UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Aufgabenblatt 6
Musterlösung

Vorlesung Verteilte Systeme
Sommersemester 2020

Aufgabe 1: Java Sicherheitsmanager

a) Der Security-Manager erlaubt es einer Java-Anwendung, Sicherheitsrichtlinien zu implementieren. D.h., die Anwendung kann festlegen, ob bestimmte (potentiell „gefährliche“) Operationen ausgeführt werden dürden, oder nicht. Dazu gehören z.B. Zugriffe auf das Dateisystem, Kommunikation über das Netzwerk, das Beenden der virtuellen Maschine oder das Erzeugen von Fenstern. Sinnvoll ist dies immer dann, wenn die Anwendung Klassen (d.h. fremden Code) nutzt, deren Quelle ggf. nicht vertrauenswürdig ist. Ein Beispiel sind Java Applets, deren Code von irgendeinem Web-Server geladen und im Web-Browser ausgeführt wird. Damit der Applet-Code keinen Schaden anrichten kann, laufen Applets immer unter Kontrolle eines Security-Managers.

Der Security-Manager schützt also die Resourcen des lokalen Rechners vor Schaden, der durch die Ausführung von fremdem bzw. unbekanntem Code verursacht werden könnte.

Technisch wird der Security-Manager durch die Klasse `SecurityManager` (bzw. eine Unterklasse davon) realisiert. Die Methoden dieser Klasse führen (auf Basis einer Sicherheitsrichtlinie, die z.B. in Form einer Policy-Datei vorliegt) eine Vorabüberprüfung durch, ob eine bestimmte Methode/Operation ausgeführt werden darf. Falls ja, kehrern sie zurück, falls nein, werfen sie die Exception `java.lang.SecurityException` (siehe Java-Dokumentation zu `java.lang.SecurityManager`).

b) Wenn ein Security-Manager definiert wird, sind alle kritischen Operationen verboten, es sei denn, sie werden explizit erlaubt (z.B. durch eine Policy-Datei). Wird der Client also ohne Angabe der Ploicy-Datei (Property `java.security.policy`) gestartet, wird der Dateizugriff auf `/tmp/test` nicht erlaubt.

Die notwendige Policy-Datei finden Sie im Archiv `l06Files.zip`[1] auf der Vorlesungswebseite.

c) Die Lösung dieser Aufgabe finden Sie im Archiv `l06Files.zip`[2] auf der Vorlesungswebseite. Die Methode `checkWrite()` des Security-Managers prüft, ob der Dateiname auf `.java` endet und wirft in diesem Fall eine `SecurityException`. Anderenfalls muss noch die Methode `checkWrite()` der Oberklasse `SecurityManager` aufgerufen werden, um die Standard-Zugriffskontrollen durchzuführen.

Aufgabe 2: Lastverteilungsstrategien

a) statische Lastverteilung: vor der Ausführung wird ein Ablaufplan erstellt; Nachteil: alle Prozesse und deren Laufzeiten müssen a-priori bekannt sein, Dynamik der Prozesse und Fremdlast können nicht berücksichtigt werden.

dynamische Lastverteilung: Prozesse werden (erst) bei ihrer Erzeugung auf einen Knoten plaziert und bleiben dort; Nachteil: spätere Änderung der Lastsituation (durch Dynamik der Prozesse oder Fremdlast) bleibt unberücksichtigt.

dynamisch-präemptive Lastverteilung: Prozesse können während ihrer Laufzeit (auch mehrfach) den Knoten wechseln; Nachteile: Migrations-Problematik, Thrashing möglich; Vorteil: sehr adaptiv, vor allem bei Fremdlast.

b) Nicht alle rechenbereiten Prozesse verursachen gleich viel Last (z.B. Prozesse, die häufig auf I/O warten). Ein besseres Lastmaß könnte die CPU-Auslastung sein (also wieviel % der Zeit rechnet die CPU), die aber nicht immer einfach zu ermitteln ist und auch stark schwanken kann. Auch Speicherbedarf, Seitenfehlerraten, E/A-Raten, Kommunikationsraten etc. könnten in Lastmetriken mit einfließen. Die (häufige) Ermittlung dieser Metriken bedeutet

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l06Files.zip`
[2] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l06Files.zip`

aber zusätzlichen Overhead, der sich durch die „bessere“ Metrik nicht immer amortisiert. In Experimenten hat sich herausgestellt, daß (aufgrund dieses Overheads) durch solche Metriken i.a. auch kein besserer Lastausgleich erzielt wird (d.h. die Laufzeit eines Programmsystems nicht reduziert wird).

