Оперативни Системи

Блокади

проф. д-р Димитар Трајанов,
проф. д-р Невена Ацковска,
проф. д-р Боро Јакимовски,
проф д-р Весна Димитрова,
проф. д-р Игор Мишковски,
проф. д-р Сашо Граматиков,
вонр. проф. д-р Милош Јовановиќ,
вонр. проф. д-р Ристе Стојанов,
доц. д-р Костадин Мишев

Цел на предавањето

- Ресурси – модел на систем
- Појава на блокада
- Детекција
- Справување
  - Превенција
  - Одбегнување
  - Дозволи и реагирај
  - Не прави ништо

Модел на систем – ресурси на систем

- Системот се состои од конечен број ресурси дистрибуирани меѓу процеси кои се натпреваруваат.
- Ресурсите се делат на неколку типови, од кои секој има одреден број на идентични инстанци
  - Меморискиот простор, CPU циклусите, датотеките и I/0 уредите (како принтери) се пример за типови ресурси.
  - Ако системот има 2 CPU, тогаш ресурсот од тип CPU има две инстанци. Слично, ресурсот од тип принтер може да има 5 инстанци
- Ако процесот бара инстанца од ресурсот, тогаш било која инстанца од ресурсот ќе го задоволи барањето.
  - Во спротивно, не се работи за ист ресурс

За проблемот

- Ситуација кога 2 или повеќе процеси се чекаат меѓусебно неограничено
- Множество процеси се блокирани кога некој од нив чека за настан што може да биде предизвикан само од некој друг процес од множеството процеси
- Се случува често кога се користат взаемни исклучувања (работа со критични секции),
- Пример:
  - Процесот А го има печатачот, а бара датотека или нејзин слог
  - Процесот В ја има датотеката или слогот, а го бара печатачот

Релевантен пример во бази на податоци

Proces A

lock(r2);

…

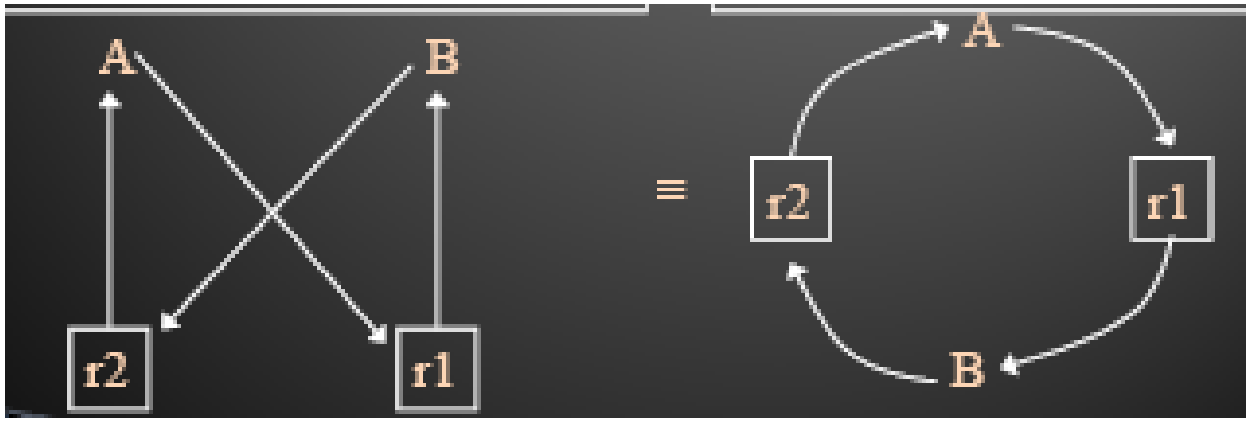
lock(r1); / се обидува

Proces B

lock(r1);

…

lock(r2); / се обидува



Блокадата е лоша

- Бидејќи:
- Таа го спречува процесот да „напредува“
- Блокадата бара интервенција за да биде прекината
- Таа ја намалува искористеноста на ресурсите
- Сите процеси чекаат
  - Ниту еден нема да направи настан кој би разбудил некој друг процес
  - Обично се чека на ослободување на ресурс, зафатен од друг блокиран процес
- Ниту еден од процесите НЕ
  - работи
  - ослободува ресурс
  - може да биде разбуден
- Сите процеси бесконечно чекаат

Блокада се случува кога 4 услови се точни истовремено:

- **Взаемно исклучување**: само еден процес во еден момент може да го користи ресурсот
- **Држи и чекај**: процес кој држи барем еден ресурс чека да добие дополнителни ресурси кои ги држат други процеси
- **Нема испразнување**: ресурсот може да биде ослободен само ако процесот го дозволи тоа, по комплетирање на неговата задача
- **Кружно чекање**: постои множество $\{P_0, P_1, \ldots, P_n\}$ чекачки процеси такви што P_0 чека за ресурс кој го држи P_1, P_1 чека за ресурс кој го држи P_2, ..., P_{n-1} чека за ресурс кој го држи P_n, и P_n чека за ресурс кој го држи P_0.

Решенија

- Во општ случај, постојат 4 стратегии за решавање на блокадите:
  - Превенција, реализирај ги ситуациите на конкурентност во ОС така што блокада да не може да се случи
  - Одбегнување, предвиди блокада и одбегни ја
  - Дозволи блокада (согледај ја) и реагирај
  - **Не прави ништо** (најчесто се користи во реалноста)

Систем со повеќе процеси и ресурси

- За да се детектира блокада во системот мора да постои формализација со која ќе бидат претставени процесите и ресурсите кои ни се на располагање

- Наједноставен начин е системот да го претставиме преку граф на алокација на ресурси.

