

Architektura Systemów Wbudowanych

Laboratorium 6

**Wieloprocusowość, wielowątkowość -
szeregowanie, synchronizacja, komunikacja
IPC**

2018

Spis treści

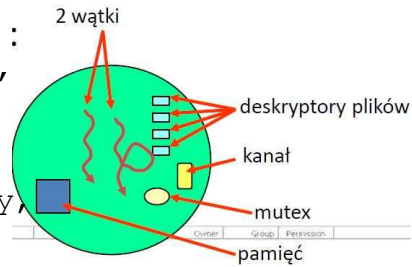
Wstęp - wątki, procesy.	- 3 -
Przygotowanie platform docelowej.....	- 4 -
Przygotowanie środowiska programistycznego QNX Momentics IDE.....	- 7 -
Przełączanie perspektyw.....	9
Tworzenie projektu.....	15
Kompilacja projektu.....	17
Uruchomienie aplikacji.....	18
Debugowanie aplikacji.....	19
Wątki przygotowanie platformy programistycznej QNX Momentics.....	20
Wątki szeregowanie.....	22
Przeanalizuj poniższy kod. Zwróć uwagę na wynik działania funkcji update_thread().	22
Synchronizacja wątków.....	25
Przygotowanie platformy i środowiska IDE.....	25
Przeanalizuj poniższy kod programu Mutex.....	25
Przeanalizuj poniższy kod programu Condvar.....	29
Przeanalizuj poniższy kod programu SEMEX.....	32
Komunikacja IPC Inter-Procces Communication.....	34
Przygotowanie platformy i środowiska IDE.....	35
Zadanie 1.....	35
Zadanie 2.....	37
Zadanie 3.....	38
Zadanie 4.....	39
Zadanie 5.....	41
Zadanie 6.....	42
Zadanie 7.....	43
Zadanie 8.....	45
Zadanie 9.....	46
Sprawozdanie.....	47

Wstęp - wątki, procesy.

Procesy są komponentami składowymi systemu, które widoczne dla każdego innego procesu oraz mogą komunikować się z każdym procesem.

- Proces:

- program załadowany do pamięci;
- identyfikowany przez **id** procesu, zwykle nazywany jako **pid**;
- wspólne zasoby procesu:
 - pamięć, włączając kod i dane,
 - otwarte pliki,
 - identyfikatory :
 - id użytkownika, id grupy,
 - timery.

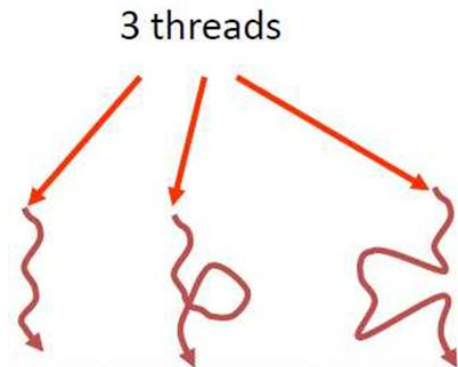


- Zasoby należące do jednego procesu są chronione przed innymi procesami.

Wątki są szczegółową implementacją danego procesu ukrytą w jego wnętrzu. Uruchamiane są wewnątrz procesów. Każdy uruchomiony proces musi posiadać przynajmniej jeden wątek. Wątki należące do jednego procesu współdzielą wszystkie jego zasoby.

- Wątek

- wątek jest pojedynczym strumieniem wykonania;
- wątek posiada pewne atrybuty:
 - priorytet,
 - algorytm kolejkowania,
 - zestaw rejestrów,
 - maska CPU dla SMP,
 - maska sygnałów,
 - i inne.



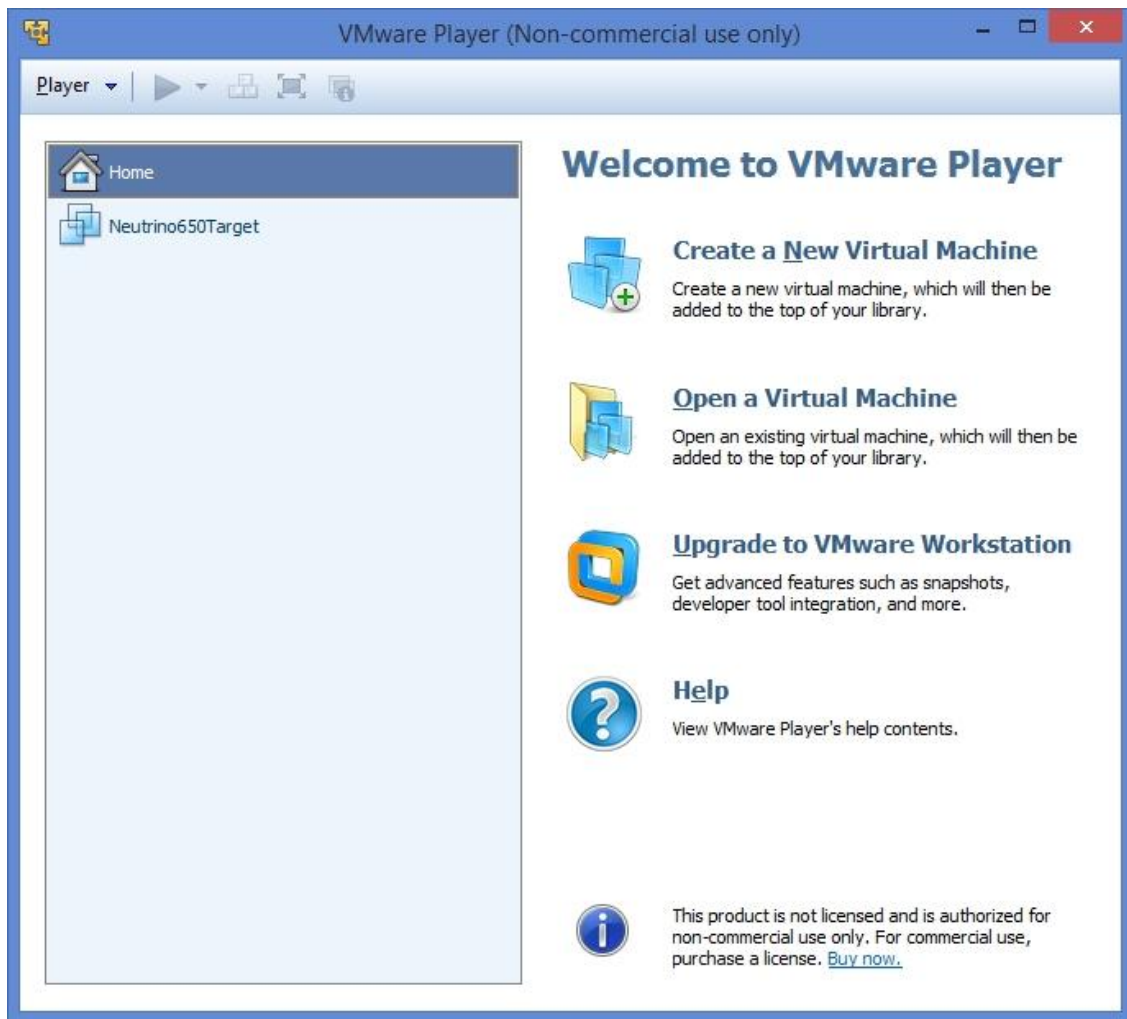
Przygotowanie platform docelowej.

Uruchom VMware Player dwuklik na ikonie:



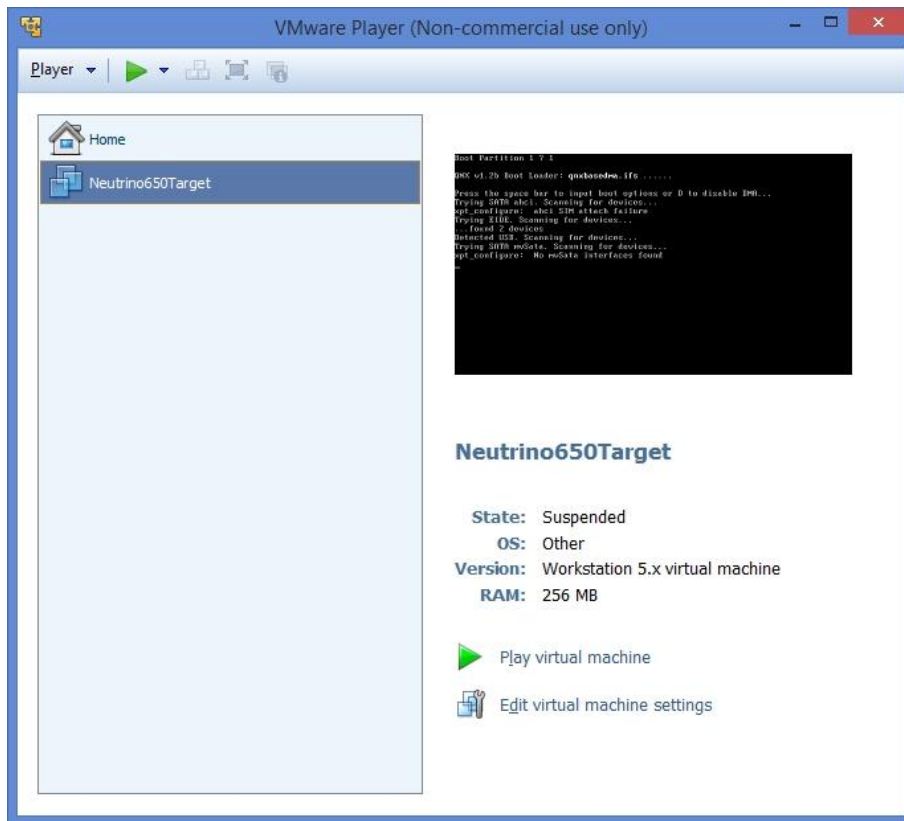
Ikonka aplikacji VMware Player.

Na ekranie pojawi się okno główne programu:



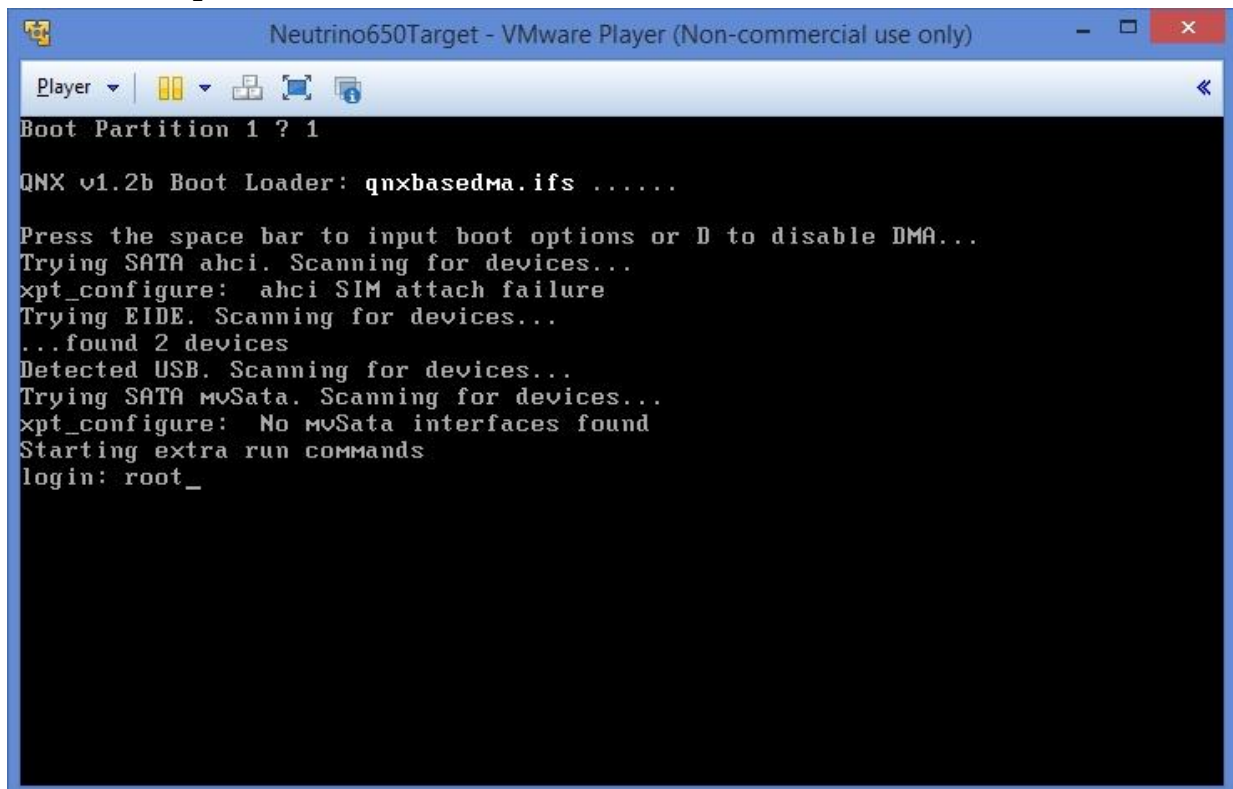
Główne okno programu VMware Player.

Po lewej stronie okna znajduje się lista obrazów systemów. Wybierz z niej obraz system QNX Neutrino - **Neurino650Target**. Okno zmieni swój wygląd na poniższy.



VMware Player okno po wybraniu obrazu systemu.

