Теми за Държавен Изпит

1 август 2024 г.

Компютърни архитектури

Компютърни архитектури. Формати на данните. Вътрешна структура на централен процесор – блокове и конвейерна обработка, инструкции.

1.Основният елемент е **централният процесор (CPU)**, който приема данни от **входните устройства**, обработва ги в информация, която трансферира в **паметта** и **изходните устройства**. Състои се от три основни блока **- аритметично и логическо устройство (ALU)**, **регистри** и **контролен блок**. В ALU се извършва набор от аритметични и логически операции, регистрите са оформени като малка многопортова памет, а контролният блок управлява входно/изходните устройства, генерира управляващи сигнали към останалите компоненти на системата, като сигнали за четене-/запис и организира изпълнението на инструкциите.

2.**Памет -** състои се от оперативна памет **RAM** (Random-Access Memory) за временно съхранение на данни и програмен код и **ROM** (Read-Only Memory) за съхраняване на системен код и информация.

3.**Системна шина -** състои се от **адресна шина**, **шина данни** и **контролна шина**, която включва сигнали за синхронизация и управление и всички други сигнали, които не са адреси и данни.

4. Схема за **директен достъп до паметта** (DMA) - позволява обмен на данни между паметта и други устройства, без участието на процесора. При персоналните компютри е част от Северният мост (**North Bridge**)

5. **Многоканален входно/изходен контролер** (I/O Controller) - осъществява интерфейс с външни устройства, както и с други входно/изходни устройства като клавиатура и мишка. Част от Южния мост (**South Bridge**)

6.**Програмируем контролер на прекъсванията** (Programmable Interrupt Controller) - осъществява синхронизацията на заявките за прекъсване от

различните входно/изходни устройства, като проверява текущия им статус. При персоналните компютри е част от Южния мост.

Архитектура **CISC** (Complex Instruction Set Computer)

1. Голям брой инструкции.
2. Много начини на адресиране (addressing modes).
3. Променлива дължина на инструкциите.
4. Повечето инструкции могат да манипулират директно с данни, намиращи се в клетки от паметта.
5. Контролният блок на процесора (Control Unit) използва микропрограмиране.

Предимства - лесно добавяне на нови инструкции без промяна в архитектурата на процесора.
Недостатъци - малък брой програмно достъпни регистри – частично се компенсира от възможността да се работи с клетки от паметта.

Архитектура **RISC** (Reduced Instruction Set Computer)

1. Ограничен (малък) набор от инструкции.
2. Само няколко начина на адресиране (addressing modes).
3. Всички инструкции имат еднаква дължина, което улеснява декодирането.
4. Повечето инструкции се изпълняват за един машинен цикъл.
5. Контролният блок на процесора (Control Unit) не използва микропрограмиране, а е реализиран чисто хардуерно.
6. Голям брой програмно достъпни регистри.

Предимства - инструкциите са прости и се изпълняват много бързо.
Недостатъци - по-дълъг програмен код на изпълняваните програми.

Изпълнение на инструкциите – може да се раздели на няколко стъпки:

1. Извличане (**Fetch**) на инструкция от паметта и вкарване в CPU;
2. Декодиране (**Decode**) на инструкцията – определяне на типа на инструкцията и участващите операнди от контролния блок на процесора.
3. Изпълнение (**Execute**) на инструкцията – подаване на операндите към АЛУ и извършване на операция с тях.
4. Запис на резултата от АЛУ в паметта (**Memory**).
5. Запис на резултата от АЛУ в регистър на процесора (**Writeback**), ако това е необходимо.

Запомнена програма – Програмите се зареждат в паметта на компютъра на определен от архитектурата на системата начален адрес, като това става на няколко стъпки:

1. Език от високо ниво -> Компилатор -> Асемблерски код;
2. Асемблерски код -> Асемблер -> Обектен файл;
3. Обектен файл + Библиотечни файлове -> Линкер -> Изпълним код;
4. Изпълним код -> Лоудър -> Зареден в паметта програмен код.

Структура и йерархия на паметта. Сегментна и странична преадресация. Система за прекъсване – приоритети и обслужване.

Структура и йерархия на паметта

Памет в персоналния компютър се нарича всеки ресурс, който има свойството да съхранява информация във времето. Паметта представлява съвкупност от битове - информация, която се съхранява и има стойност 0 или 1. С развитието на компютрите се оказва, че най-удобната адресируема единица е байтът, който представя осем бита. Паметта е организирана в клетки, наредени последователно една след друга, т.е. като едномерен линеен масив от байтове, които се номерират с последователни номера, наречени адреси. Номерацията започва от 0 и стига до последния наличен физически адрес. Паметта се изгражда като устройство, което може да извършва следните операции:

- четене - подаване на адрес и извеждане съдържанието на съответния байт
- писане - подаване на адрес и байт, който се запива на този адрес

Това е логическият модел на паметта. От архитектурна гледна точка, паметта се реализира физически като тримерни матрици. Когато се подаде двоичен адрес, има устройство което го дешифрира - адресът насочва устройството към елементите, които съдържат съответните битове. Паметта е организирана в йерархия от гледна точка на нейния обем и скорост на достъп до информацията. Тя се разбива на два големи класа: основна памет (RAM = SRAM + DRAM) и външна памет (магнитни дискове, CD, DVD). SRAM (Static RAM) или още наречена кеш памет осъществява много бърз достъп до най-често използваните данни и команди

Сегментна и странична преадресация

Система за прекъсване

Операционни системи

Файлова система. Логическа организация и физическо представяне.

Файлова система Файлова система е тази компонента на операционната система, която е предназначена да управлява постоянните обекти данни, т.е. обектите, които съществуват по-дълго отколкото процесите, които ги създават и използват. Постоянните обекти данни се съхраняват на външна памет (диск или друг носител) в единици, наричани файлове. Файловата система трябва да:

- осигурява операции за манипулиране на отделни файлове, например create, open, close, read, write, delete и др.
- изгражда пространството от имена на файлове и да предоставя операции за манипулиране на имената в това пространство.

Логическа организация на ФС Най-важната характеристика на една абстракция за потребителите са правилата, по които се именуват обектите, в случая файловете. Най-често името на файл е низ от символи с определена максимална дължина, като в някои системи освен букви и цифри са разрешени и други символи. Много често името се състои от две части, разделени със специален символ, например ".". Втората част се нарича разширение на името и носи информация за типа или формата на данните, съхранявани във файла. Например, следните имена имат разширения, показващи типа на данните във файла.

