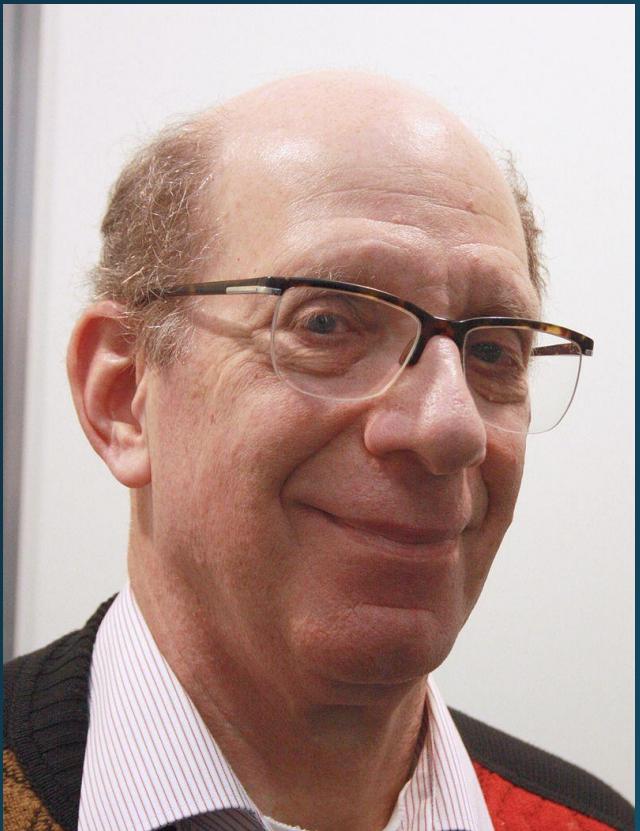


Лекция 2

*Прекъсвания
Процеси
Deadlock – мъртва хватка*

- Andrew S. Tanenbaum William Stallings



Прекъсвания

- Прекъсвания на нормалната последователност на процесора
- Целта е да се подобри работата на процесора
 - Повечето Вх/Изх. Устройства са по-бавни от процесора
 - Процесора спира за да изчака вх/изх устройства

Прекъсват нормалната последователност на изпълнение на команди от процесора.

Прекъсването предава управлението на функцията за обработване на прекъсването.

Прекъсването на обработваното задание става по такъв начин, че да е възможно възстановяване на неговата обработка.

Операционната система запазва състоянието на процесора като запазва регистрите, програмния брояч и т.н.

Определя какъв е видът на прекъсването и къде точно трябва да се предаде управлението за да се обработи възникналото прекъсване

Параметри на системите за прекъсване: Максималното време за обслужване на прекъсването се състои от:

Latency time - това е т.н. *Interrupt Latency time* - максималното време, чрез което прекъсванията са забранени, плюс времето, необходимо за стартиране на програмата за обслужване на прекъсването;

Response time - *Interrupt Response time* - това е времето, включващо *Interrupt Latency time*, плюс времето за запазване на контекста, както и времето, необходимо на ядрото да запише прекъсването (за някои от операционните системи - ОС);

Recovery time - *Interrupt Recovery time* - това е времето за възстановяване на контекста и за рестартиране на задачата, която е била прекъсната. За някои ОС трябва да се добави и времето, необходимо на оператора да реши коя е следващата задача, която да се стартира.

Апаратни и програмни прекъсвания.

Приоритети при прекъсванията.

Маскиране на прекъсванията.

Определяне на допустимия момент за прекъсване.

В компютърните системи (КС) прекъсванията на изпълняваните програми са два основни вида: *апаратни (hardware) прекъсвания* и *програмни (software) прекъсвания*:

Апаратни (hardware) прекъсвания: те са предвидени за определени аппаратни компоненти и предизвикват програмни прекъсвания. Аппаратните прекъсвания са два класа: от процесора - вътрешни и от периферията - външни;

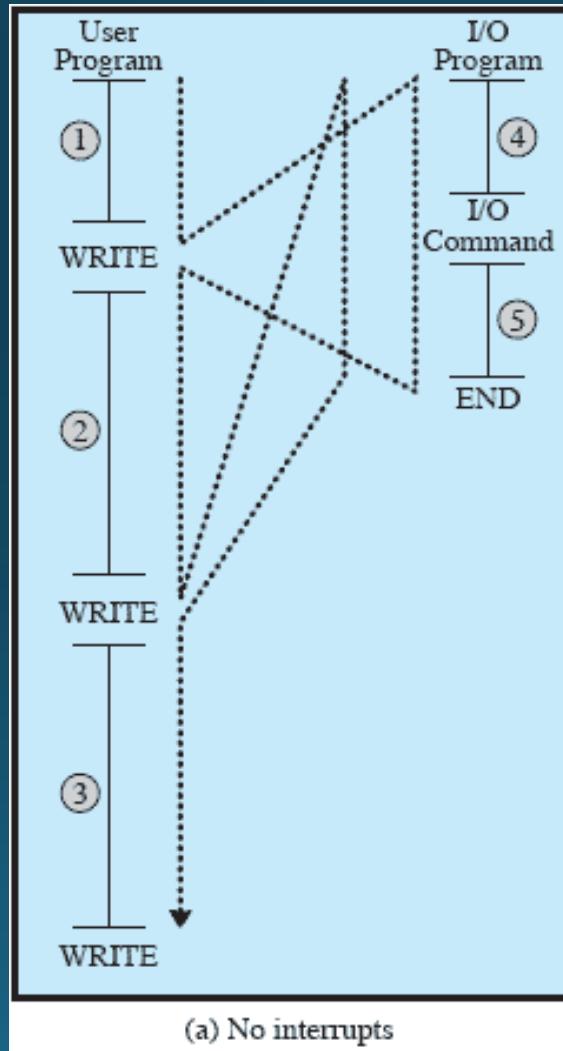
Програмни (software) прекъсвания: те са предвидени за реализиране на определени функции - прекъсване на BIOS . Тези прекъсвания не са присвоени на определени системни компоненти, а на определени функции. Например, клавишите *Ctrl* и *Break* едновременно активирани, създават прекъсване под код 23_{16}

Common Classes of Interrupts

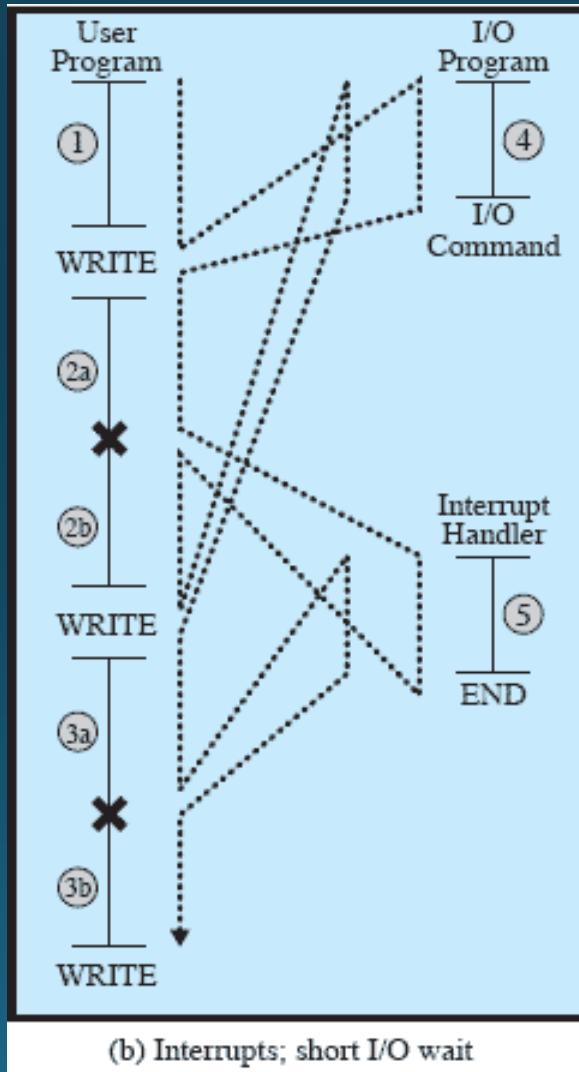
Table 1.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

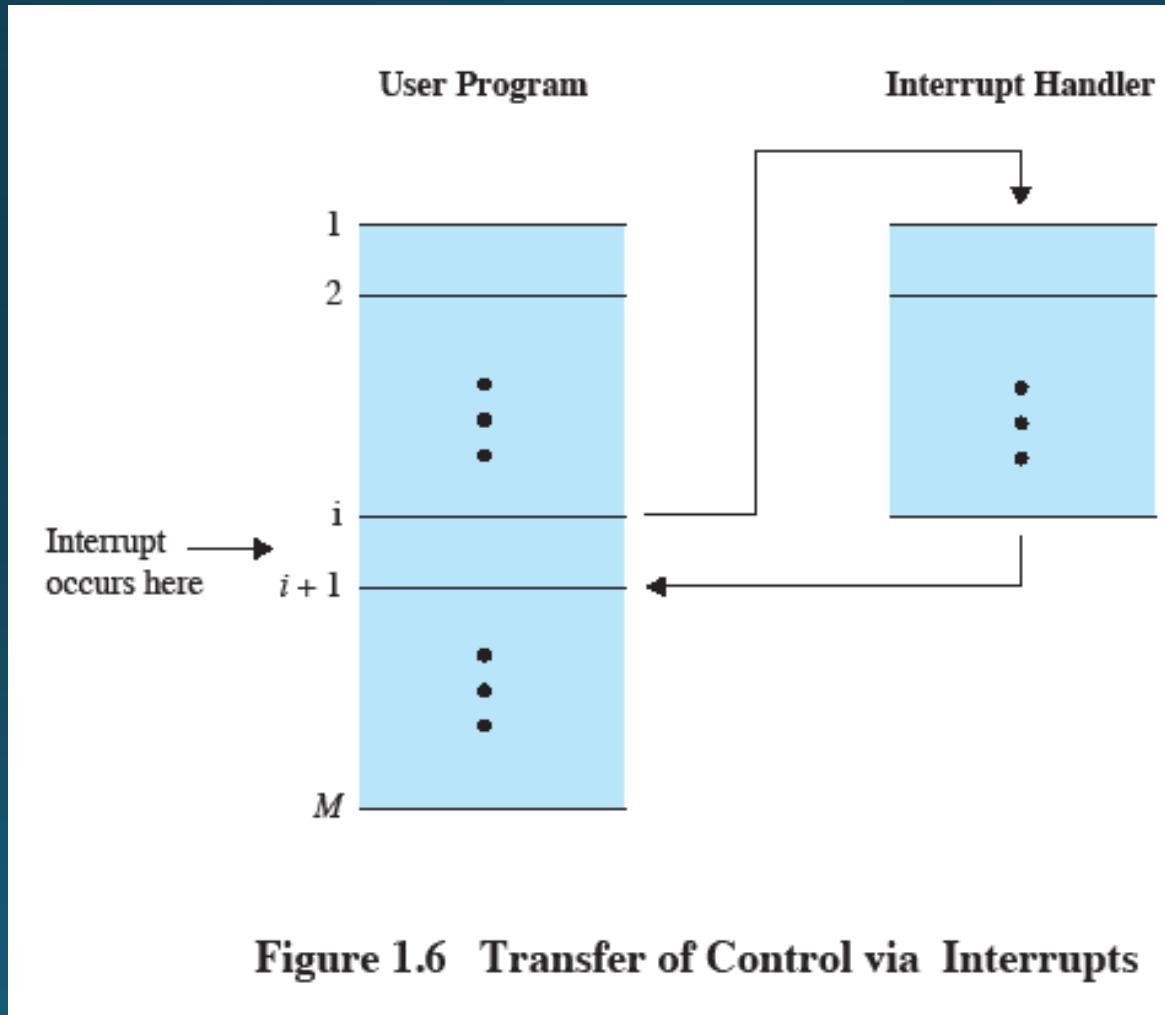
Процеси без прекъsvания



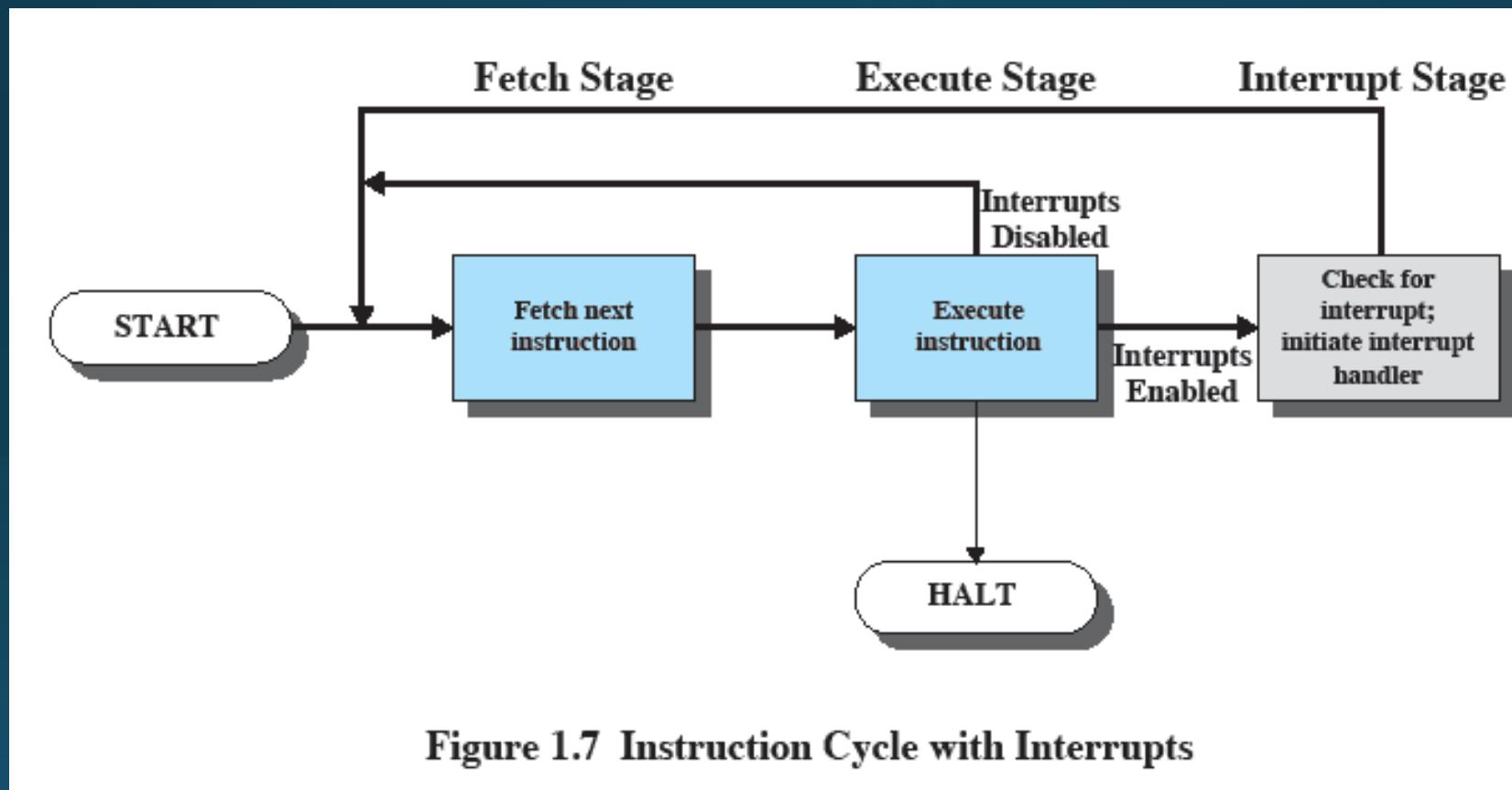
Прекъсвания и цикъл на инструкциите



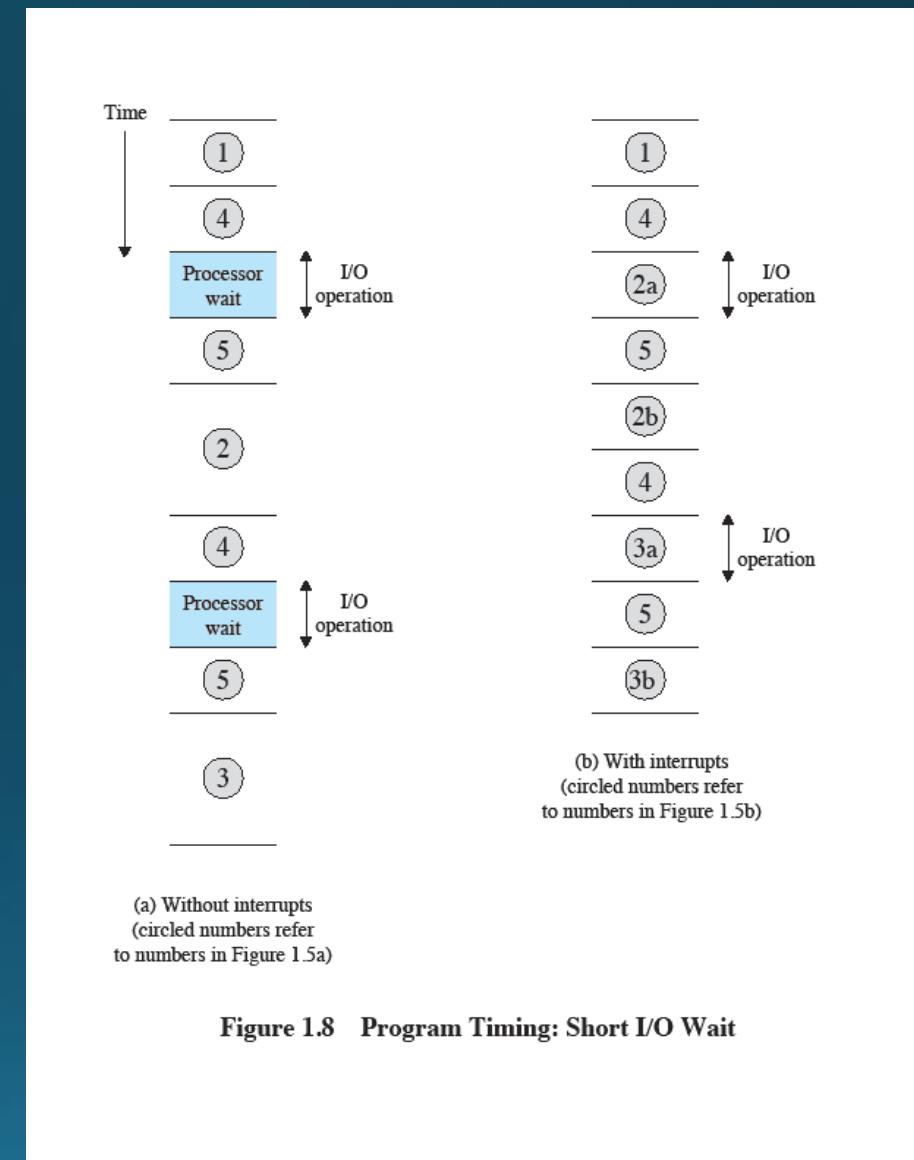
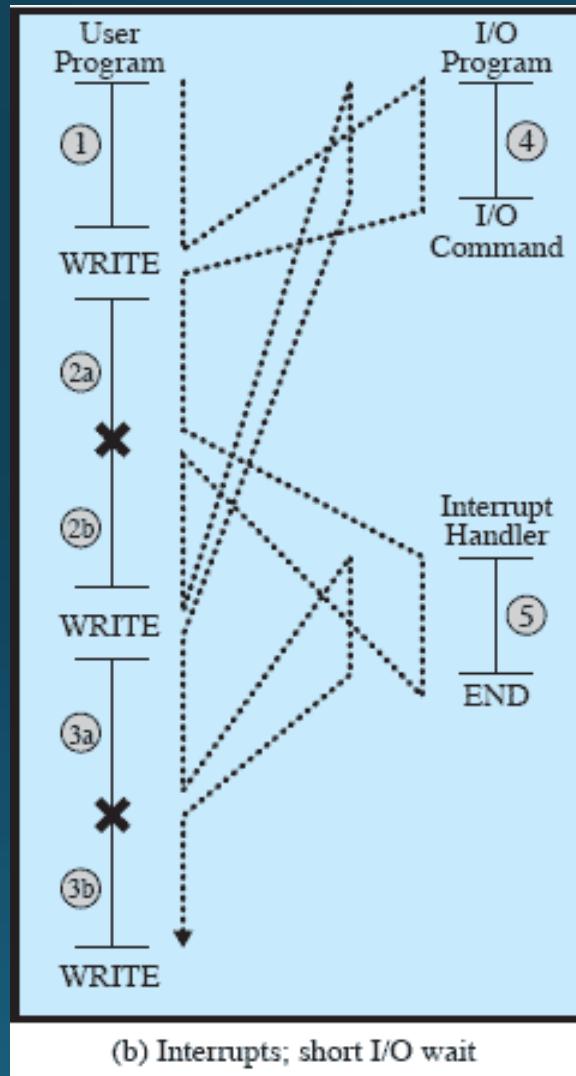
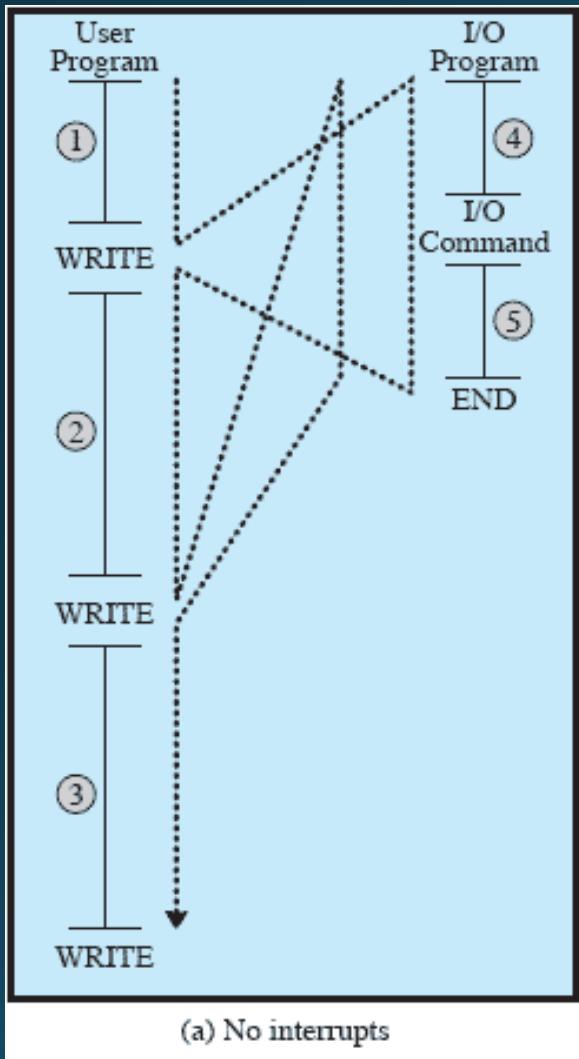
Transfer of Control via Interrupts



Цикъл на инструкции с прекъсвания



Късо I/O изчакване



Дълго I/O изчакване

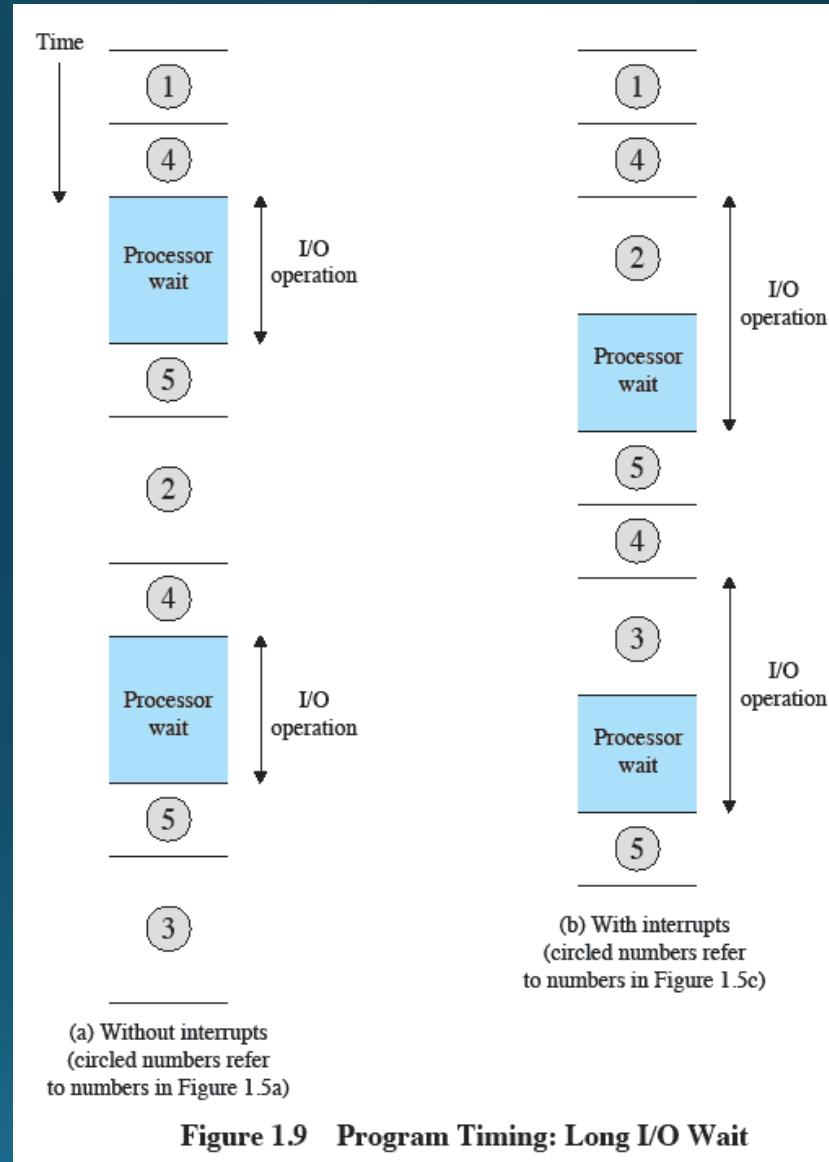
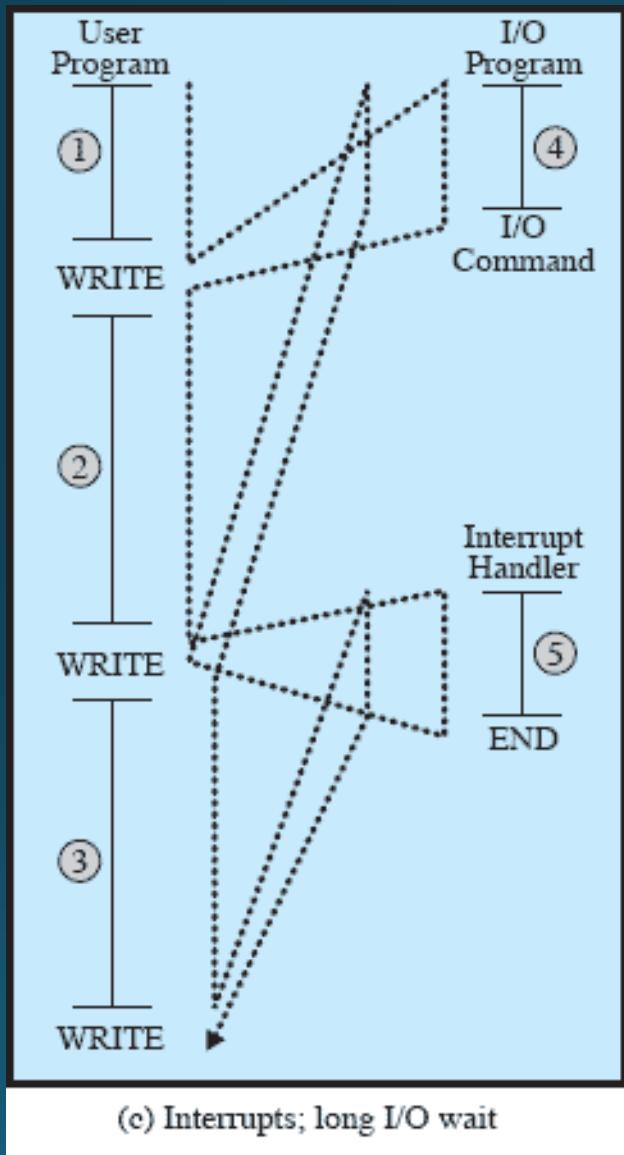
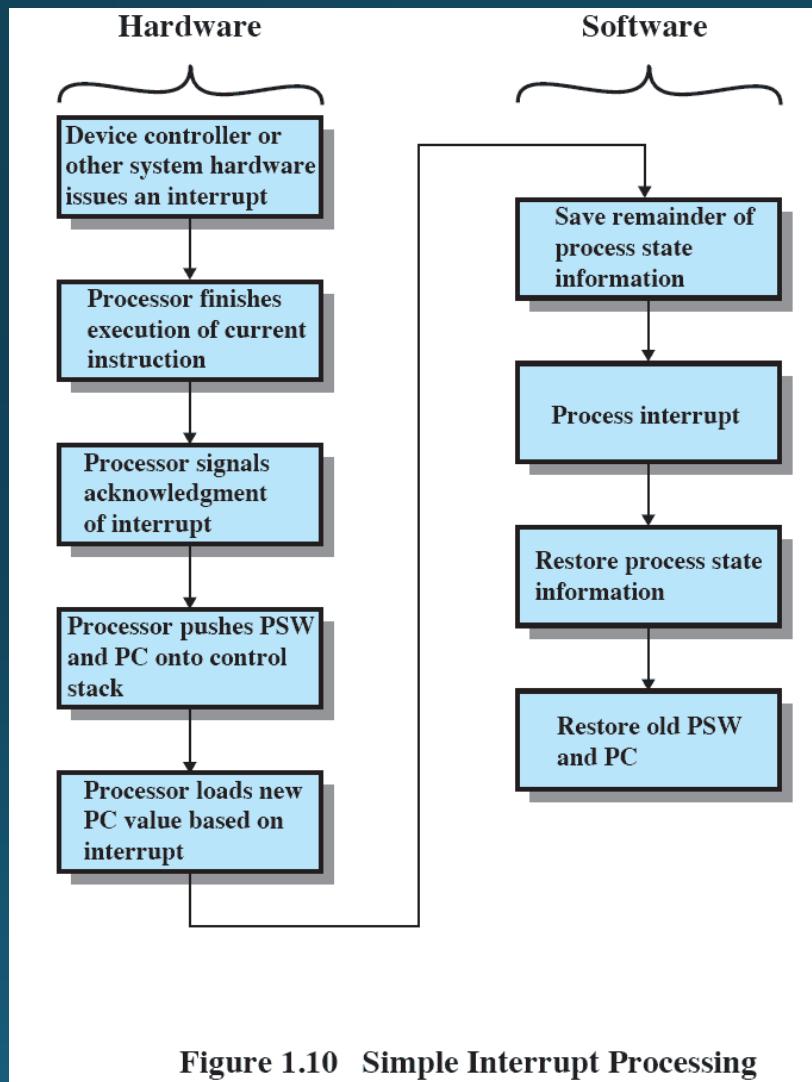


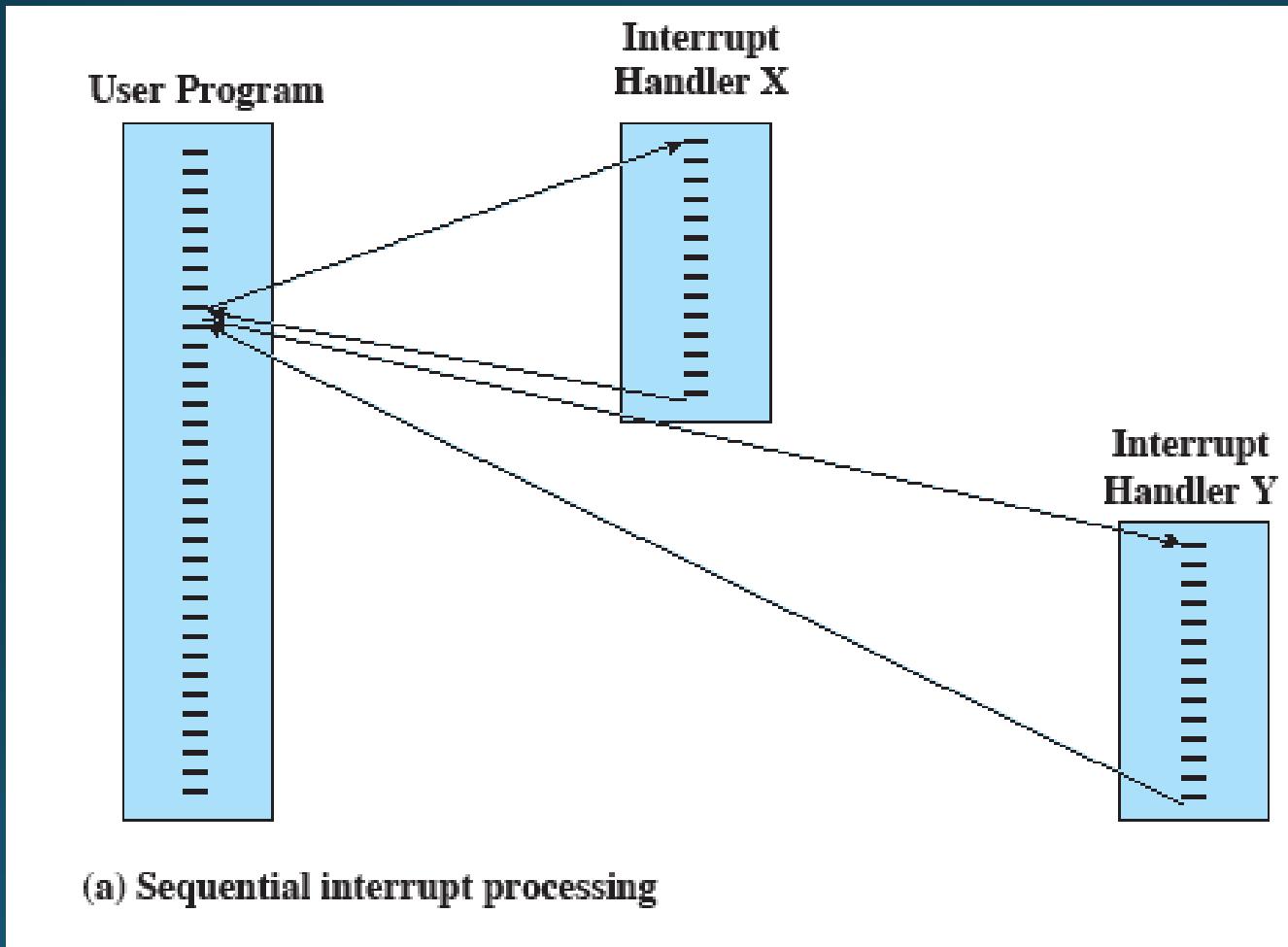
Figure 1.9 Program Timing: Long I/O Wait

Просто прекъсване

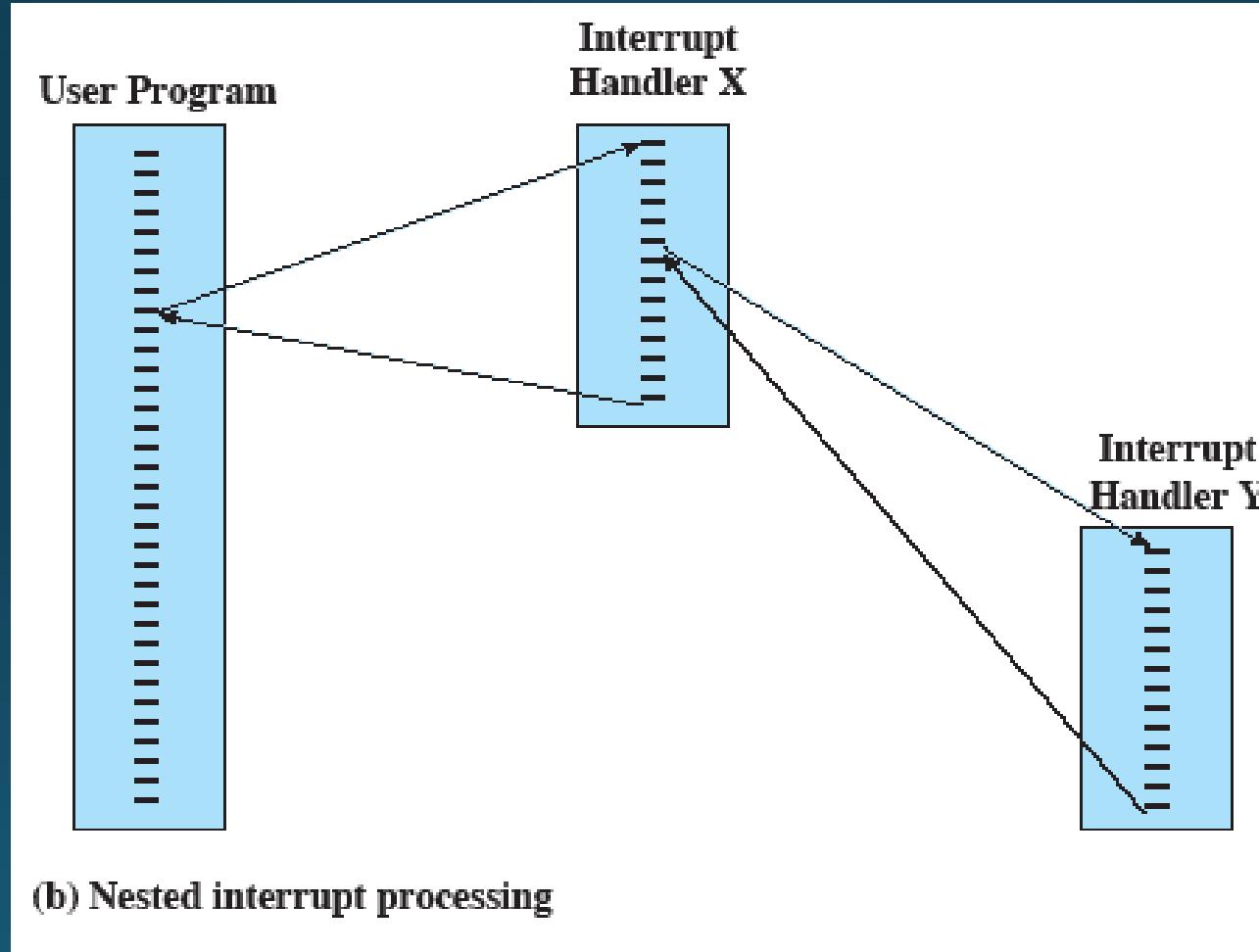


Множествени прекъsvания

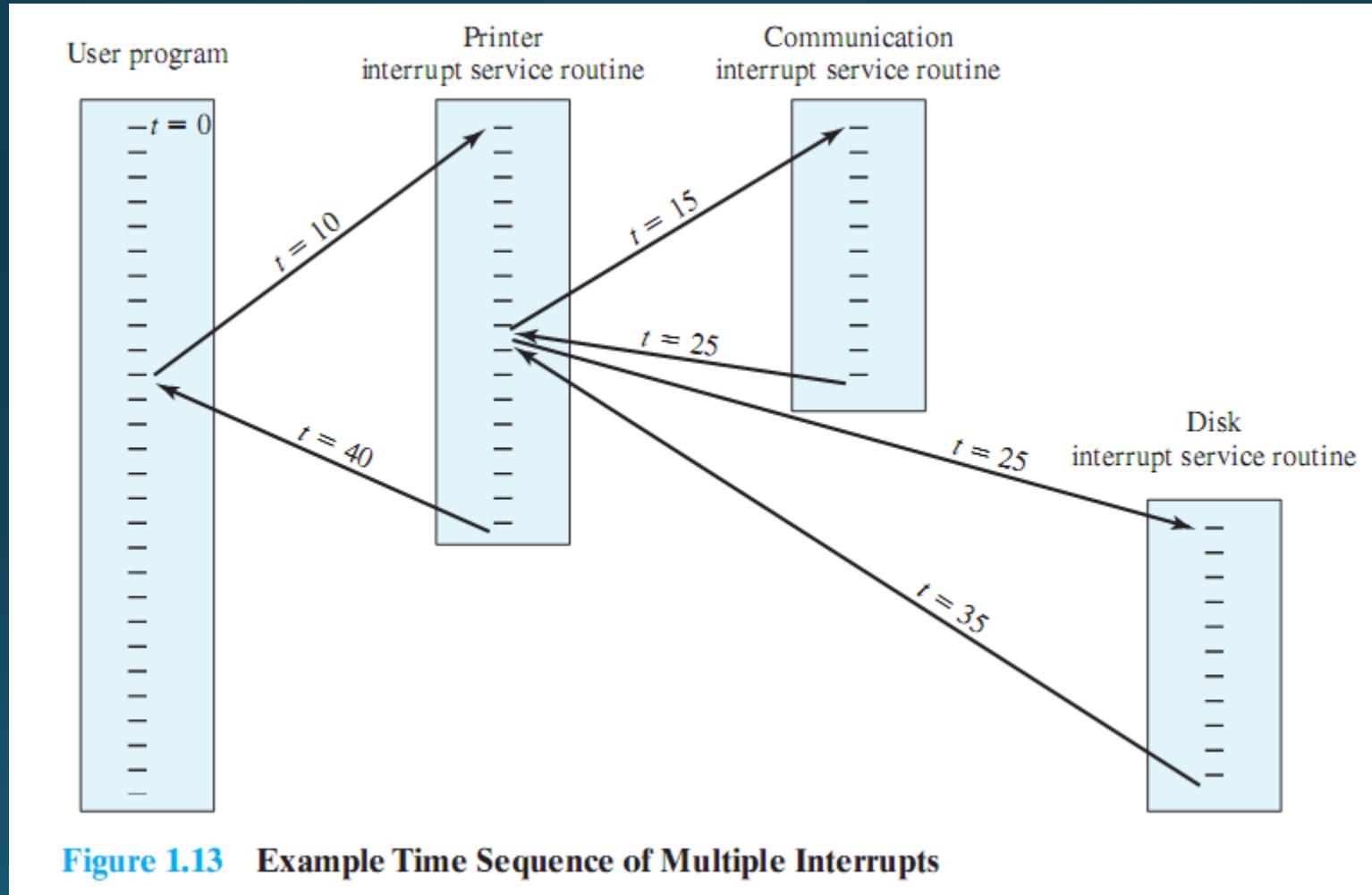
Последователни



Вложена обработка с прекъсвания



Пример за вложени прекъсвания



ПРОЦЕСИ

ПРОЦЕСИ

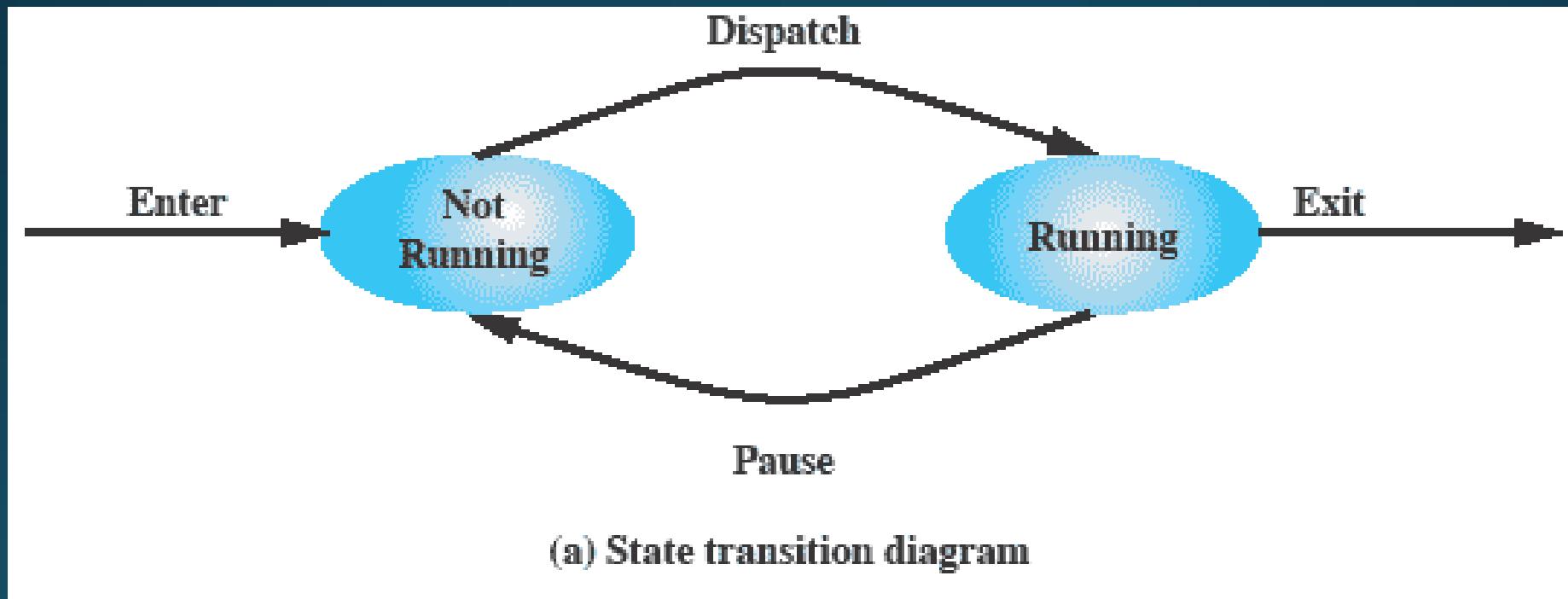
- Фундамент в структурата на ОС
- Процеса е:
 - Програма в режим на изпълнение
 - Инстанция на изпълнима програма
 - Поредна нишка на изпълнение, текущо състояние, както и свързаните с тях набор на системните ресурси .

Компоненти на процеса

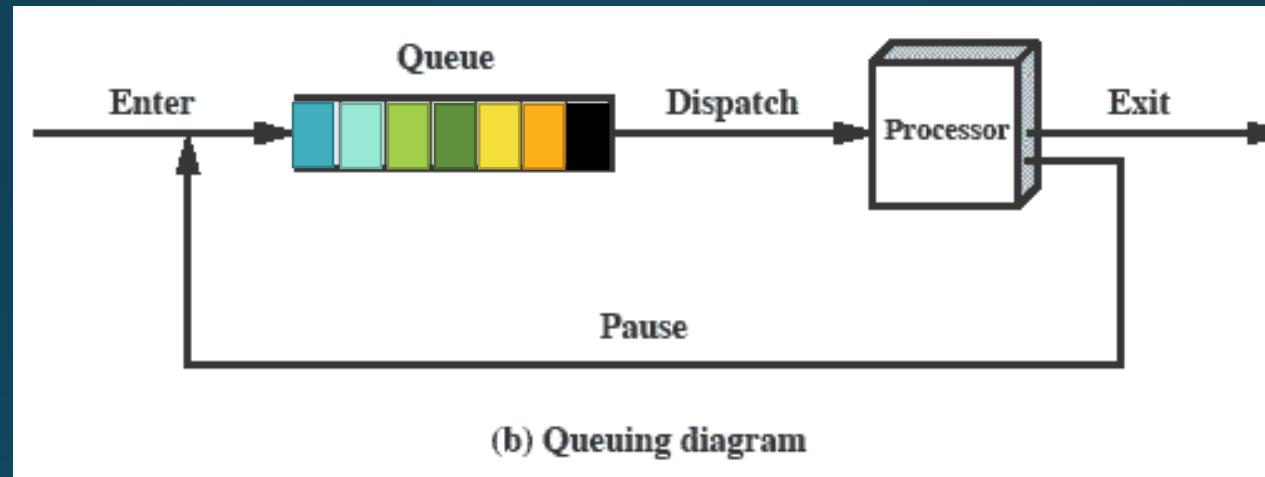
- Процесът съдържа :
 - Изпълнима програма
 - Асоциирани данни към изпълнимата програма
 - Изпълним контекст в програмата
- Изпълнимия контекст съдържа цялата информация необходима за управление на процесите от ОС

Two-State Process Model

- Състояния
 - Running
 - Not-running

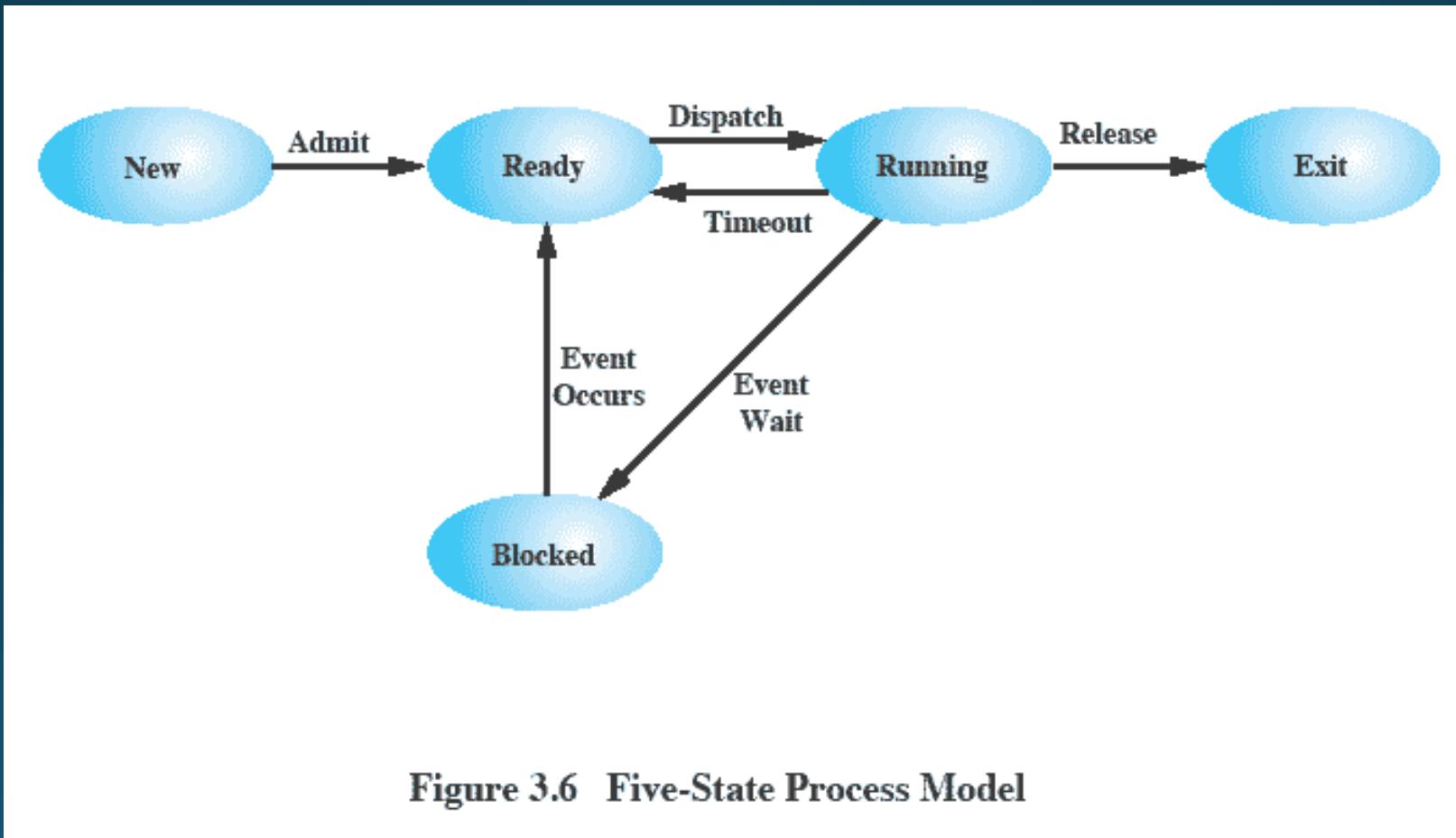


Queuing Diagram

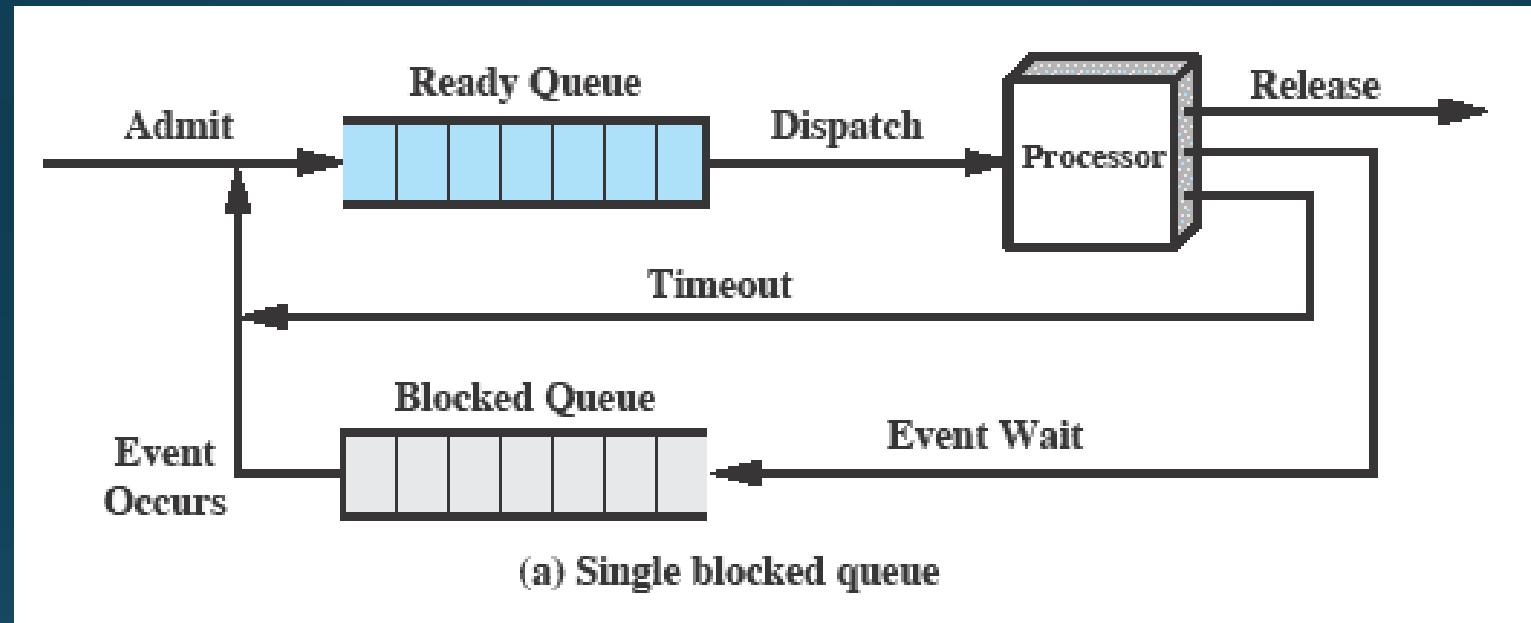


Etc ... processes moved by the dispatcher of the OS to the CPU then back to the queue until the task is completed

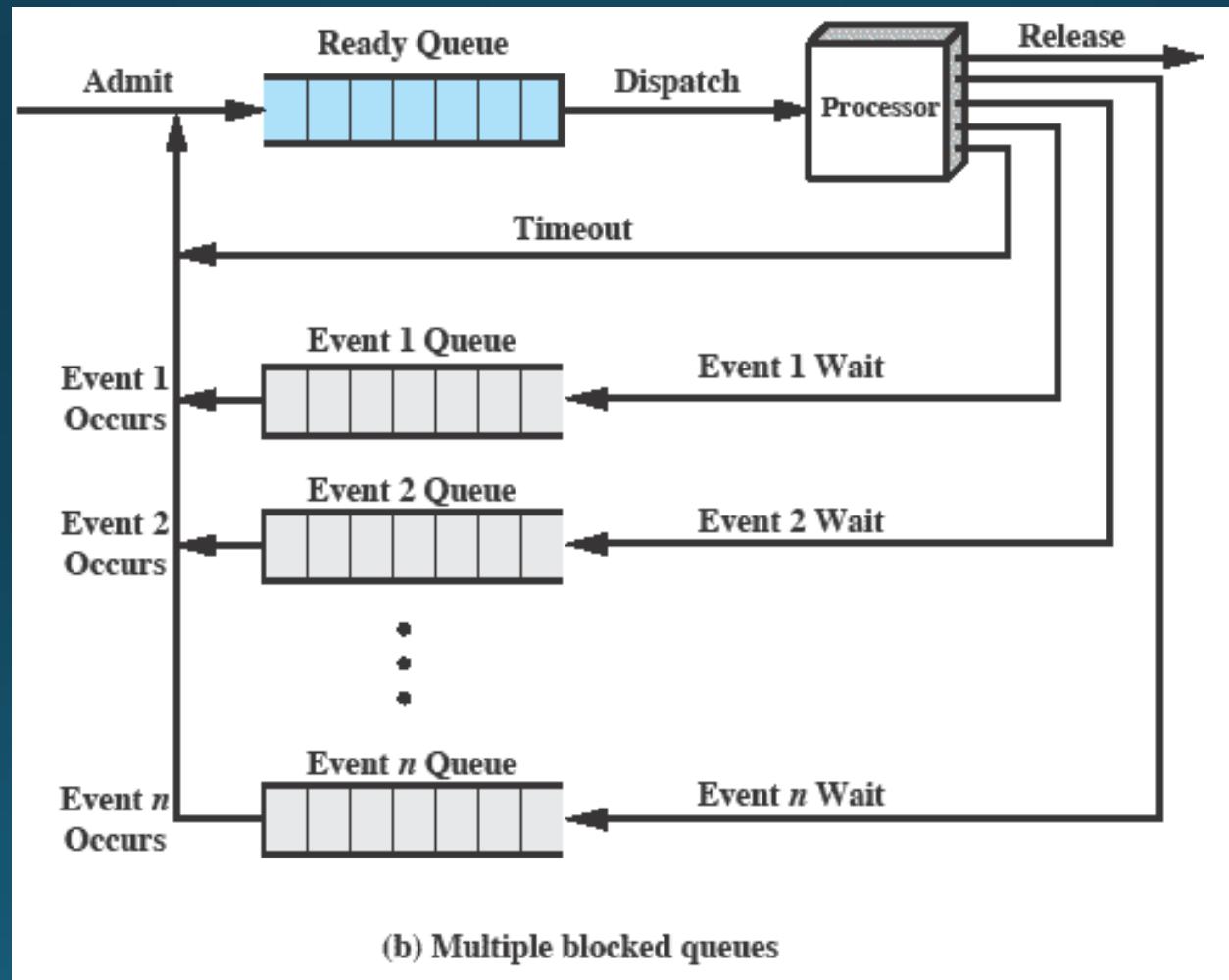
Five-State Process Model



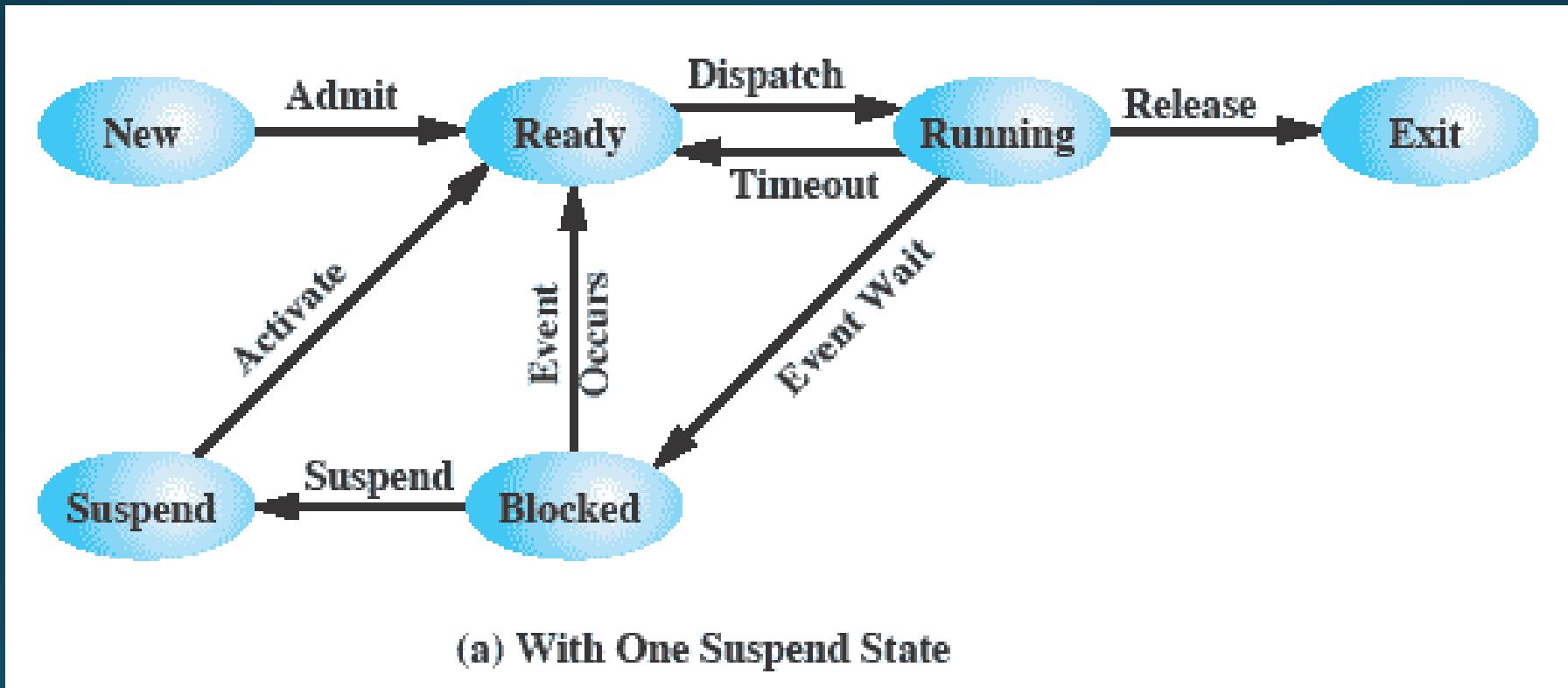
Използване на 2 опашки



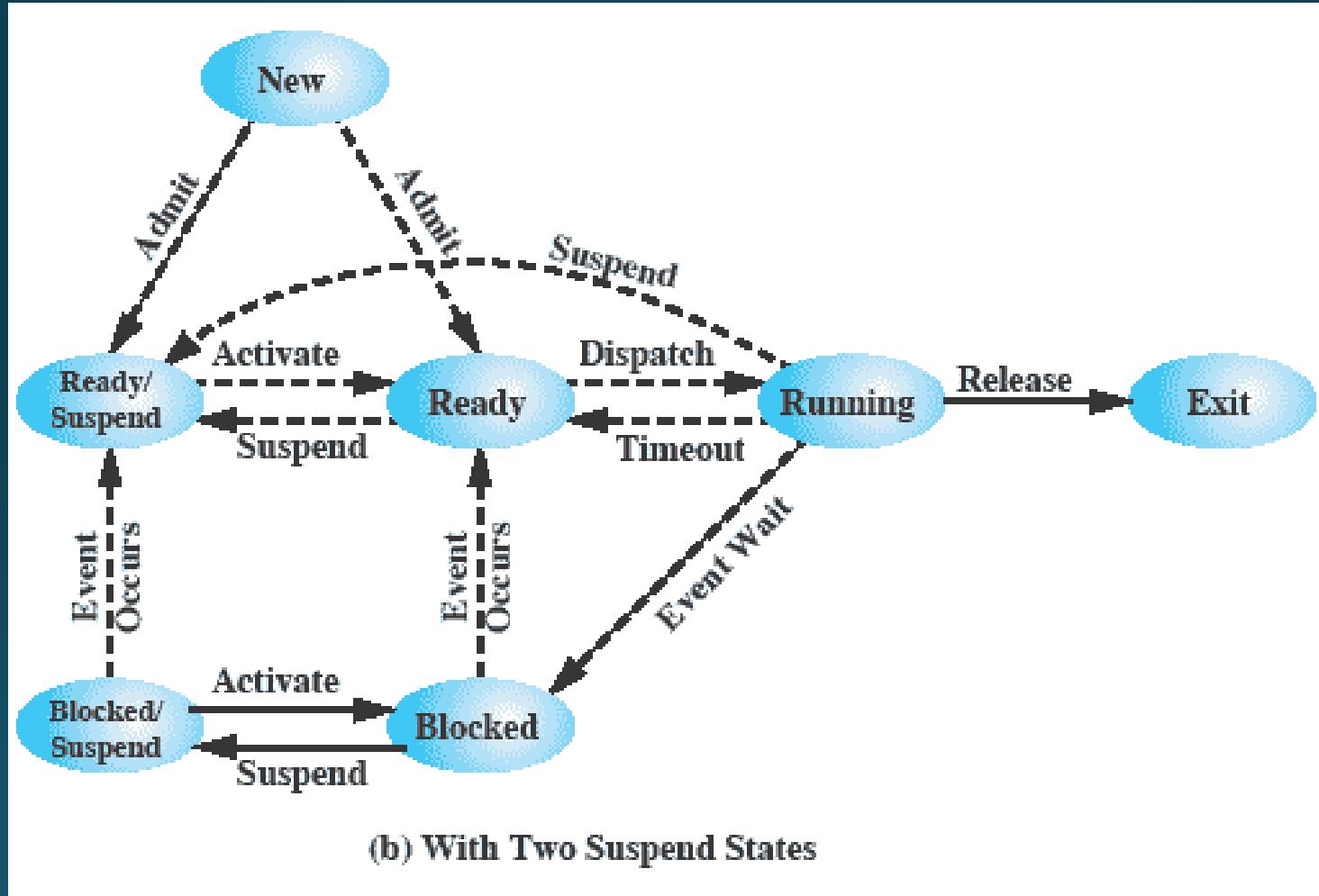
Multiple Blocked Queues



One Suspend State



Two Suspend States



Processes and Resources

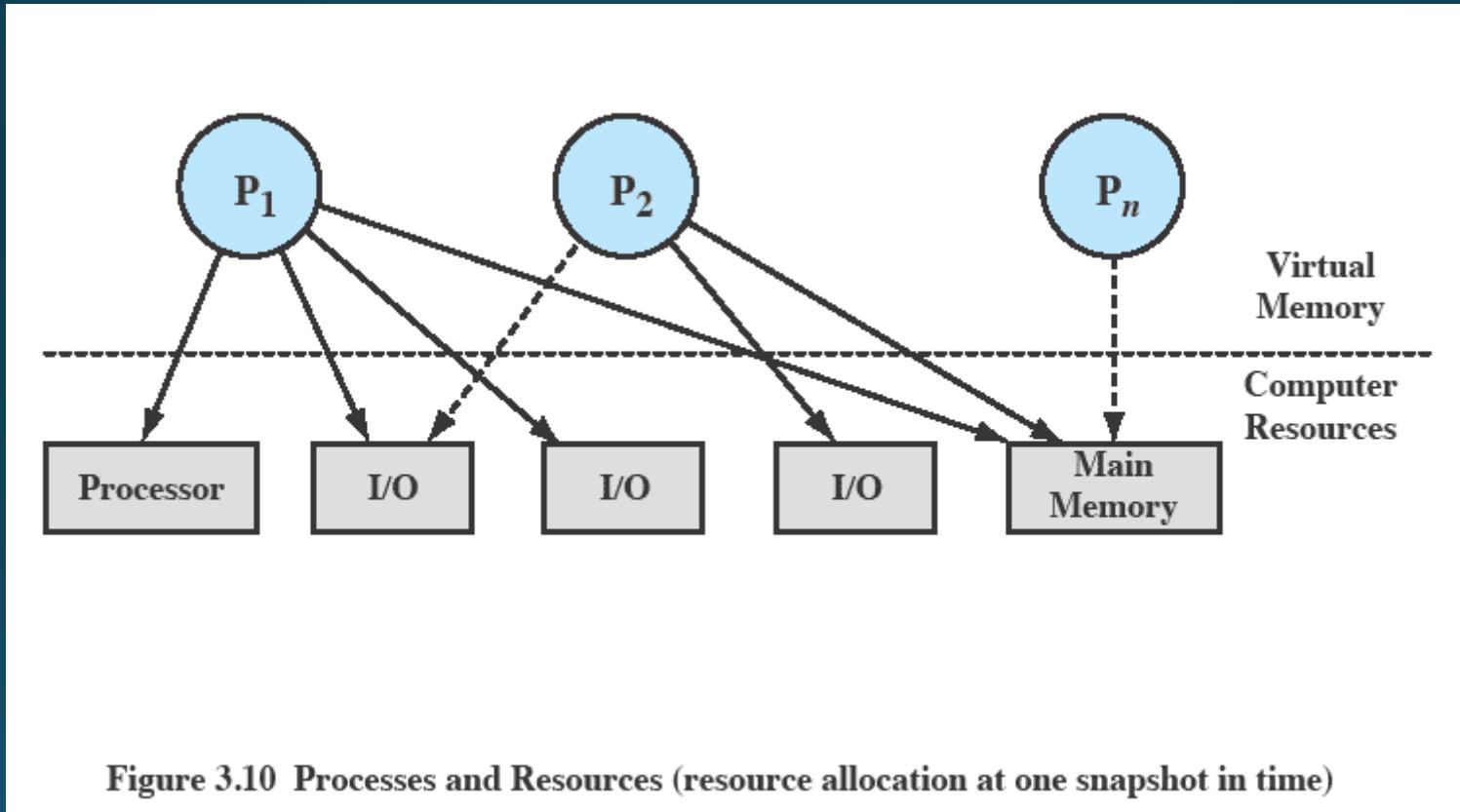
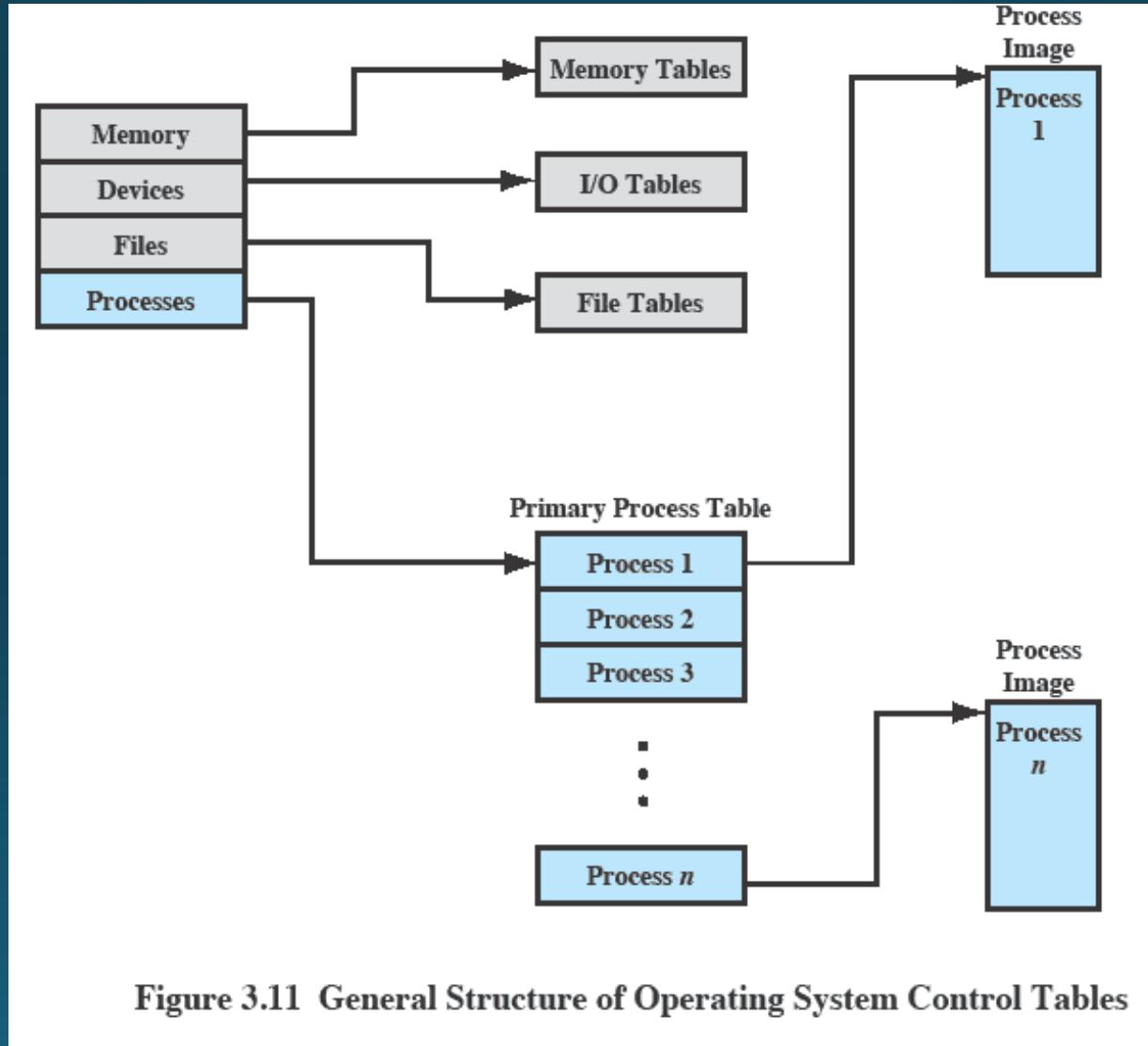


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

OS Control Tables



Атрибути на ПРОЦЕСИТЕ

- Идентификация на процеса;
- Състояние на процеса;
- Контролна информация на процеса;

Идентификация

Всеки процес има идентификационен номер

ОС използват тези идентификатори за управление и включването им в различни таблици за рефериране и управление на процесите

Информация състояние на процесите

- Регистри
 - User-visible registers
 - Control and status registers
 - Stack pointers
- Program status word (PSW)
 - Contains status information

Контролна информация на процесите

Допълнителна информация необходима за ОС да контролира и управлява активните процеси

Начин на изпълнение

- Most processors support at least two modes of execution
 - User mode
 - Less-privileged mode
 - User programs typically execute in this mode
 - System mode
 - More-privileged mode
 - Kernel of the operating system

Process Creation

- ОС създава нови процеси :
 - Задава уникален идентификатор процес;
 - Заделя , установява пространство за процеса;
 - Инициализира контролния блок;
 - Задава подходящи линкове ;
 - Създава или разширяване други структури от данни;

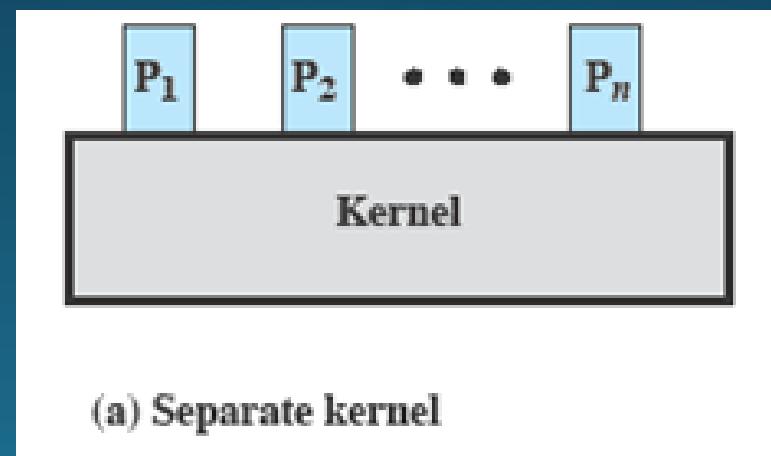
Процес ли е ОС?

- Ако ОС е сбор от програми и ако те се изпълняват от процесора – това означава ли че ОС е процес?
- Ако е така – как се контролира?
 - Кой и как контролира това?

(въпрос за размисъл ☺)

Non-process Kernel

- Процес изпълняван извън ядрото
- Концепциите на процеса се счита, че се прилагат само за потребителски програми
- Кода на ОС се изпълнява в отделен привилегирован режим



Мъртва хватка - Deadlock

Съдържание

→ Принципи на мъртвата-хватка (Deadlock)

- Предотвратяване на Deadlock
- Избягване на Deadlock
- Откриване на Deadlock
- Интегрирана Deadlock стратегия

Мъртва хватка- deadlock

Дефиниция – изпълнението на процесите е в безизходица

Видове според поделяния ресурс :

- Физически deadlock
- Логически deadlock

Класически пример

Процес А

Заяви Р1

Заяви Р2

Процес Б

Заяви Р2

Заяви Р1

Условия за възникване

- Взаимно изключване – ресурсът се използва от един процес в даден момент
- Очакване на ресурс – процесите държат разпределението им ресурси като очакват следващите
- Липса на преразпределение – ресурсите се освобождават само от процеса, който ги притежава, след като е изпълнил задачата си

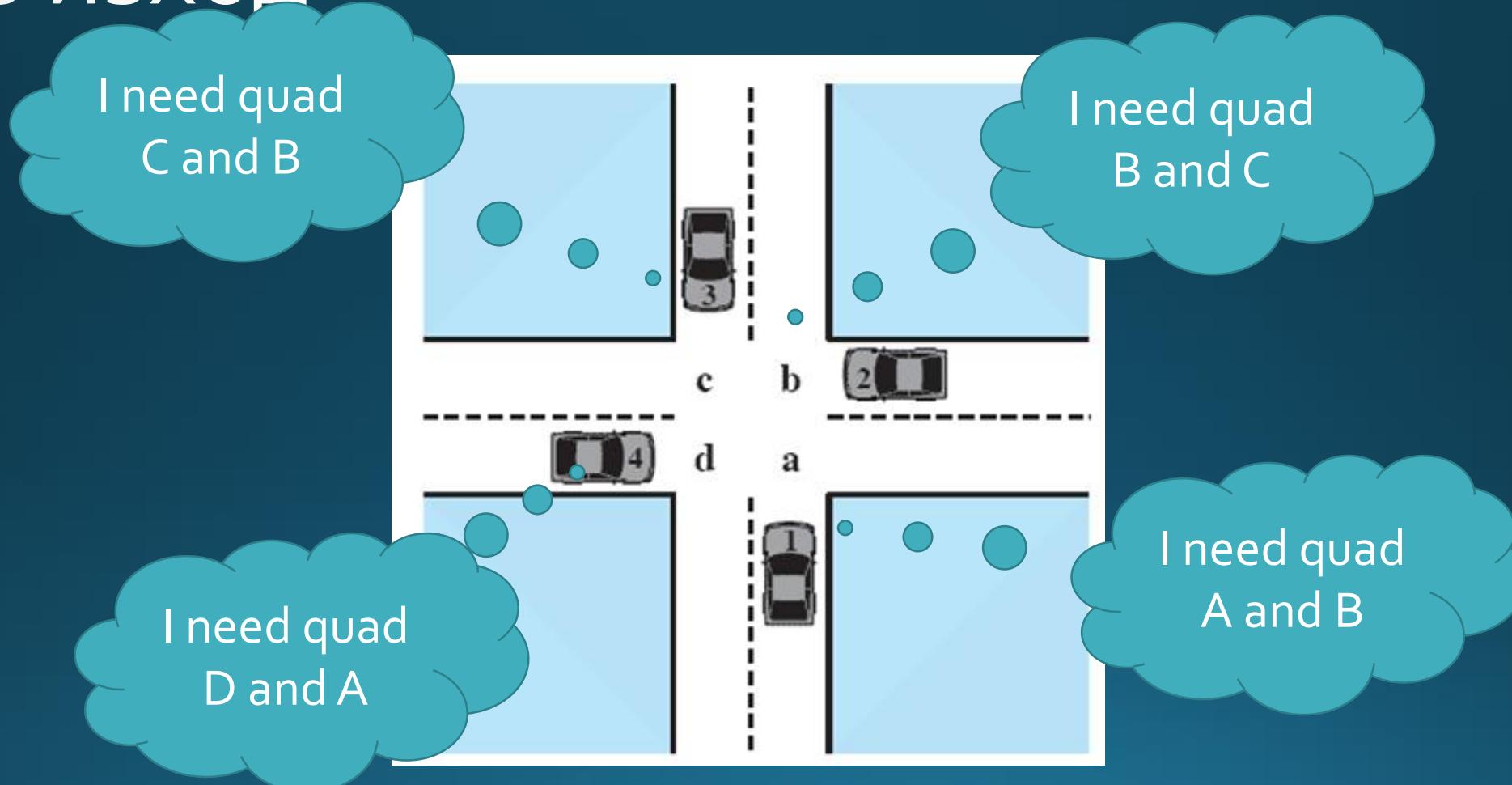
Стратегии за избягване на МХ

- Стратегия на предпазване
- Стратегия на избягване
- Стратегия на разпознаване

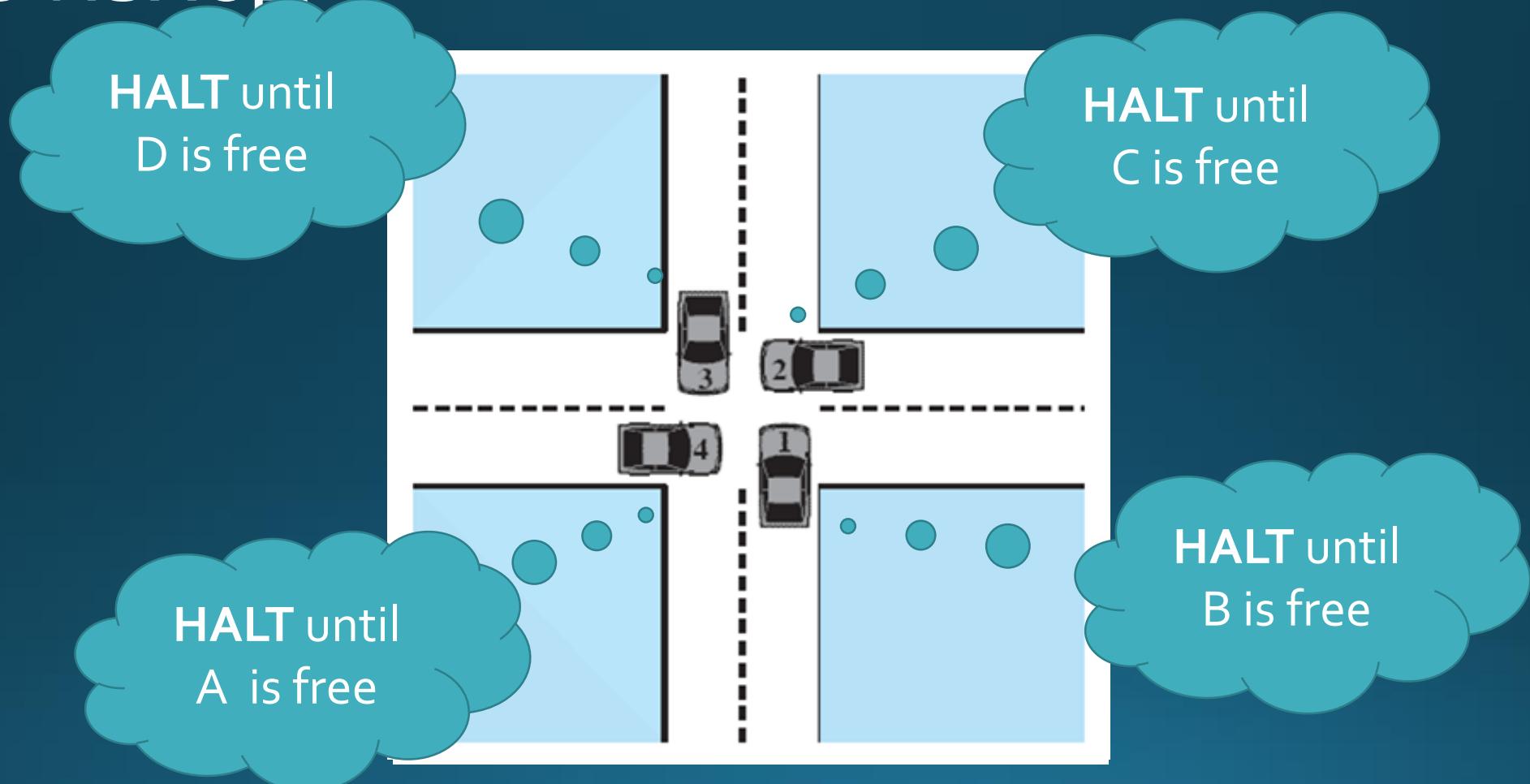
Deadlock

- Множество от процеси е блокирано без изход, когато всеки процес е блокиран в очакване на събитие, което може да се инициира само от друг блокиран процес в множеството
 - Обикновено включва процеси, които се състезават за достъп до едно и също множество ресурси

Потенциално блокиране без изход



Действително блокиране без изход

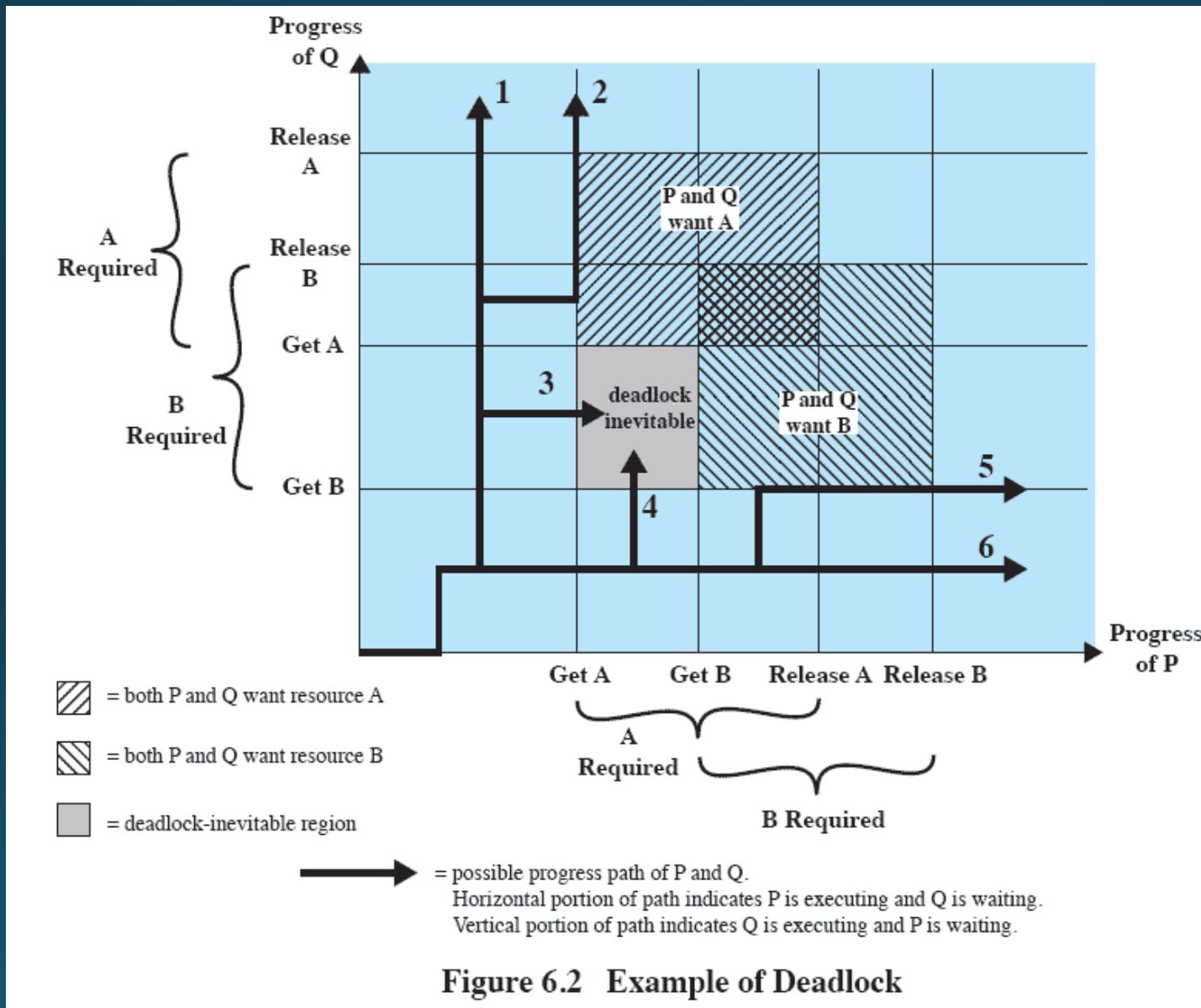


Два процеса P и Q

- Нека разгледаме двата процеса P и Q
- Всеки от тях изключително се нуждае от достъп до ресурси A и B за определен период от време.

Process P	Process Q
...	...
Get A	Get B
...	...
Get B	Get A
...	...
Release A	Release B
...	...
Release B	Release A
...	...

Съвместна диаграма на Deadlock



Алтернативна логика

- Да предположим, че Р не се нуждае от двата ресурса по едно и също време.
- Двата процеса могат да имат следната форма =>

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Диаграма на алтернативната логика

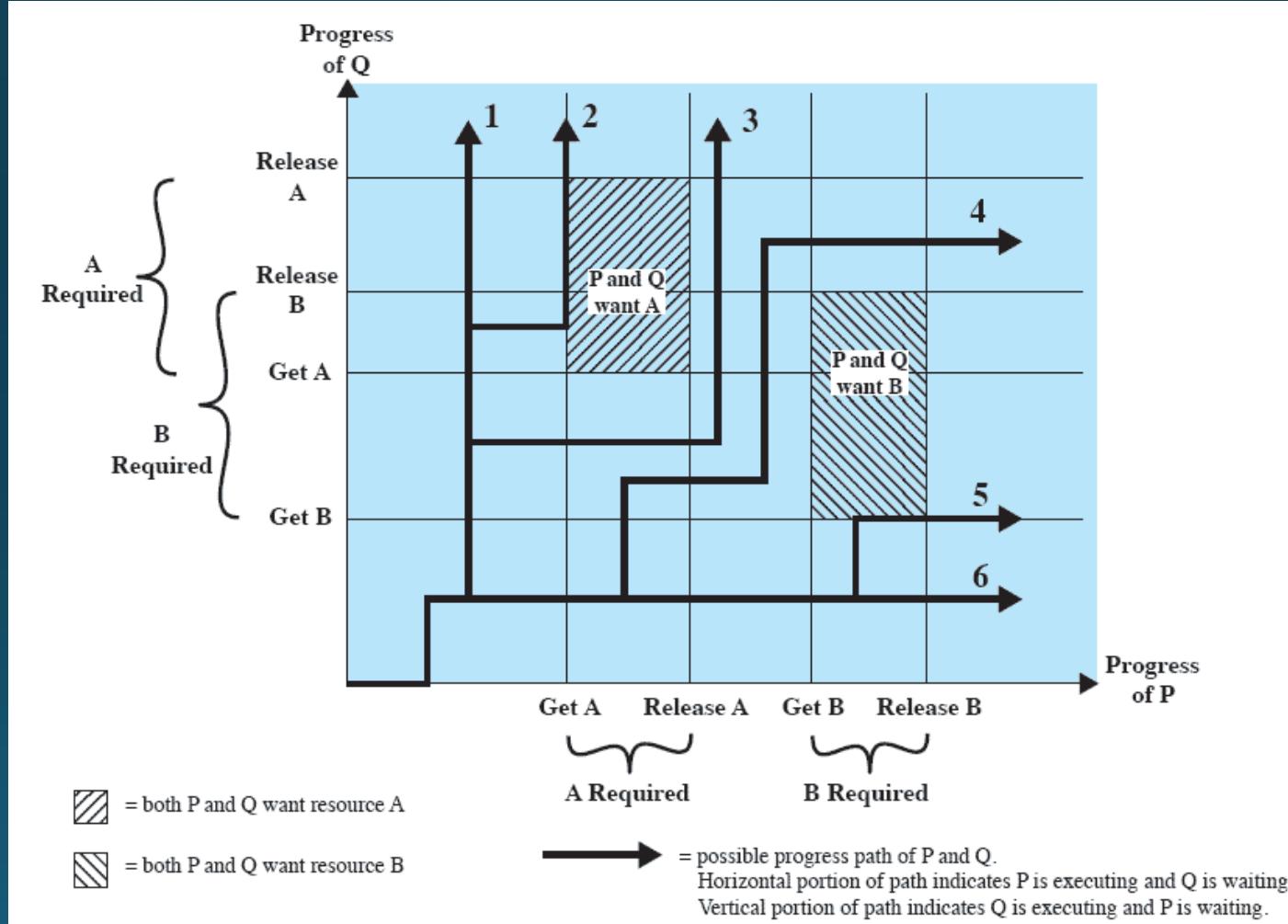


Figure 6.3 Example of No Deadlock [BACO03]

Категории ресурси

Има две основни категории ресурси:

- Преизползваеми
 - Могат безопасно да се използват само от един процес по едно и също време и не се изчерпват от тази употреба.
- Консумативни
 - Могат да бъдат създадени (*produced*) и унищожени (*consumed*).

Преизползваеми ресурси

- Примери:
 - Процесори, канали за вход/изход, първична и вторична памет, устройства, структури от данни като файлове, бази от данни и семафори
 - “Мъртва-хватка” се появява ако всеки процес държи един ресурс и изисква друг, за да продължи да се изпълнява.

Deadlock - Пример

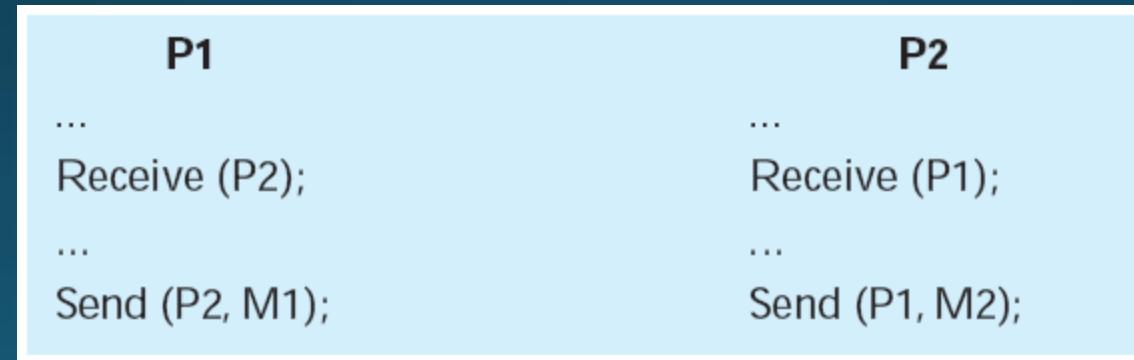
- Имаме два процеса, които се състезават за незабавен достъп до файл D и пишещо устройство T.
- Deadlock се появява, когато всеки процес държи единия ресурс и изисква достъп до другия.

Консумативни ресурси

- Примери: Прекъсвания, сигнали, съобщения, информация във входно/изходните буфери
- Deadlock може да възникне, ако получаването на съобщението е блокиращо
- Рядка комбинация от събития може да предизвика Deadlock

Пример за Deadlock

- Имаме двойка процеси, във всеки процес се опитваме да получим съобщение от другия процес и тогава да пратим съобщение.



Условия за възможен Deadlock

- Взаимно изключване
 - Само един процес може да използва даден ресурс в определено време
 - Hold-and-wait
 - Процес може да държи алокиран ресурс докато чака да получи други
- No pre-emption
 - Нито един ресурс на може насилиствено да бъде отнет от процеса, който оперира с него.

Действителният Deadlock се нуждае от ...

Всичките три предходни условия и:

- Циклично чакане
 - Затворена верига от процеси, така че всеки процес притежава най-малко един ресурс, необходим на следващия процес във веригата

Преодоляване на Deadlock

- Съществуват три подхода за справянето със “мъртвата хватка”.
 - Предотвратяване на deadlock
 - Избягване на deadlock
 - Откриване на Deadlock

Стратегия за предотвратяване на “мъртва хватка”

- Проектиране на системата по такъв начин, че възможността от възникване на deadlock да бъде изключена.
- Два основни метода
 - Индиректен – предотвратяване на трите необходими условия да се появят наведнъж
 - Директен – Предотвратяване на цикличното изчакване

Предотвратяване на Deadlock

Условия 1 & 2

- Взаимно изключване
 - Трябва да се поддържа от операционната система
- Hold and Wait
 - Изисква процесът да заяви всички необходими ресурси наведнъж

Предотвратяване на Deadlock

Условия 3 & 4

- No Preemption
 - Процесът трябва да освобождава ресурсите и да ги изисква отново
- Циклични изчаквания
 - Дефиниране на линейно подреждане на типовете на ресурсите

Съдържание

- Принципи на мъртвата хватка (Deadlock)
 - Предотвратяване на Deadlock
 - Избягване на Deadlock
 - Откриване на Deadlock
- Интегрирана Deadlock стратегия
- Проблемът “вечерящи философи”
- Механизми в UNIX, Linux, Solaris и Windows

Избягване на Deadlock

- Решението дали текущото заделяне на ресурс потенциално би довело до deadlock се прави динамично.
- Изисква информация за бъдещи заявки на процесите

Два подхода за избягване на Deadlock

- Отказ на стартирането на процес
 - Да не се стартира процес, ако неговите заявки могат да доведат до deadlock.
- Отказ при заделянето на ресурси
 - Да не се правят опити за заделяне на ресурси на един процес, ако това разпределение може да доведе до deadlock

Отказ при стартирането на процес

- Процес се стартира само ако максималните изисквания на всички текущи процеси плюс тези на новия могат да бъдат изпълнени.
- Не е оптимално
 - Предполагаме най-лошият вариант: Всички процеси ще достигнат максималните си изисквания.

Отказ при заделянето на ресурси

- Познат още като алгоритъм на банкерите
 - Стратегия на отказ за разпределение на ресурсите
- Разглеждаме система с определен брой ресурси:
 - *State* - Състоянието на системата е текущото разпределение на ресурсите за обработка
 - *Safe state* - Безопасно състояние е разпределението, при което има най-малко една последователност, която не води до deadlock
 - *Unsafe state* – състояние, което не е безопасно

Избягване на Deadlock

- Когато един процес, който отправя искане за група ресурси
 - Допускаме, че искането е одобрено,
 - Съответно актуализираме състоянието на системата
- Тогава се определя дали резултатът е безопасно състояние
 - Ако е така, заявката се изпълнява
 - Ако не е така, процесът се блокира докато състоянието не стане безопасно за изпълнение на заявката

Избягане на Deadlock: Предимства

- Не е необходимо да подменяме и отказваме процеси както при откриването на deadlock
- Не е толкова ограничително както предотвратяването на deadlock.

Избягане на Deadlock: Ограничения

- Максималните изисквания относно ресурсите тряба да се знаят предварително
- Процесите тряба да бъдат независими и без изисквания за синхронизация
- Трябва да има определен брой ресурси за разпределение
- Никой процес не може да приключи (exit), докато заема дадени ресурси

Откриване на Deadlock

- Стратегиите за предотвратяване на Deadlock са много консервативни
 - ограничаване на достъпа до ресурси и налагане на ограничения на процесите.
- Стратегиите за откриване на Deadlock правят точно обратното.
 - Изисканите ресурси се заделят, когато е възможно.
 - Редовно се прави проверка дали е възможно да възникне Deadlock

Обобщен алгоритъм за откриване

- Използваме матрицата на разпределение на ресурсите и вектора на свободните такива
- Използваме също и матрица на запитванията Q
 - където Q_{ij} показва, че ресурс j е поискан от процес I
- Първо размаркираме ('un-mark') всички процеси, които не се намират в deadlock.
 - Първоначално това са всички процеси

Стратегии за възстановяване след откриване на Deadlock

- Прекратяване на всички процеси, които се намират в Deadlock.
- Връщане назад (Back up) на всеки блокиран процес до някоя дефинирана предварително точка преди блокирането и рестартиране на всички процеси
 - Риск от повторение на deadlock
- Успешно излизане от блокирани процеси, докато deadlock ситуацията не бъде отстранена
- Успешно подменяне на ресурси докато deadlock ситуацията не бъде отстранена

UNIX Concurrency

Механизми

- UNIX осигурява разнообразие от механизми за комуникация и синхронизация на процеси чрез:
 - Pipes („тръби“ - канали)
 - Messages (съобщения)
 - Shared memory (споделена памет)
 - Semaphores (семафори)
 - Signals (сигнали)

Pipes

- Къгов буфер, позволяващ два процеса да си комуникират на база модела производител-потребител
 - FIFO опашка, в която единият процес записва данни, а другият чете.
- Два типа pipes:
 - Именовани
 - Неименовани

Съобщения

- Блок от байтове с придружаващ тип.
- UNIX предлага *msgsnd* и *msgrcv* системни извиквания за процесите така, че да ги включи в обмена на съобщения.
- Опашка от съобщения е обвързана с всеки процес, като тя функционира като пощенска кутия.

Споделена памет

- Общ блок от виртуалната памет се споделя от няколко процеса.
- Позволено е само четенето или четене и запис за един процес,
 - Определено на пред-процесно ниво.
- Ограниченията на взаимните изключвания не са част от употребата на споделената памет, но трябва да бъдат осигурени от процесите, които я използват.

Сигнали

- Програмен механизъм, който информира процеса за появата на асинхронни събития.
 - Подобно на хардуерните прекъсвания, без приоритети
- Сигнал се получава при обновяването на област от таблицата с процеси за процес, за който е изпратен сигналът.

Семафори

- Подобни на UNIX SVR4, но освен това се предлага реализация на семафори за собствена употреба.
- Три вида семафори на ядрото:
 - Двоични семафори (Binary semaphores)
 - Броящи семафори (counting semaphores),
 - “Четящи-пишещи” семафори (reader-writer semaphores).

Barriers (Бариери)

- За прилагане на реда, в който се изпълняват инструкциите, Linux предлага използването на бариери при употреба на паметта.

Table 6.6 Linux Memory Barrier Operations

rmb()	Prevents loads from being reordered across the barrier
wmb()	Prevents stores from being reordered across the barrier
mb()	Prevents loads and stores from being reordered across the barrier
Barrier()	Prevents the compiler from reordering loads or stores across the barrier
smp_rmb()	On SMP, provides a rmb() and on UP provides a barrier()
smp_wmb()	On SMP, provides a wmb() and on UP provides a barrier()
smp_mb()	On SMP, provides a mb() and on UP provides a barrier()

SMP = symmetric multiprocessor

UP = uniprocessor

Примитиви за синхронизиране на нишки в Solaris

- В допълнение на механизмите за съгласуване (concurrency mechanisms) в UNIX SVR4 има:
 - Взаимно изключване (mutex) заключване
 - Семафори
 - Множество читатели, един писател (readers/writer) заключвания
 - Условни променливи

MUTEX Lock

- Mutex се използва, за да осигури, че само една нишка в дадено време може да достъпи защищен от mutex ресурс.
- Нишката, която заключва mutex, трябва да бъде единствената, която може да го отключи.

Семафори и Read/Write заключвания

- Solaris предлага класически броящи семафори.
- Readers/Writers заключването позволява множество нишки да имат постоянен read-only достъп до защищен от lock обект.
 - Той също така позволява на една нишка да достъпи обекта за запис наведнъж, докато изключва достъпа на всички читатели.

Условни променливи

- Условна променлива се използва при изчакване докато дадено условие се изпълни (има стойност true).
- Условните променливи трябва да се използват заедно с тези заключването.

Механизми за съгласуване в Windows

- Windows предвижда синхронизирането на нишки като част от обектната си архитектура.
- Важни методи на синхронизация са:
 - Изпълними обекти за разпределение (използващи Wait функции),
 - Потребителски режим в критичните секции, слаби reader-writer заключвания и условни променливи.

Wait функции

- Wait функциите позволяват на нишката сама да блокира изпълнението си.
 - Wait функциите не връщат изпълнението, докато определено условие не се изпълни.
 - Типът на wait функцията определя множеството от използвани условия.

Dispatcher Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.

Критични секции

- Подобен механизъм на mutex lock
 - С изключението, че критичните секции могат да се използват само от нишки на един процес.
- Ако системата е мултипроцесорна, кода ще се опита да присвои spin-lock.
 - Като последна мярка, в случай че spinlock не може да бъде придобит, обекта за разпределение се използва да блокира нишката така, че ядрото да разпредели друга нишка на процесора.

Слаби Read-Writer заключвания

- Windows Vista добави потребителски режим reader-writer.
- Reader-writer заключването влиза в ядрото за блокиране единствено след опит за използване на spin-lock.
- 'Слаби' - тъй като обикновено се изисква заделяне на част от паметта, сочена от указатели.

Условни променливи

- Windows Vista добавя и условни променливи.
- Процесът трябва да декларира и инициализира CONDITION_VARIABLE
- Използва се или при критични секции или при SRW заключвания.

Предпазване от МХ

- Процесът получава всички ресурси преди да почне да се изпълнява
 - снижава се ефективността.

Решение : Заданието се разделя на подзадания.

Възможно е безкрайно отлагане – процесите чакащи популярни ресурси могат да чакат безкрайно

Модифицирано предпазване от МХ

- Процес заявил допълнителен ресурс, който не е на разположение, а се държи от друг процес, то ресурса се отнема от чакащия го процес и се предоставя на заявилия го процес.

Метод на кръгово очакване

- Ресурсите са подредени йерархично и притежават уникален номер
- Процесите заявяват ресурси само от по-горно ниво

Загубата на ефективност тук се дължи на факта че процесите могат да имат необходимост от ресурсите в друг от определения йерархичен ред

Алгоритъм на банкера

- Банкерът разполага с капитал
- Клиентите заявяват заеми
- Клиентите получават заеми по-малки от заявените
- Банкера следи за наличния ресурс

Капитал – Заеми

Разпределянето на ресурсите към процесите се извършва по същата логика

недостатъци

- Валиден е за фиксиран брой ресурс и процеси
- Клиентите трябва да върнат ресурсите в крайно време – при ОС не винаги е възможно

Възстановяване след МХ

- Принудително завършване на всички процеси в ОС
- Принудително завършване само на процесите участващи в МХ
- Принудително завършване на процесите в МХ, но един по един и след всяко прекратяване се извиква алгоритъм за откриване на МХ

Преразпределяне на ресурсите

- Отнема се ресурса на блокирания процес
- Процеса се рестартира или се връща стъпка назад преди момента на MX
- Предотвратяване на безкрайно отлагане е необходимо

Смесени стратегии

- Ресурсите се групират в отделни класове
- Класовете се подреждат линейно (кръгово)

Пространство на размяна - предварително разпределяне на памет – тъй като е известна

Основна памет – изпълнението на процес може да се премести във вторичната памет и основната да се освободи

Вътрешни ресурси – подреждане на ресурсите

Четири необходими и достатъчни условия за безизходица

Ресурси за последователно използване:

- Процесите използват споделени ресурси, изключващи едновременно използване

Допълнително придобиване:

- Процесите задържат ресурсите, които вече са им заделени, но чакат да получат допълнителни ресурси.

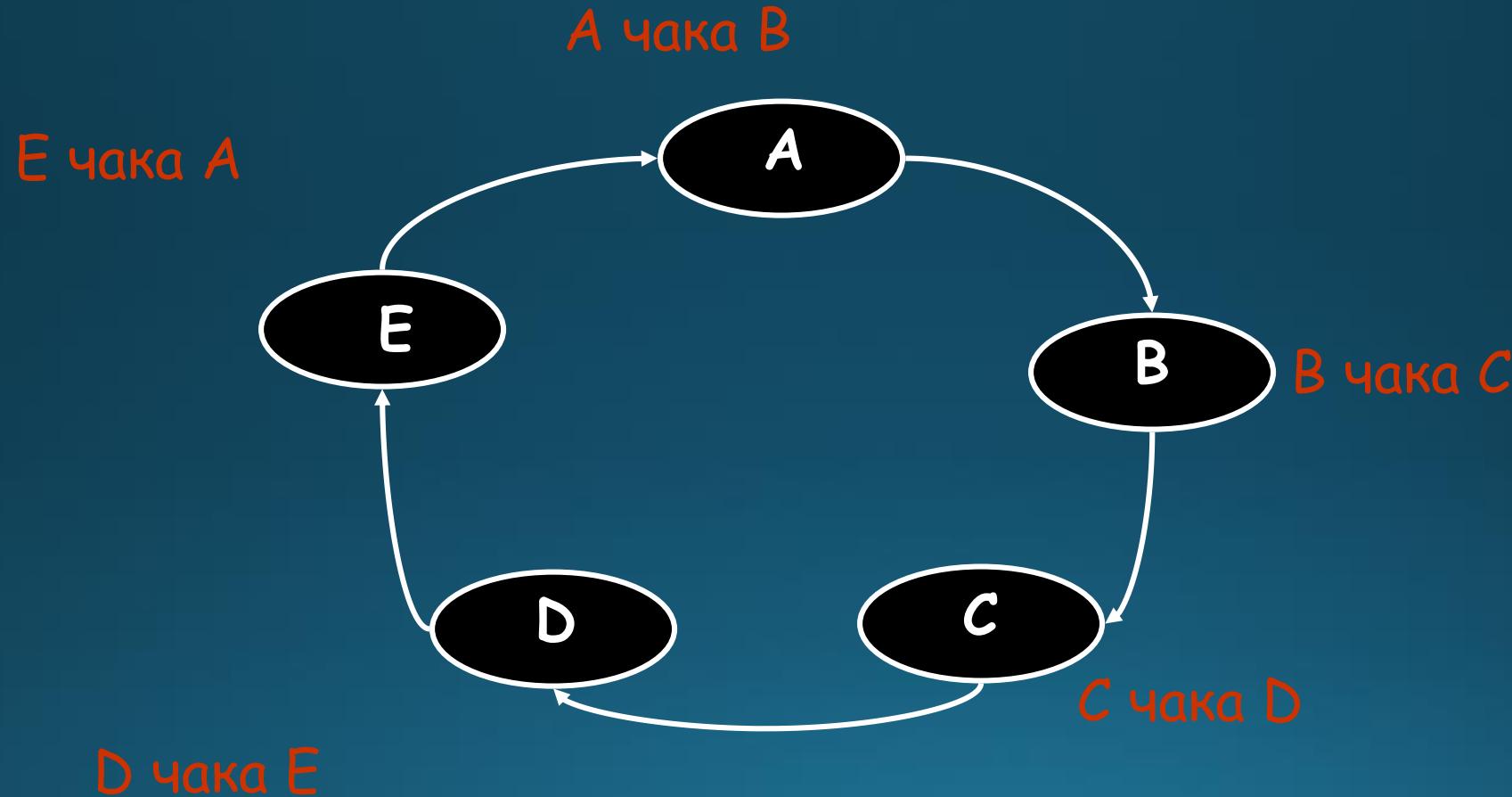
Липса на насилиствено завладяване на ресурси:

- След назначаването към процес, ресурсите не могат да бъда насила отнети, а се освобождават само доброволно.

Цикъл на изчакване:

- Цикъл, в който всеки процес задържа ресурс, необходим на следващия процес в цикъла.

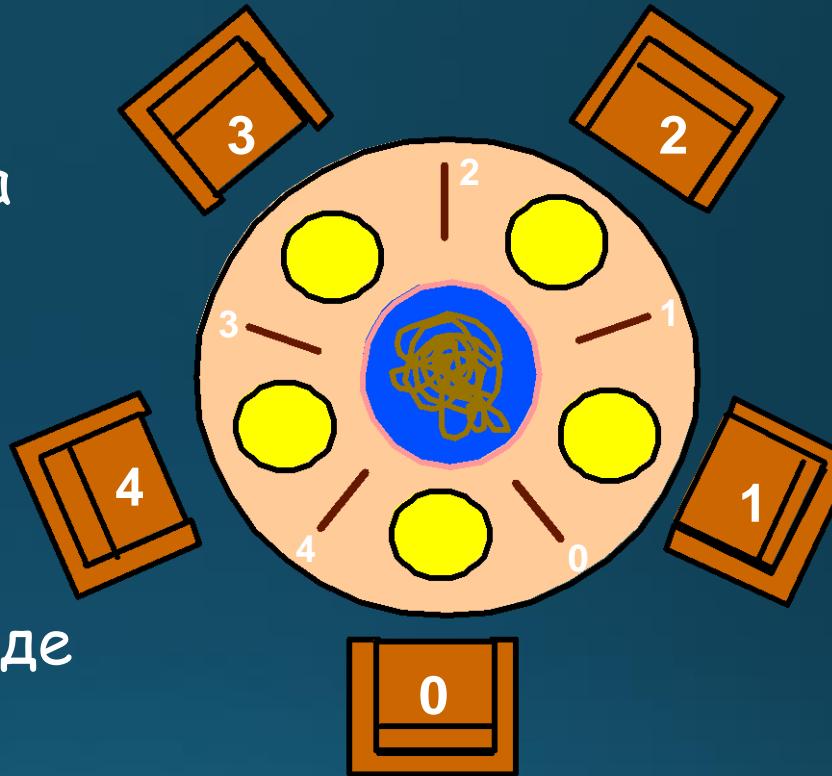
Цикъл на изчакване



Вечерята на философите

Пет философа стоят на кръгла маса. Всеки от тях прекарва живота си в последователност от **мислене и ядене**. В центъра на масата има голяма купа спагети. Всеки филосог има нужда от две вилици, за да изяде порция спагети.

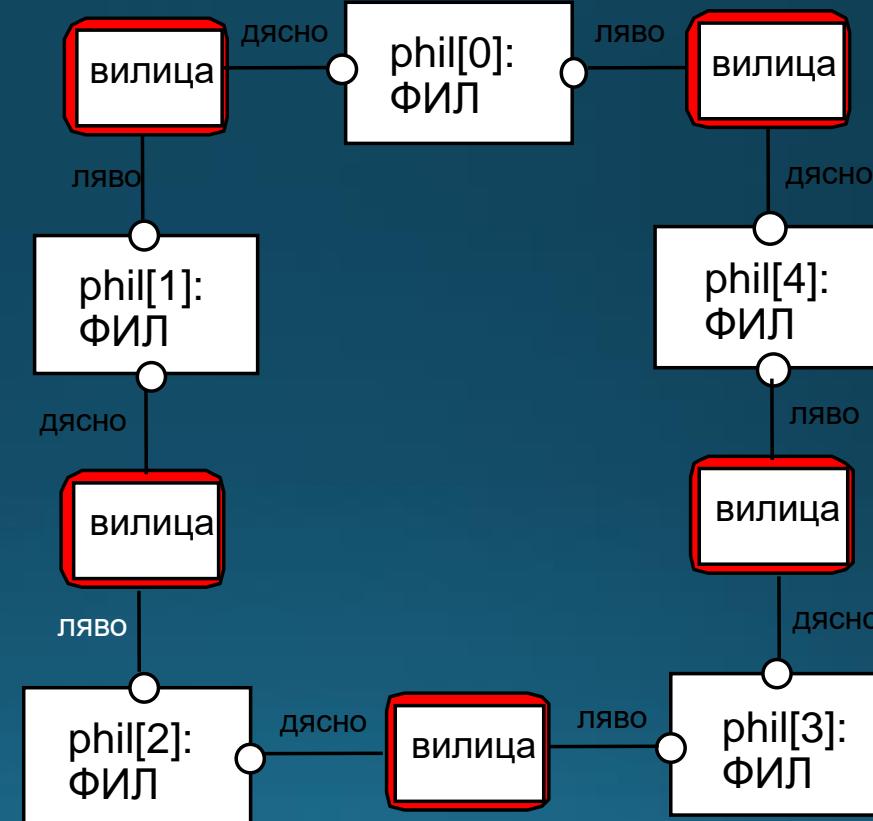
По една вилица е сложена между всяка двойка философи те са се съгласили, че всеки ще използва само вилиците, които са непосредствено вляво и вдясно.



Модел и структурна диаграма

Всяка вилица е
споделен ресурс
с възможни
действия взимам
и оставям.

Когато е гладен,
всеки ФИЛ трябва
първо да **вземе**
лявата и дясната
вилица, преди да
започне да яде.



Проблемът

- Проектиране на алгоритъм, който ще позволи на философите да се нахранят.
 - Никои двама философи не могат да използват една и съща вилица по едно и също време (взаимно изключване)
 - Нито един философ не трябва да умира от глад (избягване на deadlock и недостиг на ресурси)

Решение със семафори

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
              philosopher (3), philosopher (4));
}
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Избягане на deadlock

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

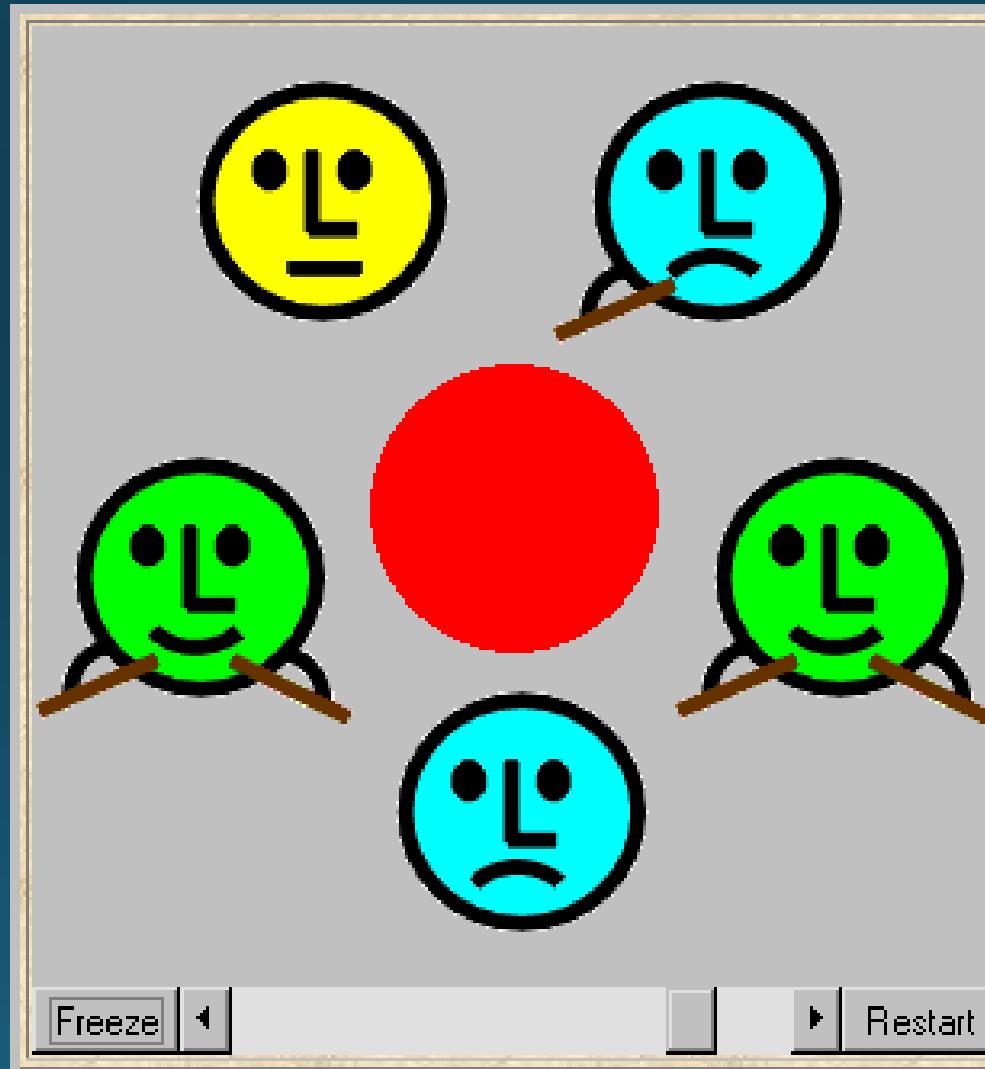
Figure 6.13 A Second Solution to the Dining Philosophers Problem

Решение с мониторами

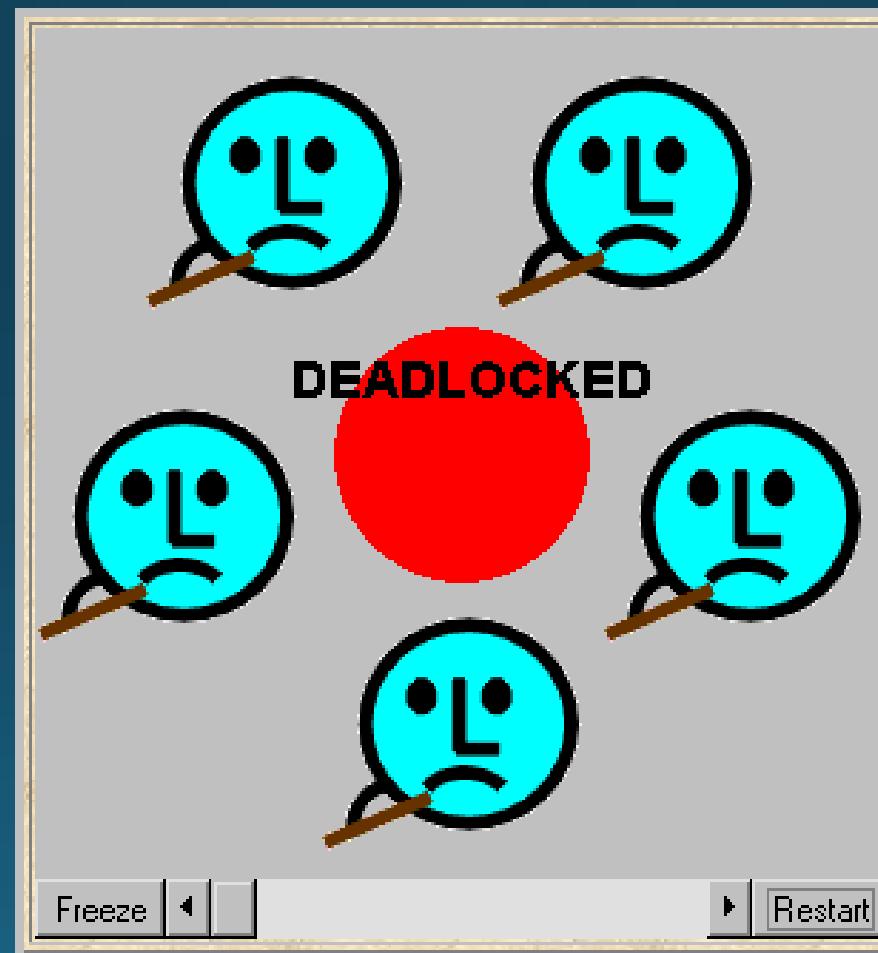
```
monitor dining_controller;
cond ForkReady[5];           /* condition variable for synchronization */
boolean fork[5] = {true};     /* availability status of each fork */

void get_forks(int pid)      /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork(left))
        cwait(ForkReady[left]);          /* queue on condition variable */
    fork(left) = false;
    /*grant the right fork*/
    if (!fork(right))
        cwait(ForkReady[right]);        /* queue on condition variable */
    fork(right) = false;
}
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left]))       /*no one is waiting for this fork */
        fork(left) = true;
    else                            /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right]))     /*no one is waiting for this fork */
        fork(right) = true;
    else                            /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```

Нормална ситуация



deadlock



Жизнен цикъл на философа

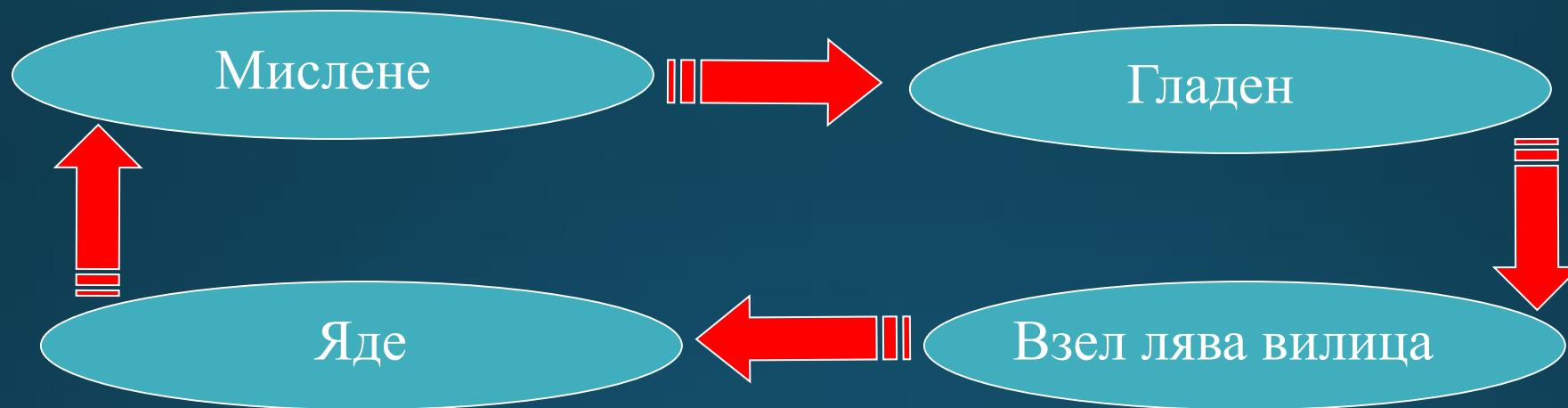
- **Мислещ философ** – няма вилици.
(На кого му трябват вилици, за да мисли?)
- **Мислещият философ** може да огладнее.
- **Гладният философ** се опитва да вземе вилицата вляво и става **гладен философ с вилица в лявата ръка**.
- Но една вилица не е достатъчна: философът започва да яде, само ако има две вилици.
- Вилицата **вдясно може да бъде взета, само ако не се използва**.
- **След ядене, може да се направи само едно**: оставяне на двете вилици и връщане към мислене.

Клас Философ

- **Философите** ще имат уникален индекс, състояние, в което се намират и могат да изпълняват две неща:
 - Рапорт за възможна промяна на състоянието (**canMove**)
 - Промяна на състоянието (**move**).

```
enum State {Thinking, Hungry, HungryWithLeftFork, Eating};  
class Philosopher {  
    enum State status = Thinking ;  
    int index;  
    Boolean canMove()  
    move()  
}
```

Успешен жизнен цикъл на един философ



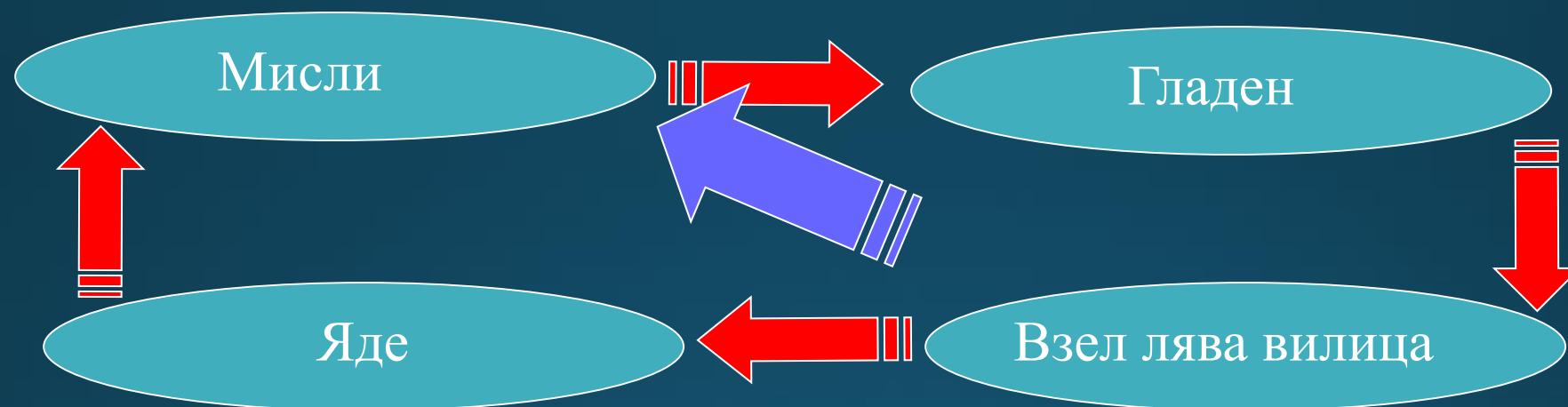
Лаком философ

- **Лакомият философ** не оставя вилицата, докато не се наяде и започне да мисли.
- Това може да доведе до безизходица.
- Това поведение е модифицирано, чрез въвеждане на случайно време за мислене и фиксирано време за ядене
 - Мислещият философ ще мисли около 80% от времето.

Благороден философ

- Благородният философ не настоява да следва успешния живот.
- След като вземе лявата вилица, но открие че дясната е заета, благородният философ оставя лявата вилица и остава да помисли още малко.
- Така че **ако всички философи са благородни, няма безизходица**, но е възможно някой да умре от глад.

Жизнен цикъл на благородния философ



Решения

- Да се позволи най-много 4 философа да седят на масата
- Да се позволи философ да вземе вилица само ако и 2 те са на разположение
- Четен вдига първо лявата после дясната, а нечетен обратно

Читател - Писател

- Ресурс може да се чете от много процеси
- Не е възможно четене и писане на общ ресурс в един и същ момент от време

Решение на проблема ч-п

- Процес контролер
- Изпращат се заявки към него
- Съобщения за освобождаване на ресурс
- Контролерът позволява първо писане
- Чака съобщение за край и предоставя за четене
- Ако има няколко 'четящи процеса' се изчакват до завършването им и тогава се предоставя ресурса за писане

Алгоритми

- Спящ бърснар
- Алгоритъм на банкера
- Приоритет на ресурси

Задача на обядващи философите

И тази задача е поставена и решена за първи път от Дейкстра чрез семафори и обща памет. Задачата се състои в следното. Пет философа седят около кръгла маса, като пред всеки от тях има чиния със спагети, а между всеки две чинии има само по една вилица. Животът на всеки философ представлява цикъл, в който той размишлява, в резултат на което огладнява и се опитва да вземе двете вилици около своята чиния. Ако успее известно време се храни, а след това връща вилиците. Задачата е да се напише програма, която да прави това, което се очаква от философа.

Решението на Дейкстра използва общ масив $state[N]$, като всеки елемент описва състоянието на съответния философ. Възможните състояния са: мисли, гладен е (опитва се да вземе двете вилици) и храни се. Достъпа до общия масив се регулира от двоичен семафор mutex. Освен него се използва масив от семафори $s[N]$, по един за всеки философ, по който той се блокира когато трябва да чака освобождаването на вилиците