Professur für Betriebssysteme und Verteilte Systeme
Wintersemester 2025/2026, Prof. Dr. Bettina Schnor

Übung zur Vorlesung
Grundlagen Betriebssysteme und Rechnernetze

3. Übungsblatt, Abgabe bis Mo, 01. Dezember 2025, 10:00 Uhr

Hinweise zur Bearbeitung und Abgabe der Übung:

- Die Übungsaufgaben und Zusatzmaterial finden Sie auf Moodle:
  `https://moodle2.uni-potsdam.de/course/view.php?id=47479`
- Abgabe in 3er-Gruppen; tragen Sie sich im Moodle-Kurs in eine Abgabegruppe ein! Auf allen Abgaben Namen und Matrikelnummern der Gruppenmitglieder nicht vergessen!
- Laden Sie Ihre Lösungen der Theorieaufgaben als PDF in Moodle hoch.
- Nutzen Sie für die Praxisaufgaben Git.UP! Klonen Sie sich für die jeweilige Aufgabe die Vorlage aus der Gruppe `GBR Vorlagen`. Sollte es für die Aufgabe keine Vorlage geben, zB: Aufgabe 1.1, erstellen Sie ein eigenes Repository. Achten Sie darauf ihre Git-Projekte im GBR Namespace zu erstellen. Für weitere Details folgen sie dem Manual auf Moodle.
- Halten Sie sich an die in den praktischen Aufgaben gegebenen Dateinamen und Methodennamen!
- Bitte keine Umlaute und Leerzeichen in Verzeichnissen und Dateinamen in der Abgabe verwenden! Ersetzen Sie Umlaute in Dateinamen (ä wird zu ae etc.)!
- Bei Problemen mit den Praxisaufgaben oder den Abgaben in GitLab legen Sie bitte in Ihrem GitLab-Repository ein Issue an und markieren darin einen Lehrenden! Bei Fragen zu den Aufgaben benutzen Sie bitte das Diskussionsforum in Moodle!
- Bitte kommentieren Sie Ihre Programme sinnvoll! Verwenden Sie sprechende Bezeichner und behandeln Sie mögliche Fehler!
- Stellen Sie sicher, dass Ihre Programme auf `tiree.lab.cs.uni-potsdam.de` compilierbar und lauffähig sind!

Gesamtpunktzahl des Aufgabenblattes: 20 Punkte.

Aufgabe 3.1: Echtzeit-Scheduling und Prioritäten

a) Gegeben sei die folgende Menge an Rechenaufträgen mit ihren Planungsdaten:

Prozess	Ankunftszeit	Rechenzeit	Deadline
P_1	0	4	8
P_2	2	3	10
P_3	6	4	14
P_4	6	2	10
P_5	8	5	15

Prüfen Sie, ob ein gültiger Ablaufplan für die gegebenen Aufträge existiert, wenn die Prozesse auf einem Single-Core-System nicht unterbrechend ausgeführt werden!

Hinweis: Deadline meint hier die absolute Deadline, also den entsprechenden Zeitpunkt!

b) Windows/NT benutzt 16 feste Prioritäten (Prio-Klassen 16 - 31). Geben Sie für folgende Prozesse geeignete Prioritäten an, so daß alle Fristen eingehalten werden! Wie sieht der zugehörige Ablaufplan aus?

Prozess	r_i	t_i	d_i
P_1	0	2	10
P_2	1	4	10
P_3	2	3	15
P_4	8	2	10
P_5	9	4	14

Mit r_i: ready time; t_i: Schätzung für maximale Ausführungszeit; d_i: Deadline.

