

Системно програмиране

Курсова работа

***Тема №14***

*В съвременния свят всички самолети кацат, но не всичките където трябва… и с каквато трябва скорост. Напишете приложение, където близките на неприятно-кацнали хора могат да проверят дали техните близки са живи или не. Имайте предвид, че при 300 човека - пътници, то ще има доста близки, които да правят постоянно проверка. Освен ако в самолета не се возят асистентите по С.Пр. - тогава никой няма да се интересува за тях.*

*Ръководител:*

***гл. ас. д-р Антония Ташева***

***ас. инж. Боян Петров***

*ФКСУ, гр. 42, Фак № 121213099*

СЪДЪРЖАНИЕ

*Изготвили:*

***Георги Челенков***

*ФКСУ, гр. 42, Фак № 121213099*

***Георги Илиев***

*ФКСУ, гр. 42, Фак № 121213248*

*Ръководител:*

***ас. Десислав Андреев***

*ТУ-София*

*31.05.2015 г.*

[УВОД 3](#_Toc452211300)

[ГЛАВА 1 4](#_Toc452211301)

[ГЛАВА 2 6](#_Toc452211302)

[ГЛАВА 3 9](#_Toc452211303)

[ГЛАВА 4 10](#_Toc452211304)

[ЗАКЛЮЧЕНИЕ 12](#_Toc452211305)

[ПРИЛОЖЕНИЕ 12](#_Toc452211306)

# УВОД

Задание(Тема №14):

В съвременния свят всички самолети кацат, но не всичките, където трябва… и с каквато трябва скорост. Напишете приложение, където близките на неприятно-кацнали хора могат да проверят дали техните близки са живи или не. Имайте предвид, че при 300 човека - пътници, то ще има доста близки, които да правят постоянно проверка. Освен ако в самолета не се возят асистентите по С.Пр. - тогава никой няма да се интересува за тях.

За реализацията на това задание, трябва да бъдат изпълнени следните подзадачи:

* Да се проектира модел на приложение с информационни структури от данни за следните *entity*-та:

**- Летища**

**- Полети(Самолети)**

**- Пътници**

* Да се имплементират структурите с подходящи полета за съответното *entity;*
* Да се имплементира **приложение - сървър** с функции:

- да дефинира и имплементира информационните структури;

- да поддържа определен набор от *летища;*

***-*** да поддържа множество *полети* от всяко *летище;*

- да поддържа множество *пътници* за всеки *полет;*

***-***да е в състояние да приеме голям брой конекции;

-да обслужва заявки(запитвания) относно **статуса** на *полети*/ **състоянието** на *пътници.*

* Да се имплементира **приложение -клиент** с функции:

- да се свързва към дадения сървър;

- да предоставя меню за избор на типа запитване към сървъра;

- да комуникира със сървъра **двупосочно**, по време на целия си *lifecycle;*

(постоянно и неограничено да обменя съобщения в двете посоки)

-да форматира и представя отговора на запитване към сървъра в конзолата (във възможно по-опростен и нагледен формат)

# ГЛАВА 1

* Да се проектира модел на приложение с информационни структури от данни за следните *entity*-та:

**- Летища**

**- Полети(Самолети)**

**- Пътници**

Реализацията изисква употребата на **C структури от данни**(*struct*). Те са стандартен feature на езика C и употребата им не налага включване на допълнителни библиотеки в проекта.

Причината за избора на този подход е очевидна - структурите са репрезентацията на обект в C.

* Да се имплементират структурите с подходящи полета за съответното *entity*

|  |  |  |
| --- | --- | --- |
| Летище(Airport) | Полет(Flight) | Пътник(Passenger) |
| * Масив от полети * Име на летище * Уникален идентификатор | * Масив от пътници * Дестинация * Флаг за статус на полет * Уникален идентификатор | * Номер на място(седалка) * Пълно име на пътника * Уникален идентификатор |
| air.png | fli.png | 2016-05-28 12_38_51-Debian x64 [Running] - Oracle VM VirtualBox.png |

* Да се имплементира **приложение - сървър**

Реализацияата на структурите от данни, поддръжката (зареждането им с реалните стойности не представлява интерес, затова при настоящата демонстрация са въведени програмно).

За реализацията на самия сървър - ползван e UNIX *socket* по познатия алгоритъм:

**create socket -> bind to port -> listen -> accept -> receive/send -> shutdown/close**

Необходимо е импортирането на библиотеките:

* *sys/socket.h* - за UNIX socket-ите
* *arpa/inet.h* - за структурата(*inet\_addr*)
* *unistd.h* - за *read/write* операции.