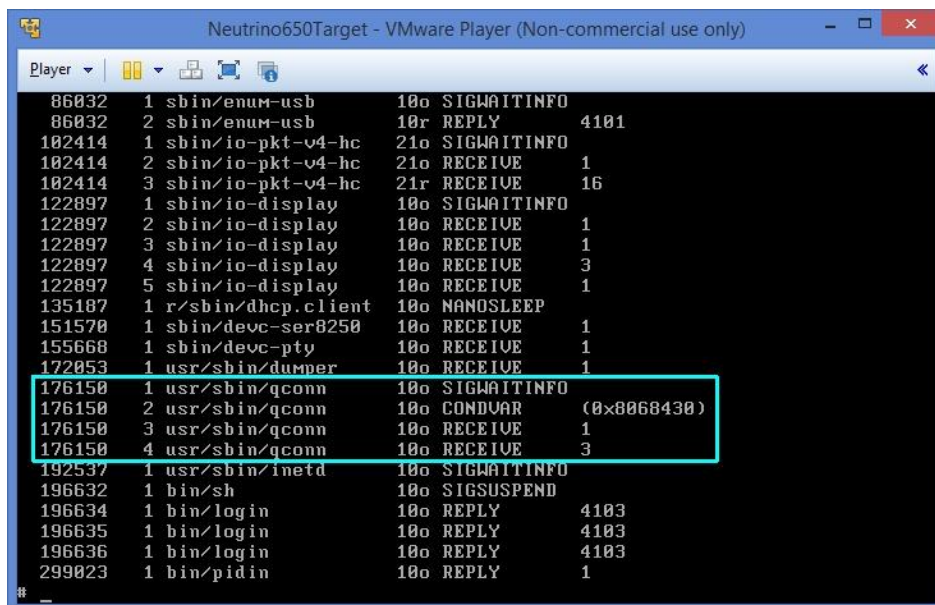
Następnie wybierając **Play virtual machine** lub wykonując dwuklik na nazwie obrazu uruchamiamy system QNX Neutrino. Na ekranie pojawi się okno konsoli systemu.



Okno konsoli systemu QNX Neutrino.

Zaloguj się jako administrator systemu **login: root**.

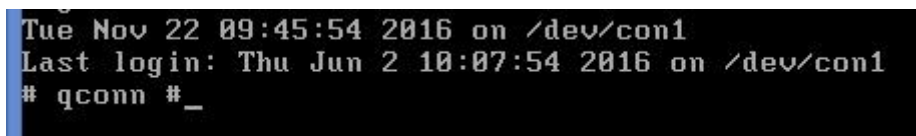
Poleceniem **pidin** sprawdź czy uruchomiony jest proces **qconn** - QNX connection.



```
Neutrino650Target - VMware Player (Non-commercial use only)
Player
86032 1 sbin/enum-usb 10o SIGWAITINFO
86032 2 sbin/enum-usb 10r REPLY 4101
102414 1 sbin/io-pkt-v4-hc 21o SIGWAITINFO
102414 2 sbin/io-pkt-v4-hc 21o RECEIVE 1
102414 3 sbin/io-pkt-v4-hc 21r RECEIVE 16
122897 1 sbin/io-display 10o SIGWAITINFO
122897 2 sbin/io-display 10o RECEIVE 1
122897 3 sbin/io-display 10o RECEIVE 1
122897 4 sbin/io-display 10o RECEIVE 3
122897 5 sbin/io-display 10o RECEIVE 1
135187 1 r/sbin/dhcp.client 10o NANOSLEEP
151570 1 sbin/devc-ser8250 10o RECEIVE 1
155668 1 sbin/devc-pty 10o RECEIVE 1
172053 1 usr/sbin/dumper 10o RECEIVE 1
176150 1 usr/sbin/qconn 10o SIGWAITINFO
176150 2 usr/sbin/qconn 10o CONDVAR (0x0068430)
176150 3 usr/sbin/qconn 10o RECEIVE 1
176150 4 usr/sbin/qconn 10o RECEIVE 3
192537 1 usr/sbin/inetd 10o SIGWAITINFO
196632 1 bin/sh 10o SIGSUSPEND
196634 1 bin/login 10o REPLY 4103
196635 1 bin/login 10o REPLY 4103
196636 1 bin/login 10o REPLY 4103
299023 1 bin/pidin 10o REPLY 1
#
```

Okno konsoli po wykonaniu polecenia **pidin**.

Jeżeli nie ma go na liście uruchomionych procesów wykonaj poniższe polecenie.

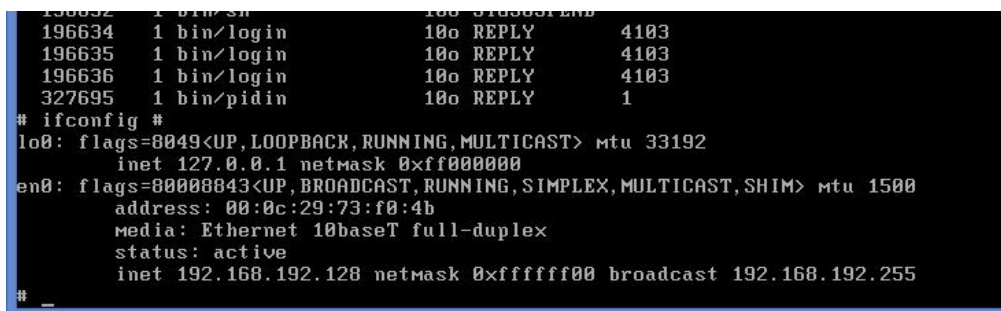


```
Tue Nov 22 09:45:54 2016 on /dev/con1
Last login: Thu Jun 2 10:07:54 2016 on /dev/con1
# qconn #_
```

Polecenie **qconn**.

Ponownie sprawdź czy uruchomiony jest proces **qconn**.

W następnej kolejności sprawdź adres IP uruchomionej platformy docelowej poleceniem **ifconfig**.



```
196632 1 bin/sh 10o SIGSUSPEND
196634 1 bin/login 10o REPLY 4103
196635 1 bin/login 10o REPLY 4103
196636 1 bin/login 10o REPLY 4103
327695 1 bin/pidin 10o REPLY 1
# ifconfig #
lo0: flags=0049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33192
inet 127.0.0.1 netmask 0xff000000
en0: flags=00008843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST,SHIM> mtu 1500
address: 00:0c:29:73:f0:4b
media: Ethernet 10baseT full-duplex
status: active
inet 192.168.192.128 netmask 0xfffff00 broadcast 192.168.192.255
#
```

Zapamiętaj adres IP będzie on potrzebny do zestawienia połączenia pomiędzy platformą docelową, a hostem środowiska IDE.

Przełączanie pomiędzy oknem maszyny wirtualnej a systemem Windows:

CTRL+G - przejście z OS Windows do maszyny wirtualnej;

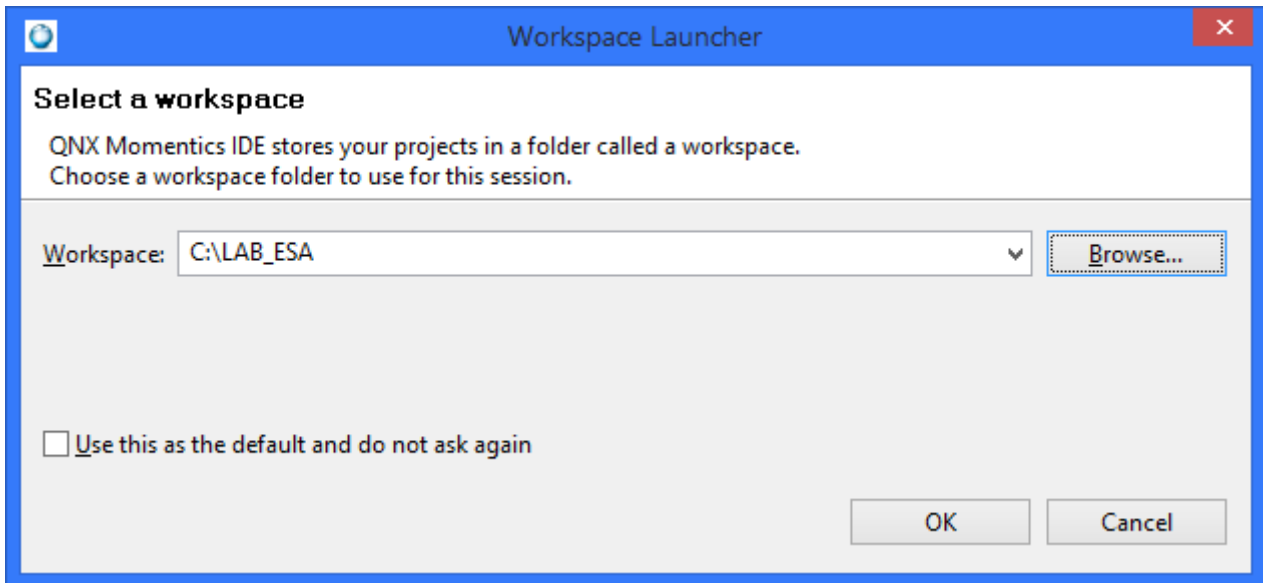
CTRL+ALT - przejście z maszyny wirtualnej do OS Windows.

Przygotowanie środowiska programistycznego QNX Momentics IDE.

Uruchom środowisko QNX Momentics IDE używając ikonki:

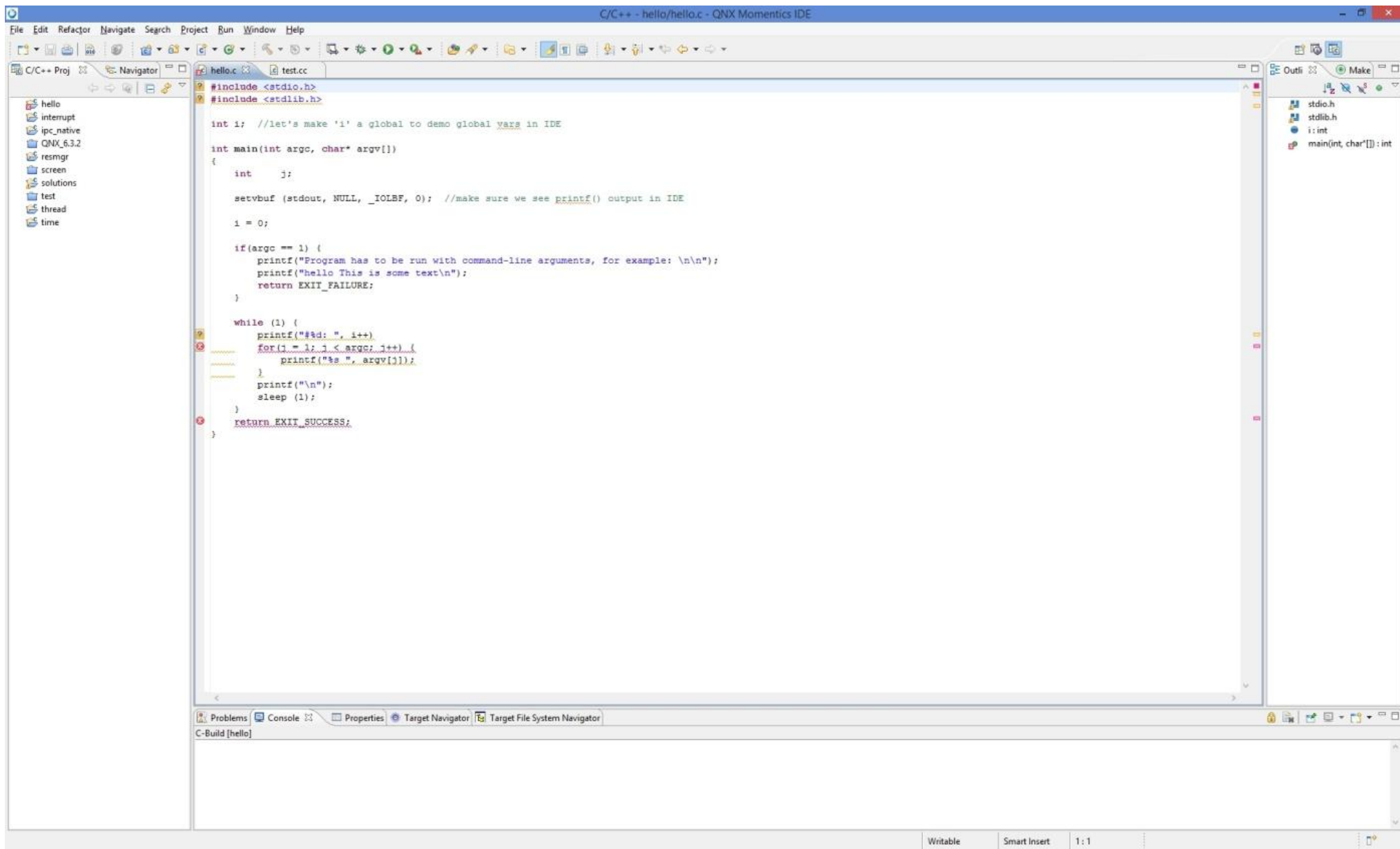


Na ekranie pojawi się okno Workspace Launcher pozwalające wybrać przestrzeń roboczą Workspace. Wybierz przestrzeń roboczą wskazaną w instrukcji lub podaną przez prowadzącego.



Okno **Workspace Launcher**

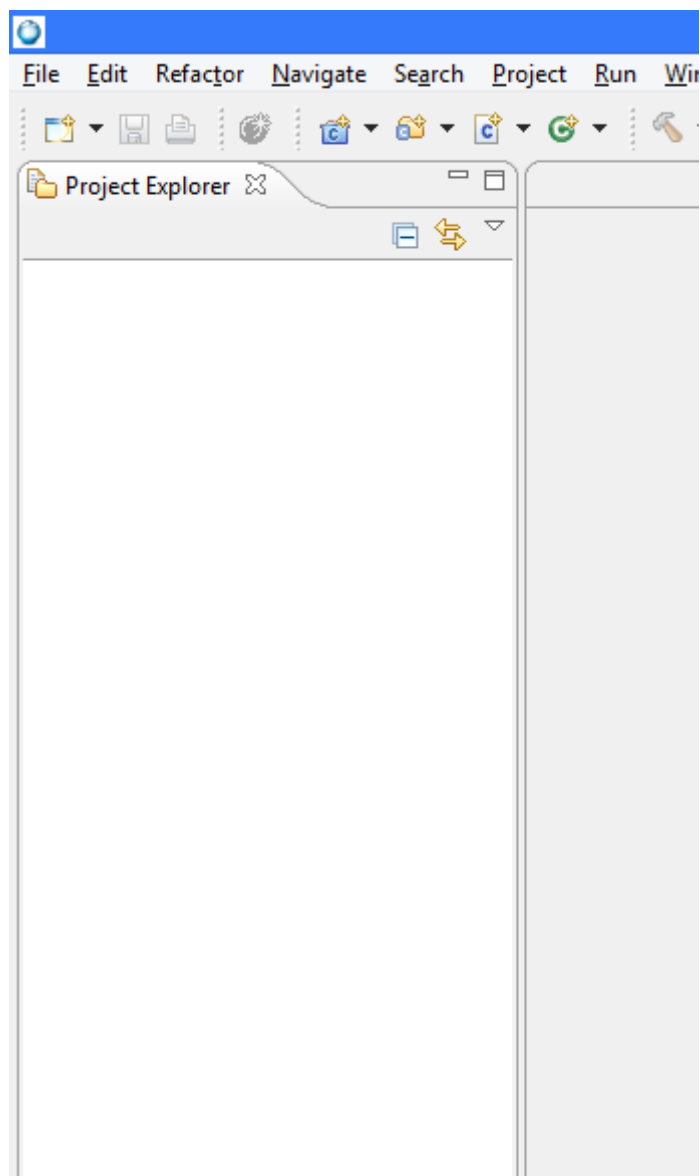
Po wybraniu przestrzeni roboczej na ekranie pojawi się okno główne środowiska.