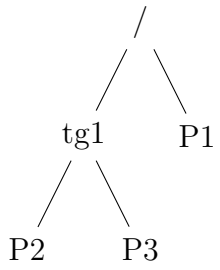
(2 + 2 Punkte)

Aufgabe 3.2: Linux CFS-Scheduling

Der Linux Kernel unterstützt seit der Kernel-Version 2.6.24 **Gruppen-basiertes Scheduling**. Beantworten Sie für ein **Einkern** Linux-System mit den folgenden **Gruppen (/ und tg1)** und **Prozessen (P1, P2 und P3)** die folgenden Fragen:

Der Prozess **P3** hat außerdem einen **nice-Wert** von -2, alle anderen Variablen haben ihren **Standardwert**.

a) Geben Sie die **CFS-Runqueues** (Red-Black-Trees) an, die der Linux-Kernel für die oben genannten Prozesse und Gruppen verwaltet, indem Sie die zugeordneten **Entitäten** angeben.



b) Diskutieren Sie, zu welchem Zeitpunkt ein Betriebssystem die **vruntime** einer Scheduling-Entität aktualisieren sollte. Überprüfen Sie Ihre Hypothese für den CFS-Scheduler.

c) Angenommen der Prozess **P3** läuft als erstes auf dem System für **10ms**. Bestimmen Sie die **vruntime** aller betroffenen Scheduling-Entitäten.

d) Bestimmen Sie die CPU-Zeit der Prozesse P1, P2 und P3 nach einer Sekunde in einem **idealen System**.

(1+1+1+2 Punkte)

Aufgabe 3.3: POSIX-Threads

In der Aufgabe 2.1 Prozesserzeugung haben Sie die Inferenz eines ML-Modells auf Kindprozesse verteilt. Benutzen Sie für die Inferenz nun POSIX-Threads.

a) Alle Threads sollen einen zuvor statisch festgelegten Bereich des `words` Arrays klassifizieren. Dabei sollen maximal 4 Threads gleichzeitig laufen. Die Ergebnisse der Klassifikation soll in ein gemeinsames Array geschrieben werden. Anschließend soll der initiale Thread nachdem alle anderen Threads beendet sind, die Gesamtergebnisse ausgeben.

b) Nutzen Sie den Debugger `gdb`, um die Ausführung des Programms zu verfolgen. Setzen Sie einen Breakpoint in der Thread-Funktionen der neu erstellten Threads und lassen Sie sich die Adressse einer lokalen Variable ausgeben. Ermitteln Sie anhand ihrer Ergebnisse die Stackgröße eines Threads.

c) Messen Sie mittels `clock_gettime` die Zeit für die gesamte Inferenz ohne die Ausgabe der Ergebnisse. Vergleichen Sie die Laufzeit mit der in der Übung präsentierten Ergebnissen für die Aufgabe 2.1. Erklären Sie die unterschiedlichen Laufzeiten.

Hinweis: Achten Sie darauf, dass die Threads keine gemeinsamen Variablen nutzen, bzw. nur auf unterschiedliche Bereiche eines gemeinsamen Arrays zugreifen.

(4 + 1 + 2 Punkte)

Aufgabe 3.4: Signalhandler für Broken Pipe

In der Vorlesung „05-2/3 IPC Pipes und Shared Memory“ wurde Ihnen ein Pipe-Beispiel vorgestellt. Auf Moodle verlinkt finden Sie im Materialorder eine abgespeckte Version des Programms (`broken_pipe.c`), welches nur noch den Test 4 enthält. Zusätzlich finden Sie auch die benötigte Datei `foo` im Material Ordner, die die ersten 588 Zeichen des Gedichtes „Knecht Ruprecht“ enthält. Erweitern Sie das Programm `broken_pipe.c` um einen Signalhandler, welcher das Signal für das Schreiben in eine Pipe ohne Leser abfängt. Benutzen Sie dazu `sigaction`!

Der Signalhandler soll dem Nutzer mitteilen, dass ein entsprechendes Signal empfangen wurde. Anschließend soll das Programm alle offene Ressourcen schließen und mit `EXIT_FAILURE` beenden. Achten Sie darauf, dass Sie im Signalhandler nur async-signal-safe Funktionen benutzen (siehe `man signal-safety`).

(4 Punkte)