Граф на алокација на ресурси

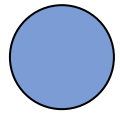
- Множество темиња V и множество лаци E
- V е партиционирано во два вида:

  $P = \{P_1, P_2, \ldots, P_n\}$, множество што ги содржи сите процеси во системот

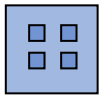
  $R = \{R_1, R_2, \ldots, R_m\}$, множество што ги содржи сите ресурси во системот
- лак на барања – насочен лак $P_i \rightarrow R_j$
- лак на доделување – насочен лак $R_j \rightarrow P_i$

Граф на алокација на ресурси (2)

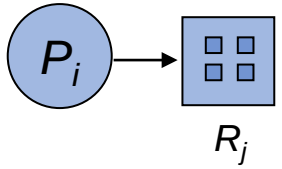
- Процес



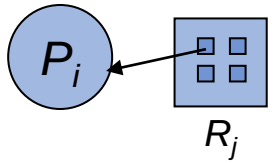
- Ресурс од 4 инстанци



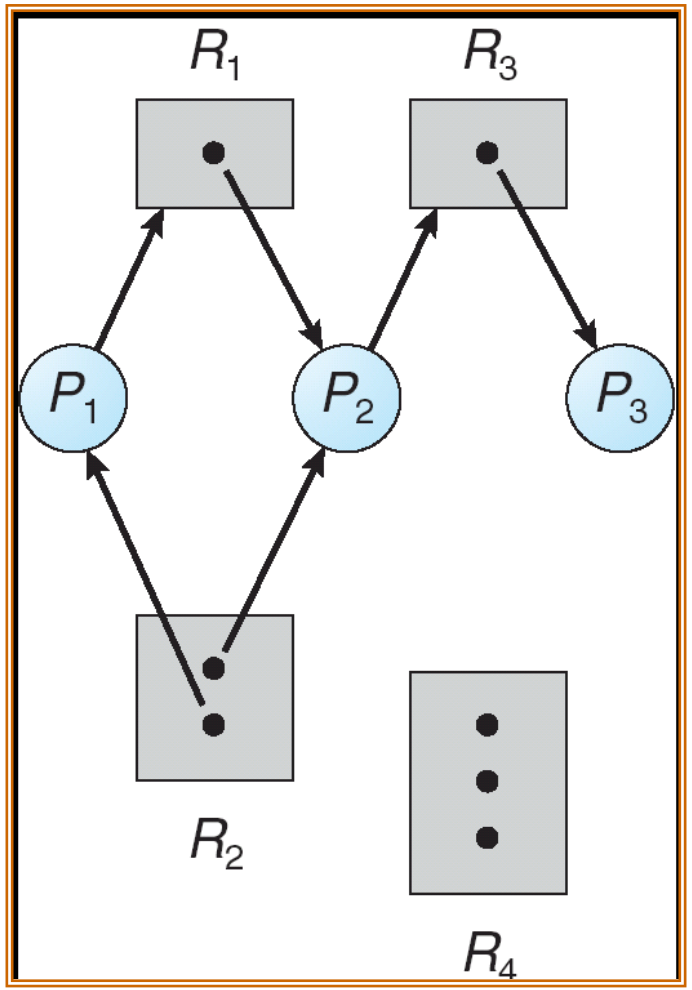
- P_i бара една инстанца на R_j



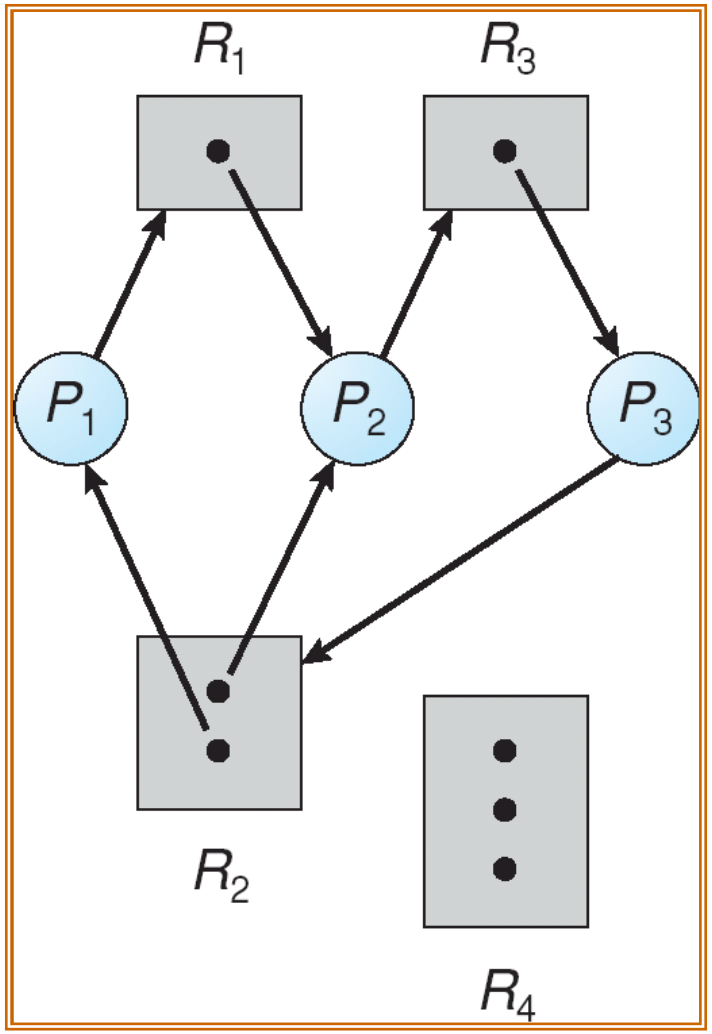
- P_i држи инстанца на R_j



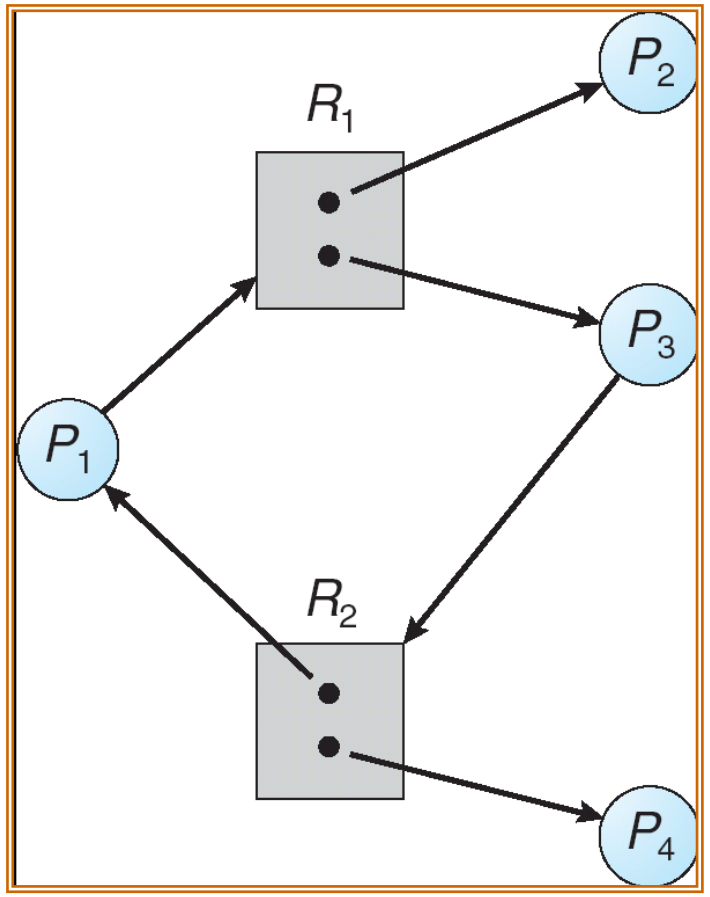
Пример на граф на алокација на ресурси



Граф со блокада