Okno główne środowiska programistycznego

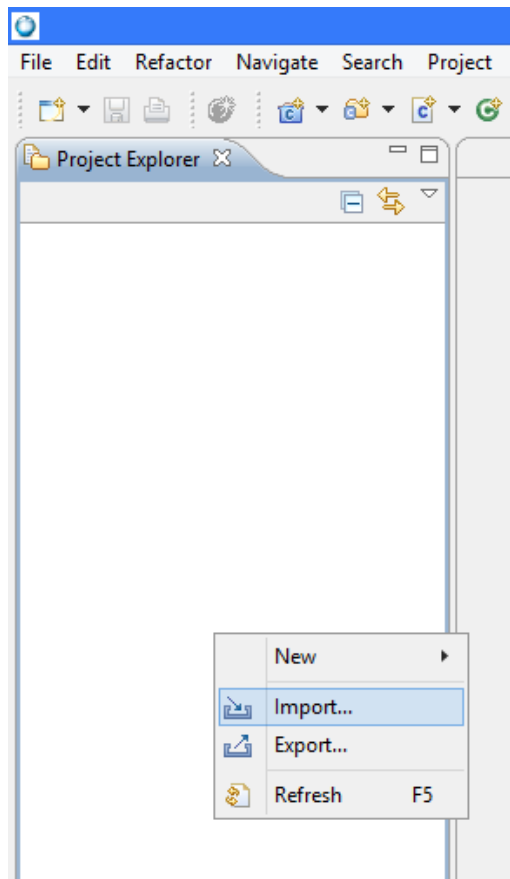
Importowanie projektów do przestrzeni roboczej

Po lewej stronie okna środowiska deweloperskiego znajduje się okno **Project Explorer**, w którym wyświetlane są projekty znajdujące się w aktualnie wybranej przestrzeni roboczej. Jeżeli w oknie nie są widoczne żadne projekty należy je zaimportować.



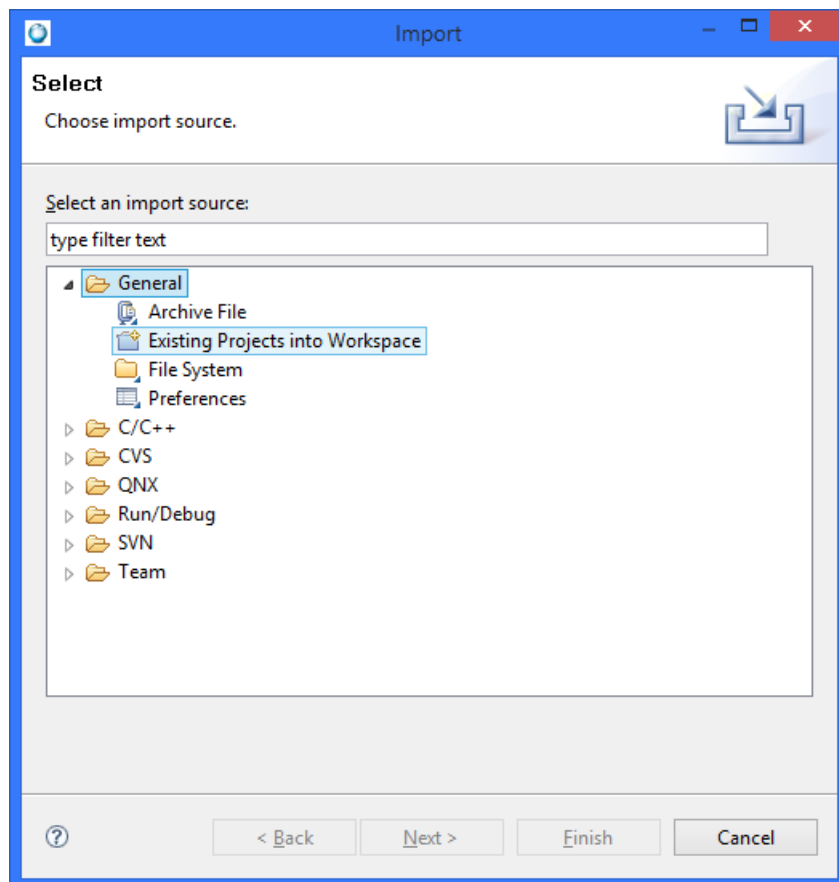
Okno Project Explorer

Aby zaimportować projekty do przestrzeni roboczej należy prawym klawiszem myszy otworzyć menu kontekstowe okna **Project Explorer**, a następnie wybrać opcję **Import**, rysunek na następnej stronie.



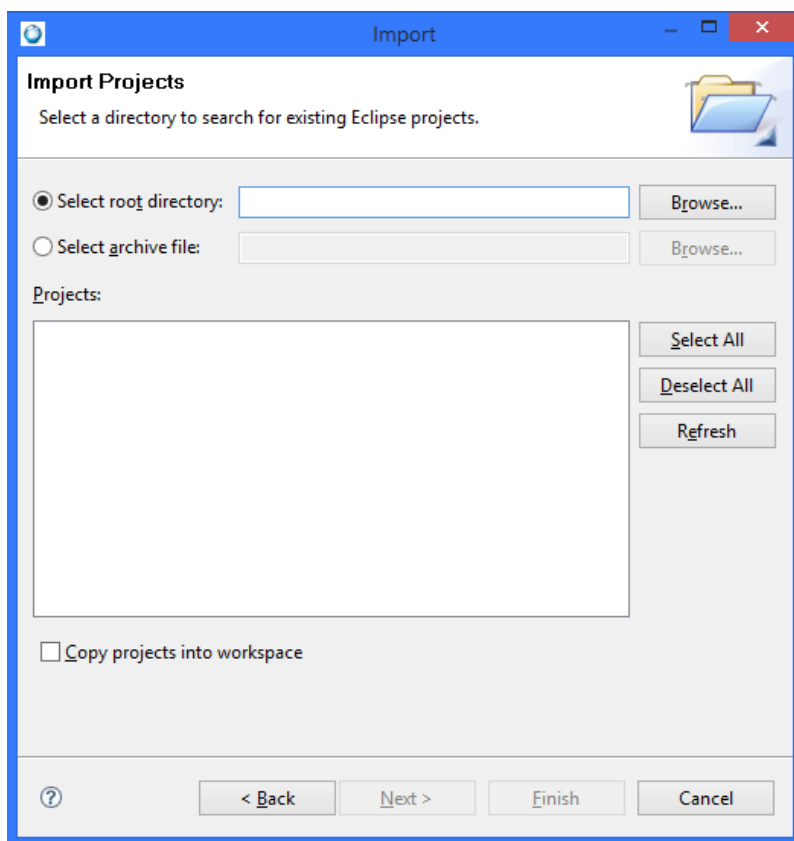
Menu kontekstowe okna Project Explorer

Na ekranie pojawia się okno **Import.**



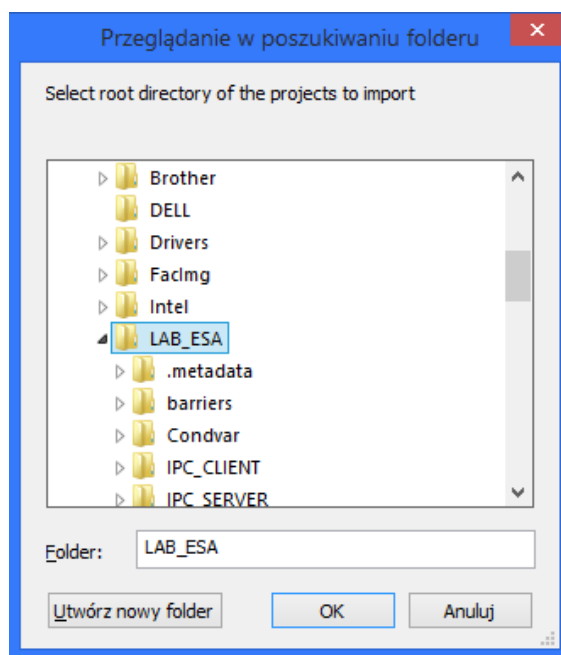
Okno Import

Wybieramy z grupy **Select an import source**, opcję **General**, a następnie **Existing Projects into Workspace** na ekranie pojawi się poniższe okno.



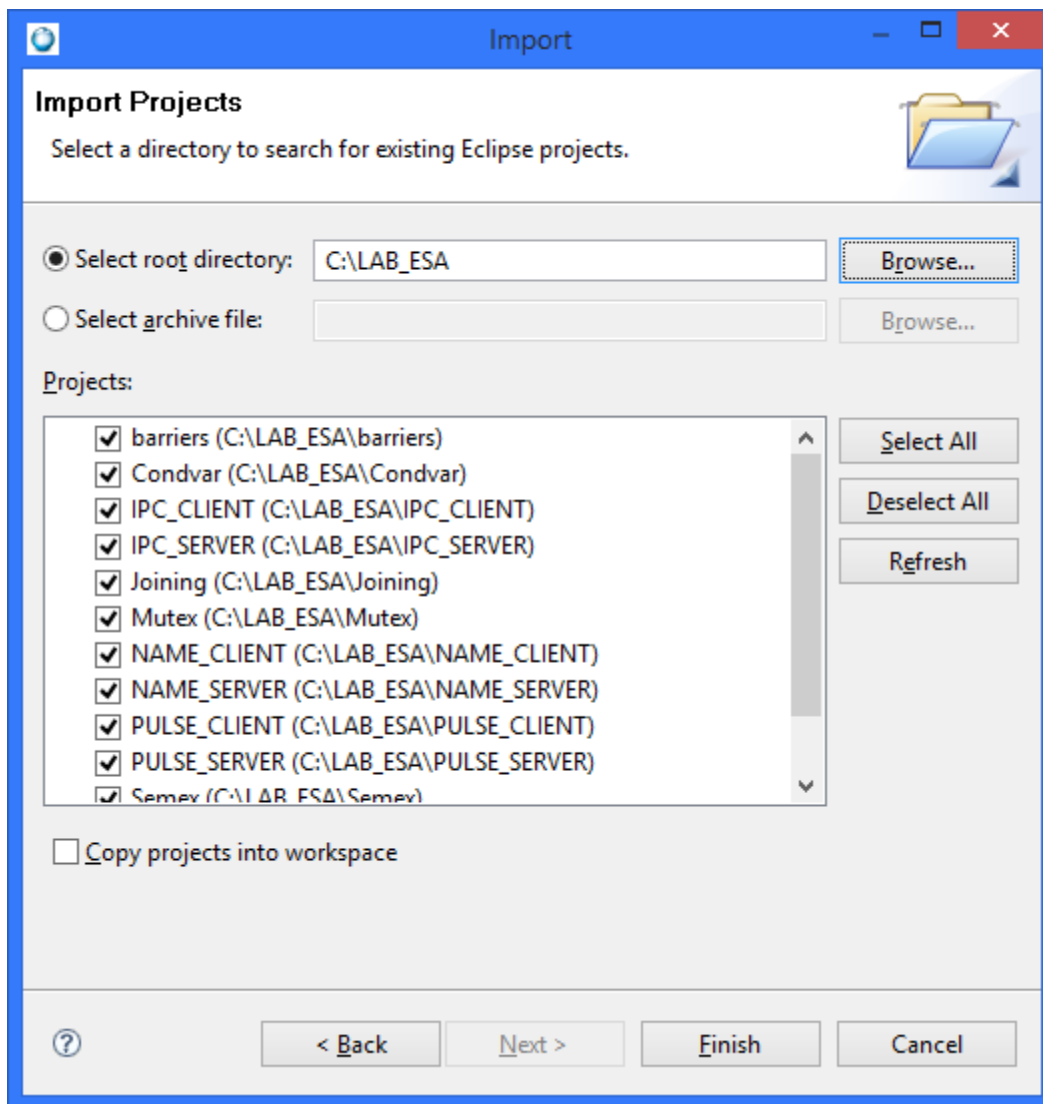
Okno Import

W polu **Select root directory** wskazujemy główny katalog naszej przestrzeni roboczej, używając klawisza **Browse**. Katalog wybieramy w oknie poniżej.