Aufgabe 3: Graphpartitionierung und List-Scheduling

Lastverteilung durch Graph-Partitionierung:
Gegeben ist eine Anzahl von nebenläufigen, kommunizierenden Prozessen und die Kommunikationslast zwischen je zwei Prozessen. Dies wird als Graph dargestellt. Die Knoten stellen die Prozesse dar und die Kanten zeigen die Kommunikationlast. Ziel: die auftretende Last möglichst gleichmäßig zu verteilen, um Überlast auf einzelnen Knoten zu verhinden. Der Graph wird so partitioniert, daß die Gewichte-Summe der geschnittenen Kanten minimal ist, d.h. die Kommunikation zwischen Knoten minimiert wird. Alle Lastverteilungsstrategien gründen ihre Entscheidungen auf irgendeine *Lastmetrik*.

List-Scheduling:
Das Programm bestehend aus mehreren abhängigen Tasks, die ggf. auf Ausgabedaten ihrer Vorgängertasks arbeiten, aber *während* der Abarbeitung nicht kommunizieren. Dies wird als DAG dargestellt. Die Knoten beinhalten Tasks mit Ausführungszeiten und die Kanten zeigen die notwendige Kommunikation mit Übertragungsdauer. Ziel: Scheduling der Tasks so, daß die Bearbeitungszeit (engl. *makespan*) minimiert wird.

Gemeisamkeiten:
Graphen mit Tasks als Knoten und Kommunikation als Kanten, Kantengewichte, Taskstruktur ist fest, Anzahl von CPUs ist vorgegeben, Schedule (Zuordnung von Prozessen zu CPUs) wird vor Programmstart berechnet, Ziel i.d.R. Minimierung der Laufzeit, ...

Unterschiede:
Programmstruktur (bei Graph-Partitionierung: nebenläufige Prozesse, die während ihrer Ausführung kommunizieren; bei List-Scheduling: Tasks, die ggf. Ausgabedaten anderer Tasks benötigen, aber *während* der Ausführung nicht kommunizieren), Bedeutung der Knotengewichte, ungerichteter vs. gerichteter Graph, ...

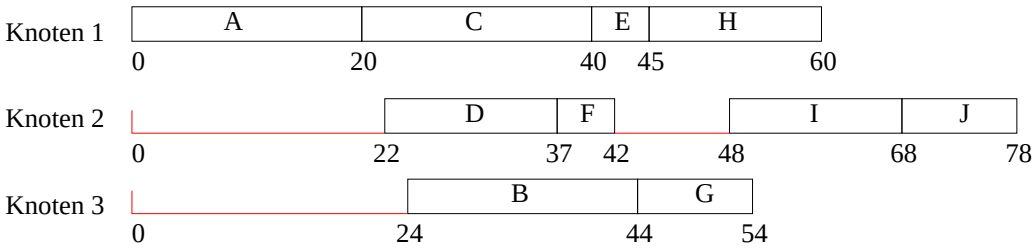
Aufgabe 4: Lastverteilung durch Graphpartitionierung

In dieser Aufteilung hat Knoten 1 die Prozesse A, E und G, Knoten 2 hat die Prozesse B, C und F, und Knoten 3 hat die Prozesse D, H und I. Der Schnitt zwischen Knoten 1 und 2 enthält jetzt die Kanten AB und EB mit einer Kommunikationslast von 5. Der Schnitt zwischen Knoten 2 und 3 enthält jetzt die Kanten CD, CI, FI und FH mit einer Kommunikationslast von 14. Der Schnitt zwischen Knoten 1 und 3 enthält jetzt die Kanten EH und GH mit einer Kommunikationslast von 8. Die Summe ist 27.

Aufgabe 5: List-Scheduling

Earliest Task First (ETF):

Schritt	ausgewählter Knoten / Static Level	Startzeit auf Prozessor p1	p2	p3	ausgewählter Prozessor
1	A / 70	0	0	0	p1
2	C / 50	20	28	28	p1
3	D / 45	40	22	22	p2
4	B / 40	40	37	24	p3
5	F / 35	40	37	44	p2
6	E / 30	40	42	44	p1
7	G / 20	52	52	44	p3
8	H / 25	45	53	53	p1
9	I / 30	60	48	54	p2
10	J / 10	76	68	76	p2



High Level First with Estimated Time (HLFET):

Schritt	ausgewählter Knoten / Static Level	Startzeit auf Prozessor p1	p2	p3	ausgewählter Prozessor
1	A / 70	0	0	0	p1
2	C / 50	20	28	28	p1
3	D / 45	40	22	22	p2
4	B / 40	40	37	24	p3
5	F / 35	40	37	44	p2
6	E / 30	40	42	44	p1
7	I / 30	50	48	50	p2
8	H / 25	45	68	53	p1
9	G / 20	60	68	44	p3
10	J / 10	76	68	76	p2

Mit HLFET ergibt sich in diesem Beispiel genau der gleiche Schedule wie mit ETF.

Aufgabe 6: Prozeßmigration

Es gibt etliche Zustandskomponenten eines Prozesses, die im BS gespeichert sind, z.B.

- die Tabelle offener Dateien (der Prozeß könnte nach der Migration nicht mehr auf diese zugreifen)
- die Signalmaske (der Prozeß würde ggf. bestimmte Signale nicht mehr richtig behandeln)
- gesetzte Alarm-Timer (der Prozeß würde das Alarm-Signal nicht mehr bekommen)
- etc.

Man müßte also zumindest diese Information des BS-Kerns mit übertragen. Alternativ kann man auch wieder Systemaufrufe an den Ursprungsknoten zurücksenden (und Signale etc. von diesem an den neuen Knoten weiterleiten).

Eine Migration in der beschriebenen Art ist natürlich prinzipiell nur zwischen Rechnern möglich, die exakt dieselbe CPU-Architektur und dieselbe BS-Version benutzen.

Aufgabe 7: Policies dynamischer Lastverteilungs-Systeme

a) Die **Transfer Policy** beschreibt, wie ein Computer entscheidet, ob (bzw. wann) Last verschoben werden soll. Typischerweise wird dies über Schwellwerte für die Last bestimmt. Wenn ein Computer eine Last hat, die größer als diese Schwelle ist, gibt er Tasks (Prozesse oder Jobs) ab. Umgekehrt kann ein Knoten, dessen Last kleiner als eine Schwelle ist, Tasks von anderen Knoten anfordern.

Sobald ein Knoten als Sender festgelegt ist, muss er wählen, welcher Task zu einem anderen Knoten gesendet werden soll (**Selection Policy**). Dabei muß sichergestell werden, dass der Overhead, der durch das Verschieben der Task verursacht wird, nicht den Nutzen des Verschiebens vernichtet. Eine Methode wäre, neue Tasks zu senden, die gerade in die Warteschlange gekommen sind. Dies stellt sicher, dass ein Task, der gesendet werden muss, nicht schon durchgeführt worden ist, d.h. keinen Zustand hat.

Die **Location Policy** beschreibt, wie man einen Empfänger-Knoten für die zu verschiebenden Tasks ermittelt (bzw. einen Sender-Knoten, wenn der Empfänger den Lastausgleich initiiert). Eine sehr einfache Möglichkeit besteht

darin, einen zufälligen Knoten zu nehmen. Besser wäre es, mehrere Knoten nach Ihrer Last zu befragen und den am wenigsten belasteten als Empfänger zu wählen.

Die **Information Policy** beschreibt, wie/wann die Last-Information ermittelt und verteilt wird. Die Last kann z.B. regelmäßig ermittelt werden, oder ereignisgesteuert (wenn sich die Lastsituation ändert, z.B. bei Erzeugung oder Beendigung von Tasks). Die Lastinformation kann global an alle Knoten oder auch nur lokal an die nächsten Nachbarn verteilt werden.

b) Empfänger-initiierte Verfahren können eine Verlangsamung des Systems (mit niedriger Last) ergeben. Die Leistung des Systems sinkt.

Wenn das System nie eine hohe Last hat, dann beschließen normalerweise alle Knoten durch die Transfer Policy, ein Empfänger zu sein. Wenn wir empfänger-initiierte Verfahren verwenden, wird jeder Knoten ständig Anfragen nach Tasks versenden, und es gibt keine Sender, um sie zu erfüllen. Diese Taskanfragen füllen das Netz mit nicht notwendigem Verkehr und können tatsächliche Taskübertragungen verlangsamen und damit die Leistung verringern.

Durch die Latenz zwischen Absenden einer Anfrage und Eintreffen einer Task kann es zudem zu Leerlaufzeiten auf Knoten kommen, obwohl vielleicht insgesamt genügend Tasks vorhanden wären, um alle Knoten zu beschäftigen.