

# **Metody Programowania Robotów**

*Zestaw instrukcji laboratoryjnych do programowania aplikacji  
w systemie operacyjnym czasu rzeczywistego QNX Neutrino*

**Paweł Małczyk, Paweł Tomulik**

Zakład Teorii Maszyn i Robotów  
Instytut Techniki Lotniczej i Mechaniki Stosowanej  
Wydział Mechaniczny Energetyki i Lotnictwa  
Politechnika Warszawska  
pmalczyk@meil.pw.edu.pl, ptomulik@meil.pw.edu.pl  
www: <http://ztmir.meil.pw.edu.pl>

październik 2015 r.



# Laboratorium 1

## Podstawy obsługi systemu operacyjnego QNX RTOS

---

### 1.1 Powłoka

Powłoka:

- Interpreter poleceń użytkownika
- Pośredniczy między użytkownikiem a systemem
- Środowisko pracy użytkownika w systemie
- Przetwarza pojedyncze polecenia lub skrypty

**Przykład 1.1.1** [Przykład prostej komendy]. Polecenia można wydawać powłoce z wiersza poleceń (terminala, konsoli) wpisując ich nazwy.

```
1 # ls
```

Komenda wyświetla zawartość bieżącego katalogu.

**Przykład 1.1.2** [Przykład komendy z argumentami]. Komenda wyświetla zawartość bieżącego katalogu wraz ze szczegółowymi informacjami nt. obiektów. Argumenty (-l) zmieniają zachowanie komendy prostej.

```
1 # ls -l
```

Formalna składnia komendy z argumentami:

```
1 # command argument1 argument2 argument3 ... argumentN
```

**Przykład 1.1.3** [Przykład złożonej komendy]. Komendy proste i komendy proste z argumentami możemy łączyć. Separatorem poleceń jest średnik (;).

```
1 # date ; ls
```

Formalna składnia złożonej komendy:

```
1 # command1; command2; command3; ... argumentN
```

**Przykład 1.1.4** [Wejście i wyjście z powłoki]. Powłoka wykonuje polecenia użytkownika. Kiedy powłoka wyświetla znak zachęty, który w domyślnym shell-u (Bourne shell) w systemie QNX jest znak #

dla użytkownika `root`, to czeka na polecenia użytkownika. Możemy uruchomić powłokę w takim trybie. Sytuację tę ilustruje poniższy przykład.

```
1 # /bin/sh
2 #
3 # exit
```

Aby wyjść z powłoki należy użyć polecenia `# exit`.

**Przykład 1.1.5** [Obsługa edytora tekstu vi]. Edytor `vi` jest zaawansowanym edytorem tekstowym często występującym w systemach Unix. Aby uruchomić edytor należy wpisać komendę:

```
1 # vi
```

Edytor `vi` jest edytorem modalnym. Oznacza to, że może znajdować się w dwóch stanach: **trybie edycji** lub **trybie poleceń**.

- Przejście do trybu edycji poprzez wydanie polecenia `i` (insert) lub `a` (append).
- Przejście z trybu edycji do trybu poleceń odbywa się poprzez naciśnięcie klawisza `Esc`.

Polecenia edytora `vi` składają się z kilku grup. Przedstawiono je zbiorczo w postaci krótkiej instrukcji na stronie 6.

W trybie edycji, tj. po wciśnięciu klawisza `i` wpiszmy do pliku następujące linie tekstu postaci, np.:

```
1 Jak dobrze wstac
2 Skoro swit
3 Jutrzenki blask
4 Duszkiem pic
5 Tytul:
6 Radosc o poranku
7 Nim w gorze tam
8 Skowronek zacznie tryl
9 Jak dobrze wczesnie wstac
10 Dla tych chwil
```

W następnej kolejności przejść do trybu poleceń, poprzez wciśnięcie klawisza `Esc` oraz zapisać plik wydając polecenie:

```
1 :w plikPoranny
```

