Exam Assignments

Operating Systems 2

TNIBES01-2

by

dr. W. M. Bergmann Tiest & dr. ir. M. Hajian

Course year: 2017-2018

CMI
Department of Technical Informatica

HOGESCHOOL ROTTERDAM

Contents

Chapter 1

Assignment 1: Input/output basics

To see what comes with an asynchronous input/output system, we are going to implement a telegraph as an input / output system on the Raspberry Pi. All communication is executed using the Morse code. Morse code consists of short (100∼ms) and long (300∼ms) pulses; see attached table. There is a pause of 100 ms between pulses (as long as a period). Between letters there is a pause of 300∼ms (3 points), and between words a pause of 700∼ms (7 points).

1.1 Assignment: Morse input / output system (hand-in assignment)

Connect a LED with a resistor of 330Ω to GPIO0 (physical pin 11) as the output. Connect a switch to GPIO2 (physical pin 13) as an input. Ground is GND (physical pin 9). Then get started with the C++ program `input_output.cpp`. With this program there is also a `Makefile`, you only need to give the command `make` to compile and link the program.

The program contains the following functions, which must be completed:

- `main()`: The input and output pin are set here. The input pin is configured as `PUD_UP` to enable the internal pull-up resistor. The function `wiringPiISR()` is used to set the *interrupt service routine*(ISR) for both transitions, i.e. from low to high and from high to low. By using an ISR it is possible to simultaneously process input and output (asynchronous input/output).
- `dot()`: transmit a point followed by a pause.
- `dash()`: transmit a line followed by a pause.
- `wait(int n)`: wait for the duration of n points.

Table 1.1: Morse Code Table

Character	Morse Code	Number	Morse Code
A	· –	0	– – – – –
B	– · · ·	1	· – – – –
C	– · – ·	2	· · – – –
D	– · ·	3	· · · – –
E	·	4	· · · · –
F	· · – ·	5	· · · · ·
G	– – ·	6	– · · · ·
H	· · · ·	7	– – · · ·
I	· ·	8	– – – · ·
J	· – – –	9	– – – – ·
K	– · –	. (point)	· – · – · –
L	· – · ·	, (comma)	– – · · – –
M	– –	? (question mark)	· · – – · ·
N	– ·	’ (apostrophe)	· – – – – ·
O	– – –	! (exclamation mark)	– – – · · ·
P	· – – ·	/ (slash)	– · · – ·
Q	– – · –	((open parenthesis)	– · – – ·
R	· – ·	) (close parenthesis)	– · – – · –
S	· · ·	& (ampersand)	· – · · ·
T	–	: (colon)	– – – · · ·
U	· · –	; (semicolon)	– · – · – ·
V	· · · –	= (is equal)	– · · · –
W	· – –	+ (plus)	· – · – ·
X	– · · –	– (minus/dash)	– · · · · –
Y	– · – –	” (quotation mark)	· – · · – ·
Z	– – · ·	@ (at sign)	· – – · – ·

- `sendMorse(char character)`: transmit an ASCII character as a morse code.

- `switchChanged()`: An interrupt service routine is called when there is a transition in the input. This routine stores the time of the transition in a `static` variable. This allows the time difference between the current and the previous transition to be determined. Then it is checked whether the transition was from high to low (switch is pressed) or from low to high (switch is released). In the first case, the calculated time indicates how long it has been since the last pulse ended. If that is more than 200 ms, the character is ready and the function is called for decrypting. The receive buffer can then be deleted. If it is more than 600 ms, the word is finished and a space must be added. In the second case, the calculated time indicates how long the pulse has lasted. It is then checked whether the pulse lasted more or less than 200 ms, and a stripe or a dot is added to the receiving buffer respectively.

- `decodeMorse(string s)`: decode an incoming morse sign according to attached tree structure, see Figure 1.1. Can you see how this routine works? Can you extend it with the numbers and punctuation marks?

