13. Sistēmprogrammēšanas elementi

Nodaļas saturs:

13.1. Pavedienprocesi

Ar sistēmprogrammēšanu šeit tiek saprastas tādas programmēšanas aktivitātes, kurās izmantojamās konstrukcijas nav C++ standarts, bet ir platformas (respektīvi, sistēmas) atkarīgas. Viena no šādām tēmām, ar ko saistās sistēmprogrammēšana, ir **paralēlie procesi**. Šajā nodaļā tiks aplūkoti t.s. viegla svara (*lightweighed*) procesi jeb **pavedienperocesi** (*threads*), kas raksturojas ar to, ka tiem netiek izdalīti neatkarīgi resursi un kam neatkarīga no citiem (paralēla) ir tikai to izpilde. Vienas programmas ietvaros ar pavedienprocesiem parasti pietiek, lai realizētu (šķietami) paralēlu darbību virkņu izpildi.

Kad kāda programma tiek palaista, tad priekš tās automātiski tiek ieveidots viens process (galvenais process), un programmas pašas ziņā ir – vai palaist vēl kādus papildus procesus paralēli galvenajam.

Kaut arī sintakse starp dažādām platformām atšķiras, tomēr galvenie principi ir līdzīgi. Šeit aplūkotas tikai pašās vienkāršākās konstrukcijas, un tās ir:

- procesa izveidošana (palaišana);
- procesu sinhronizēšana un pabeigšana.

Tālāk parādītie piemēri Windows un Linux vidēs palaiž divus paralēlos procesus (sauksim tos par A un B), un katrs no šiem procesiem izdrukā uz ekrāna vērtības no 0 līdz 3 (pievienojot priekšā savu identifikatoru, attiecīgi A vai B), tad katrs procesa beigās izdrukā, ka ir beidzies, bet kad beigušies abi procesi, tiek izdrukāts arī šāds paziņojums (sk. pirmkoda piemērus 13.1 un 13.2).

13.1.1. Pavedienprocesi Windows standartā

Lai Windows platformā strādātu ar procesiem, jāielādē bibliotēka <*windows.h*>.

Windows platformā (sk. pirmkoda piemēru 13.1) viens no būtiskiem jēdzieniem programmēšanai ir t.s. *handle* (‘rokturis’). Tā ir sava veida **norāde** (*pointer*) uz Windows specifiskiem objektiem, un tā to sauksim arī turpmāk materiālā. Lai Windows sistēmā strādātu ar pavedienprocesiem, vispirms ir jādefinē norādes uz tiem (sk. 10 rindu pirmkodā).

Rindās 13 un 14 tiek attiecīgi palaisti 2 paralēlie procesi. To veic ar funkcijas *CreateThread* palīdzību. Svarīgākie principi funkcijas *CreateThread* izsaukumā:

- Paralēlā procesa veidošana pamatā nozīmē funkcijas izsaukumu, Windows vidē ir noteikts, ka tā vienmēr ir funkcija ar atgriežamo tipu *(DWORD WINAPI)* un tieši vienu parametru ar tipu *(void *)*.
- Mūsu piemērā šī funkcija ir process, kas definēta rindās 25-33.

- Izsaucamo funkciju padod *CreateThread* parametrā #3, bet tai nododamo parametru parametrā #4 (kā redzams mūsu piemērā, funkcijai tiek padota norāde uz mūsu doto nosaukumu procesam (A vai B).
- Bez tam Windows sistēma pieprasa padot papildus procesa identifikatora informāciju caur parametru #6), kas, manuprāt, ir liekvārdība, jo pēc procesa palaišanas procesu identificē attiecīgā norāde (*handle*).
- Funkcija atgriež norādi uz izveidoto procesu.

Funkcija *WaitForMultipleObjects* nodrošina, ka programma neiet tālāk, kamēr visi pakārtotie procesi, kas tiek padoti masīvā formā caur parametru #2, nav beigušies – šādi tiek nodrošināta sinhronizācija.