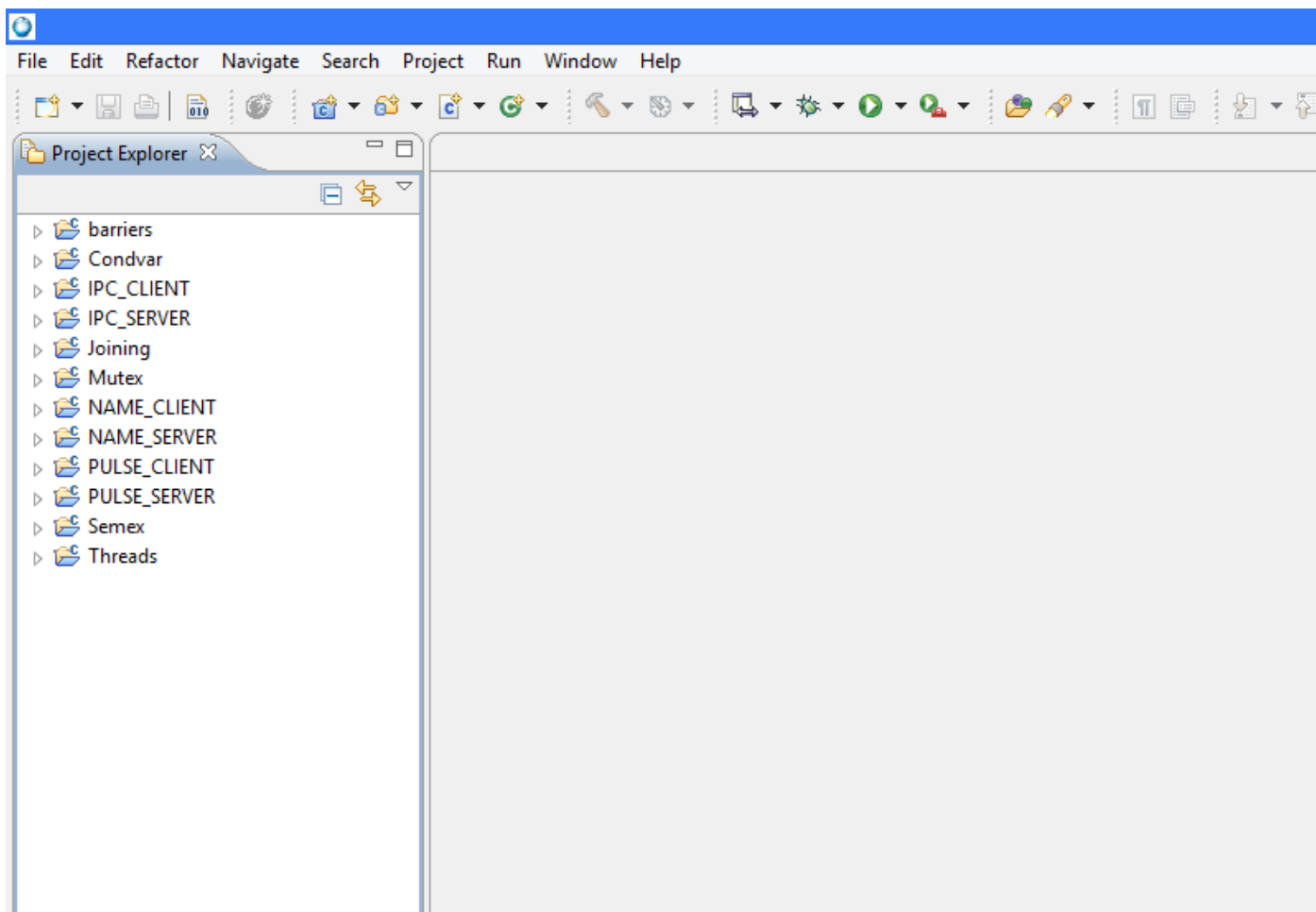
Okno wyboru folderu przestrzeni roboczej

Na ekranie pojawi się okno z listą dostępnych projektów. Środowisko domyślnie wybierze te projekty, które nie są dołączone do aktualnej przestrzeni roboczej.



Okno Import Projects

Po wciśnięciu klawisza **Finish** projekty zostaną dodane automatycznie do aktualnej przestrzeni roboczej rysunek na następnej stronie.



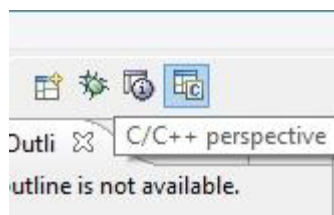
Widok okna platformy programistycznej z zaimportowanymi projektami.

Przełączanie perspektyw

Perspektywy pozwalają zorganizować widok okna głównego zgodnie z aktualnie wykonywanymi zadaniami. Okno główne w trakcie debugowania kodu powinno zawierać zupełnie inne widoki niż te, których używamy podczas pisania kodu. Perspektywy przełączamy używając kluczy, które znajdują się po prawej stronie paska narzędzi.



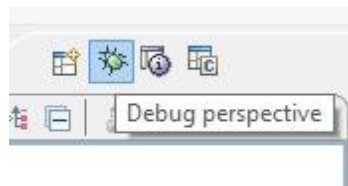
Klucze perspektyw



Klucz perspektywy C/C++

Perspektywy C/C++ używamy do:

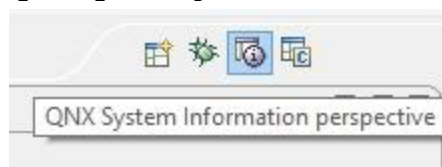
- tworzenia projektu;
- dodawania plików nagłówkowych i źródłowych;
- edytowania kodu;
- kompilacji projektu.



Klucz perspektywy Debug

Perspektywy Debug używamy do:

- analizowania krok po kroku poprawności działania oprogramowania;
- poszukiwania błędów logicznych;
- sprawdzania poprawności danych;
- śledzenia ścieżki wykonywanego kodu.



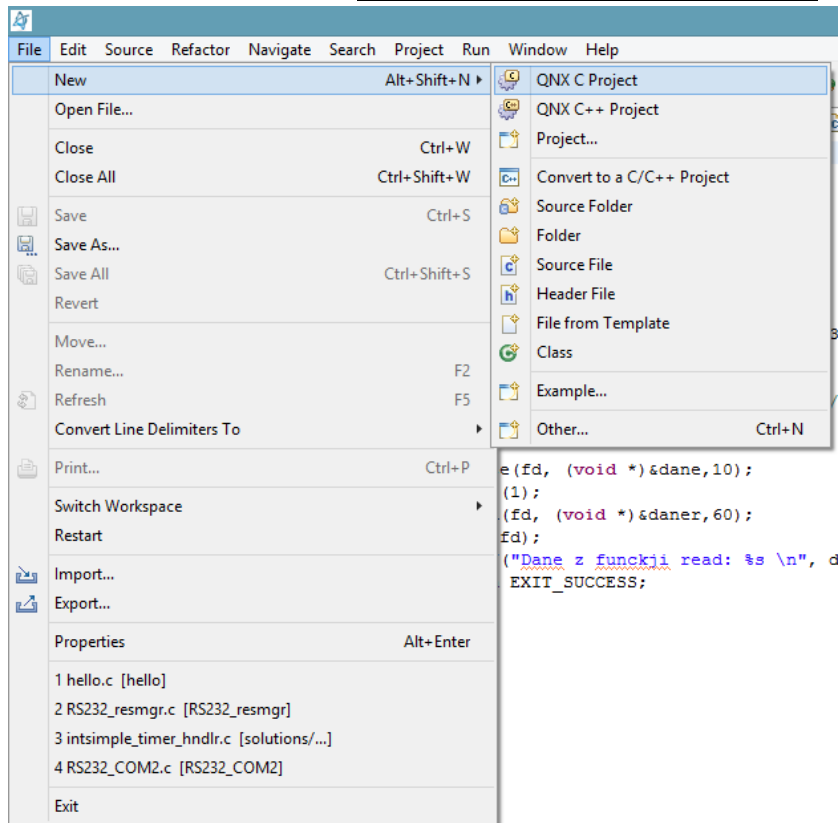
Klucz perspektywy QNX System Information

Perspektywy QNX System Information używamy do:

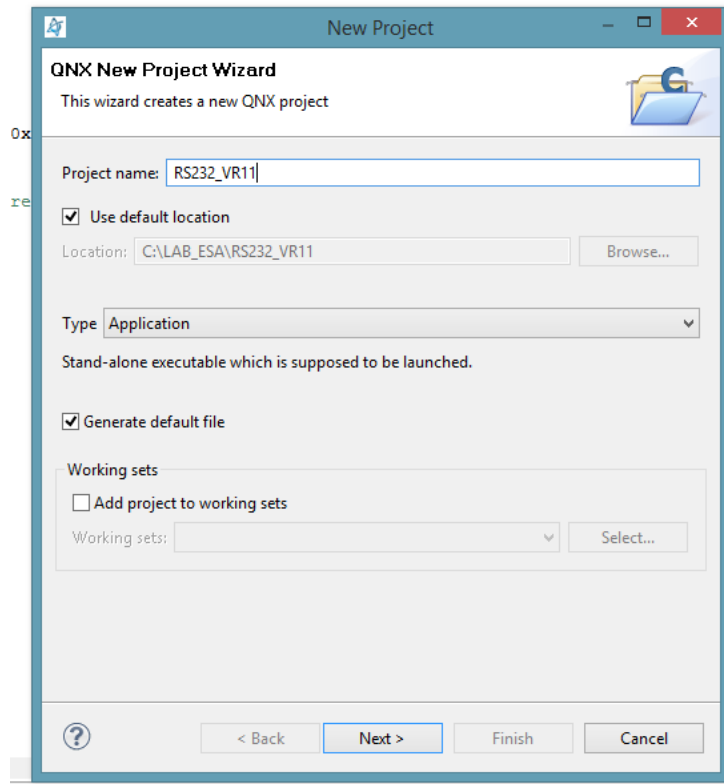
- dodawania i usuwania platformy docelowej;
- sprawdzenia stanu platformy docelowej;
- sprawdzenia stanu uruchomionych procesów i wątków;
- analizy zasobów procesów;
- analizy wzajemnych powiązań pomiędzy procesami i wątkami.

Tworzenie projektu.

Wybieramy z menu głównego opcję: **File->New->QNX C Project.**

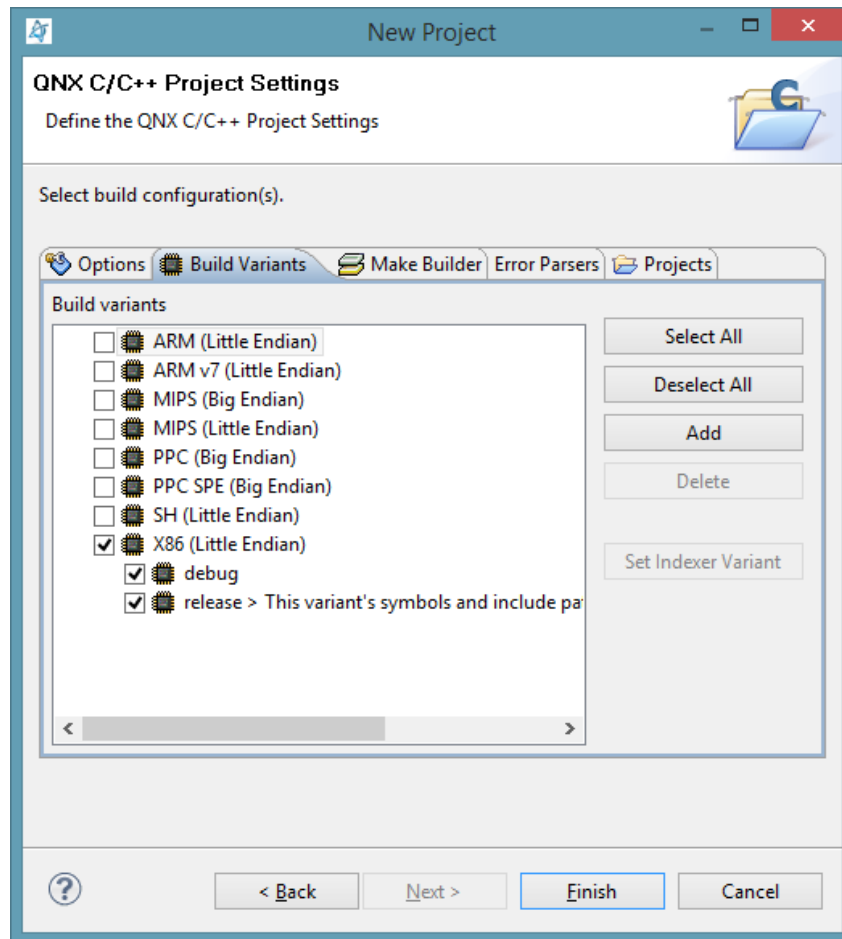


Na ekranie pojawi się okno dialogowe **New Project.**



Okno dialogowe **New Project**

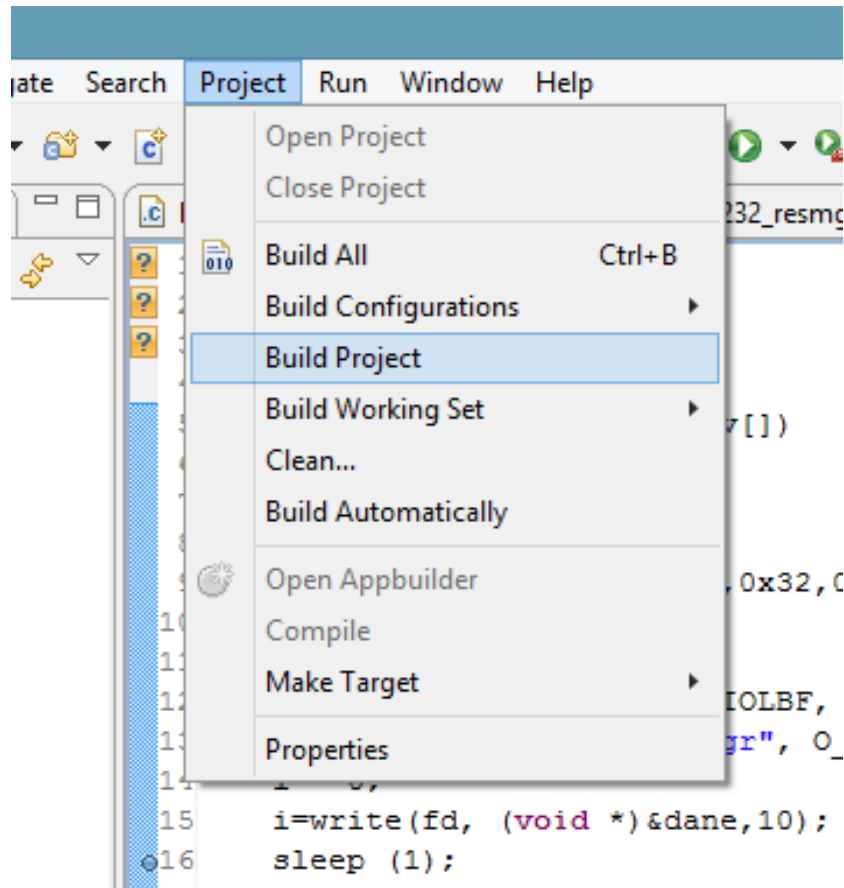
Po wpisaniu nazwy projektu w okienku Project name: wciskamy klawisz Next pojawi się okno dialogowe.