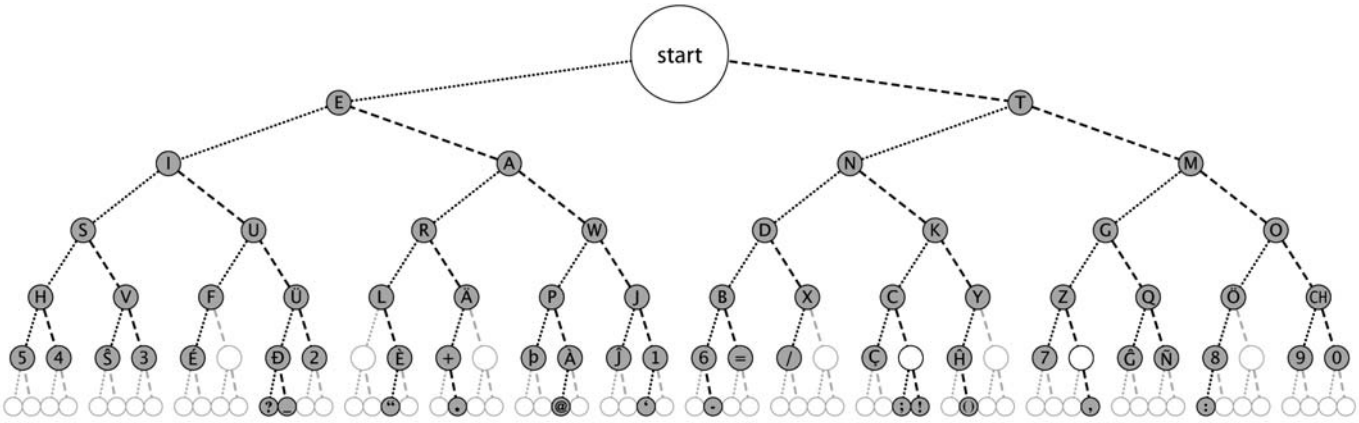

Figure 1.1: Morse Code Tree.

Complete the program such that it fulfilled the following requirements:

1. Convert the text that enters the stream `cin` to the Morse code and transmit it via the connected LED. Note: the procedure described above only processes a signal when a next pulse is received. This is acceptable in order not to make the code too complex. Give an asterisk (*) for unrecognized characters.

Chapter 2

Assignment 2: Process Table

Due to lack of time it was not possible to write the assignment in English.

2.1 Introduction

In dit onderdeel simuleren we het starten en stoppen van verschillende processen, en het parallel uitvoeren van meerdere processen. Je krijgt opdrachten via de standard input en je moet een *process table* bijhouden. Om het simpel te houden wordt hier, in tegenstelling tot de process table in Linux, niet bijgehouden bij welke gebruiker de processen horen. Wel moeten er meerdere processen tegelijkertijd uitgevoerd worden, waarbij van ieder proces telkens één instructie wordt uitgevoerd, zodat alle processen aan bod komen. Voor het uitvoeren van programma's gaan we ervan uit dat het programma al in het geheugen geladen is. Je moet ervoor zorgen dat de instructies van het programma uitgevoerd worden. Daarvoor heb je de functie `execute()` tot je beschikking, waaraan je het adres van de uit te voeren instructie meegeeft. Na iedere uitgevoerde instructie wordt het adres van de volgende instructie teruggegeven. De lengte van het programma kan vari"eren.

De volgende functies zijn beschikbaar in de library `tinbes`:

`long execute(long addr)` voert de instructie uit op geheugenlocatie `addr`. Geeft het adres van de volgende instructie van het programma terug, of 0 als het programma ge"eindigd is.

`void unblockCin()` stelt de console input in op non-blocking, dat wil zeggen: het programma wacht niet met het lezen van input totdat er op Enter gedrukt is.

`bool charactersAvailable()` geeft `true` als er karakters klaarstaan in de buffer van de console input; anders `false`.

`string readLine()` geeft een string met alle karakters uit de buffer van de

console input tot de eerstvolgende newline. Het newline-karakter wordt hierbij weggelaten.

2.2 Inleveropdracht

De volgende functionaliteit moet worden gerealiseerd:

- maak een nieuw proces
- start een proces
- pauzeer een proces
- be"eindig een proces
- lijst met processen

Hiervoor zijn in het programma `processes.cpp` de volgende onderdelen aanwezig, die aangevuld moeten worden. Er is ook een `Makefile`, zodat je alleen het commando `make` hoeft te geven om het programma te compileren en te linken.

- Global variables: Maak een process table. Deze moet voor ieder proces bevatten: de naam, het id-nummer, de status (running/paused), en het executieadres van het proces. Namen zijn maximaal 10 tekens; alle ASCII-waarden zijn toegestaan behalve 0. Er zijn maximaal 25 processen.
- `main()`: Check regelmatig of er data op de input aanwezig is met de functie `charactersAvailable()`. Als dat zo is, lees dan de string met `readLine()` en voer dan de bijbehorende actie uit. Zoniet, ga dan verder met het uitvoeren van processen.
- `newProcess()`: Op de input krijg je het woord `RUN` gevolgd door een newline, de naam van het proces gevolgd door een newline, en het adres van de eerste instructie gevolgd door een newline. Maak een nieuw proces met deze naam en geef het id-nummer van het proces terug op de output.
- `executeProcesses()`: Laat van alle lopende (niet-gepauzeerde) processen één voor één een instructie uitvoeren. Als een proces eindigt, verwijder het dan uit de process table en geef een melding terug op de output dat het proces ge"eindigd is met de naam van het proces.
- `removeProcess(int i)`: Haal het proces met het opgegeven regelnummer uit de tabel en zorg dat de tabel weer aaneengesloten is.
- `listProcesses()`: Op de input krijg je het woord `LIST` gevolgd door een newline. Geef een lijst terug van alle processen, met id-nummer, naam, en status.