Граф со циклус, но без блокада



Основни факти

- ако графот нема циклуси $\Rightarrow$ нема блокада

- ако графот има циклус $\Rightarrow$
  - ако има само по една инстанца од ресурсот, тогаш има блокада
  - ако има повеќе инстанци од ресурсот, има можност за блокада

Пример

- Претпоставете ја следнава ситуација со ресурси во системот:
- Постојат три типа (класи) на ресурси R1, R2 и R3.

- Постојат по две инстанци од секој од ресурсите.

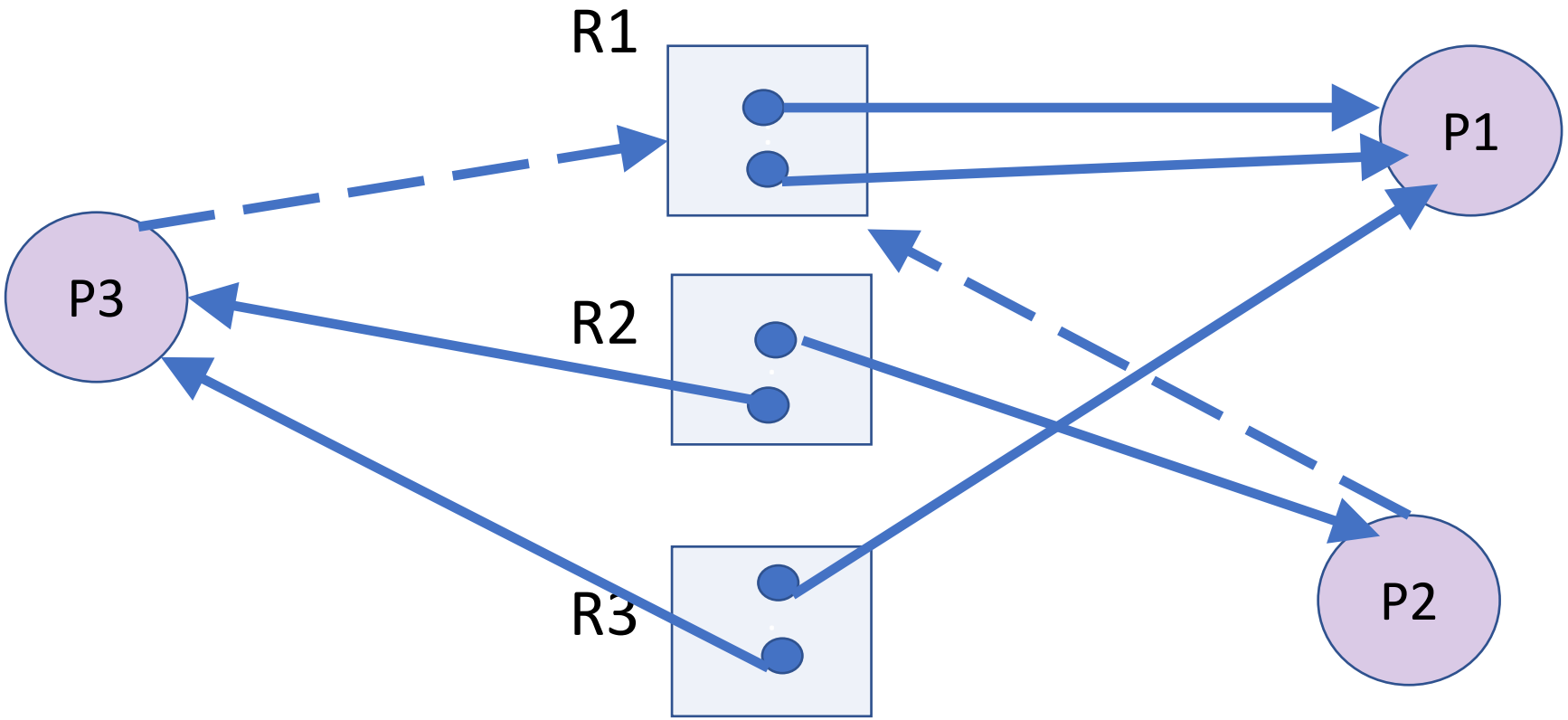
- Постојат три процеси означени од P1 до P3.

- Некои од инстанците на ресурсите веќе се доделени на процесите, и тоа: две инстанци од R1 држи P1, една инстанца од R2 ја држи P2, а втората ја држи P3, една инстанца од R3 ја држи P1, а другата ја држи P3

- Некои процеси побарале дополнителни инстанци, и тоа: P3 бара една инстанца од R1 и P2 бара една инстанца од R1.