Pēc procesa darbības beigām jāpaziņo sistēmai par resursa atbrīvošanu (funkcija *CloseHandle*), kas pēc būtības ir kaut kas līdzīgs faila aizvēršanai.

Funkcijā process tiek papildus izmantota sistēmas funkcija *Sleep*, kas nodrošina gaidīšanu attiecīgo skaitu milisekunžu.

Pirmkods 13.1. Paralēlie procesi Windows vidē (*sys1threadswin.cpp*)

```
01 #include <windows.h>
02 #include <iostream>
03 using namespace std;
04
05 const int ITERATION_COUNT = 4;
06 DWORD WINAPI process (void *ptr);
07
08 int main()
09 {
10     HANDLE thread_a, thread_b;
11     char id_a={'A'}, id_b={'B'};
12     DWORD id2_a, id2_b;
13     thread_a = CreateThread (NULL, 0, process, (void*)&id_a, 0,
                  &id2_a);
14     thread_b = CreateThread (NULL, 0, process, (void*)&id_b, 0,
                  &id2_b);
15     HANDLE tt[2];
16     tt[0] = thread_a;
17     tt[1] = thread_b;
18     WaitForMultipleObjects (2, tt, TRUE, INFINITE);
19     CloseHandle (thread_a);
20     CloseHandle (thread_b);
21     cout << "END OF " << id_a << " " << id_b << endl;
22     return 0;
23 }
24
25 DWORD WINAPI process (void *ptr)
26 {
27     for (int i=0; i<ITERATION_COUNT; i++)
28     {
29         cout << *((char*)ptr) << i << endl;
30         Sleep (rand()%2*100);
31     };
32     cout << "End of the thread " << *((char*)ptr) << endl;
33 }
```

Programmas darbības piemērs:

```
A0
B0
A1
B1
A2
A3
B2
B3
End of the thread End of the thread B
A
END OF A B
```

Pamēģiniet laist programmu no komandrindas vairākkārtīgi, iegūstot dažādus rezultātus (funkcijā process notiek gadījuma skaitļu izmantošana), bez tam dažādu rezultātu iegūšana parāda, ka, pirms tas netiek speciāli pateikts (šeit ar *WaitForMultipleObjects*), tikmēr nekāda sinhronizācija starp procesiem nenotiek.

13.1.2. Pavedienprocesi POSIX standartā

Atšķirībā no Windows standarta, POSIX nav kādas konkrētas operētājsistēmas standarta, bet gan standarts, kuru pēc tradīcijas atbalsta *Unix/Linux* sistēmas.

Pirmkoda piemērs 13.2 parāda to pašu programmu, kas iepriekš, tikai POSIX variantā (attiecīgi to var nokompilēt tikai *Unix/Linux* sistēmā).

Lai POSIX standartā strādātu ar procesiem, jāielādē bibliotēka <*pthread.h*>.

Pēc līdzības ar Windows *handle*, vispirms ir jāizveido procesu objekti ar tipu *pthread_t* (rinda 9).

Procesu palaiž ar funkciju *pthread_create* (rindas 12 un 13), kas izsauc funkciju, kam ir viens parametrs *(void *)* ar atgriežamo tipu *(void *)*. Funkcija un tās parametrs tiek padots caur parametriem #3 un #4. Funkcija *pthread_create* caur parametru #1 inicializē procesa objektu.

Sinhronizāciju POSIX standartā veic ar funkciju *pthread_join* (rindas 14 un 15), un tas tiek veikts katram sinhronizējamam procesam atsevišķi (nevis ar vienu kopēju komandu, kā ir *Windows*).

Pirmkods 13.2. POSIX paralēlie procesi (*sys2threadsposix.cc*)

```
01 #include <iostream>
02 #include <pthread.h>
03 using namespace std;
04
05 void *process (void *ptr);
06
07 int main()
08 {
09     pthread_t thread_a, thread_b;
10     char id_a='A', id_b='B';
11     char *res_a, *res_b;
12     pthread_create (&thread_a, NULL, process, (void*)&id_a);
13     pthread_create (&thread_b, NULL, process, (void*)&id_b);
14     pthread_join (thread_a, (void**)&res_a);
15     pthread_join (thread_b, (void**)&res_b);
```

```
16      cout << "END OF " << *res_a << ' '<< *res_b << endl;
17      pthread_exit (NULL);
18      return (0);
19  }
20
21  void *process (void *ptr)
22  {
23      for (int i=0; i<4; i++)
24      {
25          cout << "" << *((char*)ptr) << i << endl;
26          sleep (rand()%2);
27      };
28      cout << "End of the thread " << *((char*)ptr) << endl;
29      return ptr;
30  }
```

Programmas darbības piemērs:

```
A0
B0
B1
A1
B2
A2
B3
End of the thread B
A3
End of the thread A
END OF A B
```

13.2. Logu programmēšana Windows

Nākošais piemērs demonstrē vienkāršu logu programmu Windows vidē, kas dara sekojošas darbības vai spēj veikt sekojošus uzdevumus:

- Izveido logu ar nosaukumu „The Hello World program”;
- Loga pirmajā rindiņā izvada tekstu „Hallo World”, kuru pārzīmē arī mainot loga izmērus vai aktivizējot logu;
- Loga otrajā rindiņā skaita sekundes kopš programmas palaišanas;
- Loga trešajā rindiņā izvada nospiestā taustiņa kodu;
- Var tikt aizvērta arī, nospiežot taustiņu *Esc*.

Lai palaistu programmu, projekta tipam Dev-C++ vidē jābūt *Win32 GUI* (pierastā *WIN32 Console* vietā, sk. *sys3window.dev*), tāpēc tas nav iespējams ārpus Dev-C++ projekta (pa taisno no *cpp* faila).

Programma sastāv no šādām galvenajām daļām (pirmkoda piemērs 13.3):

- Funkcija *WinMain* (rindas 10-59) Windows sistēmā aizstāj pierasto *main*. Tādējādi Windows sistēmā C++ vairs nav „tīrs”, bet ir ticis inkorporēts sistēmā.
- Tiek inicializēta struktūra loga izveidošanai (rindas 17-32).
- Tiek izveidots logs (rindas 34-48).
- Visa programmas darbība ir „ķert” notikumus un tos apstrādāt, ko reprezentē t.s. notikumu cikls (rindas 50-55).

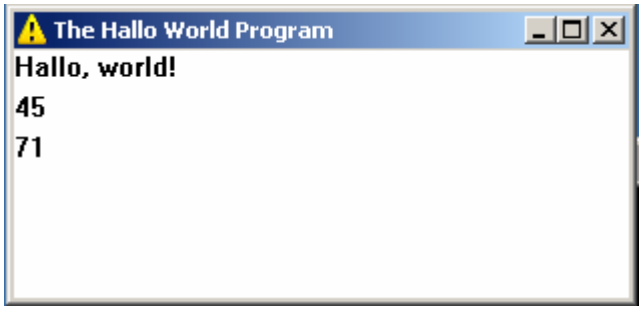
- Katra atsevišķa notikuma apstrādi veic funkcija *WndProc* ar fiksētu interfeisu (rindas 7, 20, 61-125). Šo funkciju (atšķirībā no *WinMain*) drīkst nosaukt citādi, tikai tas attiecīgi jāpiereģistrē (rinda 20).

Funkcijā *WndProc* redzami galvenie *Windows* notikumi:

- WM_CREATE – loga izveidošana;
- WM_PAINT – loga pārzīmēšana (piemēram, mainot izmēru vai aktivizējot), ;
- VM_TIMER – taimera notikums;
- VM_DESTROY – loga likvidēšana;
- VM_KEYDOWN – taustiņa nospiešana, funkcijas parametrs #3 (šeit *wParam*) nosaka nospiestā taustiņa kodu.

Teksta izvadei uz ekrāna tiek lietota funkcija *TextOut*.

Lai veiktu izvadi logā, Windows sistēmā tiek noteiktā veidā izmantots t.s. aparatūras konteksts (*device context*, *DC*) un zīmēšanas parametru konfigurēšanas struktūra (*paint structure*), rindas 66, 77-80, 85, 90, 93, 112-115.



Attēls 13.1. Programmas (13.3) darbības piemērs

Pirmkods 13.3. Vienkāršs logs Windows sistēmā (*sys3window.cpp*)

```
01 #include <windows.h>
02 #include <string>
03 #include <stdio.h>
04
05 using namespace std;
06
07 LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
08                                      LPARAM lParam);
09
10 int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
11                         LPSTR lpszCmdLine, int nCmdShow)
12 {
13      HWND          hWnd;
14      MSG               msg;
15      WNDCLASSEX     wc;
16
17      //fill the WNDCLASSEX structure with the appropriate values
18      wc.cbSize = sizeof(WNDCLASSEX);
19      wc.style = CS_HREDRAW | CS_VREDRAW;
20      wc.lpfnWndProc = WndProc;
21      wc.cbClsExtra = 0;
22      wc.cbWndExtra = 0;
23      wc.hInstance = hInst;
24      wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
```

```
25      wc.hCursor = LoadCursor(NULL, IDC_ARROW);
26      wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
27      wc.lpszMenuName = NULL;
28      wc.lpszClassName = "Hallo World";
29      wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);
30
31      //register the new class
32      RegisterClassEx(&wc);
33
34      //create a window
35      hWnd = CreateWindowEx(
36           WS_EX_APPWINDOW,
37           "Hallo World",
38           "The Hallo World Program",
39           WS_OVERLAPPEDWINDOW | WS_VISIBLE,
40           CW_USEDEFAULT,
41           CW_USEDEFAULT,
42           CW_USEDEFAULT,
43           CW_USEDEFAULT,
44           NULL,
45           NULL,
46           hInst,
47           NULL
48      );
49
50      //event loop - handle all messages
51      while(GetMessage(&msg, NULL, 0, 0))
52      {
53           TranslateMessage(&msg);
54           DispatchMessage(&msg);
55      }
56
57      //standard return value
58      return (msg.wParam);
59 }
60
61 LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
62                                        LPARAM lParam)
63 {
64      static int Ticks = 0;
65      //device context used for drawing
66      HDC hDC;
67
68      //find out which message is being sent
69      switch(nMsg)
70      {
71           case WM_CREATE:
72                //create the timer (0.1 seconds)
73                SetTimer(hWnd, 1, 100, NULL);
74                break;
75
76           case WM_PAINT:
77                PAINTSTRUCT ps;
78                hDC = BeginPaint (hWnd, &ps);
79                TextOut (hDC, 0, 0, "Hallo, world!", 13);
80                EndPaint (hWnd, &ps);
```

```
81                  break;
82
83              case WM_TIMER: //when the timer goes off (only one)
84                  //get the dc for drawing
85                  hDC = GetDC(hWnd);
86                  if (Ticks % 10 == 0)
87                  {
88                      char text[20];
89                      sprintf (text, "%i", Ticks/10);
90                      TextOut (hDC, 0, 20, text, strlen(text));
91                  };
92                  Ticks = (Ticks+1) % 1000000;
93                  ReleaseDC(hWnd, hDC);
94                  break;
95
96              case WM_DESTROY:
97                  //destroy the timer
98                  KillTimer(hWnd, 1);
99                  //end the program
100                  PostQuitMessage(0);
101                  break;
102
103              case WM_KEYDOWN:
104                  if (wParam == VK_ESCAPE)
105                  {
106                      SendMessage(hWnd, WM_DESTROY, 0, 0);
107                  }
108                  else
109                  {
110                      char text[20];
111                      sprintf (text, "%i", wParam);
112                      hDC = GetDC(hWnd);
113                      TextOut (hDC, 0, 40, "       ", 7);
114                      TextOut (hDC, 0, 40, text, strlen(text));
115                      ReleaseDC(hWnd, hDC);
116                  };
117                  break;
118
119              default:
120                  //let Windows handle every other message
121                  return(DefWindowProc(hWnd, nMsg, wParam,
                    lParam));
122      }
123
124      return 0;
125 }
```