- `suspendProcess(int i)`: Op de input krijg je het woord `SUSPEND` gevolgd door een newline, en een id-nummer gevolgd door een newline. Pauzeer het bijbehorende proces. Als het proces niet bestaat of al gepauzeerd is, geef dan een foutmelding terug.
- `findProcess(int i)`: Zoek het proces met het opgegeven nummer in de process table. Als het bestaat, geef dan het regelnummer terug; zoniet, geef dan –1 terug.
- `resumeProcess(int i)`: Op de input krijg je het woord `RESUME` gevolgd door een newline, en een id-nummer gevolgd door een newline. Herstart het bijbehorende proces. Als het proces niet bestaat of al actief is, geef dan een foutmelding terug.
- `killProcess(int i)`: Op de input krijg je het woord `KILL` gevolgd door een newline, en een id-nummer gevolgd door een newline. Be"eindig het bijbehorende proces en verwijder het uit de process table. Als het proces niet bestaat, geef dan een foutmelding terug.

Chapter 3

Assignment 3: Memory allocation table

In dit onderdeel implementeren we het alloceren van geheugen en het opslaan en weer terugzoeken van verschillende datatypes. Data kan alleen opgeslagen worden in gealloceerde geheugengebieden, dus voordat er data opgeslagen kan worden, moet de gebruiker eerst een geheugengebied aanvragen. Hierdoor wordt voorkomen dat het ene proces data van het andere proces kan veranderen. Verder moeten de verschillende datatypes omgezet worden naar bytes om ze te kunnen opslaan. Om de data weer op te vragen, moeten tenslotte de verschillende datatypes weer gereconstrueerd worden uit losse bytes.

De volgende functies zijn beschikbaar in de library `tinbes`:

`void storeByte(byte b, long addr)` slaat een byte op in een geheugenplaats

`byte recallByte(long addr)` haalt een byte op uit een geheugenplaats

`long getMem()` geeft de totale grootte van het geheugen terug

`string readLine()` geeft een string met alle karakters uit de buffer van de console input tot de eerstvolgende newline. Het newline-karakter wordt hierbij weggelaten.

De volgende functionaliteit moet worden gerealiseerd:

- alloceer n bytes geheugen
- geef geheugen vrij
- sla een character op (1 byte)
- sla een integer op (2 bytes, two's complement-notatie)

- sla een floating-point-getal op (4 bytes, IEEE 754-notatie)
- sla een string op
- zoek een character op
- zoek een integer op
- zoek een floating-point-getal op
- zoek een string op

Hiervoor zijn in het programma `memory.cpp` de volgende onderdelen aanwezig, die aangevuld moeten worden. Er is ook een `Makefile`, zodat je alleen het commando `make` hoeft te geven om het programma te compileren en te linken.

- Global variables: Maak een memory allocation table. Deze moet bijhouden wat het beginadres en de grootte van de gealloceerde geheugengebieden zijn.
- `allocate(long size)`: Op de input krijg je het woord `ALLOC` gevolgd door een newline, en een getal gevolgd door een newline. Alloceer de aangegeven hoeveelheid geheugen en geef het beginadres terug. Geef een foutmelding als er niet genoeg ruimte is. Deze functie maakt gebruik van `findFreeSpace()`.
- `findFreeSpace(long size)`: Ga na of er een vrij geheugengebied is van tenminste de aangegeven grootte. Hiervoor moet je een gesorteerde lijst maken van de beginadressen van de reeds gealloceerde stukken geheugen en hun groottes, en het eerste aaneengesloten stuk vrije ruimte tussen twee opeenvolgende gealloceerde gebieden (of tussen een gealloceerd gebied en het eind van de beschikbare ruimte) vinden dat groot genoeg is. Als zo'n gebied er is, geef dan het beginadres terug. Zoniet, geef dan –1 terug.
- `sort()`: Sorteer de tabel op beginadres.
- `deallocate(long a)`: Op de input krijg je het woord `DEALLOC` gevolgd door een newline, en een adres gevolgd door een newline. Check of het adres klopt en dealloceer het aangegeven gebied. Geef een foutmelding terug als het niet klopt.
- `freeMem()`: Op de input krijg je het woord `FREEMEM` gevolgd door een newline. Geef de totale hoeveelheid vrij geheugen terug.
- `store(long a, string type, string data)`: Op de input krijg je het woord `SET` gevolgd door een newline, dan een adres gevolgd door een newline, dan het type (één van de woorden `CHAR`, `INT`, `FLOAT` of `STRING`) gevolgd door een newline, en dan een tekst gevolgd door een newline. De tekst bevat ofwel een getal, ofwel een string, waarbij het newline-karakter