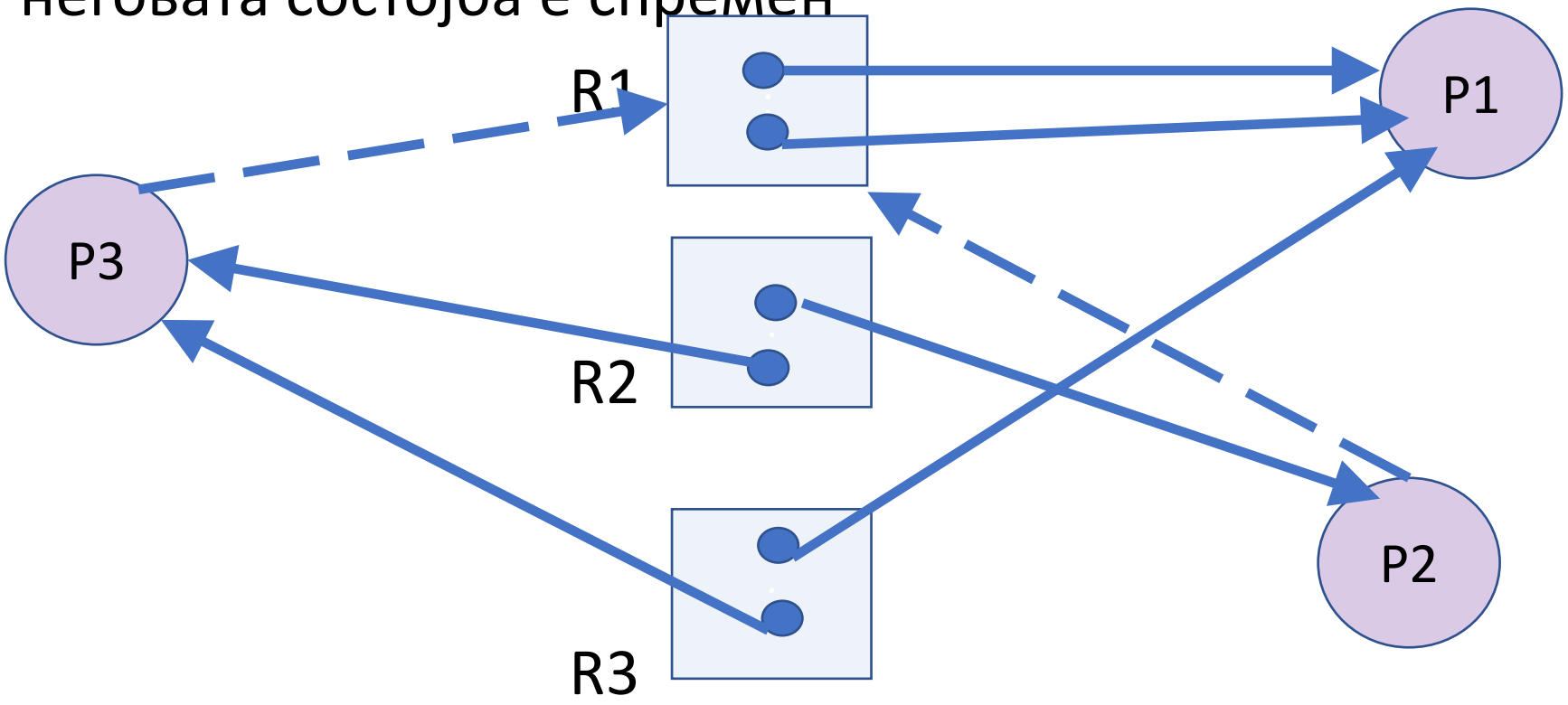
Пример

- Кој процес може прво да се изврши?