W zakładce **Build Variants** wybieramy platformę sprzętową, na którą będziemy kompilować nasze oprogramowanie. Po dokonaniu wyboru wciskamy klawisz **Finish**. Na ekranie pojawi się okno główne środowiska z perspektywą C/C++ i z wstępnie przygotowanym projektem.

Kompilacja projektu.

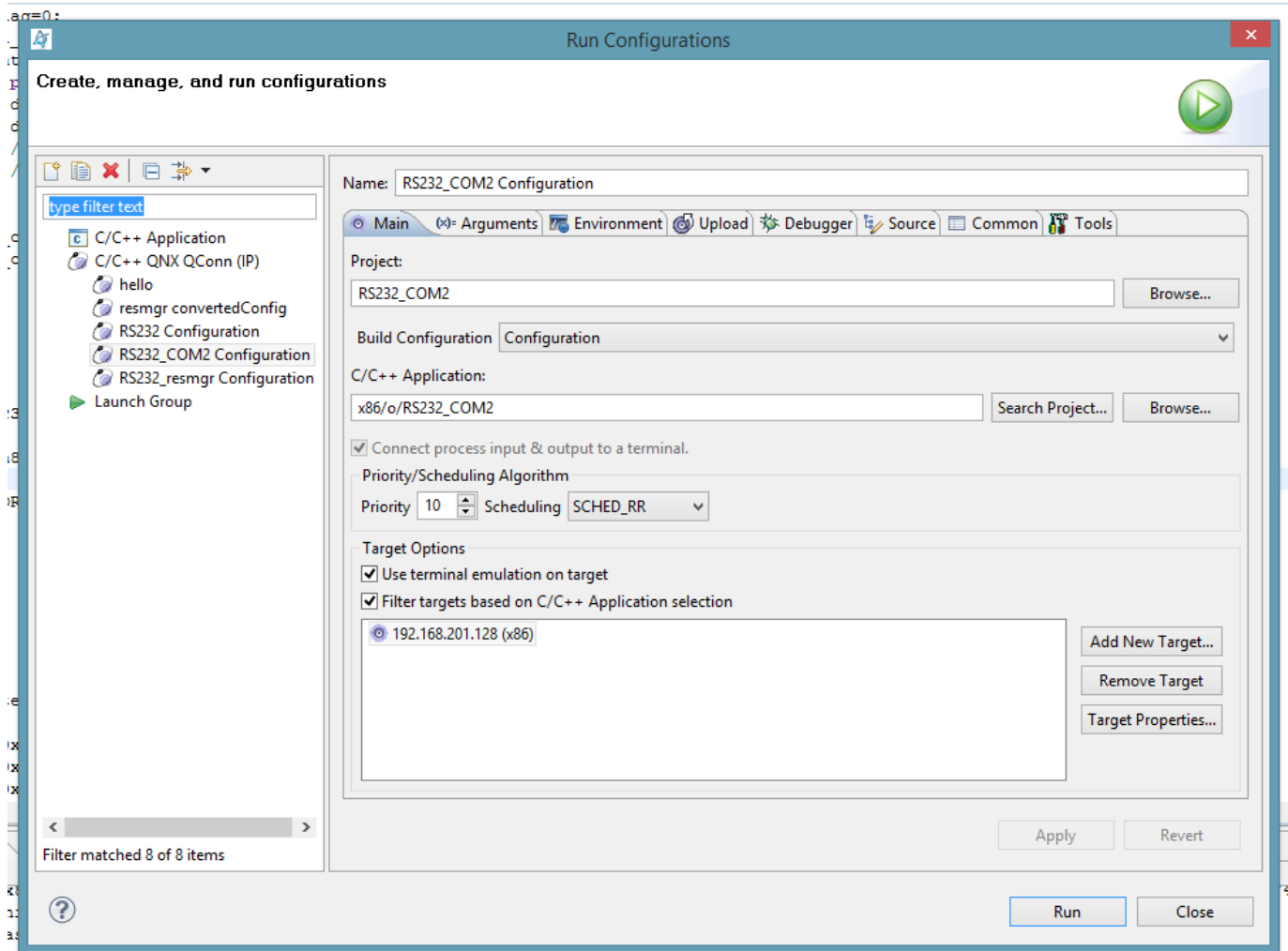
Wybieramy z menu głównego opcję **Project->Build Project**. Wynik kompilacji pojawi się w zakładce **Console**. W zakładce **Problems** wyświetlone zostaną ewentualne błędy i problemy.



Menu Project

Uruchomienie aplikacji.

Z menu głównego wybieramy opcję za pierwszym razem **Run->Run Configuration**, przy kolejnych uruchomieniach **Run->Run**.

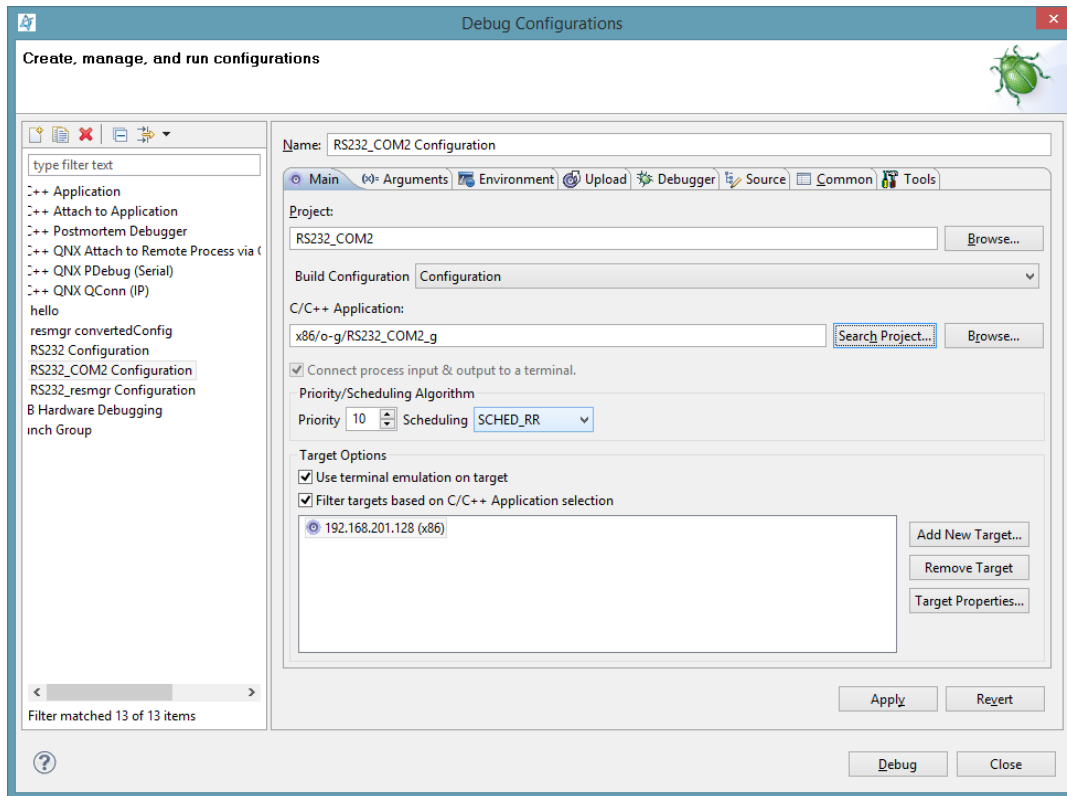


Okno **Run Configurations**

W polu po lewej stronie tworzymy nową konfigurację uruchamiania C/C++ QNX QConn IP. W polu **Project Browse** wybieramy projekt z przestrzeni roboczej. W polu C/C++ Application **Search Project** wybieramy aplikację do uruchomienia. W zakładce „**Arguments**” możemy prowadzić argumenty wejściowe aplikacji. **Run** uruchamia aplikację na platformie docelowej. W polu **Target Option** wybieramy odpowiednią platformę docelową.

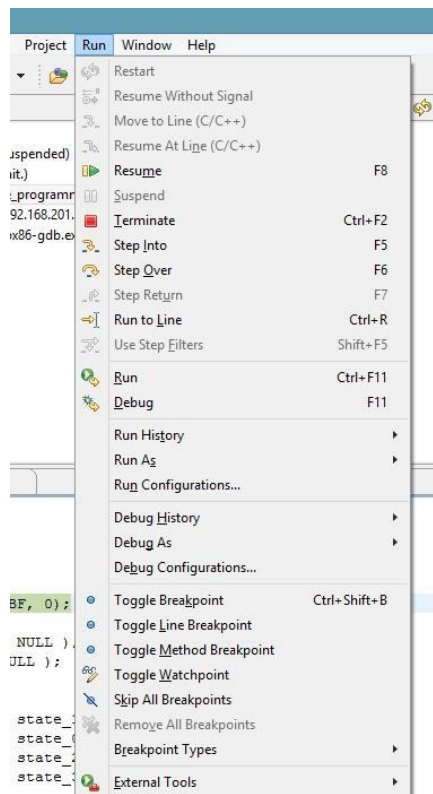
Debugowanie aplikacji.

Podobnie jak w pkt 3. używamy opcji **Run->Debug Configuration** przy pierwszym uruchomieniu debugera w kolejnych uruchomieniach **Run->Debug**.



Okno Debug Configurations

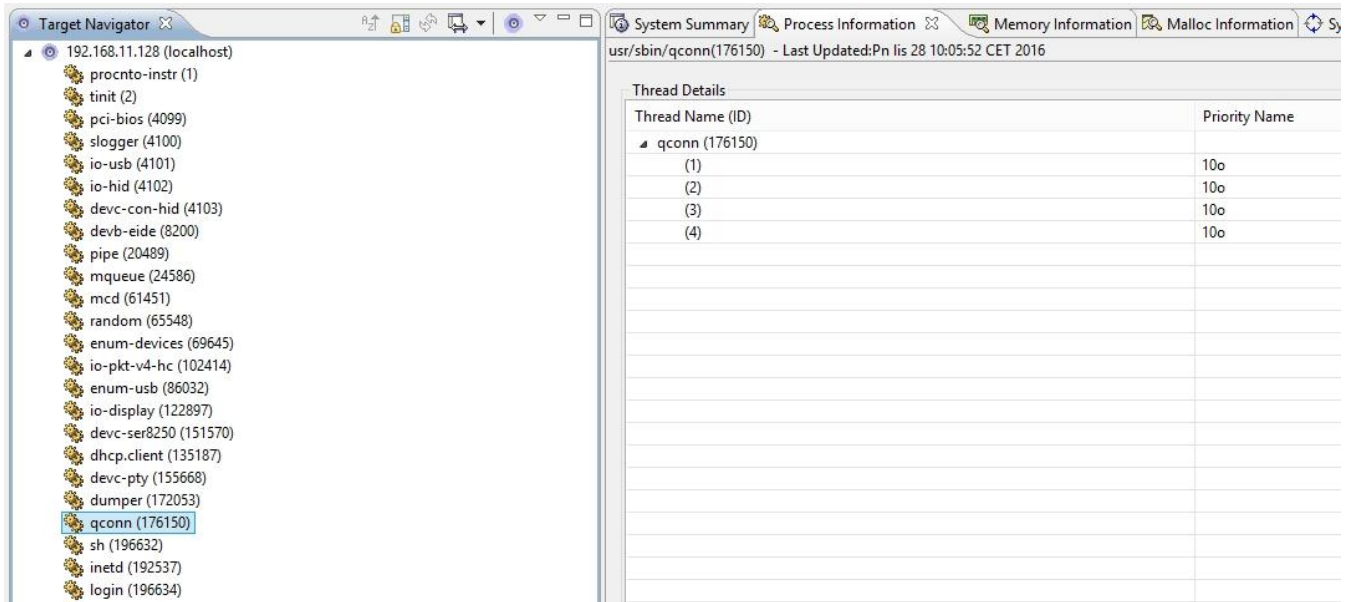
W menu głównym opcja **Run/Debug** w czasie debugowania kodu posiada następujące polecenia.



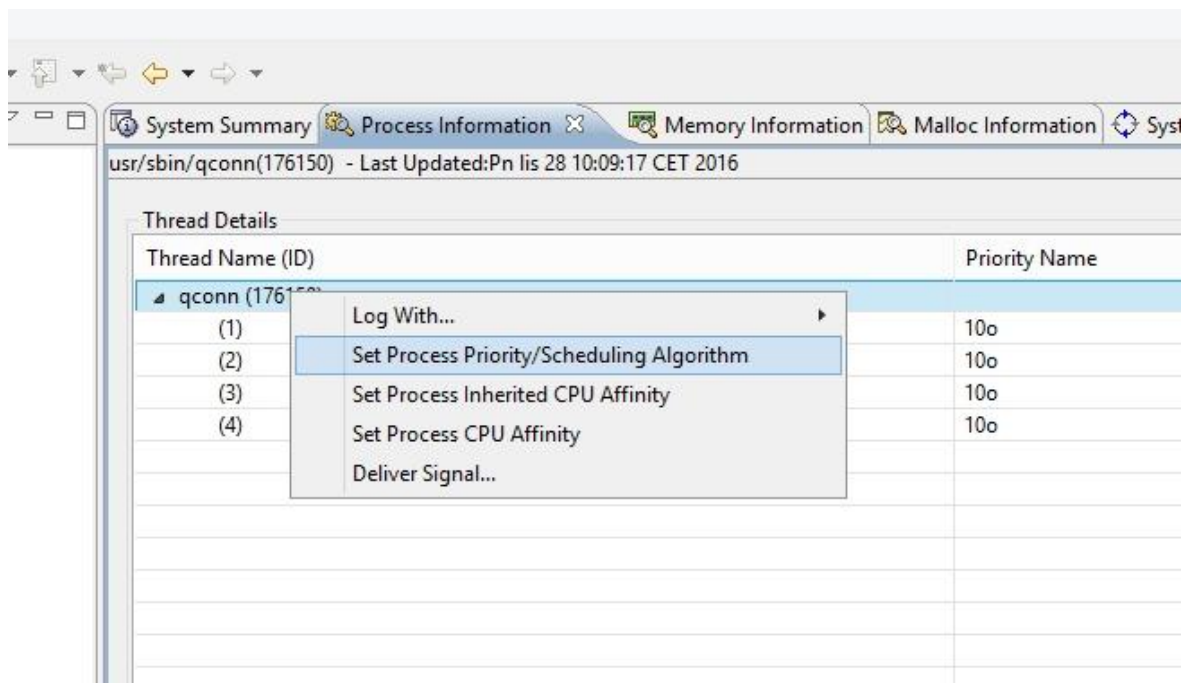
Menu **Run/Debug**

Wątki przygotowanie platformy programistycznej QNX Momentics.