vervangen is door `\n`. Check of het adres in een gealloceerd geheugengebied ligt, en of het datatype erin past, en sla het dan op. Geef anders een foutmelding terug. Characters kunnen direct als enkele bytes opgeslagen worden; integers moeten als 16-bits two's complement opgeslagen worden, floating-point-getallen als 32-bits single-precision IEEE 754, en strings als een serie bytes gevolgd door een byte met waarde 0.

- `retrieve(long a, string type)`: Op de input krijg je het woord `GET` gevolgd door een newline, dan een adres gevolgd door een newline, dan één van de woorden `CHAR`, `INT`, `FLOAT` of `STRING` gevolgd door een newline. Check of het adres in een gealloceerd geheugengebied ligt, en haal het bijbehorende getal of de string op, en geef dit weer op de output.

Chapter 4

Assignment 4: File Systems

In this assignment we will implement a virtual file system. Most filesystems have a directory tree with multiple layers, but for this assignment we only use one layer. The files are stored on a virtual disk of the Raspberry Pi memory. The locations of the files are stored in a *file allocation table* (FAT). The following functions are available in the library `tinbes`:

`int writeByte(int b, long addr)` : write a byte to an address on the virtual disk. Returns 0 for success, otherwise –1.

`int readByte(long addr)` : read a byte from an address on the virtual disk. Returns the value for success, otherwise –1.

`long getSize()` : returns the total number of available bytes.

`string readLine()` : gives to the next new line a string with the characters from the buffer which have been received from the console input. The new line character is therefor omitted.

4.1 Deliverables: File allocation table

The following functionality should be realized:

- write to a file.
- read from a file.
- delete a file.
- copy a file.
- rename a file.
- print a list of the given files and their size.

- geef de maximale grootte van een op te slaan bestand weer
- organize the location of the files on the virtual disk such that all the empty spaces become contiguous.

Hence the following items in the program `file_system.cpp` are given, which must be completed. There is also a `Makefile`, so by using the command `make` you can compile and link the program.

- Global variables: Create a file allocation table (FAT). This must keep track of the start_address and the size of the stored files. File names are up to 10 characters; all ASCII values are allowed except 0. File length is up to 1024 bytes. Maximum number of files is 25.
- `store(long fileSize, string fileName, char *data)`: At the input you get the word `STORE` followed by a newline, a filename followed by a newline, then the length followed by a newline, and then the contents of the file. Save this file in your filesystem. Give an error message if there is not enough free space or if the file name already exists. This function uses `findFreeSpace ()` and `findFile ()`.
- `findFile(string fileName)`: Check whether a file with the specified name exists. If so, return the index in the FAT. If not, return -1.
- `retrieve(string fileName)`: At the input you get the word `RETRIEVE` followed by a newline and a filename followed by a newline. Display the contents of the file on the output. Give an error message if the file does not exist.
- `erase(string fileName)`: At the input you get the word `ERASE` followed by a newline and a filename followed by a newline. Delete this file. Give an error message if the file does not exist.
- `copy(string fileName1, string fileName2)`: At the input you get the word `COPY` followed by a newline, a filename followed by a newline, and another filename followed by a newline. Copy the contents of the first file into a new file with the second file name. Give an error message if the file does not exist, if there is not enough free space, or if there is already a file with the second name. This function uses `retrieve()` and `store()`.
- `rename(string fileName1, string fileName2)`: An the input you get the word `RENAME` followed by a newline, a filename followed by a newline, and another filename followed by a newline. Change the name of the first file to the second file name. Give an error message if the file does not exist, or if there is already a file with the second name.
- `files()`: At the input you get the word `FILES` followed by a newline. Display a list of the files present and their size.

- `freespace()`: At the input you get the word `FREESPACE` followed by a newline. Display the size of the largest possible file to be saved.
- `pack()`: At the input you get the word `PACK` followed by a newline. change the format of the files in your filesystem in such a way that the free space becomes a continuous piece.