Причината да изберем socket пред pipe е, че задачата налага комуникация с отдалечен сървър по IP, а не комуникация между вътрешни за сървъра процеси. Освен това те превъзхождат pipe-овете и по други важни показатели - позволяват обмен на данни между множество процеси и поддържат двупосочен обмен за разлика от pipe(ако ползваме тях ще ни трябват 2 - по един за всяка посока).

Приемането на връзките се осъществява, като за всеки клиент се създава нова POSIX нишка (*pthread*).

Този подход е инстинктивно най-правилното решение що се отнася до достъп, поддръжка на **множество** опити за конекции **едновременно и независимо една от друга**.

Налага се включването на библиотека *pthread.h* в проекта, както и компилиране на кода с *-pthread* аргумент.

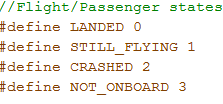
* Да се имплементира **приложение - клиент**

За реализацията на клиентското приложение - ползван e UNIX *socket* по познатия алгоритъм:

**create socket -> connect to server -> receive/send -> shutdown/close**

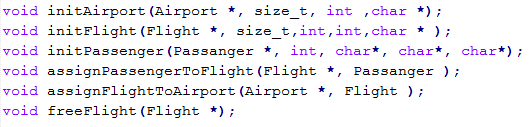
Имплементиран е безкраен цикъл с възможност за изход при определен избор(*case*) от менюто - в нашия случай - "0".

Изборът, предлаган за момента, е запитване( case = "1" ) за статус на полета/състояние на пътник. Възможните състояния са дефинирани предварително



# ГЛАВА 2

Сървър

Структурите за основните "обекти" - Летище, Полет, Пътник са предварително инициализирани с данни от допълнителни функции, извикани в основната нишка на изпълнение(***main***) преди имплементацията на стандартния алгоритъм за запускане на сървър.

Базовата имплементация необходима за пускането на сървър е по стандартен модел, като отговарящите в програмата функции/блокове са:

* ***create socket***

create_sock.png

* ***bind to port***

bind.png

* ***listen***

listen.png

* ***accept***

accept.png

* ***receive****/****send***

read.png

write.png

* ***shutdown****/****close***

close.png

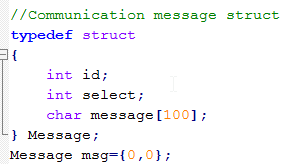
Особености по базовата имплементация няма освен зададената дължина на опашката за чакащи accept връзки, която, в случая е конфигурирана на 3. Така сървърът може да приема голям брой конекции, като същевременно не позволява опашката от чакащи да стане прекалено дълга.

След приемането на конекция, сървърът създава нишка за току-що свързалия се клиент.

thread.png

Нишката е със стандартна конфигурация(няма подадени допълнителни атрибути) и изпълнява зададена *handler* функция, която осъществява обработката на запитването от клиента, а след това връща и отговора.

Комуникацията представлява структура за съобщение, унифицирана като формат за обмен между *сървъра* и *клиента*:



Съдържа уникален идентификатор за съобщението, избора на клиента, както и текстово съобщение, което да бъде предадено за показване.

Алгоритъмът, по който действа handler-функцията е следният:

**read***(while read message size > 0)* **-> determine client selection -> send back response**

В случай, че клиентът е избрал да провери статуса на полет/ състоянието на пътник, се извиква функция със специално предназначение(*API*):

api.png

Тя претърсва пътниците на всички полети от съответното летище, намира   
търсения от клиента пътник и връща код за неговото състояние.

*Handler* функциятата, на свой ред, изготвя подходящо съобщение според получения код, форматира го в стандартизираната структура за съобщение и го изпраща като отговор за показване при клиента.

Клиент

Базовата имплементация необходима за пускането на *клиент* е по стандартен модел, като отговарящите в програмата функции/блокове са:

* ***create socket***

cr_sock.png

* ***connect to server***

conn.png

* ***receive*/*send***

snd.png

recv.png

* ***shutdown/close***

cls.png

Алгоритъмът, по който действа *клиент приложението* е следния:

**communicate-> determine selection-> send(request)-> receive(response)-> show message**

*(until exit signal)*

Поддържа се цикъл с условие за изход - подаден сигнал за изход(*signalExit*).

sigExit.png

На всяка итерация се показва менюто, предназначено за *клиента*, което, в базовия си вид, му дава избор между **изход**(*case = 0*) и проверка **статуса на полет/състоянието на пътник**(*case = 1*).