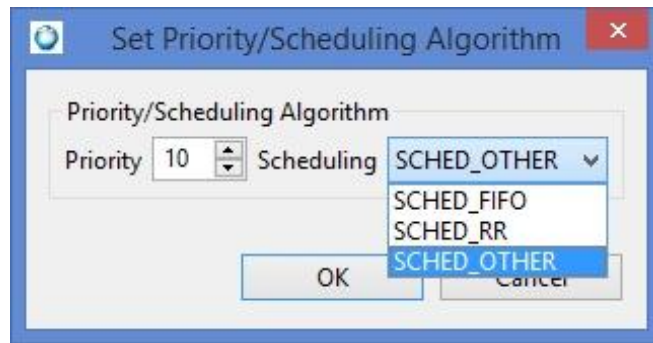
Przed uruchomieniem własnego kodu zwiększ priorytet procesu qconn z poziomu środowiska IDE, do poziomu 12. W tym celu wybierz perspektywę QNX System Information. Wybierz zakładkę Process Information. Następnie z listy procesów uruchomionych na platformie docelowej wybierz proces qconn.



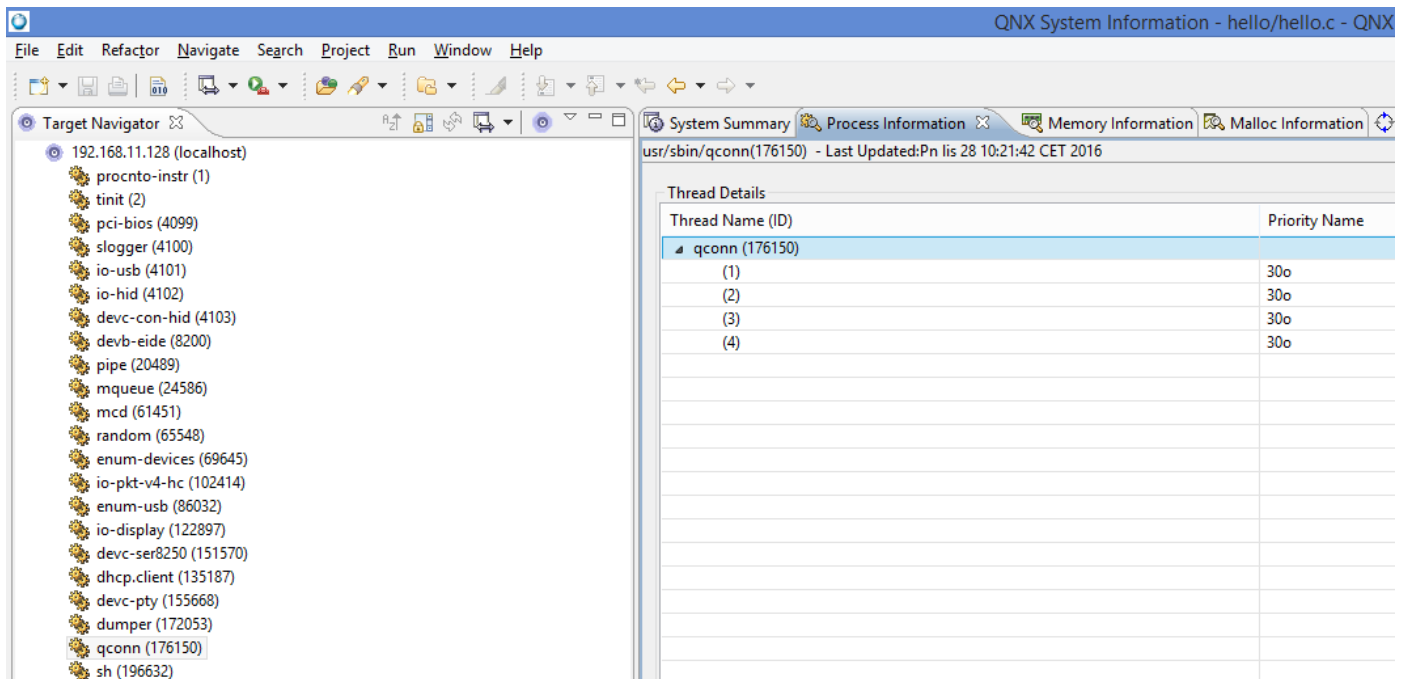
W zakładce Process Information wciśnij prawy klawisz myszy na nazwie procesu głównego qconn. Pojawi się menu kontekstowe - rysunek poniżej.



Wybierz opcję Set Process Priority/Scheduling Algorithm na ekranie pojawi okno.



W polu Priority wpisz wartość 12. Algorytm szeregowania pozostaw bez zmian. Wciśnij klawisz OK zatwierdzając zmiany. Wynik zmian powinien być widoczny w zakładce Process Information, jak na poniższym rysunku.



Zmianę ustawień możesz sprawdzić również w wirtualnej maszynie używając polecenia pidin.

```

135187 1 r/sbin/dhcp.client 10o NANOSLEEP
151570 1 sbin/devc-ser8250 10o RECEIVE 1
155668 1 sbin/devc-pty 10o RECEIVE 1
172053 1 usr/sbin/dumper 10o RECEIVE 1
176150 1 usr/sbin/qconn 30o SIGWAITINFO
176150 2 usr/sbin/qconn 30o CONDUAR (0x8068430)
176150 3 usr/sbin/qconn 30o RECEIVE 1
176150 4 usr/sbin/qconn 30o RECEIVE 3
192537 1 usr/sbin/inetd 10o SIGWAITINFO
196632 1 bin/sh 10o SIGSUSPEND
196634 1 bin/login 10o REPLY 4103

```

Wątki szeregowanie

Przeanalizuj poniższy kod. Zwróć uwagę na wynik działania funkcji `update_thread()`.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <sched.h>

/*
The number of threads that we want to have running
simultaneously.
*/

#define NumThreads      4

volatile int      var1;
volatile int      var2;
/*
The header of the main function of thread. Thread starts its
job in this function.
*/
void      *update_thread (void *);

char      *progrname = "threads";

int main(int argc, char *argv[]) {
    pthread_t      threadID [NumThreads]; // a place to
                                           // hold the thread
                                           // ID's
    pthread_attr_t      attrib;           // scheduling
                                           // attributes
    struct sched_param  param;           // for setting
                                           // priority

    int            i, policy;
    int            oldcancel;

    setvbuf (stdout, NULL, _IOLBF, 0);
    var1 = var2 = 0; /* initialize to known values */
    printf ("%s: starting; creating threads\n", progrname);
/*
We want to create the new threads using Round Robin
scheduling, and a lowered priority, so set up a thread
attributes structure. We use a lower priority since these
threads will be hogging the CPU
*/
    pthread_getschedparam( pthread_self(), &policy, &param );
    pthread_attr_init(&attrib);
```

```

pthread_attr_setinheritsched(&attrib, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy (&attrib, SCHED_RR );
param.sched_priority -= 5;          // change priority a couple
                                   // levels
pthread_attr_setschedparam (&attrib, &param);
pthread_setcanceltype( PTHREAD_CANCEL_ASYNCHRONOUS, &oldcancel);
*/
Create the threads. As soon as each pthread_create call is
done, the thread has been started.
*/

    for (i = 0; i < NumThreads; i++) {
        pthread_create( &threadID [i], &attrib, &update_thread,
                        (void *) i);
    }
/* Let the other threads run for a while */
    sleep (120);
/* and then kill them all*/

    printf ("%s:  stopping; cancelling threads\n", progame);

    for (i = 0; i < NumThreads; i++) {
        pthread_cancel (threadID [i]);
    }

    printf ("%s:  all done, var1 is %d, var2 is %d\n", progame,
            var1, var2);
    fflush (stdout);
    sleep (1);
    return EXIT_SUCCESS;
}

void *update_thread (void *i)
{
    while (1) {
        if(var1 != var2) {
            printf ("thread %d, var1 (%d) != var2 (%d)!\n", (int) i,
                    var1, var2);
            var1 = var2;
        }

        var1++;
        //sched_yield(); /* for faster processors, to cause problem
            to happen */
        var2++;
    }
    return (NULL);
}

```

1. Jaki jest wynik działania powyższego programu, czy wartości zmiennych var1 i var2 będą sobie równe, czy też różne?

2. Uzupełnij kod projektu **Threads**, tak aby uruchomiony został jeden wątek o priorytecie o pięć niższym niż priorytet procesu głównego, algorytm szeregowania Round Robin. W helpie sprawdź funkcje, których wywołania należy uzupełnić.
3. Sprawdź wynik działania uruchomionego kodu. Czy wynik jest zgodny z przewidywaniami z pkt 3. Do sprawdzeń użyj **perspektywy QNX System Information**. Wynik działania programu przedstaw prowadzącemu.
4. Zmodyfikuj kod programu tak, aby proces główny tworzył 16 wątków, które będą rozpoczynały pracę w funkcji **update_thread()**. Jaki będzie wynik działania programu, czy wartości zmiennych var1 i var2 będą sobie równe, czy też różne?
5. Sprawdź wynik działania uruchomionego kodu. Do sprawdzeń użyj **perspektywy QNX System Information**. Wynik działania programu przedstaw prowadzącemu.
6. Jakie może być rozwiązanie ewentualnych problemów?
7. Zmodyfikuj kod w taki sposób, aby proces główny tworzył trzy wątki, które będą rozpoczynały pracę w różnych funkcjach, będą miały różny priorytet oraz różny sposób szeregowania.
8. Sprawdź wynik działania uruchomionego kodu. Do sprawdzeń użyj **perspektywy QNX System Information**. Wynik działania programu przedstaw prowadzącemu.

Synchronizacja wątków

W systemach wielowątkowych wspólne zasoby procesu takie, jak:

- pamięć:
 - kod programu;
 - dane;
- otwarte pliki;
- identyfikatory:
 - id użytkownika,
 - id grupy;
- timery,

są współdzielone przez wszystkie wątki należące do procesu. Wątki konkurują pomiędzy sobą o dostęp do poszczególnych zasobów. Może to prowadzić do niepożądanych sytuacji, w których wielokrotne zapisy tego samego obszaru pamięci mogą nadpisywać oczekiwane dane. W takich sytuacjach wątki czytające nie wiedzą kiedy dane są stabilne. Operowanie na niewłaściwych danych prowadzi do wypracowywania błędnych wyników, a w konsekwencji może być przyczyną podejmowania niewłaściwych decyzji, które mogą mieć katastrofalne skutki.

Rozwiązaniem w/w problemów jest stosowanie metod synchronizacji pracy wątków. QNX Neutrino dostarcza wielu metod synchronizacji wątków. Do najważniejszych możemy zaliczyć wymienione poniżej:

- mutex - wzajemne wykluczenie wątków;
- condvar - oczekiwanie na zmienną;
- semaphore - oczekiwanie na licznik;
- rwlock - synchronizacja wątków piszących i czytających;
- join - synchronizacja do zakończenia wątku;
- spinlock - oczekiwanie na alokację pamięci;
- sleepn - podobnie do condvars, z dynamiczną alokacją;
- barrier - oczekiwanie na określoną liczbę wątków.

Na laboratorium będziemy badać działanie trzech pierwszych metod.

Przygotowanie platformy i środowiska IDE

Uruchom QNX Neutrino za pomocą maszyny wirtualnej. Sprawdź poleceniem **pidin** czy jest uruchomiony proces **gconn**. Jeśli nie ma go na liście pracujących procesów uruchom go poleceniem **gconn #**. Sprawdź adres IP poleceniem **ifconfig**. Następnie uruchom środowisko QNX Momentics IDE. Wybierz przestrzeń roboczą wskazaną przez prowadzącego.

Przeanalizuj poniższy kod programu Mutex.

```
/*  
The exercise is to use the mutex construct that we learned about to  
modify the source to prevent our access problem.  
*/
```

```
#include <stdio.h>  
#include <sys/neutrino.h>  
#include <pthread.h>  
#include <sched.h>
```

```

/*
The number of threads that we want to have running simultaneously.
*/

#define NumThreads      4

/*
The global variables that the threads compete for. To demonstrate
contention, there are two variables that have to be updated
"atomically". With RR scheduling, there is a possibility that one thread
will update one of the variables, and get preempted by another thread,
which will update both. When our original thread runs again, it will
continue the update, and discover that the variables are out of sync.
*/

volatile int    var1;
volatile int    var2;

void    *update_thread (void *);

char    *progrname = "mutex";
pthread_mutex_t var_mutex;

main () {
    pthread_t      threadID [NumThreads]; // a place to hold
                                           // the thread ID's
    pthread_attr_t  attrib;              // scheduling
                                           // attributes
    struct sched_param param;           // for setting
                                           // priority

    int            i, policy;

    setvbuf (stdout, NULL, _IOLBF, 0);

    var1 = var2 = 0;    /* initialize to known values */

    printf ("%s: starting; creating threads\n", progrname);
/*
Mutex initialization
*/
    pthread_mutex_init(&var_mutex, NULL );

/*
We want to create the new threads using Round Robin scheduling, and a
lowered priority, so set up a thread attributes structure. We use a
lower priority since these threads will be hogging the CPU.
*/

    pthread_getschedparam( pthread_self(), &policy, &param );
    pthread_attr_init (&attrib);
    pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy (&attrib, SCHED_RR);
    param.sched_priority -= 2;    // drop priority a couple
                                // levels

```

```

pthread_attr_setschedparam (&attrib, &param);
attrib.flags |= PTHREAD_CANCEL_ASYNCHRONOUS;

/*
Create the threads.  As soon as each pthread_create call is done, the
thread has been started.
*/
for(i = 0; i < NumThreads; i++) {
    pthread_create(&threadID [i], &attrib, &update_thread,
                  (void*)i);
}
/*
Let the other threads run for a while
*/

sleep(20);

/*
And then kill them all
*/

printf("%s:  stopping; cancelling threads\n", progame);
for(i = 0; i < NumThreads; i++) {
    pthread_cancel (threadID [i]);
}

printf("%s:  all done, var1 is %d, var2 is %d\n", progame,
       var1, var2);
fflush (stdout);
sleep(1);
exit(0);
}

/*
The actual thread.