- file.c - програма на Си
- file.o - програма в обектен код
- file.exe - програма в изпълним код
- file.txt - текстов файл

Какво включва пространството от имена или какви са типовете файлове? Преди всичко имена на **обикновени файлове (regular files)**, съдържащи програми, данни, текстове или каквото друго потребителят пожелае. Но пространството от имена би могло да включва и имена на външни устройства, системни структури и услуги. Такова решение е прието в много от съвременните операционни системи. Външните устройства са специален тип файлове, наречени **специални файлове (character special и block special device file)** в Unix, Linux и др. Системните примитиви на файловата система са приложими както към обикновените файлове така и към специалните файлове. Следователно, всяка операция за четене или писане, осъществявана от програмата, е четене или писане във файл. Най-същественото пре-

димство на този подход е, че позволява да се пишат програми, които не зависят от устройствата, тъй като действията, изпълнявани от програмата зависят само от името на файла. Трябва да се съхранява информация за файловете във файловата система и за тази цел се използват специални системни структури, наричани **каталог, справочник, директория, directory, folder, VTOC**, които осигуряват връзката между името и данните на файла и реализират организацията на файловете. В много системи каталогът е тип файл с фиксирана структура на записите и съдържа по един запис за всеки файл. И така, типовете файлове, поддържани от файловите системи на Unix, Linux и др. са:

- обикновен файл
- каталог
- специален файл
- програмен канал, FIFO файл
- символни връзки

Типът **програмен канал (pipe)** и **FIFO файл** се използва като механизъм за комуникация между конкурентни процеси. Чрез **символните връзки (symbolic link, soft link, junction)** един файл може да има няколко имена, евентуално в различни каталози, или както се казва реализират се няколко връзки към файл. Този тип файл е един от механизмите за осигуряване на общи файлове за различните потребители. Друг начин за работа с общи файлове са **твърдите връзки (hard links)**, но те не са тип файл, а са няколко имена на обикновен файл. Тясно свързан с въпроса за типовете файлове е вътрешната структура на файл. Файлът най-често представлява последователност от обекти данни. Какви са тези обекти данни? Възможни са два отговора - запис или байт: Файлът е последователност от записи с определена структура и/или дължина. Основното в този подход е, че всеки системен примитив read или write чете или пише един запис. Такъв подход е използван в DOS/360, OS/360, CP/M. Другата възможност, реализирана в много от съвременните операционни системи, Unix, Linux, MINIX, MSDOS, Windows и др., е последователност от байтове. Това е структурата на файл, реализирана от файловата система. Всяка по-нататъшна структура на файла може да се реализира от програмите на потребителско ниво, като операционната система не помага за това, но и не пречи. Този подход позволява разработката на прости системни примитиви, които освен това да са приложими както за обикновени файлове, така и за останалите типове файлове. Всеки файл има име и данни. В допълнение операционната система може да съхранява и друга информация за файла, която ние ще наричаме атрибути на файла. Атрибутите, реализирани в различните системи се различават, но някои общи са: време на създаване, време на достъп, време на промяна, собственик, права на достъп, парола за достъп и флагове като read-only, system, archive и др.

Йерархична организация на файловата система
Вътрешните възли на дървото трябва да са каталози, а листата могат да бъдат каталози, обикновени файлове, специални файлове и други типове фалове, ако се поддържат такива. За потребителя каталогът представлява група от файлове и каталози (подкаталози). Този подход, използван при файловите системи на повечето съвременни операционни системи Unix, Linux, MSDOS, MINIX, Windows и др., дава възможност за естествено и удобно групиране на файловете, отразяващо предназначението и формата на данните, съхранявани в тях. Всеки връх в дървото, каталог или друг тип файл, има име, което потребителят избира да е уникално в рамките на съдържащия го каталог и което ние ще наричаме собствено име. Тогава всеки файл ще има име, което уникално го идентифицира в рамките на цялата йерархична структура и ние ще го наричаме пълно име. Използват се два начина за формиране на пълно име.

- абсолютно пълно име (absolute path name) - всеки файл или каталог притежава едно абсолютно пълно име, което съответства на единствения път в дървото от корена до съответния файл или каталог
- относително пълно име (relative path name) - този начин за формиране на пълно име е свързан с понятието текущ или работен каталог (current/working directory). Във всеки един момент от работата на потребителя със системата, той е позициониран в един от каталозите на дървото. Операционната система предоставя средства за избор на текущ каталог. Относителното пълно име на файл или каталог съответства на пътя в дървото от текущия в даден момент каталог до съответния файл или каталог. Като в този случай е разрешено и движение нагоре по дървото.

Пълното име на файл, абсолютно или относително, се състои от компоненти - собствените имена на каталозите в пътя и на самия файл, разделени със специален символ-разделител, като напр. » "; "/". Движението нагоре по дървото се записва чрез специално име, което обикновено е "..". Ако първият символ в пълното име е символа-разделител, това означава, че името е абсолютно пълно. Всяко име, което не започва със символа-разделител, се приема за относително пълно или собствено име. Абсолютно пълно име на файл е /home/ivan/letter. Ако текущ каталог е /home, то относителното пълно име на същия файл е ivan/letter. Ако текущ каталог е /home/emi, то относителното пълно име на същия файл е ../ivan/letter. Следователно, ако от текущ каталог /home/emi искаме да създадем копие на файла под име *copyletter*, имаме различни възможности за именуване на оригиналния файл и копието:
cp/home/ivan/letter/home/emi/copyletter
cp/home/ivan/lettercopyletter
cp../ivan/lettercopyletter

Управление на процеси и междупроцесни комуникации.

Ще разгледаме системните примитиви по стандарта POSIX, които реализират основните операции за процеси, а именно създаване на процес, завършване на процес и други.

1.Създаване на процес

```
#include <sys/types.h>
pid_t fork(void);
```

Единственият начин да се създаде процес в Unix/Linux системите е чрез fork. Процесът, който изпълнява fork се нарича процес-баща, а новосъзданият е процес-син. При връщане от fork двата процеса имат еднакви образи, с изключение на връщаната стойност: в процеса-баща fork връща pid на процеса-син при успех или -1 при грешка, а в сина връща 0. Сложността при този примитив се състои в това, че един процес "влиза"във функцията fork, а два процеса "излизат от"fork с различни връщани значения. Да разгледаме по-подробно това, което става когато процес-баща изпълнява fork и така ще разберем какво е общото между процесите баща и син.

Алгоритъм на fork

1. Определя уникален идентификатор за новия процес и създава запис в таблицата на процесите, като инициализира полетата в него (група и сесия от процеса-баща, състояние “новосъздаден” и т.н.).
2. Създава U area, където повечето от полетата се копират от процеса-баща (файловите дескриптори, текущ каталог, управляващ терминал, потребителските идентификатори - euid, ruid, egid и rgid).
3. Създава образ на новия процес - копие на образа на процеса-баща (регионът за код обикновено е общ за двата процеса).
4. Създава динамичната част от контекста на новия процес: Слой 1 е копие на слой 1 от контекста на бащата. Създава нов слой 2, в който записва съхранените регистри от работата в слой 1, като регистър PC е изменен така, че синът да започне изпълнението си в fork от стъпка 7.
5. Изменя състоянието на процеса-син в "готов в паметта".
6. В процеса-баща връща pid на новосъздадения процес-син.
7. В процеса-син, връща 0.

Следователно, когато по-късно планировчикът избере новосъздадения процес за текущ той ще заработи според най-горното ниво на динамичната част от контекста си (слой 2) - в системна фаза на fork и ще върне 0. И така процесите син и баща имат следното общо помежду си: - Двата процеса изпълняват една и съща програма. Дори процесът-син, който преди това не е работил, започва изпълнението на потребителската програма от оператора след fork. Но връщаните стойности в двата процеса са различни, което позволява да се определи кой процес е бащата и кой сина и да се раздели тяхната функционалност макар, че изпълняват една и съща програма. -

Процесът-син наследява от бащата файловите дескриптори, т.е. двата процеса ще разделят достъпа до файловете, които бащата е отворил преди да изпълни fork. Това позволява пренасочване на входа и изхода и свързване на процеси с програмни канали. - Двата процеса имат един и същ текущ каталог, управляващ терминал, група процеси и сесия. - Двата процеса имат еднакви права - реален и ефективен потребителски идентификатори, което гарантира неизменност на привилегиите на потребител при работа в системата.
Има два начина за използване на fork. Процес иска да създаде свое копие, което да изпълнява една операция, докато другото копие изпълнява друга, например при процеси сървери. Процес иска да извика за изпълнение друга програма, тогава първо създава свое копие, след което в едното копие (обикновено в процеса-син) се извиква другата програма. Това се използва от командния интерпретатор.