При избрана последната опция, се изпраща сигнал към *сървъра*, който да извести за направения избор и да предизвика изпращане на подходящ отговор. След това този отговор се **чете** и **визуализира** от *клиентското* *приложение*.

# 

# ГЛАВА 3

Сървър

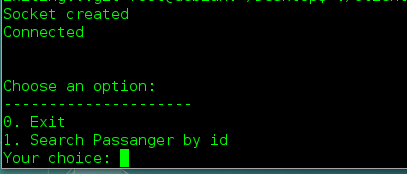
Не се налагат специални инструкции за ползване на сървърното приложение. При правилна конфигурация, то функционира без нужда от намеса.

Клиент

При пускане на клиентското приложение веднага се прави опит за свързване към конфигурирания сървър.

Ако това не стане по някаква причина, като например сриване на сървъра, блокировка при спиране на стара инстанция на сървъра - зает порт), се получава подходящо известие в конзолата на клиента.

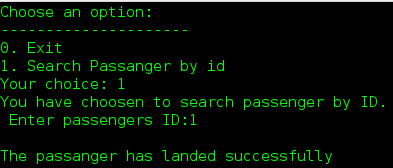
В случай че свързването се осъществи, се показва менюто за клиента и всичко, което се изисква от него е да избере опция и да следва указанията.



Възможните сценарии на този етап са следните:

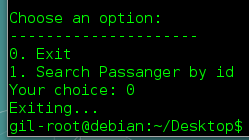
* *клиента* иска да отправи запитване

**пускане -> избор "1" -> въвеждане на ID на пътник(номер на седалка)**

****

* *клиента* иска да спре приложението

**пускане -> избор "0"**



# ГЛАВА 4

Сървър

Възможните слаби места на *сървъра* са следните:

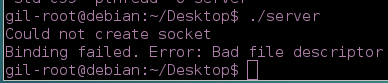
* **неуспешно създаване/*bind*-ване на *socket***

Много рядко - възможните причини са малко и има малка вероятност да се случи (грешни параметри, подадени на функцията за създаване/недостиг на ресурси в системата). Предвидена е проверка за всеки случай.

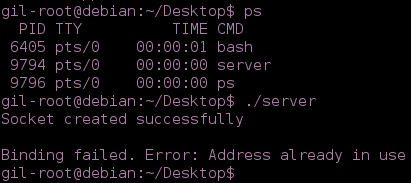
При подаден невалиден протокол на функцията за създаване

invProto.png

се задействат съответно валидациите за създаване и *bind*-ване на *socket*

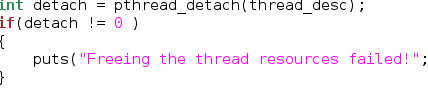


Също при зает порт(блокирана инстанция) се активира подходяща валидация.



* **неуспешно създаване на нова нишка за клиент след определен брой завършили изпълнението си стари нишки**

Причина за това може да е неуспешно освобождаване на ресурсите заемани от вече терминираните нишки. За да се избегне такава ситуация е предвидено експлицитно *detach*-ване на нишката след създаването й.



* **предвидени са и стандартни валидации на всяко четене/пращане**

Клиент

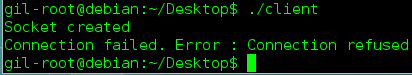
Възможните слаби места на *клиента* са следните:

* **неуспешно създаване на *socket* - приложима е снимката от валидацията за сървъра**
* **неуспешно свързване към *сървър***

Най-вероятната причина е разминаване в конфигурирания порт с този от сървъра. Например при порт *8888* на *сървъра* и този на *клиента*:

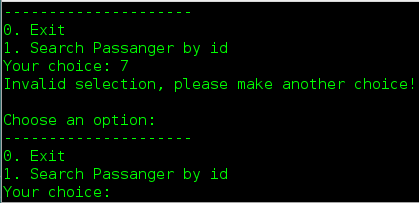
ipwrong.png

се задейства следната валидация



* **невалиден избор от менюто за клиента**

Избирането на несъществуващ case в менюто задейства валидация



# ЗАКЛЮЧЕНИЕ

Имайки предвид представеното задание, реализацията на приложенията за клиент и сървър покрива напълно заложените изисквания и очакваните резултати.

За съжаление има сериозен пропуск - липсва ключовата валидация дали сред пътниците няма асистенти по СПР ☺ .

# ПРИЛОЖЕНИЕ

**Github:** https://github.com/Chelenkoff/passenger-flight-checker