The thread ensures that var1 == var2.  If this is not the case, the
thread sets var1 = var2, and prints a message.
Var1 and Var2 are incremented.
Looking at the source, if there were no "synchronization" problems, then
var1 would always be equal to var2.  Run this program and see what the
actual result is...
*/

void do_work(){
    static int var3;

    var3++;

/*
For faster/slower processors, may need to tune this program by modifying
the frequency of this printf -- add/remove a 0
*/

```

```

if( !(var3 % 10000000) )
    printf ("%s: thread %d did some work\n", progname,
            pthread_self());
}

void *
update_thread (void *i){
    while (1) {
        pthread_mutex_lock(&var_mutex);
        if (var1 != var2) {
            int lvar1, lvar2;
            lvar1 = var1;
            lvar2 = var2;
            var1 = var2;
            pthread_mutex_unlock(&var_mutex);
            printf("%s: thread %d, var1 (%d) is not equal to var2
                (%d)!\n", progname, (int) i, lvar1, lvar2);
        }else
            pthread_mutex_unlock(&var_mutex);
        /* do some work here */
        do_work();
        pthread_mutex_lock(&var_mutex);
        var1++;
        var2++;
        pthread_mutex_unlock(&var_mutex);
    }
    return (NULL);
}

```

9. Uruchom program z projektu **Mutex**. Sprawdź jego działanie. Jaki jest wynik działania programu?
10. Uzupełnij kod projektu **Mutex**, tak aby praca wątków była synchronizowana przy pomocy mutex'a.
11. Sprawdź wynik działania zmodyfikowanego kodu. Sprawdź w jakim stanie są inne wątki, gdy jeden z nich wykonuje kod sekcji krytycznej. Do sprawdzeń użyj **perspektywy QNX System Information**. Wynik działania programu przedstaw prowadzącemu.

Przeanalizuj poniższy kod programu Condvar.

```
/*
Objectives:
This code was a two-state example. The producer (or state 0) did
something which caused the consumer (or state 1) to run. State 1 did
something which caused a return to state 0. Each thread implemented one
of the states.
```

```
This example will have 4 states in its state machine with the following
state transitions:
State 0 -> State 1
State 1 -> State 2 if state 1's internal variable indicates "even"
State 1 -> State 3 if state 1's internal variable indicates "odd"
State 2 -> State 0
State 3 -> State 0
```

```
And, of course, one thread implementing each state, sharing the same
state variable and condition variable for notification of change in the
state variable.
```

```
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <sched.h>

/*
Our global variables.
*/
volatile int          state;          // which state we are in

/*
Our mutex and condition variable
*/

pthread_mutex_t      mutex ;
pthread_cond_t       cond  ;

void    *state_0 (void *);
void    *state_1 (void *);
void    *state_2 (void *);
void    *state_3 (void *);

char    *progname = "condvar";

main () {
    setvbuf (stdout, NULL, _IOLBF, 0);

    pthread_mutex_init( &mutex, NULL );
    pthread_cond_init( &cond, NULL );
    state = 0;

    pthread_create (NULL, NULL, state_1, NULL);
    pthread_create (NULL, NULL, state_0, NULL);
```

```

pthread_create (NULL, NULL, state_2, NULL);
pthread_create (NULL, NULL, state_3, NULL);

sleep (20);      // let the threads run
printf ("%s:  main, exiting\n", progname);
}

/*
State 0 handler (was producer)
*/

void *
state_0 (void *arg){
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state != 0) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("%s:  transit 0 -> 1\n", progname);
        state = 1;
        /* don't chew all the CPU time */
        delay(100);
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock (&mutex);
    }
    return (NULL);
}

/*
State 1 handler (was consumer)
*/
void *
state_1 (void *arg){
    int i;
    //int new_state = 2;
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state != 1) {
            pthread_cond_wait (&cond, &mutex);
        }
        if( ++i & 0x1 ){
            state = 3;
        }
        else{
            state = 2;
        }
    }

    // "clever" ways to set next state
    //     state = (++i & 0x1)? 3:2;
    //
    //     state = new_state ^= 0x1;
    //

```

```

    printf ("%s: transit 1 -> %d\n", progame, state);
    pthread_cond_broadcast (&cond);
    pthread_mutex_unlock (&mutex);
}
return (NULL);
}

void *
state_2 (void *arg){
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state != 2) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("%s: transit 2 -> 0\n", progame);
        state = 0;
        pthread_cond_broadcast (&cond);
        pthread_mutex_unlock (&mutex);
    }
    return (NULL);
}

void *
state_3 (void *arg){
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state != 3) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("%s: transit 3 -> 0\n", progame);
        state = 0;
        pthread_cond_broadcast (&cond);
        pthread_mutex_unlock (&mutex);
    }
    return (NULL);
}

```

12. Uruchom program z projektu **Condvar**. Sprawdź jego działanie. Jaki jest wynik działania programu?
13. Zmodyfikuj kod programu **Condvar** w taki sposób, aby wszystkie wątki *consumer*, wykonały pracę po zmianie stanu zmiennej warunkowej.
14. Sprawdź wynik działania uruchomionego kodu z pkt 11. Do sprawdzeń użyj **perspektywy QNX System Information**. Wynik działania programu przedstaw prowadzącemu.

Przeanalizuj poniższy kod programu SEMEX.

```
/*
This module demonstrates POSIX semaphores.
Operation:
A counting semaphore is created, primed with 0 counts. Five consumer
threads are started, each trying to obtain the semaphore. A producer
thread is created, which periodically posts the semaphore, unblocking
one of the consumer threads.
*/

#include <stdio.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <malloc.h>

/*
Our global variables, and forward references
*/

sem_t *mySemaphore;

void *producer (void *);
void *consumer (void *);

char *progname = "semex";

#define SEM_NAME "/Semex"

main () {
    int i;

    setvbuf (stdout, NULL, _IOLBF, 0);

    // #define Named
    #ifdef Named
        mySemaphore = sem_open (SEM_NAME, O_CREAT, S_IRWXU, 0);
        /* not sharing with other process, so immediately unlink */
        sem_unlink( SEM_NAME );
    #else // Named
        mySemaphore = malloc (sizeof (sem_t));
        sem_init (mySemaphore, 1, 0);
    #endif // Named

    for(i = 0; i < 5; i++){
        pthread_create (NULL, NULL, consumer, (void *) i);
    }
    pthread_create (NULL, NULL, producer, (void *) 1);
    sleep (20); // let the threads run
    printf ("%s: main, exiting\n", progname);
}
```



```

/*
Producer
*/

void *
producer (void *i){
    while (1) {
        sleep (1);
        printf ("%s: (producer %d), posted semaphore\n", progname,
                (int) i);
        sem_post (mySemaphore);
    }
    return (NULL);
}

/*
Consumer
*/

void *
consumer (void *i){
    while (1) {
        sem_wait (mySemaphore);
        printf ("%s: (consumer %d) got semaphore\n", progname,
                (int) i);
    }
    return (NULL);
}

```

15. Uruchom program z projektu **Semex**. Sprawdź jego działanie. Jaki jest wynik działania programu? Jakiego rodzaju semafora używa program?
16. Zmodyfikuj kod programu **Semex** w taki sposób, aby wątki do synchronizacji swojej pracy używały innego rodzaju semafora.
17. Uruchom i sprawdź działanie zmodyfikowanego kodu.

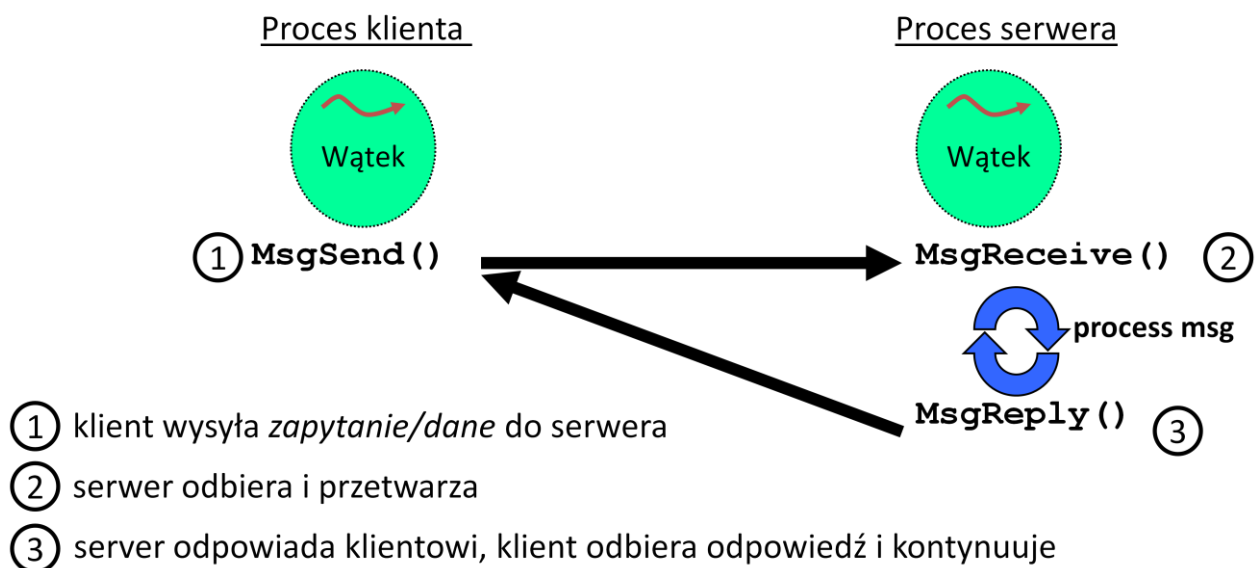
Komunikacja IPC Inter-Proces Communication

QNX Neutrino wspiera różnorodne mechanizmy komunikacji IPC:

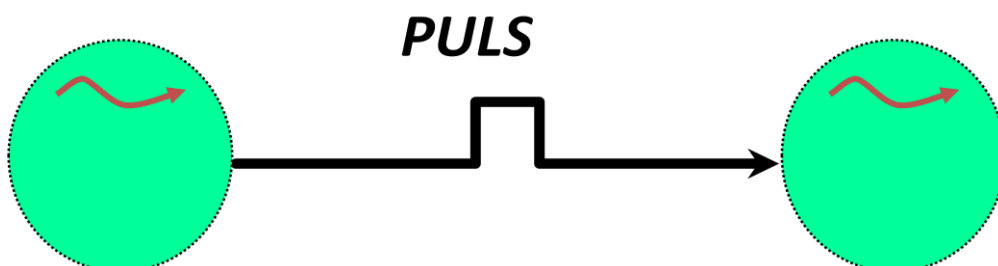
- rodzima komunikacja QNX Neutrino - (API unikalne dla QNX):
 - komunikaty QNX Neutrino;
 - pulsy QNX Neutrino;
- mechanizmy zgodne z POSIX/UNIX (API przenośne):
 - sygnały;
 - pamięć dzielona;
 - potoki (wymagany proces *pipe*);
 - kolejki komunikatów POSIX (wymagany proces *mqueue* lub *mq*);
 - gniazda TCP/IP (wymagany proces *io-net*)

Na laboratorium zapoznamy się z rodzimą komunikacją QNX Neutrino.

Komunikacja bazująca na komunikatach QNX Neutrino oparta jest na modelu klient-serwer. Jest dwukierunkowa i synchroniczna. Oznacza to iż klient po wysłaniu komunikatu do serwera czeka na jego odpowiedź. Dopiero po otrzymaniu informacji zwrotnej może kontynuować pracę. W trakcie przesyłania komunikatów mamy do czynienia z fizycznym kopiowaniem danych pomiędzy obszarami pamięci.



Do komunikacji asynchronicznej używane są pulsy - krótkie powiadomienia.



Zalety pulsu:

- nieblokujący dla nadawcy;
- ustalony rozmiar:
 - 32-bitowa wartość,
 - 8-bitowy kod(-128 do 127);
- jednokierunkowe (nie wymagają odpowiedzi);
- szybkie i „niedrogie”

Przygotowanie platformy i środowiska IDE

Uruchom QNX Neutrino za pomocą maszyny wirtualnej. Sprawdź poleceniem **pidin** czy jest uruchomiony proces **qconn**. Jeśli nie ma go na liście pracujących procesów uruchom go poleceniem **qconn #**. Sprawdź adres IP poleceniem **ifconfig**. Następnie uruchom środowisko QNX Momentics IDE. Wybierz przestrzeń roboczą wskazaną przez prowadzącego.

Zadanie 1.

Przeanalizuj kod źródłowy poniższego programu **server**. Odszukaj w nim fragmenty odpowiedzialne za zestawienie połączenia po stronie serwera. Wypisz odpowiednie funkcje w sprawozdaniu.

```
/*
server.c
  A QNX msg passing server. It should receive a string from a client,
  calculate a checksum on it, and reply back the client with the checksum.
  The server prints out its pid and chid so that the client can be
  made aware of them.
  Using the comments below, put code in to complete the program. Look
  up function arguments in the course book or the QNX documentation.
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <process.h>

#include "msg def.h" //layout of msg's should always defined by
//a struct, here's it's definition
int calculate_checksum(char *text);

int main(void) {
  int chid;
  int pid;
  int rvid;
  cksum_msg_t msg;
  int status;
  int checksum;
```

```

    setvbuf (stdout, NULL, _IOLBF, 0); //set IO to stdout to be line
buffered

    chid = ChannelCreate(0); //PUT CODE HERE to create a channel
    if(-1 == chid) { //was there an error creating the
channel?
        perror("ChannelCreate()"); //look up the errno code and print
        exit(EXIT_FAILURE);
    }

    pid = getpid(); //get our own pid
//print our pid/chid so client can be told where to connect
    printf("Server's pid: %d, chid: %d\n", pid, chid);
    while(1) {
        //PUT CODE HERE to receive msg from client
        rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
        if(rcvid == -1) { //Was there an error receiving msg?
            perror("MsgReceive"); //look up errno code and print
            break; //try receiving another msg
        }

        checksum = calculate_checksum(msg.string_to_cksum);
        //PUT CODE HERE TO reply to client with checksum
        status = MsgReply(rcvid, EOK, &checksum, sizeof(checksum) );
        if(-1 == status) {
            perror("MsgReply");
        }
    }
    return 0;
}

int
calculate_checksum(char *text){
    char *c;
    int cksum = 0;

    for (c = text; *c != NULL; c++)
        cksum += *c;
    return cksum;
}

```

Na podstawie powyższego kodu zmodyfikuj kod źródłowy projektu **server** tak, aby mógł odbierać komunikaty od klienta. Po modyfikacjach uruchom program **server**.

Zadanie 2.

Przeanalizuj kod źródłowy poniższego programu **client**. Odszukaj w nim fragmenty odpowiedzialne za zestawienie połączenia po stronie klienta. Wypisz odpowiednie funkcje w sprawozdaniu.