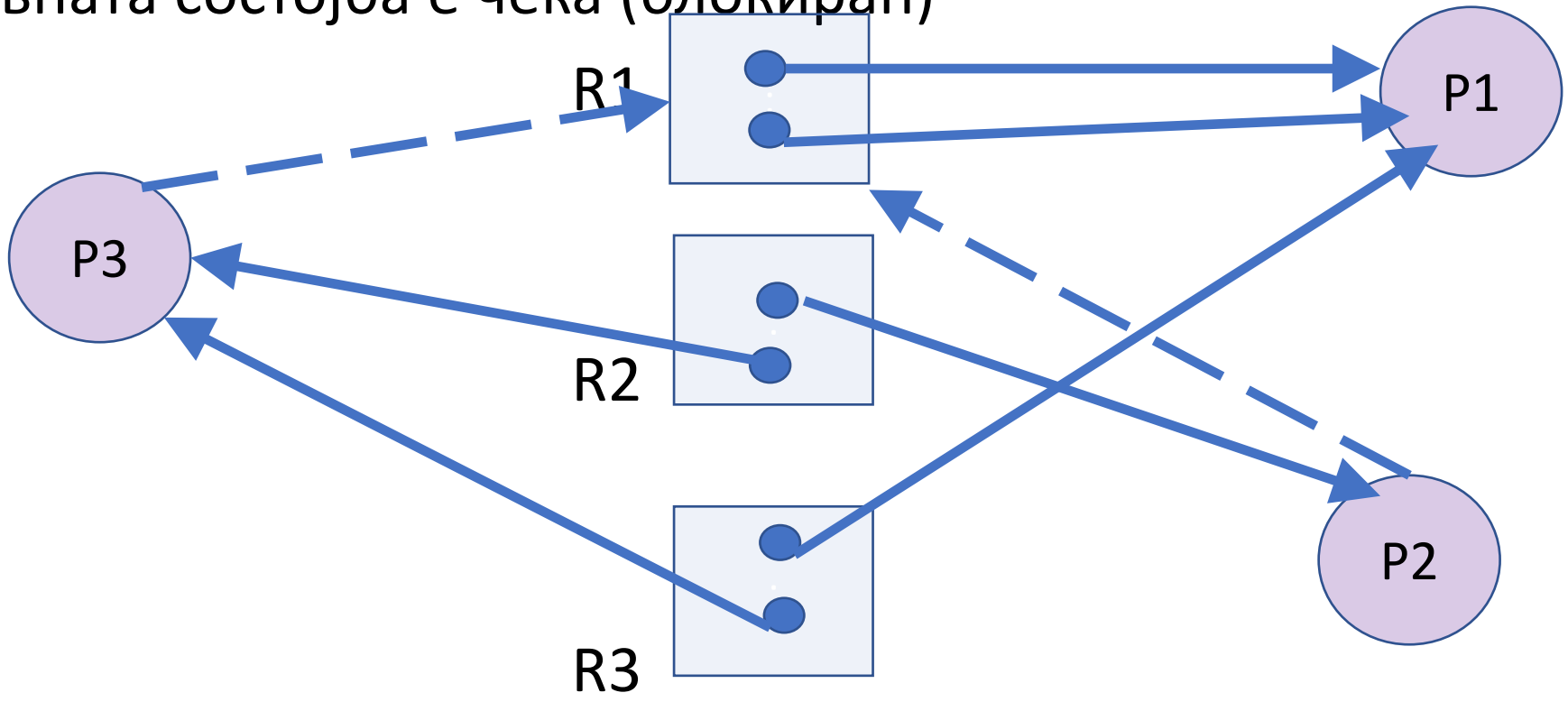
Пример

- Кој процес може прво да се изврши?
- P1, неговата состојба е спремен



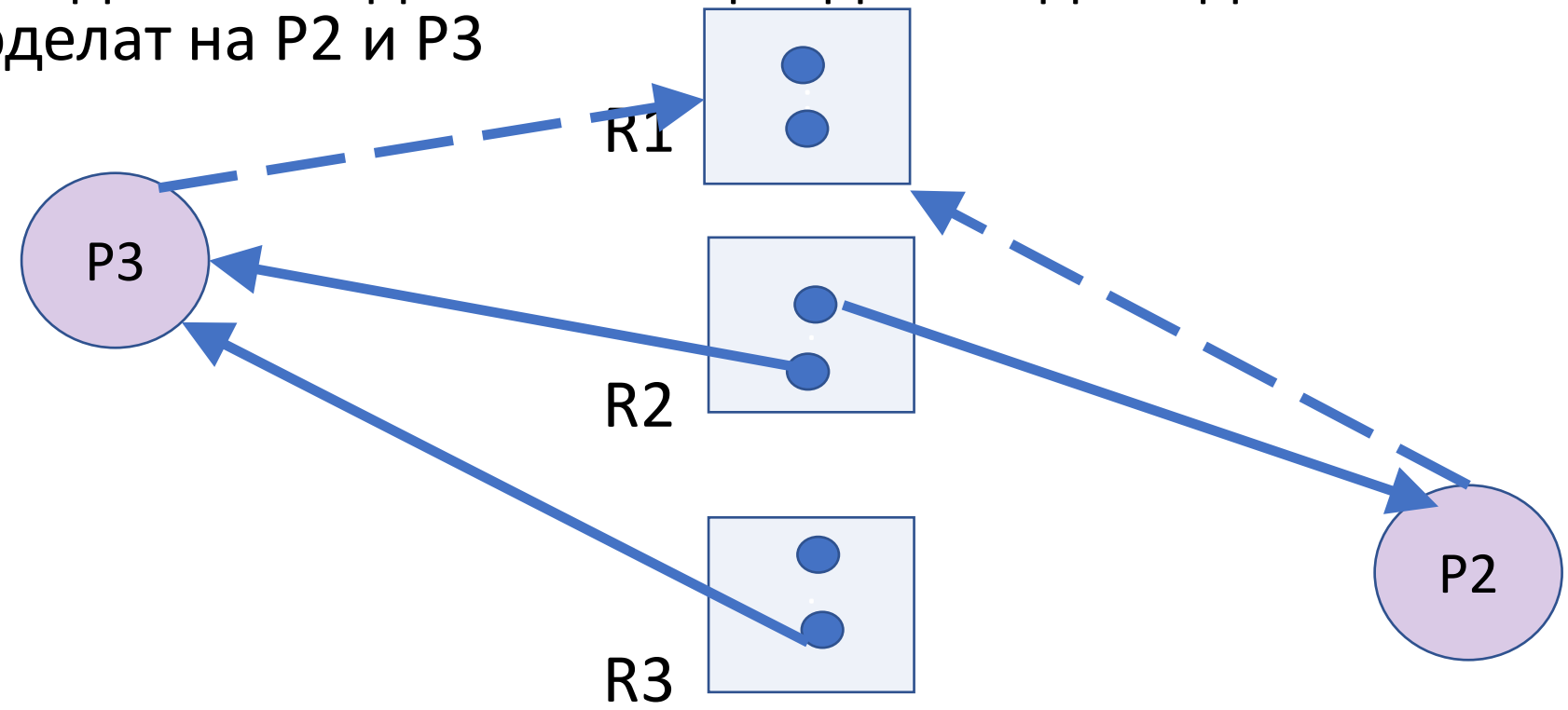
Пример

- Која е состојбата на P2 и P3?
- Нивната состојба е чека (блокиран)



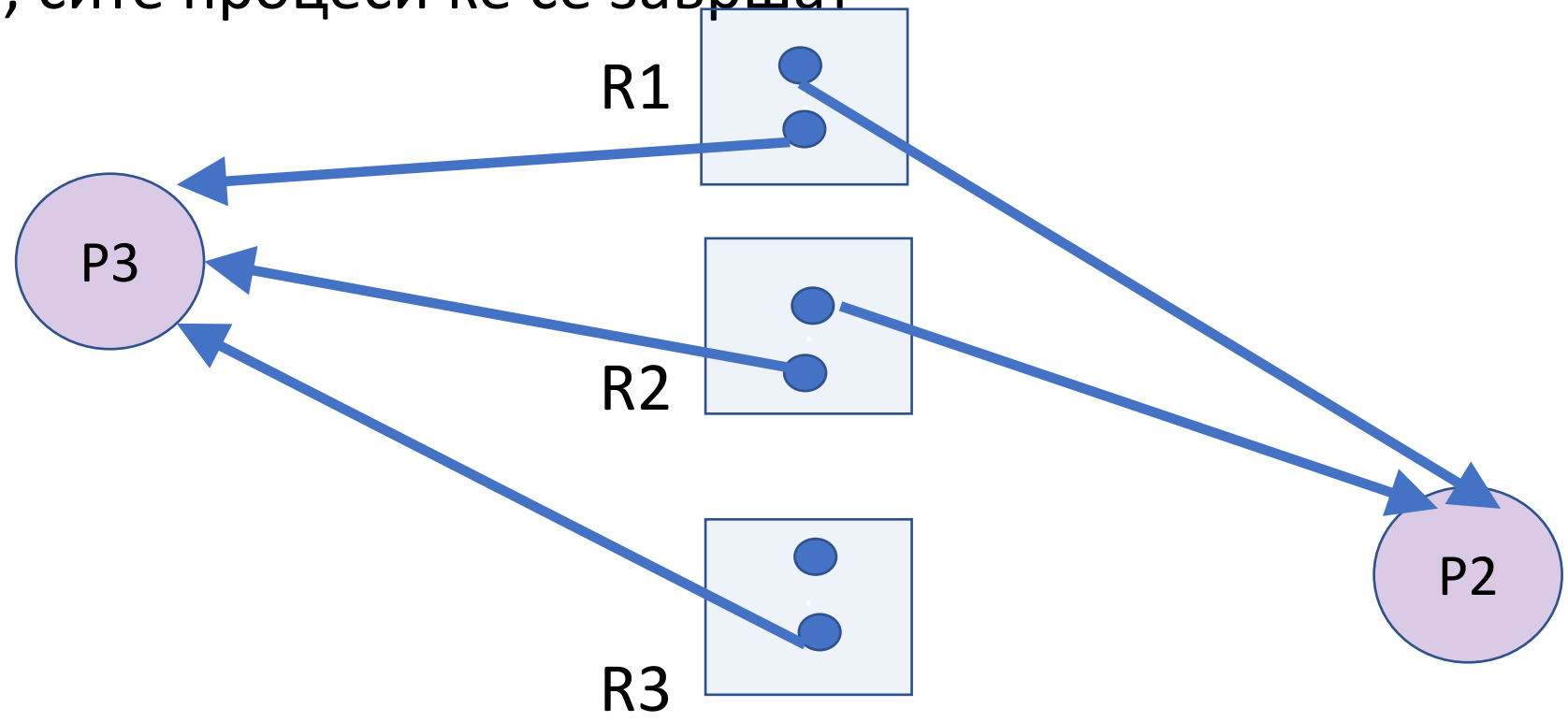
Пример

- Која е состојбата откако P1 ќе терминира?
- Има две слободни инсатнци од R1и една од R3 кои може да се доделат на P2 и P3



Пример

- Дали системот е во блокада?
- Не, сите процеси ќе се завршат



2. Одбегнување на блокада

- Постојат повеќе алгоритми за одбегнување на блокада, но за сите мора однапред да знаеме повеќе елементи:
  - колку ресурси се на располагање во системот
  - колку максимум ресурси му требаат на секој процес кој е стартуван да се изврши
  - како се распределени ресурсите по процесите кои се стартувани

- За да решиме проблем каде што имаме повеќе инстанци од ист тип, претставувањето со граф на алокација на ресурси може да не е прегледно

Банкаров алгоритам – повеќе инстанци од ист тип

- Формален метод со кој може да детектираме блокада во некој систем доколку однапред знаеме кои се елементите на системот е Банкаровиот метод:
  - Се знае колку ресурси се слободни во системот – **Available** вектор
  - Секој процес **мора однапред да знае колку максимум инстанци** од секој ресурс му требаат – **MAX** матрица
  - Оттука, може да знаеме уште колку ресурси му се потребни на секој процес за да се заврши – **Need** матрица
  - Кога процесот бара ресурс, можеби ќе мора да чека, ако ресурсите не се достапни во тој момент, те тековно доделени на други процеси
  - Ресурсите кои се тековно алоцирани знаеме кај кој процес се доделени - **Allocation** матрица
  - Кога процесот ќе ги добие сите ресурси што му требаат, мора да ги врати за конечно време

Структури податоци за банкаровиот алгоритам

Нека n = број на процеси и m = број на типови ресурси

- ***Available***: Вектор со должина m. Ако *available[j]* = k, тогаш постојат k слободни инстанци на ресурсот од тип R_j
- ***Max***: n x m матрица. Ако *Max* [*i,j*] = k, тогаш на процесот P_i може да му требаат најмногу k инстанци од ресурсот од тип R_j.
- ***Allocation***: n x m матрица. Ако Allocation[*i,j*] = k тогаш на P_i се тековно алоцирани k инстанци од R_j.
- ***Need***: n x m матрица. Ако *Need*[*i,j*] = k, тогаш на P_i може да му требаат уште k инстанци од R_j за да се заврши

Need [i,j] = Max[i,j] – Allocation [i,j]

Алгоритам за барање ресурси за процесот P_i

Request = вектор на побарувања за процесот P_i. Ако $Request_i[j] = k$ тогаш процесот P_i бара k инстанци од ресурсот R_j.

1. Ако $Request_i \leq Need_i$ оди на чекор 2. Инаку, пријави грешка бидејќи процесот ги достигна максималните барања
2. Ако $Request_i \leq Available$, оди на чекор 3. Инаку P_i мора да чека, бидејќи ресурсите не му се достапни.
3. Замисли дека му ги додели бараните ресурси на P_i модифицирајќи ја состојбата според:

   Available = Available – Request;

   $Allocation_i$ = $Allocation_i$ + $Request_i$;

   $Need_i$ = $Need_i$ – $Request_i$;

   - *ако е сигурна ⇒ додели ресурси на Pi.*
   - *ако е несигурна ⇒ Pi мора да чека, врати ја состојбата од пред да ги направиш измените*

Пример за банкаров алгоритам

- 5 процеси P_0 до P_4;

  3 типови ресурси:

  A (10 инстанци), *B* (5 инстанци), и *C* (7 инстанци).

- Во време T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Пример за банкаров алгоритам (2)

- Содржината на матрицата *Need* е *Max* – *Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	**3 3 2**
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Системот е во сигурна состојба бидејќи секвенцата < P_1, P_3, P_4, P_0, P_2> го задоволува условот за сигурност

Решение: Пример за банкаров алгоритам (3)

- Содржината на матрицата *Need* е *Max – Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	**3 3 2**
P_1	2 0 0	**1 2 2**	
P_2	3 0 2	6 0 0	
P_3	2 1 1	**0 1 1**	
P_4	0 0 2	4 3 1	

- Според векторот Available можат да се извршат процесите, P1 и P3, бидејќи P1 бара 1 инстанца од ресурсот A, а слободни се 3, 2 инстанци од B, а слободни се 3 и 2 инстанци од C, а слободни се 2. Слично, P3 не побарува ниту една инстанца од A за да се заврши, а бара 1 инстанца од B, а слободни се 3 и една инстанца од C, а аслободни се 3.

Пример за банкаров алгоритам (3)

- Содржината на матрицата *Need* е *Max* – *Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	**5 3 2**
P_1	0 0 0	0 0 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Нека се извршува P1 прв. Тогаш, во интервалот додека се извршува P1 ги содржи сите ресурси кои му се потребни, т.е P1 поседува 3 инстанци од A, 2 од B и 2 инстанци од C.
- Кога ќе се заврши, P1 ќе ги ослободи сите ресурси кои ги држел и тие ќе бидат достапни на другите процеси за да се довршат. По извршувањето на P1, векторот Available ќе има вредност 5 3 2

Пример за банкаров алгоритам (3)

- Содржината на матрицата *Need* е *Max* – *Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	**7 4 3**
P_1	0 0 0	0 0 0	
P_2	3 0 2	6 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 2	4 3 1	

- Следно може да се извршат P3 и P4 бидејќи нивите потреби за ресурси се помали од слободните во векторот Available. Нека се извршува P3 прв.

- Кога ќе се заврши, P3 ќе ги ослободи сите ресурси кои ги држел и тие ќе бидат достапни на другите процеси за да се довршат. По извршувањето на P3, векторот Available ќе има вредност 7 4 3

Пример за банкаров алгоритам (3)

- Содржината на матрицата *Need* е *Max* – *Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 0 0	0 0 0	**7 5 3**
P_1	0 0 0	0 0 0	
P_2	3 0 2	6 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 2	4 3 1	

- Следно може да се извршат секој од преостанатите процеси P0, P2 или P4 Нека се извршува P0 прв.

- Кога ќе се заврши, P0 ќе ги ослободи сите ресурси кои ги држел и тие ќе бидат достапни на другите процеси за да се довршат. По извршувањето на P0, векторот Available ќе има вредност 7 5 3

Пример за банкаров алгоритам (3)

- Содржината на матрицата *Need* е *Max* – *Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 0 0	0 0 0	**10 5 5**
P_1	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 2	4 3 1	

- Следно може да се извршат секој од преостанатите процеси P2 или P4. Нека се извршува P2 прв.

- Кога ќе се заврши, P2 ќе ги ослободи сите ресурси кои ги држел и тие ќе бидат достапни на другите процеси за да се довршат. По извршувањето на P2, векторот Available ќе има вредност 10 5 5

Пример за банкаров алгоритам (3)

- Содржината на матрицата *Need* е *Max* – *Allocation*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 0 0	0 0 0	**10 5 7**
P_1	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	

- Остана процесот P4.

- Кога ќе се заврши, P4 ќе ги ослободи сите ресурси кои ги држел. По извршувањето на P4, векторот Available ќе има вредност 10 5 7, колку што е максималниот број на ресурси во системот.

Корисност?

- Теоретски убаво, но практично бескорисно
  - Како однапред да го дознаеме бројот на ресурси по процес?
  - Бројот на активни процеси **динамички се менува**, нови корисници се вклучуваат и исклучуваат!
  - Некој ресурс може да биде недостапен (расипување и сл.)!

- Во практиката, многу малку постоечки системи го користат банкаровиот алгоритам за избегнување блокади!

3. Дозволи, детектирај и реагирај

- Дозволи ја блокадата
- Анализирај ја ситуацијата
- Одбери процес - жртва и отстрани го
- Направи надоместување (recovery)

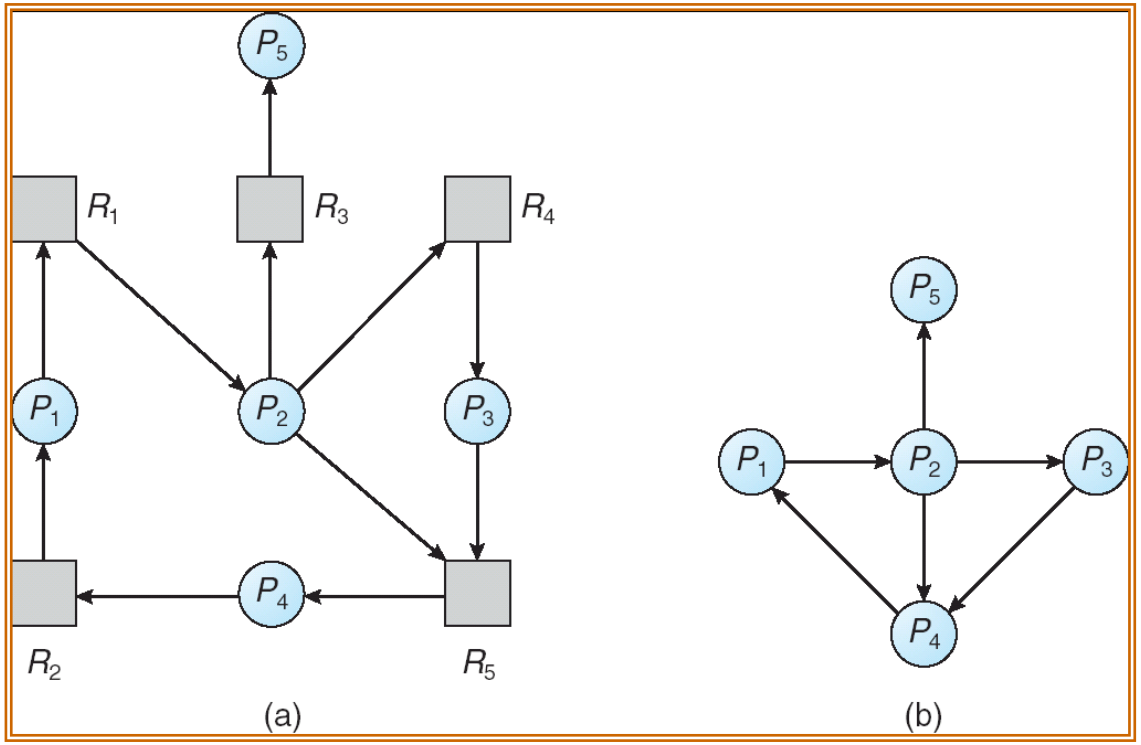
- Прашања:
  - Како да препознаеме дека се случува блокада?
  - Потребен е формален алгоритам (детектирање циклуси во ориентиран граф, матричен алгоритам)
- Како да ја одбереме жртвата?
  - Да се препознае кој од процесите се блокира...

Една инстанца од секој ресурс

- Направи *wait-for* граф на чекање
  - Јазлите се процеси.
  - $P_i \rightarrow P_j$ ако P_i го чека P_j.
- Периодично стартувај алгоритам кој бара циклуси низ графот. Ако има циклус, има блокада.
- Алгоритам кој детектира циклус во граф има комплексност од n^2 операции, каде n е бројот на лакови во графот

Граф на алокација и кореспондирачки граф на чекање

- Пример:



граф на алокација на ресурси

кореспондирачки wait-for граф

4. Не прави ништо!

- Наједноставен пристап
- Алгоритам на ној (глава во песок)
  - Колку често се јавува блокадата?
  - Математичар наспроти инженер!
- Многу системи потенцијално може да страдаат од блокади
- Во UNIX
  - потенцијално страда од блокади што не се ни детектираат
  - Табела на процеси е конечна величина
  - Ако fork паѓа бидејќи табелата е полна, треба да се почека случајно време и да се обиде пак

Баланс – точност и удобност

- Во секој ОС табелите се конечни ресурси и тоа води кон проблеми
  - Табели на процеси, конци,
  - Отворени датотеки
  - Swap простор на дискот
- Веројатно секој корисник преферира повремена блокада отколку низа ограничувања во користењето на ресурсите баланс помеѓу точноста и удобноста

Windows 11 - Deadlocks

- **Resource Allocation Graph**: Windows 11 uses the Resource Allocation Graph algorithm to detect deadlocks. This algorithm represents the resource allocation relationship among processes as a directed graph, and checks for cycles in the graph. If a cycle exists, it indicates a deadlock.
- **Timeout Mechanism**: Windows 11 employs a timeout mechanism to prevent deadlocks from occurring. **If a process is unable to acquire a resource within a specified time period, it releases all the resources it has acquired and goes into a wait state**. This mechanism ensures that a process does not hold resources indefinitely.
  - the timeout mechanism in Windows is implemented by setting a maximum wait time for processes to acquire resources and checking for cycles in the wait-for graph to detect deadlocks.

Windows 11 - Deadlocks

- **Preemption:** Windows 11 uses preemption to resolve deadlocks. If a process holds a resource and is unable to acquire another resource, the operating system may preempt the resource and allocate it to another process. This mechanism ensures that resources are used efficiently and prevents deadlocks from occurring.
- **Deadlock Detection and Recovery**: Windows 11 has built-in mechanisms to detect and recover from deadlocks. When a deadlock is detected, the operating system may terminate one or more processes involved in the deadlock to break the cycle and recover from the deadlock.
  - Random Process Termination, Priority-Based Process Termination, Resource-Usage-Based Process Termination, Deadlock Avoidance (is implemented to prevent the deadlocks – BY NOT GRANTING THE RESOURCE

Linux - Deadlocks

- **Lock ordering**: The Linux kernel uses a strict lock ordering protocol to prevent deadlocks. Lock ordering means that locks are always acquired in a specific order, and released in the reverse order. This ensures that no process can acquire a lock that is held by another process, preventing circular dependencies and deadlocks.
- **Lockdep**: The Linux kernel includes a tool called "lockdep" that checks for potential deadlocks at runtime. Lockdep analyses the lock dependencies in the kernel code and reports any potential circular dependencies. This helps developers to identify and fix potential deadlocks before they occur.

Linux Deadlocks

- **RCU (Read-Copy-Update):** RCU is a synchronization mechanism used in the Linux kernel to allow concurrent read access to shared data structures without the need for locks. RCU uses a deferred deletion mechanism to remove data structures (used in updates), ensuring that no process can access a deleted data structure.
- **Wait-for graph analysis**: The Linux kernel uses a wait-for graph to detect deadlocks. The wait-for graph is a directed graph that represents the dependencies between processes and the resources they require. When a process is waiting for a resource, it is added as a node to the wait-for graph. If a process is waiting for another process that is also waiting for a resource, a cycle is detected and a deadlock is assumed. In this case, the kernel takes action to resolve the deadlock by either releasing the resources or terminating one of the processes involved.

Linux Deadlocks

- **Priority inheritance**: Priority inheritance is a mechanism used in the Linux kernel to prevent **priority inversion**, which can lead to deadlocks. Priority inheritance ensures that the priority of a low-priority process is temporarily increased to the priority of the highest-priority process that is waiting for a resource held by the low-priority process. This prevents the high-priority process from being blocked and potentially causing a deadlock.

Заклучок

- Блокадите се потенцијален проблем за секој систем
- Може да се избегнуваат со водење сметка кои состојби се сигурни (постои низа од настани кои гарантираат дека сите процеси ќе завршат), а кои не се
- Банкаровиот алгоритам избегнува блокади е алгоритам за одбегнување блокади

Прашања?