2.Завършване на процес

```
void exit(int status);
```

Системата не поставя ограничение за времето на съществуване на процес. Има процеси, например процес init (с pid 1), които съществуват вечно в смисъл от boot до shutdown. Когато процес завършва той изпълнява exit. Процесът може явно да го извика или неявно в края на програмата, но може и ядрото вътрешно да извика exit без знанието на процеса, например когато процеса получи сигнал, за който не е предвидил друга реакция. Значението на аргумента status е кода на завършване, който се предава на процеса-баща, когато той се заинтересува. Този системен примитив не връща нищо, защото няма връщане от него, винаги завършва успешно и след него процесът почти не съществува, т.е. той става зомби.
Алгоритъм на exit

1. Изпълнява close за всички отворени файлови дескриптори и освобождава текущия каталог.
2. Освобождава паметта, заемана от образа на процеса и потребителската област.
3. Сменя състоянието на процеса в зомби. Записва кода на завършване в таблицата на процесите. Ако процесът завършва по сигнал, код на завършване е номера на сигнала.
4. Урежда изключването на процеса от йерархията на процесите. Ако процесът има синове, то техен баща става процесът init и ако някой от тези синове е зомби изпраща сигнал "death of child"на init. Изпраща същия сигнал и на процеса-баща на завършващия процес.

3.Синхронизация със завършване на процеса-син

```
#include <sys/wait.h>
#include <sys/types.h>
```

```
pid_t wait(int *status);
```

Процес-баща може да синхронизира работата си със завършването на свой процес-син, т.е. да изчака неговото завършване ако още не завършил и да разбере как е завършил, чрез wait. Функцията на системния примитив връща pid на завършилия син, а чрез аргумента status кода му на завършване.
Алгоритъм на exit

1. Ако процесът няма синове, това е грешка и функцията връща -1.
2. Ако процесът има син в състояние зомби, т.е. синът вече е изпълнил exit, освобождава записа му от таблицата на процесите, като взема кода на завършване и неговия pid и ги връща.
3. Ако процесът има синове, но никой от тях не е зомби, той се блокира, като чака сигнал "death of child". Когато получи такъв сигнал, което означава, че някой негов син току що е станал зомби, продължава както в точка 2.

Сега след като разгледахме системните примитиви fork, exit и wait, става по-ясно значението на състоянието зомби. След fork двата процеса - баща и син съществуват едновременно и се изпълняват асинхронно. Това означава, че бащата може да изпълни wait както преди така и след exit на сина. Освен това не бива да се задължава процес-баща да изпълнява wait, той може да завърши веднага след като е създал син. Но тогава в системата ще останат вечни зомбита, които заемат записи в таблицата на процесите. Затова когато процес завършва неговите синове се осиновяват от процеса init, който се грижи за изчистване на системата от зомбита-сираци.

4.Изпълнение на програма
Когато с fork се създава нов процес, той наследява образа си от бащата, т.е. продължава да изпълнява същата програма. Но чрез exec всеки процес може да смени образа си с друга програма по всяко време от своя живот, както веднага след създаването си така и по-късно, дори няколко пъти в своя живот. За този системен примитив има няколко функции в стандартната библиотека, които се различават по начина на предаване на аргументите и използване на променливите от обкръжението на процеса.

```
int execl(const char *name, const char *arg0
    [, const char *arg1]..., 0);
int execlp(const char *name, const char *arg0
    [, const char *arg1]..., 0);
int execv(const char *name, const char *argv[]);
int execvp(const char *name, const char *argv[]);
```

Първият аргумент name указва към името на файл, съдържащ изпълним код, от който ще се създаде новия образ. При execvp и execlp name може да е собствено име на файл и тогава файлът се търси в каталозите от променливата PATH. В останалите случаи name трябва да е пълно име на файл. Останалите аргументи в execl и execlp са указатели към аргументите, които ще се предадат на функцията main когато новият образ започне изпълне-

нието си. Във функциите execv и execvp има един аргумент argv, който е масив от указатели на аргументите за функцията main, който също трябва да завършва с елемент NULL.

Алгоритъм на exec

1. Намира файла, чието име е в аргумента name и проверява дали процесът има право за изпълнение на този файл.
2. Проверява дали файлът съдържа изпълним код.
3. Освобождава паметта, заемана от стария образ на процеса.
4. Създава нов образ на процеса, използвайки изпълнимия код във файла и копира аргументите на exec в новия потребителски стек.
5. Изменя значенията на някои регистри в областта за съхранени регистри в слой 1 от динамичната част на контекста, напр. на PC, указател на стек. Така, когато процесът се върне в потребителска фаза ще заработи от началото на функцията main на новия образ.
6. Ако програмата е set-UID прави съответните промени на потребителските идентификатори на процеса.

При успех, когато процесът се върне от exec в потребителска фаза, той изпълнява кода на новата програма, започвайки от началото, но това си остава същия процес. Не е променен идентификатора му, позицията му в йерархията на процесите дори голяма част от потребителската област, като файловите дескриптори, текущия каталог, управляващия терминал, групата на процесите, сесията. При грешка по време на exec става връщане в стария образ, така че функцията връща -1 при грешка, а при успех не връща нищо защото няма връщане в стария образ.

Разделянето на създаването на процес и извикването на програма за изпълнение в два системни примитива играе важна роля. Това дава възможност програмата на процес-баща да определя поне в началото функционалността на свой процес-син и например, да се реализира пренасочване на входа и изхода, и свързването на роднински процеси чрез програмни канали (заслуга за това има и наследяването на файловите дескриптори).

5.Информация за процес

```
1 pid_t getpid(void);
2 pid_t getppid(void);
```

Системният примитив getpid връща идентификатора на процеса, който го изпълнява, а getppid връща идентификатора на неговия процес-баща. И двата примитива винаги завършват успешно.

Компютърни мрежи

Компютърни мрежи и протоколи – OSI модел. Маршрутизация. Протоколи IPv4, IPv6, TCP, DNS.

OSI модел

Съвременните мрежови архитектури следват принципите на модела OSI (Open Systems Interconnection), създаден от Международната организация по стандартизация ISO (International Standards Organization) за връзка между отворени системи. Отворена система е система, чиито ресурси могат да се използват от другите системи в мрежата. OSI моделът е абстрактен модел на мрежова архитектура, който описва предназначението на слоевете, но не се обвързва с конкретен набор от протоколи. Поради това OSI моделът се нарича още еталонен (опорен) модел и всъщност дава препоръки (Reference Model). В еталонния модел има седем слоя – физически, канален, мрежов, транспортен, сесиен, представителен, приложен.

Разпределена маршрутизация

Обединената мрежа (internet) – това е съвкупност от няколко мрежи, наричани също така подмрежи (subnet), които се свързват помежду си с маршрутизатори. Организацията на съвместно транспортно обслужване в съставната мрежа се нарича междумрежово взаимодействие (internetworking). Една от основните функции на мрежовия слой е предаването на пакети между крайните възли в обединените мрежи (отделните подмрежи). Този слой на практика осъществява междумрежово свързване. Маршрут се нарича последователността от маршрутизатори, през които трябва да премине пакета от източника (изпращача) до целта (получателя). Задачата за избор на маршрут от няколко възможни се решава от маршрутизаторите и крайните възли на основание на маршрутните таблици. Записите в таблиците могат да се попълват ръчно или от протоколите за маршрутизация. Протоколите за маршрутизация (например RIP или OSPF) следва да бъдат отличавани от мрежовите протоколи (например IP или IPX). Докато първите събират и предават по мрежата само служебна информация за възможните маршрути, вторите са предназначени за предаване на потребителски данни.

Мрежовите протоколи и протоколите за маршрутизация се реализират във вид на програмни модули на крайните възли (хостовете) и на междинните – маршрутизаторите. Маршрутизаторът представлява сложно многофункционално устройство (специализиран компютър), в задачите на което влиза: построяване на маршрутни таблици, определяне на тяхна основа на маршрута, буферизация, фрагментация и филтрация на постъпващите пакети, поддръжка на мрежовите интерфейси. Главните функции на маршрутизаторите са: избор на най-добрия път за пакетите до адреса на получателя и комутация на приетия пакет от входящия интерфейс към съответстващия изходящ интерфейс.

Маршрутизацията е процес за определяне на най-добрият път (маршрут), по който пакетът може да бъде доставен до получателя. Възможните пътища за предаване на пакетите се наричат маршрути. Най-добрите маршрути до известните получатели се съхраняват в маршрутната таблица. В зависимост от начина на попълване на маршрутните таблици се различават два вида маршрутизация:

- Статична маршрутизация - данните се въвеждат от мрежовия администратор;
- Динамична маршрутизация - информацията постъпва от съседните маршрутизатори като се използва протокол за динамична маршрутизация.

Маршрутизаторът оценява достъпните пътища до адреса на получателя и избира най-рационалния маршрут на база на някакъв критерий - метрика. Най-малката метрика означава най-добър маршрут. Метриката на статичен маршрут винаги е равна на 0. Процесът на маршрутизация на дейтаграмите се състои в определянето на следващия възел (next hop) по пътя на предаването на дейтаграмата и прехвърлянето ѝ към този възел, който се определя като цел или междинен маршрутизатор. Нито възелът изпращач, нито някой междинен маршрутизатор разполагат с информация за цялата верига, по която се предава дейтаграмата. Всеки маршрутизатор, а също така и възелът изпращач, базирайки се на адреса на получателя на дейтаграмата, определят единствено следващия възел от нейния маршрут.

Протоколите за маршрутизация се делят на:

- Външни - пренасят маршрутната информация между автономните системи (EGP, BGP)
- Вътрешни - прилагат се единствено в границите на определена автономна система. (RIP, OSPF)

Различават следните класове протоколи за динамична маршрутизация:

- Протоколи с векторно разстояние (**Distance vector**) — протоколи за маршрутизация на база векторно разстояние, които използват за търсене на най-добрия път разстоянието до отдалечената мрежа. Всяко пренасочване на пакета от маршрутизатора се нарича hop (хоп, скок). За най-добър се счита пътят до отдалечената мрежа с най-малък брой хопове. Векторът определя направлението към отдалечената мрежа. Примери за протоколи за маршрутизациа с векторно разстояние са RIP и IGRP.
- Протоколи със състояние на връзките (**Link state**) — обикновено се наричат "първият – най-краткия път"(SPF). Всеки маршрутизатор създава три различни таблици. Първата от тях проследява непосредственото свързване на съседите, втората — определя топологията на

цялата обединена мрежа, а третата е маршрутната таблица. Устройството, действащо според протокол от типа състояние на връзките, има повече сведения за обединената мрежа, от колкото всеки протокол с векторно разстояние. Примери за протоколи за IP маршрутизация за състоянието на връзките са протоколите OSPF и IS-IS.

IPv4

IPv6

Основни характеристики на IPv6 протокол:

1. Дължината на IP адреса е увеличена до 16 байта, като така се предоставя на потребителите практически неограничено адресно пространство;
2. Опростена структура на заглавието, съдържащо само 8 полета (вместо 13 в IPv4 протокола). Последното позволява на маршрутизаторите по-бърза обработка на пакетите, т. е. повишава се тяхната производителност;
3. Подобрена поддръжка на незадължителни параметри, понеже в новото заглавие изискваните по-рано полета стават незадължителни, а измененият начин за представяне на незадължителните параметри ускорява обработването на пакетите от маршрутизаторите за сметка на отсъствието на обработване на тези параметри;
4. Подобрена система за безопасност. Автентификацията и конфиденциалността са основни характеристики на новия IP протокол;
5. Предвидена е възможност за разширяваня на типовете (класовете) на предоставяните услуги, които могат да се появят в резултат на очаквания растеж на мултимедийния трафик. Отделено е по-голямо внимание на типа на предоставяните услуги. За тази цел в заглавието на пакета на IPv4 е било предвидено 8 разрядно поле.

TCP

TCP (протоколът за управление на предаването) е протокол от транспортния и сеансовия слой на OSI/RM модела. TCP протоколът осигурява надеждно предаване на данните между приложните процеси, понеже използва установяване на логическо съединение между взаимодействащите процеси. Логическото съединение между два приложни процеса се идентифицира от двойка сокети (IP адрес, номер на порт), всеки от които описва един от взаимодействащите процеси. Информацията, постъпваща към TCP протокол в рамките на логическото съединение от протоколите от по-горния слой, се разглежда от TCP протокола като неструктуриран поток от байтове и се записва в буфер. За предаването към мрежовия слой от буфера се изрязва сегмент, непревишаващ 64 Кбайта (максималния размера на IP пакет). На практика дължината на сегмента се ограничива от стойността 1460 байта, като така се осигурява той да се разположи в Ethernet кадър с TCP и IP

заглавия. TCP съединението е ориентирано към пълно дуплексно предаване. Управлението на потока от данни в TCP протокола се осъществява с използването на механизма на плаващия прозорец с променлив размер. При предаването на сегмента възелът изпращач включва таймер и очаква потвърждение. Отрицателни квитанции не се изпращат, а се използва механизмът за таймаут. Възелът цел като получи сегмента формира и изпраща обратно сегмент (с данни, ако има такива, или без данни) с номер за потвърждение, равен на следващия пореден номер на очаквания байт. За разлика от много други протоколи, TCP протоколът потвърждава получаването не на пакети, а на байтове от потока. Ако времето за очакване на потвърждението изтече, изпращачът изпраща сегмента отново. Въпреки че изглежда прост в работата си протокол, в него има редица нюанси, които могат да доведат до някои проблеми. Първо, понеже сегментите при предаването по мрежата могат да се фрагментират, е възможна ситуация, при която част от предадения сегмент ще бъде приета, а останалата част ще се окаже изгубена. Второ, сегментите могат да пристигат във възела на получателя в произволен ред, което може да доведе до ситуация, при която байтовете от 2345 до 3456 вече са получени, но потвърждението за тях не може да бъде изпратено, понеже байтовете от 1234 до 2344 все още не са получени. Трето, сегментите могат да се задържат в мрежата толкова дълго, че при изпращача да изтече интервала за очакване на потвърждението и той да ги изпрати отново. Предаденият повторно сегмент може да премине по друг маршрут и може да бъде фрагментиран по друг начин, или сегментът може по маршрута да попадне случайно в претоварена мрежа. Като резултат за възстановяването на изходния сегмент ще се изисква достатъчно сложна обработка.

TCP осигурява своята надеждност чрез използването на следните механизми:

- Данните от приложението се разбиват на блокове (сегменти) с определен размер, които ще се изпращат.
- Когато TCP изпраща сегмент, той устанавява таймер, очаквайки, че от отдалечената страна ще пристигне потвърждение за този сегмент, че е получен. Ако потвърждението не е получено до изтичането на времето, сегментът се изпраща повторно.
- Когато TCP приема данните от отдалечената страна на съединението, то изпраща потвърждение. Това потвърждение не се изпраща незабавно, а обикновено след части от секундата.
- TCP осъществява изчисляване на контролна сума за своето заглавие и данните. Тази контролна сума се изчислява и в двата края на съединението, като целта й е да се открият всякакви изменения в данните по време на предаването им. Ако сегментът е с невярна контролна сума, TCP го отхвърля и потвърждение не се генерира. (Очаква се, че изпращачът ще отработи таймаута и ще направи повторно предаване.)

Понеже TCP сегментите се инкапсулират във вид на IP дейтаграми, а IP дейтаграмите могат да са с нарушена последователност при предаването, също така и TCP сегментите могат да са с нарушена последователност при предаването. След получаването на данните TCP може при необходимост да измени тяхната последователност, като резултат приложението получава данните в правилен ред.

- Понеже IP дейтаграмата може да се дублира, приемащата страна на TCP трябва да отхвърли дублираните сегменти.
- TCP осъществява контрол на потока от данни. Всяка страна на TCP съединението има определено буферно пространство. TCP на приемащата страна позволява на отдалечената страна да изпраща данни само в този случай, ако получателят може да ги разположи в буфера. Последното предотвратява препълването на буферите на бавни хостове от бързи хостове.

TCP съединение
Съединението е съвкупност от информация за състоянието на потока от данни, включваща сокетите, номерата на изпратените, приетите и потвърдените октети, размерите на прозорците. Всяко съединение уникално се идентифицира в Интернет от двойката сокети. Съединението се характеризира за процеса с име, което представлява указател към TCB (Transmission Control Block) структура, съдържаща информация за съединението. Отварянето на съединението от процеса се реализира с извикването на функцията OPEN, на която се предава сокета, с която трябва да се установи съединението. Функцията връща името на съединението. Различават се два типа отваряне на съединението: активно и пасивно. Работата с TCP протокол между двама абонати винаги преминава през следните три етапа: установяване (отваряне) на съединение, обмен на данни и затваряне на съединението.

Отваряне и затваряне на TCP съединение
За да се отвори (установи) TCP съединение, е необходимо да се извършат следните действия: 1. Страната - инициатор на съединението (клиент) изпраща SYN сегмент (вдигнат бит SYN в полето за флагове на заглавието на TCP), посочвайки името на домейна (или IPdst) и номера на порта на сървъра, към който клиентът иска да се присъедини, и началния номер на последователността на клиента (поле Sequence Number в заглавието на TCP). 2. Сървърът отговаря със сегмент SYN, съдържащ своя начален номер на последователността на сървъра заедно с вдигнат флаг SYN. Също така е вдигнат флаг ACK и е попълнено полето "номер на потвърждението"(Acknowledgement number), където се записва полученият от клиента номер на последователността + 1. 3. Клиентът трябва да потвърди пристигането на SYN сегмента от сървъра с използването на ACK флага и с нова стойност в полето за потвърждение (полученият от сървъра номер на последователността + 1). Предаването на тези три сегмента е достатъчно за установяването на съединението (често този процес се нарича три стъп-

ково ръкостискане, three - way handshaking). След това между страните (клиента и сървъра) е възможен двустранен обмен на данни по установеното съединение. При едностранно затваряне на съединението страната - инициатор на затварянето трябва да изпрати по установеното съединение FIN сегмент (с вдигнат флаг FIN), а също така и да получи ACK отговор от отдалечената страна с уведомление за получаването на FIN сегмента. След това съединението с възможност за двустранен обмен на данни преминава в еднопосочно състояние (едната страна е затворила съединението, а втората е активна и поддържа отворено съединението). За да се затвори напълно съединението, активната страна трябва да формира FIN сегмент и да получи за него потвърждение. Така при установяването на съединението на двете страни е необходимо да изпратят и да приемат 3 сегмента, а при затварянето му – 4.

DNS

DNS (Domain Name System, система за символни имена на домейните) е важна за работата на Интернет, понеже за свързването с отдалечената система е необходима информация за неговия IP адрес, а за хората е полесно да запомнят буквени (обикновено повече осмислени) адреси, отколкото последователността от цифри на IP адреса. Първоначално преобразуването между адреси на домейни и IP адреси се е реализирало с използването на специален текстов файл hosts, който се е създавал централизирано и автоматично се е разпращал на всяка от машините в своята локална мрежа. С растежа на Интернет е възникнала необходимост от ефективен, автоматизиран механизъм, какъвто е станала системата DNS. В TCP/IP стека се използва DNS, която има йерархична дървовидна структура, допускаща използване в името на домейна произволен брой съставни части. Съвкупността от имена, за които няколко от старшите съставни части съвпадат, образуват домейн (domain) от имена. Имената на домейните се назначават централизирано, ако мрежата е част от Интернет, иначе - локално. Съответствието между имената на домейните и IP адресите може да се устанавява както със средствата на локалния хост (с използването на файла hosts), така и с помощта на централизираната DNS система, основана на разпределена база за съответствие от вида «име на домейн – IP адрес». Име на домейн е и символното име на компютъра.

DNS функционира по схемата "клиент-сървър". В качеството на клиентска част се използва процедурата за разрешаване на имената - resolver, а в качеството на DNS - сървъра (BIND). DNS е разпределена система. Всеки DNS сървър съхранява имената на следващия слой от йерархията и освен таблицата на изобразяване на имената съдържа препратки към DNS сървърите на своите поддомейни, което опростява процедурата за търсене. За ускоряване на търсенето на IP адреси в DNS сървърите се прилага процедура за кеширане на преминалите през тях отговори за определено време, обикновено от няколко часа до няколко дни.

Примери за имена на домейни за организации са:

- com - комерсиални организации
- edu - образователни организации
- gov - правителствени организации
- org - некомерсиални организации
- net - организации за поддръжка на компютърните мрежи

DNS притежава следните характеристики:

- Разпределеност на администрирането. Отговорността за отделните части от йерархическата структура се поема от различни хора или организации.
- Разпределено съхраняване на информацията. На всеки възел от мрежата задължително трябва да се пазят само тези данни, които влизат в неговата зона на отговорност и (възможно) адресите на кореновите DNS сървъри.
- Кеширане на информацията. Възелът може да съхранява някакво количество данни, които не са за собствената зона на отговорност. Целта е намаляване на натоварването на мрежата.
- Йерархична структура, в която всички възли са обединени в дърво, и всеки възел може или самостоятелно да определя работата на по-долу стоящите възли, или да ги делегира (предава) на други възли.
- Резервиране. За съхраняването и обслужването на своите възли (зони) отговарят (обикновено) няколко сървъра, разделени както физически, така и логически, като по този начин те осигуряват съхраняването на данните и отказоустойчивост (продължаване) на работата, даже в случай на отказ на един от възлите.