```
/*
client.c
```

```
A QNX msg passing client. It's purpose is to send a string of text
to a server. The server calculates a checksum and replies back with it.
The client expects the reply to come back as an int.
```

```
This program must be started with commandline args. See
if(argc != 4) below.
```

```
To complete the exercise, put in the code, as explained in the comments
below. Look up function arguments in the course book or the QNX
documentation.
```

```
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h> // #define for ND_LOCAL_NODE is in here

#include "msg def.h"

int main(int argc, char* argv[])
{
    int coid; //Connection ID to server
    cksum_msg_t msg;
    int incoming_checksum; //space for server's reply
    int status; //status return value used for ConnectAttach
and MsgSend
    int server_pid; //server's process ID
    int server_chid; //server's channel ID

    setvbuf (stdout, NULL, _IOLBF, 0);

    if(4 != argc) {
        printf("This program must be started with commandline arguments, for
        example:\n\n");
        printf(" cli 482834 1 abcdefghi \n\n");
        printf(" 1st arg(482834): server's pid\n");
        printf(" 2nd arg(1): server's chid\n");
        printf(" 3rd arg(abcdefghi): string to send to server\n"); //to make
        it easy, let's not bother handling spaces
        exit(EXIT_FAILURE);
    }
}
```

```

server_pid = atoi(argv[1]);
server_chid = atoi(argv[2]);

printf("attempting to establish connection with server pid: %d, chid
      %d\n", server_pid, server_chid);
//PUT CODE HERE to establish a connection to the server's channel
coid = ConnectAttach(ND_LOCAL_NODE, server_pid, server_chid,
                    _NTO_SIDE_CHANNEL, 0);
if(-1 == coid) { //was there an error attaching to server?
  perror("ConnectAttach"); //look up error code and print
  exit(EXIT_FAILURE);
}

msg.msg_type = CKSUM_MSG_TYPE;
strcpy(msg.string_to_cksum, argv[3]);
printf("Sending string: %s\n", msg.string_to_cksum);

//PUT CODE HERE to send message to server and get the reply
status = MsgSend(coid, &msg, sizeof(msg), &incoming_checksum,
                sizeof(incoming_checksum) );
if(-1 == status) { //Was there an error sending to server?
  perror("MsgSend");
  exit(EXIT_FAILURE);
}

printf("received checksum=%d from server\n", incoming_checksum);
printf("MsgSend return status: %d\n", status);

return EXIT_SUCCESS;
}

```

Na podstawie powyższego kodu zmodyfikuj kod źródłowy projektu **client** tak, aby mógł wysyłać komunikaty do serwera. Po modyfikacjach uruchom program **client**.

Zadanie 3.

Zmień kod programu **client** tak, aby komunikat do serwera nadawany był cyklicznie.

Zastanów się, jak zmodyfikować kody powyższych programów **server** i **client**, aby odpowiedzieć na następujące pytania:

W jakim stanie jest client nim server gotowy będzie do odbierania komunikatów?

W jakim stanie jest client gdy server odbierze komunikat?

W jakim stanie jest server nim klient wyśle do niego komunikat?

Dokonaj odpowiednich modyfikacji kodu i udziel odpowiedzi na wyżej postawione pytania. Zaprezentuj wyniki prowadzącemu w perspektywie **QNX System Information**.

Zadanie 4.

Przeanalizuj kod źródłowy poniższego programu **pulse_server**.

```
/*
 pulse_server.c
 A QNX msg passing server. It should receive a string from a client,
 calculate a checksum on it, and reply back the client with the checksum.
 The server prints out its pid and chid so the client can be made
 aware of them.
 Using the comments below, put code in to complete the program. Look
 up function arguments in the course book or the QNX documentation.
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <process.h>

#include "msg def.h" //layout of msg's should always defined by a
struct, here's
//it's definition

int
calculate_checksum(char *text);

int
main(void) {
    int chid;
    int pid;
    int rvid;
    msg_buf_t msg;
    int status;
    int checksum;

    setvbuf (stdout, NULL, _IOLBF, 0); //set IO to stdout to be line
buffered
```

```

chid = ChannelCreate(0); //PUT CODE HERE to create a channel
if(-1 == chid) { //was there an error creating the
channel?
    perror("ChannelCreate()"); //look up the errno code and print
    exit(EXIT_FAILURE);
}

pid = getpid(); //get our own pid
//print our pid/chid so client can be told where to connect
printf("Server's pid: %d, chid: %d\n", pid, chid);

while(1) {
    //PUT CODE HERE to receive msg from client
    rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
    if(rcvid == -1) { //Was there an error receiving msg?
        perror("MsgReceive"); //look up errno code and print
        continue; //try receiving another msg
    }
    else if (rcvid == 0) {
        printf("a pulse was received, its code is %d, its value is
%d\n",
                msg.pulse.code, msg.pulse.value.sival_int);
        continue;
    }

    checksum = calculate_checksum(msg.cksum.string_to_cksum);
    //PUT CODE HERE TO reply to client with checksum
    status = MsgReply(rcvid, EOK, &checksum, sizeof(checksum) );
    if(-1 == status) {
        perror("MsgReply");
    }
    return 0;
}

int
calculate_checksum(char *text)
{
    char *c;
    int cksum = 0;

    for (c = text; *c != NULL; c++)
        cksum += *c;
    return cksum;
}

```

Na podstawie powyższego kodu zmodyfikuj kod źródłowy projektu **pulse_server** tak, aby mógł odbierać pulsy do klienta. Po modyfikacjach uruchom program **pulse_server**.

Zadanie 5.

Przeanalizuj kod źródłowy poniższego programu **pulse_client**.

```
/*
 pulse_client.c

 A QNX msg passing client. It's purpose is to send a string of text
 to a server. The server calculates a checksum and replies back with it.
 The client expects the reply to come back as an int
 This program must be started with commandline args. See
 if(argc != 4) below.
 To complete the exercise, put in the code, as explained in the
 comments below. Look up function arguments in the course book or the QNX
 documentation.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h> // #define for ND_LOCAL_NODE is in here

#include "msg def.h"

#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL
#define MY_OTHER_PULSE_CODE _PULSE_CODE_MINAVAIL + 7

int main(int argc, char* argv[])
{
    int coid; //Connection ID to server
    int status; //status return value used for ConnectAttach and MsgSend
    int server_pid; //server's process ID
    int server_chid; //server's channel ID
    int some_value = 25; //make up some kind of number for a value

    setvbuf (stdout, NULL, _IOLBF, 0);

    if(4 != argc) {
        printf("This program must be started with commandline arguments, for
        example:\n\n");
        printf(" cli 482834 1 abcdefghi \n\n");
        printf(" 1st arg(482834): server's pid\n");
        printf(" 2nd arg(1): server's chid\n");
        printf(" 3rd arg(abcdefghi): string to send to server\n"); //to make
        //it easy, let's not bother handling spaces
        exit(EXIT_FAILURE);
    }

    server_pid = atoi(argv[1]);
    server_chid = atoi(argv[2]);
}
```

```

printf("attempting to establish connection with server pid: %d, chid
      %d\n", server_pid, server_chid);
//PUT CODE HERE to establish a connection to the server's channel
coid = ConnectAttach(ND_LOCAL_NODE, server_pid, server_chid,
                    _NTO_SIDE_CHANNEL, 0);
if(-1 == coid) { //was there an error attaching to server?
    perror("ConnectAttach"); //look up error code and print
    exit(EXIT_FAILURE);
}

status = MsgSendPulse(coid, getprio(0), MY_PULSE_CODE, 0);
if(status == -1) {
    perror("problmem with 1st MsgSendPulse");
}

status = MsgSendPulse(coid, getprio(0), MY_OTHER_PULSE_CODE,
                    some_value);
if(status == -1) {
    perror("problmem with 2nd MsgSendPulse");
}

return EXIT_SUCCESS;
}

```

Na podstawie powyższego kodu zmodyfikuj kod źródłowy projektu **pulse_client** tak, aby mógł wysyłać pulsy do serwera. Po modyfikacjach uruchom program **pulse_client**.

Zadanie 6.

Zmodyfikuj program **pulse_client** tak aby pulsy wysyłane były cyklicznie. Dodaj własny pulse. Sprawdź w jakim stanie jest proces **pulse_client** po wysłaniu pulsu.

Zadanie 7.

Przeanalizuj kod źródłowy poniższego programu **name_lookup_server**. Odszukaj w nim fragmenty odpowiedzialne za zestawienie połączenia po stronie serwera. Zamieść odpowiednie funkcje w sprawozdaniu.

```
/*
  name_lookup_server.c

  A QNX msg passing server. It should receive a string from a client,
  calculate a checksum on it, and reply back the client with the checksum.
  The server prints out its pid and chid so that the client can be
  made aware of them.
  Using the comments below, put code in to complete the program. Look
  up function arguments in the course book or the QNX documentation.
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <process.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

#include "msg_def.h" //layout of msg's should always defined by a
struct,
// here's it's definition

int
calculate_checksum(char *text);

int
main(void) {
  int ravid;
  msg_buf_t msg;
  int status;
  int checksum;
  name_attach_t* attach;

  setvbuf (stdout, NULL, _IOLBF, 0); //set IO to stdout to be line
buffered
  //PUT CODE HERE to attach a name
  attach = name_attach(NULL, SERVER_NAME, 0);
  if(NULL == attach) { //Was there an error creating the channel?
    perror("name_attach()"); //look up the errno code and print
    exit(EXIT_FAILURE);
  }
}
```

```

while(1) {
//PUT CODE HERE to receive msg from client, store the receive id in
rcvid
rcvid = MsgReceive(attach->chid, &msg, sizeof msg, NULL);
if(rcvid == -1) { //was there an error receiving msg?
perror("MsgReceive"); //look up errno code and print
continue; //try receiving another msg
}
else if(rcvid == 0) {
printf("received pulse, code = %d\n", msg.pulse.code);
continue;
}
printf("received msg: %s\n", msg.cksum.string_to_cksum);
checksum = calculate_checksum(msg.cksum.string_to_cksum);

//PUT CODE HERE TO reply to client with checksum, store the return
status //in status
status = MsgReply(rcvid, EOK, &checksum, sizeof checksum);
if(-1 == status) {
perror("MsgReply");
}
}
return 0;
}

int
calculate_checksum(char *text)
{
char *c;
int cksum = 0;

for (c = text; *c != NULL; c++)
cksum += *c;
return cksum;
}

```

Na podstawie powyższego kodu zmodyfikuj kod źródłowy projektu **name_lookup_server** tak, aby mógł odbierać komunikaty do klienta. Po modyfikacjach uruchom program **name_lookup_server**.

Zadanie 8.

Przeanalizuj kod źródłowy poniższego programu **name_lookup_client**. Odszukaj w nim fragmenty odpowiedzialne za zestawienie połączenia po stronie klienta. Wypisz odpowiednie funkcje w sprawozdaniu.

```
/*
   name_lookup_client.c

   A QNX msg passing client.  It's purpose is to send a string of text
   to a server.  The server calculates a checksum and replies back with it.
   The client expects the reply to come back as an int.
   This program must be started with commandline args.  See
   if(argc != 4) below
   To complete the exercise, put in the code, as explained in the
   comments below.  Look up function arguments in the course book or the QNX
   documentation.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h>      // #define for ND_LOCAL_NODE is in here
#include <sys/iofunc.h>
#include <sys/dispatch.h>

#include "msg def.h"

int main(int argc, char* argv[])
{
    int coid;                //Connection ID to server
    msg_buf_t msg;
    int incoming_checksum;  //space for server's reply
    int status;             //status return value used for ConnectAttach
and
                                //MsgSend

    setvbuf (stdout, NULL, _IOLBF, 0);

    if(2 != argc) {
        printf("This program must be started with commandline arguments, for
            example:\n\n");
        printf("    client abcdefghi    \n\n");
        printf(" where (abcdefghi) is string to send to server\n"); //to
make
                                //it easy, let's not bother handling
spaces
        exit(EXIT_FAILURE);
    }
}
```

```

printf("client: attempting to establish connection with server %s\n",
SERVER_NAME);
//PUT CODE HERE to establish a connection to the server's channel, store
the connection id in the variable 'coid'
coid = name_open(SERVER_NAME, 0);
if(-1 == coid) { //was there an error attaching to server?
    perror("ConnectAttach"); //look up error code and print
    exit(EXIT_FAILURE);
}

msg.cksum.msg_type = CKSUM_MSG_TYPE;
strcpy(msg.cksum.string_to_cksum, argv[1]);
printf("Sending string: %s\n", msg.cksum.string_to_cksum);

//PUT CODE HERE to send message to server and get the reply
status = MsgSend(coid, &msg, sizeof msg, &incoming_checksum,
                sizeof(incoming_checksum));
if(-1 == status) { //was there an error sending to server?
    perror("MsgSend");
    exit(EXIT_FAILURE);
}

printf("received checksum=%d from server\n", incoming_checksum);
printf("MsgSend return status: %d\n", status);

return EXIT_SUCCESS;
}

```

Na podstawie powyższego kodu zmodyfikuj kod źródłowy projektu **name_lookup_client** tak, aby mógł wysyłać komunikaty do serwera. Po modyfikacjach uruchom program **name_lookup_client**.

Zadanie 9.

Zmodyfikuj kody programów **name_lookup_client** i **name_lookup_server** tak aby komunikaty były wysyłane i odbierane cyklicznie.

Sprawozdanie.

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

Grupa:

