING;
        return;
    }
    // If running process done -> Kill process and run next.
    if (current -> cycles_todo == 0) {
        current -> state = PS_DEAD;
        if (current -> next != head) {
            current -> next -> state = PS_RUNNING;
        }
    }

    // for (struct process *c = head -> next; c != head; c = c -> next) {
    //     if (c -> cycles_todo == 0) {
    //         c -> state = PS_DEAD;
    //         if (c -> next != head) {
    //             c -> next -> state = PS_RUNNING;
    //         }
    //     }
    // }
}

/*
Returns the process with the minimal cycles_todo.
Die Hilfsfunktion shortest funktioniert sowohl für SPN als auch SRT.
Bei SPN gilt für alle Ready-Prozesse: cycles_done = 0. Deshalb prüfen wir auch
nur
```

```
cycles_todo. Bei SRT interessiert auch nur cycles_todo. Deshalb können wir
shortest bei
SPN wiederverwenden.
*/
struct process *shortest(struct process *head) {
    int duration;
    int shortest = (int) INFINITY;
    struct process *result = NULL;
    // Iteriere durch Prozesse p
    for (struct process *p = head -> next; p != head; p = p -> next) {
        // Wenn (p READY)
        if (p -> state == PS_READY) {
            // Berechne Dauer von p (= cycles_todo. Wir ignorieren cycles_done,
weil es bei spn keine Fragmentierung gibt.)
            duration = p -> cycles_todo;
            // Wenn p kürzer als bisherige ->
            if (duration < shortest) {
                shortest = duration;
                result = p;
            }
        }
    }
    if (!result) {
        result = head;
    }
    return result;
}

/* Shortest Process Next */

void spn(struct process *head) {
    // Define bool for idle (true when no process is running.)
    bool idle = true;
    // Define null pointer for shortest process.
    struct process *shortestProcess = NULL;

    /*
    Check all processes for state == RUNNING. If process done -> kill process.
    Else -> set idle = false;
    */
    for (struct process *p = head -> next; p != head; p = p -> next) {
        if (p -> state == PS_RUNNING) {
            if (p -> cycles_todo == 0) {
                p -> state = PS_DEAD;
            } else {
                idle = false;
            }
            break;
        }
    }

    // If idle (= no process running) -> set shortest process to RUNNING.
    if (idle) {
        shortestProcess = shortest(head);
        if (shortestProcess != head) {
            shortestProcess -> state = PS_RUNNING;
        }
    }
```

```
}

/* Shortest Remaining Time */

void srt(struct process *head) {
    // Define pointers for shortest process and currently active process.
    struct process *shortestRem = shortest(head);
    struct process *current = active(head);

    // If no process RUNNING (in the very beginning) -> set shortest process to
RUNNING and return.
    if (!current) {
        current = shortestRem;
        current -> state = PS_RUNNING;
        return;
    }

    // If shortestRem != head (= there are processes left) -> check current
processes' state.
    if (shortestRem != head) {
        // If current process done -> kill process and set shortest to RUNNING.
        if (current -> cycles_todo == 0) {
            current -> state = PS_DEAD;
            shortestRem -> state = PS_RUNNING;
        // If current process not done and there's a shorter process ready ->
set current to READY and shortest to RUNNING.
        } else if ((current -> cycles_todo) > (shortestRem -> cycles_todo)) {
            current -> state = PS_READY;
            shortestRem -> state = PS_RUNNING;
        }
    // If shortestRem == head (= no processes left) -> kill current process ->
done.
    } else {
        if (current -> cycles_todo == 0) {
            current -> state = PS_DEAD;
        }
    }
}
/* Highest Response Ration Next */

/*
Returns the response ratio of a process p.
*/
float responseRatio(struct process *p) {
    float waited = (float) (p -> cycles_waited);
    float serviceTime = (float) (p -> cycles_todo);

    return (waited + serviceTime) / serviceTime;
}

/*
Returns the process with the highest response ratio by comparing all ready-
processes' ratios.
If no process ready -> return NULL.
*/
struct process *highestRatio(struct process *head) {
    int highest = 0;
    int compare;
```

```
    struct process *result = NULL;

    for (struct process *p = head -> next; p != head; p = p -> next) {
        if (p -> state == PS_READY) {
            compare = responseRatio(p);
            if (compare > highest) {
                highest = compare;
                result = p;
            }
        }
    }
    if (!result) {
        result = head;
    }
    return result;
}

void hrrn(struct process *head){
    // Define pointers for current process and next process.
    struct process *current = active(head);
    struct process *next = NULL;

    // If no active process -> set highestRatio-process to RUNNING and return.
    if (!current) {
        current = highestRatio(head);
        current -> state = PS_RUNNING;
        return;
    }

    // If current process done -> Kill current process and set highestRatio-
process to RUNNING.
    if (current -> cycles_todo == 0) {
        next = highestRatio(head);
        if (next != head) {
            current -> state = PS_DEAD;
            next -> state = PS_RUNNING;
            return;
        // If next == head (= no processes left) -> kill current (last) process
-> done.
        } else {
            current -> state = PS_DEAD;
        }
    }
}
```