DNS протоколът използва в работата си TCP или UDP, с номер на порт 53 за отговори на заявките. Традиционно заявките и отговорите се изпращат във вид на една UDP дейтаграма. TCP се прилага за AXFR (full zone transfer) заявки. Обичайно този механизъм изпълнява репликация на информация на зоната между сървърите, но той също така може да се използва за злонамерени цели (като получаване на различна информация за осъществяване на спам, разпределени DoS атаки и т.н.). Когато се обръщаме към сървъра с конкретно запитване (например с използване на домейни само от първо и второ ниво), тогава браузърът, използвайки resolver, изпълнява следния алгоритъм:

1. Търси запис с конкретното запитване във файла hosts, ако не открие, тогава
2. Изпраща запитване към известен DNS кеширащ сървър (като правило, локален), ако на този сървър такъв запис не е открит, тогава

3. DNS кеширащият сървър се обръща към DNS-ROOT сървъра със запитване за адреса на DNS сървъра, отговарящ за домейна от първо ниво. Ако получи адреса, тогава
4. DNS кеширащият сървър се обръща към DNS сървъра, отговарящ за домейна от първо ниво, със запитване за адреса на DNS сървъра, отговарящ за домейна от второ ниво. Ако получи адреса, тогава
5. DNS кеширащият сървър изпраща запитване към DNS сървъра, отговарящ за домейна от второ ниво. Ако получи адреса, тогава
6. DNS кеширащият сървър кешира адреса и го предава на клиента
7. Клиентът се обръща към откритията IP адрес

Бази от данни

Бази от данни. Релационен модел на данните.

В основата на релационния модел стои математическото понятие n-членна релация. Всяка релация от

Дефиниция **Домейн** наричаме множество от допустими стойности за даден атрибут.

Дефиниция **Релация** наричаме

Дефиниция **Кортежи** наричаме множеството от атрибутите на дадена релация.

Дефиниция **Атрибути** наричаме означенията (имената) на колоните на релацията.

Дефиниция **Схема на релация** наричаме името на релацията и множеството от атрибутите й (Movies(title, year, length)).

Дефиниция **Схема на релационна база от данни** наричаме множеството от всички схеми на релации в базата от данни.

Дефиниция **Релационна алгебра** наричаме алгебра, чиито операнди са релации или променливи, които представят релации.

Основни операции

- Обединение - $R \cup S$ (бинарна, комутативна, асоциативна)
- Разлика - $R - S$ (бинарна)

- Декартово произведение - $R \times S$ (бинарна, комутативна, асоциативна), множеството от всички двойки, при които първият елемент е произволен от R, а вторият от S
- Проекция - $\pi_{<attrlist>}(R)$, където $< attrlist >$ е списък с атрибути
- Селекция - $\sigma_{<predicate>}(R)$, където $< predicate >:=< attr >< op >< attr|const >, op \in \{=, \neq, <, >, \leqslant, AND, OR, ...\}$

Допълнителни операции

- Сечение - $R \cap S$ (бинарна, комутативна, асоциативна)
- Частно
- Съединение - $R \bowtie_C S$, където C е условието на свързване; Декартово произведение на R и S и избор на кортежите, които удовлетворяват условието C
- Естествено съединение - $R \bowtie S$; Свързване по всички атрибути с еднакви имена и автоматично отстраняване на повтарящата се колона

Бази от данни. Нормални форми.

Нормализацията на база от данни е процес, насочен към преобразуването на релационни схеми, при който се налагат ограничения към новополучените релационни схеми, елиминиращи някои нежелани свойства. Една релационна схема е в **първа нормална форма (1НФ)**, ако всеки компонент на всеки кортеж съдържа атомарна стойност. Един атрибут X от дадена релационна схема с множество от функционални зависимости се нарича **първичен атрибут**, ако е част от ключа (първичния ключ) Една релационна схема е във **втора нормална форма (2НФ)**, ако тя е в 1НФ и всеки непървичен атрибут е в пълна функционална зависимост от ключа, т.е. зависи от целия ключ, а не от негово подмножество. Нека **Аксиоми (Армстронг)**

1. Рефлексивност
   Ако $Y \subseteq X$, то $X \to Y$
2. Разширение
   Ако $X \to Y$, то $XW \to YW$
3. Транзитивност
   Ако $X \to Y$ и $Y \to Z$, то $X \to Z$

Дефиниция $K = \{A_1...A_n\}$ е **ключ** за релацията R, ако:

1. Множеството K функционално определя всички атрибути на R.
2. За нито едно помножество на K, (1) не е вярно.

Дефиниция Множеството K е **суперключ**, ако удовлетворява (1), но не и (2).

Дефиниция Релацията R е във **2NF**, ако е в $1NF$ и всеки неключов атрибут е в пълна функционална зависимост от ключа (зависи от целия ключ, а не от негово подмножество).

Дефиниция Атрибут, който е част от ключа, се нарича **първичен атрибут**.

Дефиниция Релацията R е в **3NF** $\iff$ $\forall$ ФЗ, или лявата страна е суперключ или дясната страна е първичен атрибут.

Дефиниция Релацията R е в **BCNF** $\iff$ $\forall$ нетривиална зависимост $A_1...A_n \to B_1...B_m$ от R, съответното множество от атрибути $\{A_1...A_n\}$ е суперключ за R.

Многозначни зависимости Терминът "многозначна зависимост"се използва, когато два атрибута или множество атрибути са независими помежду си. Това състояние обобщава идеята за ФЗ в този смисъл, че всяка функционална предполага съответна многофункционална зависимост. Съществуват схеми в BCNF, които съдържат излишни данни. Най-често това се получава при опит да се поставят две или повече връзки "много към много"в една релация. Това излишество е резултат от независимостта на атрибутите. Например, в Tutor/Student Cross- Reference без проблем могат да се въведат два различни номера на социална осигуровка за един и същ ръководител, а това не е желателно

Дефиниция Релацията R е в **4NF** $\iff$ $\forall$ нетривиална многозначна зависимост $A_1...A_n \to\to B_1...B_m, A_1...A_n$ е суперключ.

Изкуствен интелект

Търсене в пространството от състояния. Генетични алгоритми.

Обща постановка - Решаването на много задачи, традиционно смятани за интелектуални, може да бъде сведено до последователно преминаване от една формулировка на задачата до друга, еквивалентна на първата или по-проста от нея, докато се стигне до това, което се смята за решение на задачата.

Дефиниция **Състояние** наричаме едно описание на задачата в процеса на нейното решаване. Състоянията биват начални, междинни и крайни (целеви).

Дефиниция Оператор наричаме правило/алгоритъм, по който от дадено състояние се получава друго.

Дефиниция Пространство на състоянията наричаме съвкупността от всички възможни състояния, които могат да се получат от дадено начално състояние. Представяме пространството от състояния чрез ориентиран граф с възли - състоянията и дъги - операторите.

Основни типове задачи за търсене в ПС

Търсене на път до определена цел - търси се път от дадено начално състояние до определено целево състояние.

Според наличната информация:

- Неинформирано (сляпо) търсене - DFS, BFS, UCS, DLS, IDS
- Информирано (евристично) търсене - Best First Search, Beam Search, Hill Climbing, A*

Според разглежданото пространство:

- Глобално търсещи - DFS, BFS, UCS, IDS, Best First Search, A*
- Локално търсещи - Beam Search, Hill Climbing, Simulated Annealing, Genetic Algorithm

Характеристики на алгоритмите за търсене:

- Пълнота - един алгоритъм е пълен тогава, когато ако съществува решение, то той гарантира, че ще го намери
- Оптималност - един алгоритъм е оптимален тогава, когато ако има два или повече пътя, които водят до решение, то той гарантира, че ще намери както най-близкото решение (оптималното), така и най-късия път до това решение
- Сложност
  - по време = брой изследвани (обходени) възли
  - по памет = максимален брой възли в паметта

Параметри на търсенето:

- d - дълбочина на "най-плитката"цел
- m - максимална дълбочина на пространството от състояния
- b - коефициент на разклонение на графа на състоянията

Формиране на стратегия при игри за двама играчи

Разглеждат се т. нар. интелектуални игри с пълна информация, които се

играят от двама играчи и върху хода на които не оказват влияние случайни фактори. Двамата играчи играят последователно и всеки от тях има пълна информация за хода на играта. Най-често се решава задачата за намиране на най-добър (първи) ход на играча, който трябва да направи текущия ход.

Основни типове игри:

- с перфектна/неперфектна информация
- детерминистични/включюващи елемент на шанс
- с пълна/непълна информация

Дървото на състоянията при тези задачи е дърво на възможните позиции в резултат на възможните ходове на двамата играчи. Общият брой възли в дървото на състоянията е от порядъка на b^d

Алгоритми за решаване на задачи от този тип

- Минимаксна процедура - метод за намиране на най-добрия ход на първия играч при предположение, че другият играч също играе оптимално. Предполага се, че двамата играчи имат противоположни интереси, изразени в това, че единият търси стратегия, която води до получаване на позиция с максимална оценка (максимизиращ играч, MAX), а другият търси стратегия, която води до получаване на позиция с минимална оценка (минимизиращ играч, MIN). Минимаксната процедура е метод за получаване на оценките на възлите от по-горните нива на ДС, които позволяват на първия играч да избере най-добрия си ход
- Минимаксна процедура с α-β отсичане - при α-отсичането не е необходимо да се извършва генериране и търсене върху всяко поддърво, което произлиза от минимизиращ възел, β-стойността на който е помалка или равна на α-стойността на съответния максимизиращ родител. При β-отсичането не е необходимо да се извършва генериране и търсене върху всяко поддърво, което произлиза от максимизиращ възел, α-стойността на който е по-голяма или равна на β-стойността на съответния минимизиращ родител.

Намиране на цел при спазване на ограничителни условия

Дадени са:

- Множество от променливи $v_1, v_2, ..., v_n$ със съответни области на допустимите стойности Dv_i - дискретни (крайни или изброими безкрайни) или непрекъснати
- Множество от ограничения (допустими/недопустими комбинации от стойности на променливите)

Видове ограничения

- Унарни - ограничения, включващи една променлива
- Бинарни - ограничения, включващи две променливи
- От по-висок ред - ограничения, включващи три или повече променливи
- Основани на предпочитания (меки) - показват коя стойност е по-добра за дадена променлива

Целево състояние: множество от свързвания със стойности на променливите $v_1 = c_1, ..., v_n = c_n$, които удовлетворяват всички ограничения.

Целта на задачата е да се инициализират по такъв начина всички променливи със стойности от съответните им домейни, че и същевременно всички ограничения да бъдат удовлетворени.

Алгоритми за решаване на задачи от този тип

- Генериране и тестване
- Търсене с възврат (Backtracking)
- Разпространяване на ограничения (Constraint propagation)
  - Forward checking
  - Arc Consistency
- Локално търсещи
  - MinConflicts

Методи за информирано (евристично) търсене на път до определена цел

Реализират пълно изчерпване по гъвкава стратегия или търсене с отсичане на част от графа на състоянията. Приложими са при наличие на специфична информация за предметната област, позволяваща да се конструира оценяваща функция (евристика), която връща не булева оценка (числова оценка в предварително определен интервал). Тази оценка може да служи например за мярка на близостта на оценяваното състояние до целта или на необходимия ресурс за достигане от оценяваното състояние до целта. Най-често се използва евристична оценяваща функция h, която връща като резултат приблизителната стойност на определен ресурс, необходим за достигане от оценяваното състояние до целта.

Програмна реализация: Чрез използване на работна памет (списък на т. нар. открити възли или списък от натрупани/изминати пътища, започващи от началния възел – фронт на търсенето). Възможните състояния, които следва да се обходят се сортират въз основа оценяваща функция, която казва кое състояние е по-добро (best-first - винаги вземаме най-доброто първо).

Оценяваща функция може да е изцяло евристична (като при Greedy) или да в комбинация от евристична функция и така, която дава точна оценка (като при A*)

Greedy Best-first search

Сортиране на списъка в съответствие с евристичната функция (например h(node) = <оценка на цената на пътя от node до целта>).
Оценка на метода Best-first search:

- Методът е ефективен, но не е нито пълен (опасност от зацикляне), нито оптимален
- Времевата сложност е $O(b^m)$, но използването на подходяща евристика може да доведе до съществено подобрение
- Пространствената сложност е $O(b^m)$, тъй като се съхраняват всички достигнати състояния

Beam search

Ограничаване на списъка до първите l най-добри възела от него (в съответствие с евристиката). При $l = 1$ се доближава до метода Hill climbing.
Оценка на метода Beam search:

- Методът е ефективен, но не е нито пълен (опасност от зацикляне), нито оптимален
- Времевата сложност е $O(b^m)$, като тя зависи от разгледаното "изрязано"пространство и евристиката. Използването на подходяща евристика може да доведе до съществено подобрение.
- Пространствената сложност е $O(bl)$

Hill climbing

Списъкът се ограничава до най-добрия му елемент (в съответствие с евристиката), и то само ако той е по-добър от своя родител. Търсенето е еднопосочно, без възможност за възврат.
Оценка на метода Hill climbing:

- Методът е ефективен, но не е нито пълен (опасност от зацикляне), нито оптимален
- Времевата сложност е $O(bm)$, като тя зависи от дълбочината на пространството и евристиката.
- Пространствената сложност е $O(b)$ - константа

A*

Комбинира търсене с равномерна цена на пътя с търсене на най-добър

път. Списъкът се сортира в съответствие с функцията $f(node) = g(node) + h(node)$. Тук функцията g връща като резултат цената на изминатия път от началния възел до node, а евристичната функция h връща като резултат приблизителна стойност на цената на оставащата част от пътя от node до целта.

Оценка на метода A*:

- Стратегията е пълна, ако разклоненията (наследниците) на всеки възел от ГС са краен брой и цената на преходите е положителна, и оптимална, ако евристиката е приемлива (оптимистична), т.е. никога не надценява стойността (цената) h^* на оставащия път (ако $h^*(node) \geqslant h(node)$ за всеки възел node).
- Времевата и пространствената сложност на метода са експоненциални. Можем да кажем, че са приблизително $O(bd)$, като зависят пряко от грешката на евристичната функция и дължината на най-късият път до решението.

Генетични алгоритми

Дефиниция Генетичен алгоритъм наричаме вариант на стохастично (локално) търсене в лъч, при което новите състояния се генерират чрез комбиниране на двойки родителски състояния вместо чрез модифициране на текущото състояние.

Основни принципи:

- Състоянията се представят като низове над дадена крайна азбука. (често като 0 и 1)
- Оценяващата функция (fitness function) оценява близостта до целта на дадено състояние, като има по-големи стойности за по-добрите състояния
- Алгоритъмът започва с множество (популация) от k случайно генерирани състояния (поколение 0)

Селекция - в генерирането на състоянията от следващото поколение участват някои от най-добрите представители на текущото поколение, избрани на случаен принцип

Кръстосване - избират се двойка "родителски"състояния и се определя т.н. точка на кръстосването им. Състоянието-наследник се получава чрез конкатенация на началната част на първия и крайната част на втория родител

Мутация - извършване на случайни промени в случайно избрана малка част от новата популация с цел да се осигури възможност за достигане на

всяка точка от пространството на състоянията и да се избегне опасността от попадане в локален екстремум

Видове кръстосване:

- В единична точка - избира се една точка на кръстосване и полученият низ от началото си до точката на кръстосване е копие на началната част на единия родител, а останалата му част е копие на съответната част на втория родител
- В две точки - избират се две точки на кръстосване и полученият низ от началото си до първата точка на кръстосване е копие на съответната част от първия родител, частта на резултата от първата точка на кръстосване до втората точка на кръстосване е копие на съответната част на втория родител и останалото е копие на оставащата след втората точка на кръстосване част на първия родител
- Аритметично - извършва се определена операция (аритметична, логическа, т.н.) между двамата родители

Софтуерни технологии

Съвременни софтуерни технологии.

