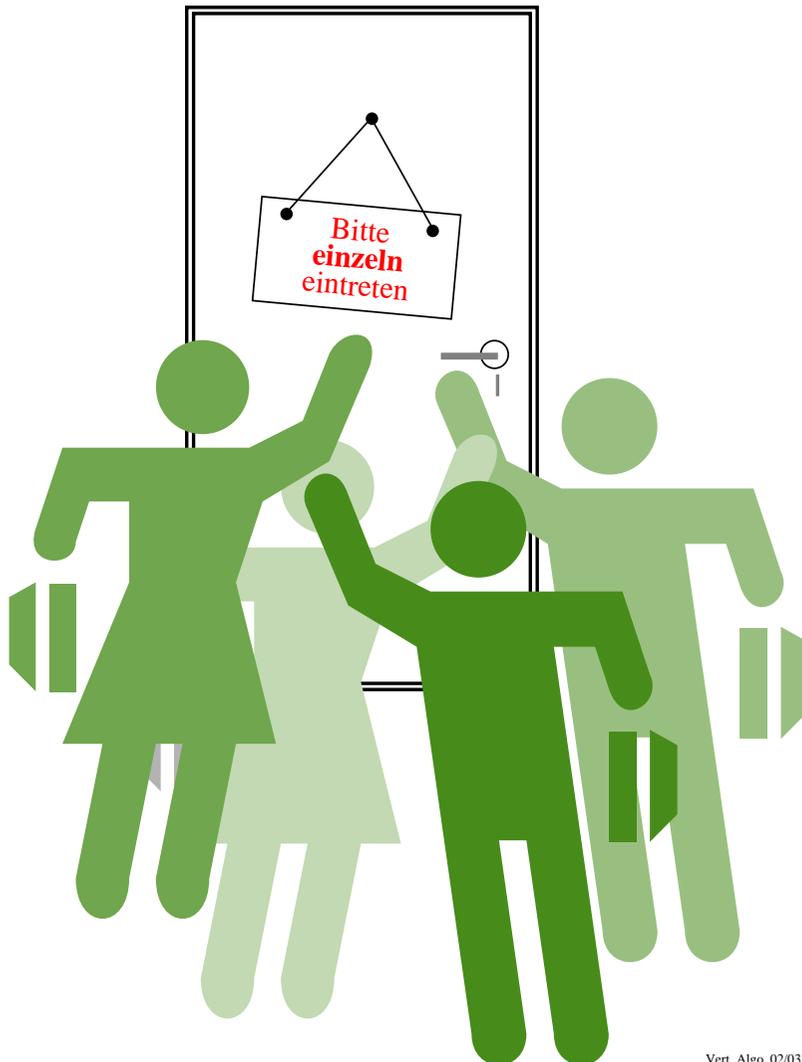


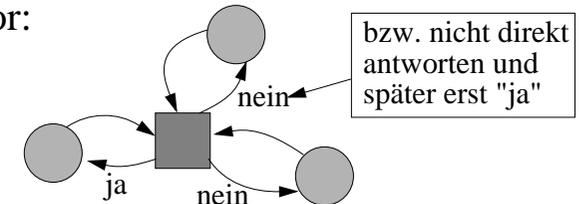
Wechselseitiger Ausschluss



Wechselseitiger Ausschluss: Request- (bzw. Permission)-basierte Prinzipien

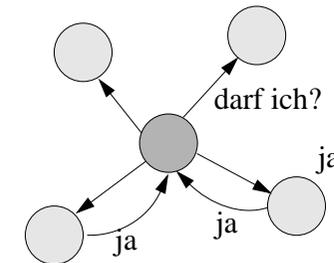
1) Zentraler Monitor: *Einen fragen*

- Engpass, schlecht skalierbar, nicht fehlertolerant



2) *Alle fragen*

- dezentral und symmetrisch, aber viele Nachrichten
- Algorithmen von Lamport (1978) und Ricart / Agrawala (1981) --> Vorlesung "Verteilte Systeme"



3) Kompromiss: dezentrale, symmetrische Lösung, bei der nur einige Prozesse um Erlaubnis gefragt werden?

- es müssen natürlich "ausreichend viele" sein

- Lassen sich Schemata als Ausprägung eines gemeinsamen allgemeinen Prinzips verstehen?

- > "ein für alle mal" verifizieren
- > abstraktere Sicht liefert hoffentlich tieferes Verständnis
- > ggf. neue Varianten mit interessanten Eigenschaften

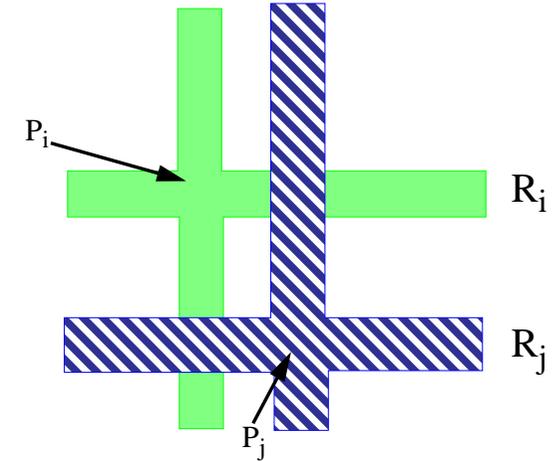
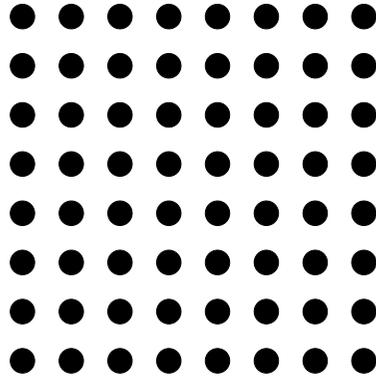
Siehe dazu Artikel von B. Sanders (ACM TOCS 5, 284-299, '87)

Maekawa's \sqrt{n} -Algorithmus (1985)

"... the algorithm is optimal in terms of the number of messages..."

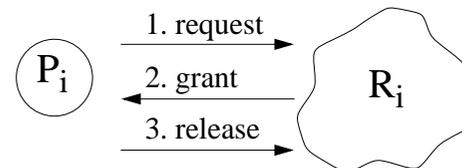
Idee in etwa:

- Anordnung der Prozesse in einem $\sqrt{n} \times \sqrt{n}$ - Gitter



- Prozess P_i hat eine Menge von Prozessen R_i , die er (mit request-Nachrichten) *um Erlaubnis fragen* muss
 - hier symbolisiert durch Prozesse in der Spalte / Zeile von P_i
- Die "request-granting" Mengen für je zwei Prozesse *überschneiden* sich garantiert! ($\forall i, j: R_i \cap R_j \neq \emptyset$)

- Grundidee:

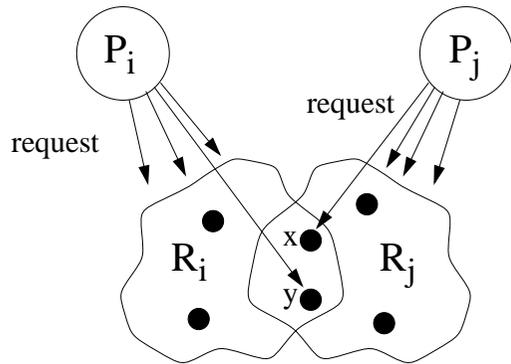


Ein Prozess wartet auf "grant" seiner Menge. Erst dann darf er den kritischen Abschnitt betreten. Nach Verlassen Menge mit "release" informieren.

- Nachrichtenkomplexität: $3 |R_i|$
--> minimale Mächtigkeit der R_i ?

Eine Erlaubnis ("grant") wird zu einem Zeitpunkt nur einem Bewerber erteilt.

Deadlock-Problematik



Beachte: Zweckmässigerweise ist oft $P_k \in R_k$, falls dies möglich ist.
Im Szenario könnte dann z.B. $x = P_j$ und $y = P_i$ sein.

- y antwortet P_i mit "grant", nicht jedoch P_j
 - x antwortet P_j mit "grant", nicht jedoch P_i
- \implies **Deadlock**, P_i und P_j warten auf weitere Zusage!

Lösung erfordert weitere Nachrichtentypen zur Deadlockvermeidung (bzw. Deadlockbehebung \rightarrow Symmetriebrechung)

- \implies soll hier nicht behandelt werden (\rightarrow Literatur)
- \implies erhöht Nachrichtenkomplexität jedoch nur um konstanten Faktor

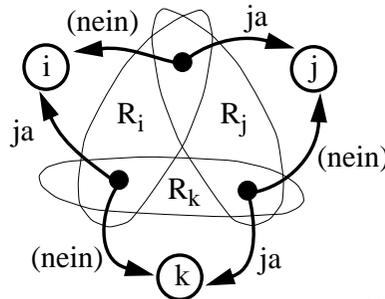
Gitteranordnung ist nicht optimal

- Zu jedem Paar von Prozessen P_i, P_j sind mindestens *zwei* Elemente im Schnitt von R_i, R_j (statt minimal einem)
- "Minimale" R_i können mit Hilfsmitteln der *projektiven Geometrie* bestimmt werden ($\rightarrow |R_i| \approx \sqrt{n}$ statt $2\sqrt{n}-1$)
 - *Endliche projektive Ebene* entsteht aus affiner Ebene unter Hinzunahme von "uneigentlichen Punkten" (=Parallelenbündel) und uneigentlichen Geraden (=Menge aller u. Punkte) und erfüllt folgende 3 Axiome:
 - zu je zwei Punkten gehört genau eine damit inzidente Gerade
 - zu je zwei Geraden gehört genau ein gemeinsamer Punkt
 - es gibt 4 Geraden, wovon keine 3 durch den selben Punkt gehen
 - Für eine *endliche projektive Ebene der Ordnung k* gilt:
 - jede Gerade enthält k Punkte
 - jeder Punkt liegt auf k Geraden
 - Ebene hat $k(k-1)+1$ Punkte und ebensoviele Geraden
 - *Idee*: Punkte \Leftrightarrow Prozesse, Geraden \Leftrightarrow request-granting-Mengen R_i .
 $n = k(k-1)+1 \rightarrow$ Jede Menge enthält $O(\sqrt{n})$ Elemente
 - *Leider* existiert nicht für jedes k eine endliche projektive Ebene der Ordnung k (mindestens jedoch dann, wenn k eine Primzahlpotenz ist); 1988 gezeigt (3000 Stunden Rechenzeit, erschöpfende Suche): es gibt keine projektive Ebene der Ordnung 10

Ist auch ein Deadlock möglich, wenn $|R_i \cap R_j| = 1$ für alle i, j ?

Ja, siehe nebenstehendes Szenario!

- Prozesse i, j und k wenden sich gleichzeitig an ihre entsprechenden Mengen
- jeweils ein Prozess daraus antwortet mit "ja"



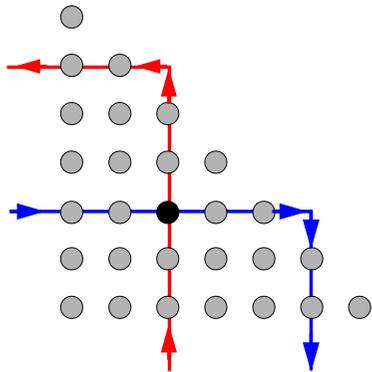
Beispiel $n=13, k=4$:
(Überprüfe, ob $|R_i \cap R_j| = 1$)

Beachte: Ummumerierung, so dass $i \in R_i$ (für alle i) ist möglich und zweckmässig!

- | | | |
|---------------------------|--------------------------|-----------------------------|
| $R_1 = \{1, 2, 3, 4\}$ | $R_6 = \{2, 6, 9, 12\}$ | $R_{10} = \{3, 7, 9, 11\}$ |
| $R_2 = \{1, 5, 6, 7\}$ | $R_7 = \{2, 7, 10, 13\}$ | $R_{11} = \{4, 5, 9, 13\}$ |
| $R_3 = \{1, 8, 9, 10\}$ | $R_8 = \{3, 5, 10, 12\}$ | $R_{12} = \{4, 6, 10, 11\}$ |
| $R_4 = \{1, 11, 12, 13\}$ | $R_9 = \{3, 6, 8, 13\}$ | $R_{13} = \{4, 7, 8, 12\}$ |
| $R_5 = \{2, 5, 8, 11\}$ | | |

Eine $\sqrt{2}\sqrt{n}$ Request-granting-Menge

- Prozesse werden *dreiecksförmig* angeordnet:

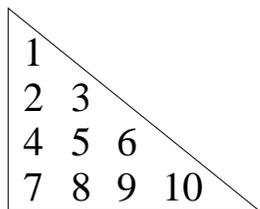


Schema 1 für R_i : Nimm alle Prozesse der Zeile von P_i sowie alle Prozesse derjenigen Spalte, die eins weiter rechts liegt als der rechteste Prozess der Zeile

Schema 2 für R_i : Nimm alle Prozesse der Spalte von P_i sowie alle Prozesse derjenigen Zeile, die eins weiter oben liegt als der oberste Prozess der Spalte

- Für beide Schemata gilt jeweils: Je zwei solche Mengen schneiden sich an mindestens einer Stelle (wieso?)

- Beispiel:



Mächtigkeit der R_i :
ca. $\sqrt{2}\sqrt{n}$ (wieso?)

Schema 1	Schema 2
$R_1 = \{1, 3, 5, 8\}$	$R'_1 = \{1, 2, 4, 7\}$
$R_2 = \{2, 3, 6, 9\}$	$R'_2 = \{1, 2, 4, 7\}$
$R_3 = \{2, 3, 6, 9\}$	$R'_3 = \{1, 3, 5, 8\}$
$R_4 = \{4, 5, 6, 10\}$	$R'_4 = \{1, 2, 4, 7\}$
$R_5 = \{4, 5, 6, 10\}$	$R'_5 = \{1, 3, 5, 8\}$
$R_6 = \{4, 5, 6, 10\}$	$R'_6 = \{2, 3, 6, 9\}$
$R_7 = \{7, 8, 9, 10\}$	$R'_7 = \{1, 2, 4, 7\}$
$R_8 = \{7, 8, 9, 10\}$	$R'_8 = \{1, 3, 5, 8\}$
$R_9 = \{7, 8, 9, 10\}$	$R'_9 = \{2, 3, 6, 9\}$
$R_{10} = \{7, 8, 9, 10\}$	$R'_{10} = \{4, 5, 6, 10\}$

- Schema 1 und 2 sind jeweils für sich unausgewogen:
Prozess 10 bzw. 1 kommt viel häufiger als andere vor

- aber: jede Zahl kommt genau 8 mal in *allen* Mengen vor (Lastsymmetrie!)
(Denkübung: Beweis, dass *stets* alle gleich oft vorkommen)
- Vorschlag: "alternierende" Benutzung der beiden Schemata (wie?)
Frage: Überschneiden sich R_i (Schema 1) und R'_j (Schema 2) jeweils?

**Man kann zeigen:
 $|R_i| = O(\sqrt{n})$ ist optimal
(für symmetrische Lösungen).**

Wir lassen das hier aber weg.