Дефиниция Софтуерен процес наричаме последователност от стъпки включващи дейности, ограничения и ресурси, които осигуряват постигането на някакъв вид резултат. По друг начин казано, процесът е съвкупност от процедури, организирани така че да се изграждат продукти, задоволяващи определени цели и стандарти.

Дефиниция Модел на софтуерен процес наричаме опростено описание на начина на разработване на софтуера, представено от определена гледна точка.

Модели:

1. Модел на водопада (**Waterfall**) - най-старият метод на структурирано разработване на софтуер, който предлага систематизиран, последователен подход към разработването на софтуер, който включва следните дейности: събиране на софтуерни изисквания, оценяване, изготвяне на график, проследяване, анализ и проектиране, генериране на код и тестване, доставяне и поддръжка
Предимства: ясно разграничителен процес, на всяка стъпка се документира процеса, всяка дейност трябва да се завърши напълно, докато се продължи към следващата, ясно дефинирани интерфейси между стъпките и ролите на разработчиците на софтуера
Недостатъци: моделът дава по-скоро структура на управлението на проект,

отколкото насоки как се извършват отделните дейности, произлязъл е от областта на хардуера и не отчита същността на софтуера като творчески процес на решаване на проблем, потребителят трудно формулира всичките си изисквания в началото, разделянето на проекта на отделни етапи не е гъвкаво (трудно е да се реагира на променящи се изисквания на клиента)

2. Модел на бързата разработка (**RAD**) - основата цел е кратък цикъл на разработка. Дейностите са комуникация, планиране (няколко софтуерни екипа работят паралелно), моделиране (на данните, на процес, бизнес моделиране), конструиране (използване на готови софтуерни компоненти и автоматизация на код) и внедряване.
Предимства:
Недостатъци: изисква много човешки ресурси за големи приложения, които трябва да се разделят на модули; когато функционалността на софтуерната система не може да бъде подходящо разделена в отделни модули; когато е важна високата производителност на софтуерното приложение; когато за разработката се разчита на нови и не достатъчно усвоени технологии **Дефиниция** Функционални изисквания - описание на услугите, които системата трябва да предоставя, начинът по който системата трябва да реагира на конкретни входни данни и поведението и в конкретни ситуации **Дефиниция** Нефункционални изисквания - ограничения върху услугите или функционалността на системата като времеви ограничения, ограничения върху процеса на разработване, използваните стандарти и т.н. **Дефиниция** Анализ на изискванията - проверка на извлечените изисквания дали правилно, точно, пълно, атомарно описват изискванията на клиента **Гъвкави (agile) софтуерни технологии** Extreme Programming - най-известната и широко използвана гъвкава методология

Софтуерни архитектури

Архитектури на софтуерни системи.

Дефиниция Архитектура на дадена софтуерна система е съвкупност от структури, показващи различните софтуерни елементи на системата, външно видимите им свойства и връзките между тях.

Дефиниция Компонент наричаме изчислителна (софтуерна) единица, която има определена функционалност и е достъпна чрез добре дефинирани интерфейси и има изрично специфицирани зависимости (входен и изходен интерфейс)

Дефиниция Конектор наричаме first class entity, представляващ механизъм за взаимодействие (протокол) между компонентите и правилата за комуникация (трансфер на данни)

Дефиниция Архитектурен стил наричаме семейство от системи по отношение на модел (образец) на структурна организация.

Архитектурни стилове

1. **Pipe-and-Filter** - всеки компонент в системата прехвърля данните в последователен ред към следващия компонент. Конекторите (pipes) между компонентите представляват действителните механизми за пренос на данни.

$$Encrypt \xrightarrow{send} Decrypt \xrightarrow{input} Authenticate$$

Името pipe-and-filter идва от системата Unix, където е възможно да се свържат процеси, използвайки тръби (pipes). Тръбите предават текстов поток от един процес в друг и имат задължението да прехвърлят данните от изхода на филтър до входа на следващия филтър. Филтрите представляват изчислителни единици в системата - те съдържат функционалността и четат данни чрез входни интерфейси, обработват ти и ги изпращат до изходните интерфейси. Вариации на pipe-and-filter:

- Последователна обработка - пакетна (batch) или поточна (pipeline/stream) $Filter \xrightarrow{pipe} Filter \xrightarrow{pipe} Filter$
- Паралелна обработка - $Filter \begin{matrix} \xrightarrow{pipe} \\ \xrightarrow{pipe} \end{matrix} \begin{matrix} Filter \\ Filter \end{matrix} \xrightarrow{pipe} Filter$

Предимства - лесен за внедряване и интуитивен;
Недостатъци - поради последователните стъпки на изпълнение е трудно да се реализират интерактивни приложения; ниска производителност - всеки филтър трябва да анализира данните, трудно се споделят глобални данни и филтрите трябва да се съгласуват относно формата на данните

2. **Shared data** - активно се използва в системи, където компонентите трябва да прехвърлят големи количества данни. Споделените данни могат да се разглеждат като конектор между компонентите. Вариации на shared data стила са:

- Blackboard (черна дъска) - когато се изпращат данни към конектора за споделени данни, всички компоненти трябва да бъдат информирани за това (с други думи споделените данни са активен агент)
- Repository (хранилище) - споделените данни са пасивни, до компонентите не се изпращат известия

Предимства - скалируемост (могат да се добавят лесно нови компоненти); конкурентност (всички компоненти могат да работят паралелно); високо ефективен при обмен на големи количества данни; централизирано управление на данните
Недостатъци - трудно се прилага в разпределена среда; споделените данни трябва да поддържат единен модел на данни; промените в модела могат да

доведат до ненужни разходи; може да се превърне в тясно място (bottleneck) в случай на твърде много клиенти
3. **Client-server** - системата е проектирана като набор от сървъри, които предлагат услуги и редица клиенти, които използват тези услуги. Тънък клиент (thin client) - клиентът реализира функционалността на потребителския интерфейс, а сървърът реализира функцията за управлението на данни и приложната обработка Тежък клиент (fat client) - клиентът може да внедри част от функционалността за обработка на приложения
Предимства - централизация на данните; сигурност; лесна реализация на архивиране (back-up) и възстановяване (recovery)
Недостатъци - работният товар на сървъра може да нарасне прекалено при голям брой клиенти; необходимост от излишък/отказоустойчивост (redundancy, fault-tolerance)
4. **Layered** - представя системата като организирана от йерархично-подредени слоеве
Предимства - вътрешната структура на слоевете е скрита, ако се поддържат интерфейси; абстракция - минимизиране на сложността; по-добра кохезия (всеки слой съдържа подобни задачи)
Недостатъци - за много системи е трудно да се разграничават отделните слоеве, което води до значителни усилия за проектиране; строгите ограничения за комуникация на слоя компрометират производителността

5. **Object-oriented**