Kolejno, zamknąć zapisany plik komendą:

```
1 :q
```

Otworzyć ponownie plik:

```
1 vi plikPoranny
```

Wykonać następujące eksperymenty:

- Wykasować tekst **Tytul**: litera po literze poleceniem **x**. Wykasować także pustą linię, powstałą w miejscu tekstu **Tytul**: (polecenie **dd**).
- Klawiszami **h**, **j**, **k**, **l** przejść do linii z tekstem **Radosc o poranku**.
- Wykasować bieżącą linię za pomocą komendy **dd**.
- Przejść do pierwszej linii pliku (**h**, **j**, **k**, **l**) i wkleić usunięty tekst kombinacją klawiszy **Shift + P**, tak, aby cały tekst wyglądał następująco:

```
1 Radosc o poranku
2 Jak dobrze wstac
3 Skoro swit
4 Jutrzenki blask
5 Duszkiem pic
6 Nim w gorze tam
7 Skowronek zacznie tryl
8 Jak dobrze wczesnie wstac
9 Dla tych chwil
```

Przeprowadzić kolejne eksperymenty:

- W trybie poleceń, znaleźć wszystkie wystąpienia słowa **dobrze** wpisując:

```
1 /dobrzespace
```

- Przeszukać tekst w przód poprzez naciśnięcie klawisza **N**. Przeszukiwanie w tył nastąpi poprzez naciśnięcie klawisza **Shift + N**.
- Przejść do pierwszej linii poleceniem **:1** oraz zamienić tekst **Radosc** na **Tytul: Radosc** poprzez sekwencję:

```
1 :s/Radosc/Tytul: Radosc
```

- Zamienić wszystkie wystąpienia słowa **dobrze** na **DOBRZE** w zakresie od bieżącej linii **.** do ostatniej linii **\$**:

```
1 :..,$s/dobrzespace/DOBRZE
```

Zapisać i zamknąć plik poleceniem:

```
1 :wq!
```

## Vi Reference Card

### Modes

Vi has two modes: insertion mode, and command mode. The editor begins in command mode, where cursor movement and text deletion and pasting occur. Insertion mode begins upon entering an insertion or change command. [ESC] returns the editor to command mode (where you can quit, for example by typing :q!). Most commands execute as soon as you type them except for “colon” commands which execute when you press the return key.

### Quitting

exit, saving changes  
quit (unless changes)  
quit (force, even if unsaved)

### Inserting text

insert before cursor, before line  
append after cursor, after line  
open new line after, line before  
replace one char, many chars

### Motion

left, down, up, right  
next word, blank delimited word  
beginning of word, of blank delimited word  
end of word, of blank delimited word  
sentence back, forward  
paragraph back, forward  
beginning, end of line  
beginning, end of file  
line **n**  
forward, back to char **c**  
forward, back to before char **c**  
top, middle, bottom of screen

### Deleting text

Almost all deletion commands are performed by typing **d** followed by a *motion*. For example **dw** deletes a word. A few other deletions are:

character to right, left  
to end of line  
line  
line

### Yanking text

Like deletion, almost all yank commands are performed by typing **y** followed by a *motion*. For example **y\$** yanks to the end of line. Two other yank commands are:

line  
line

### Changing text

The change command is a deletion command that leaves the editor in insert mode. It is performed by typing **c** followed by a *motion*. For example **cw** changes a word. A few other change commands are:

:x  
:q!  
:q!

### Putting text

put after position or after line  
put before position or before line

### Registers

i , I  
a , A  
o , O  
r , R

### Markers

( , )  
{ , }  
0 , \$  
1G , G  
**n**G or **n**

h , j , k , l  
w , W  
b , B  
e , E  
c , C  
0 , \$  
1G , G  
fc , Fc  
tc , Tc  
H , M , L

### Search for strings

search forward

search backward

repeat search in same, reverse direction

### Replace

x , x  
D  
dd  
:d

~  
join lines  
repeat last text-changing command  
g , c  
g , &

### Regular expressions

any single character except newline  
zero or more repeats  
[...]  
[^ ...]  
~ , \$  
\< , \>  
\n

### Counts

Nearly every command may be preceded by a number that specifies how many times it is to be performed. For example **5dw** will delete 5 words and **3fe** will move the cursor forward to the 3rd occurrence of the letter **e**. Even insertions may be repeated conveniently with this method, say to insert the same line 100 times.

### Ranges

Ranges may precede most “colon” commands and cause them to be executed on a line or lines. For example **:3,7d** would delete lines 3–7. Ranges are commonly combined with the **:s** command to perform a replacement on several lines, as with **:,\$s/pattern/string/g** to make a replacement from the current line to the end of the file.

lines **n-m**  
current line  
last line  
marker **c**  
all lines  
all matching lines

### Files

w file  
w >file  
r file  
!program  
:n  
:p  
:e file  
:!program  
replace line with program output

### Other

toggle upper/lower case  
join lines  
repeat last text-changing command  
undo last change, all changes on line  
u , U

**Przykład 1.1.6** [Utworzenie i uruchomienie skryptu]. Powłoka, jako interpreter, wykonuje pewien program. Kolejne komendy programu mogą być wpisywane na bieżąco w terminalu lub cały program może być dostarczony do powłoki w postaci skryptu. Należy utworzyć plik o nazwie **skrypt** edytorem tekstu vi o treści:

```
1 date ; ls
```

W następnej kolejności uruchomić skrypt powłoki wydając polecenie:

```
1 # /bin/sh skrypt
```

Przykład ilustruje skrypt powłoki. Na ogół skrypty składają się z plików, w których są zapisane komendy, interpretowane przez powłokę. Skrypt można uruchomić wpisując w wiersz poleceń jego nazwę. Jednak bezpośrednie wpisanie jego nazwy kończy się niepowodzeniem:

```
1 # ./skrypt
2 sh: ./skrypt: cannot execute - Permission denied
```

W tej sytuacji należy zapewnić, aby skrypt miał odpowiednie atrybuty oraz upewnić się, że uruchamiany jest właściwy interpreter poleceń. Aby zmienić atrybuty pliku należy użyć następującego polecenia:

```
1 # chmod a+x ./skrypt
```

Uruchomić skrypt w linii poleceń.

**Przykład 1.1.7** [Prosty skrypt powłoki]. Należy także uzupełnić plik **skrypt**, tak, żeby miał postać:

```
1 #!/bin/sh
2 # wypisz date i wyświetl zawartosc katalogu
3 date ; ls
```

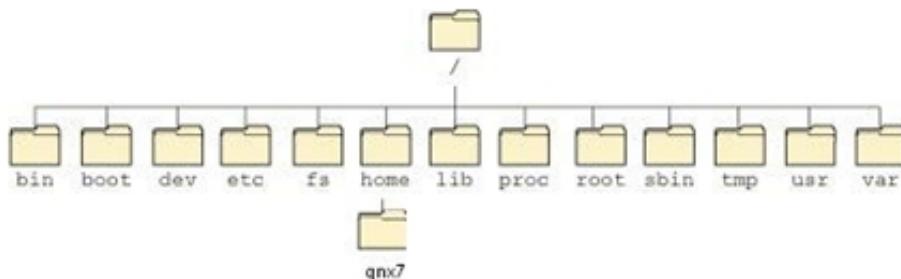
Znak **#** stanowi znak komentarza w skrypcie. Wiersze zaczynające się od tego znaku są ignorowane przez interpreter poleceń i traktowane jako komentarz, oprócz pierwszej linii z komendą **#!/bin/sh**. Pierwsza linia kodu przekazuje informacje o rodzaju powłoki, która powinna wykonać skrypt.

**Przykład 1.1.8** [Prosty skrypt powłoki]. Dokumentacja systemu jest dostępna w formie elektronicznej na stronach [www.qnx.com](http://www.qnx.com). Skrócony opis interesującego nas polecenia systemowego uzyskujemy poprzez wpisanie w okno terminala polecenia **use polecenie**, np.:

```
1 # use date
```

## 1.2 System plików

W systemie QNX Neutrino prawie wszystkie zasoby są plikami. Dane, urządzenia, bloki pamięci, a nawet pewne usługi są reprezentowane przez abstrakcję plików. Mechanizm plików pozwala na jednolity dostęp do zasobów zarówno lokalnych, jak i zdalnych, za pomocą poleceń i programów usługowych wydawanych z okienka poleceń. Typowe drzewo plików w systemie QNX przedstawiono na rysunku 1.



Rysunek 1: Drzewo plików w systemie QNX

- Katalog główny / jest miejscem montowania twardego dysku lub pamięci flash.
- /bin - zawiera podstawowe komendy systemowe (np. ls, chmod).
- /boot - zawiera pliki i katalogi związane z obrazami systemu operacyjnego.
- /dev - katalog przynależy do menadżera zasobów, w którym przechowywane są informacje o urządzeniach dostępnych w systemie.
- /etc - zawiera pliki i programy używane do administracji i konfiguracji systemu.
- /fs - w katalogu są montowane dodatkowe systemy plików.
- /home - katalogi domowe użytkowników.
- /lib - zawiera współdzielone biblioteki używane przez inne programy, procesy.
- /proc - katalog, w którym są zawarte informacje o procesach i przestrzeni nazw.
- /root - katalog domowy dla użytkownika root.
- /sbin - zawiera niezbędne pliki wykonywalne (np. sterowniki, programy inicjujące, konfiguracyjne, menadżery).
- /tmp - zawiera pliki tymczasowe.
- /usr - zawiera współdzielone dane, tylko do odczytu.
- /var - zawiera zmienne dane (np. cache, logi).

Bardziej obszerny opis struktury i hierarchii katalogów w systemach klasy Unix, do których należy QNX, można znaleźć pod adresem [https://en.wikipedia.org/wiki/Unix\\_filesystem](https://en.wikipedia.org/wiki/Unix_filesystem). W systemie QNX występują różne typy plików. Ich zestawienie przedstawiono w tabeli 1.

Pliki są zorganizowane w katalogi. Katalogi mają strukturę drzewa z korzeniem / (root). Aby wskazać na konkretny obiekt w systemie plików, należy podać jego ścieżkę. Rozróżniamy ścieżki absolutne i względne. Ścieżka absolutna zaczyna się od znaku /, np. /etc/passwd. Ścieżka relatywna zaczyna się od znaku innego niż /, np. etc/passwd.

**Tabela 1:** Typy plików w systemie QNX

Oznaczenie	Typ pliku	Opis
d	Katalog (ang. directory)	Plik zawierający inne pliki i katalogi.
l	Dowiązanie symboliczne (ang. symbolic link)	Dodatkowa nazwa pliku, który jest umieszczony w innym miejscu.
n	Plik specjalny	Np. blok pamięci współdzielonej.
c	Specjalny plik znakowy (ang. character device)	Urządzenie z dostępem znakowym (konsole, porty szeregowe, równoległe).
p	Plik specjalny FIFO	Bufor cykliczny w pamięci operacyjnej.
b	Specjalny plik blokowy (ang. block device)	Urządzenie z dostępem blokowym (dysk, partycja dyskowa).
s	Gniazdko (ang. socket)	Plik komunikacji sieciowej.

**Przykład 1.2.1** [Wylistowanie zawartości katalogu]. Pokazać zawartość bieżącego katalogu:

```
1 # ls -l
2 total 419571
3 drwxr-xr-x  2 root      root          3072 Feb 23 2014 bin
4 drwxr-xr-x  4 root      root          1024 Feb 23 2014 boot
5 dr-xr-xr-x  2 root      root           0 Oct 22 15:07 dev
6 drwxr-xr-x 10 root      root          3072 Feb 23 2014 etc
7 drwxr-xr-x  2 root      root          1024 Feb 23 2014 home
8 drwxr-xr-x  4 root      root          5120 Feb 23 2014 lib
9 drwxr-xr-x  3 root      root          1024 Feb 23 2014 libexec
10 dr-xr-xr-x  2 root      root         214798336 Oct 22 15:07 proc
11 drwxr-xr-x  2 root      root          1024 Sep 30 21:43 root
12 drwxr-xr-x  2 root      root          3072 Feb 23 2014 sbin
13 -rw-rw-rw-  1 root      root           11 Oct 22 12:14 skrypt
14 drwxr-xr-x  2 root      root          1024 Oct 22 12:30 tmp
15 drwxr-xr-x  7 root      root          1024 Feb 23 2014 usr
16 drwxr-xr-x  4 root      root          1024 Feb 23 2014 var
```

Pokazać zawartość innego katalogu:

```
1 # ls -la /usr/
```

Argument **-l** służy do listowania w formacie tzw. długim, natomiast argument **-a** do listowania ukrytych plików. Sprawdzić inne parametry polecenia ls następująco:

```
1 # use ls
```

**Przykład 1.2.2** [Obejrzenie zawartości pliku]. Oprócz listowania zawartości katalogów, istotna jest możliwość oglądania zawartości plików (np. skryptowych):

```
1 # cat /skrypt
2 # cat -n /skrypt      # -n numerowanie linii
3 # cat -n /etc/passwd
```

**Przykład 1.2.3** [Wyświetlanie liczby wierszy, słów i bajtów zawartych w pliku]. Aby uzyskać informację o całkowitej liczbie linii, słów i znaków, zawartych w pliku można użyć polecenie wc (ang. word count):

```
1 # wc /skrypt
```

Dostępne argumenty polecenia przedstawiono w tabeli 2.

**Tabela 2:** Opcje polecenia wc

Argumenty	Opis
-l	Oblicza liczbę linii
-w	Oblicza liczbę słów
-m	Oblicza liczbę znaków
-c	Oblicza liczbę znaków

**Przykład 1.2.4** [Poruszanie się po systemie plików]. System operacyjny jest wyposażony w zestaw instrukcji, które umożliwiają poruszanie się po drzewie plików. Najważniejsze zestawiono w tabeli 3 i zilustrowano w przykładzie.

**Tabela 3:** Poruszanie się po systemie plików

Polecenie	Opis
pwd	Wyświetlenie nazwy katalogu bieżącego (ang. print working directory)
cd	Zmiana katalogu bieżącego (ang. change directory)
cd ..	Przejście do katalogu nadzawanego
cd /	Przejście do katalogu głównego

Wprowadzić następujące komendy w wierszu poleceń.

```
1 # pwd      # Nazwa katalogu bieżacego (dla naszego przypadku)
2 /
3 # cd /usr    # Przejscie do katalogu uzytkownika
4 # pwd
5 /usr
6 # cd ..      # Przejscie do katalogu nadzawanego
7 # pwd
8 /
9 # ls .       # Wylistowanie nazw plikow w biezacym katalogu
10 ...
11 # ls -l      # Wylistowanie ze szczegolami
```

**Przykład 1.2.5** [Utworzenie i kasowanie katalogów]. Pliki w systemie plików można tworzyć i usuwać. Katalogi można również modyfikować. Najczęściej używane polecenia służą do tworzenia, kopiowania, przenoszenia oraz usuwania katalogów. Zestawienie podstawowych polecień podano w tabeli 4.

Sprawdzić działanie następujących komend:

**Tabela 4:** Tworzenie i usuwanie katalogów i plików

Polecenie	Opis
touch plik	Utworzenie pustego pliku lub zmiana daty modyfikacji istniejącego
cd	Zmiana katalogu bieżącego (ang. change directory)
rm [-Rf] plik	Usunięcie pliku: -i - żądanie potwierdzenia; -f - bezwarunkowe kasowanie pliku; -R - kasowanie zawartości katalogu z podkatalogami
mkdir katalog	Utworzenie katalogu o nazwie katalog
rmdir katalog	Usunięcie katalogu o nazwie katalog

```

1 # pwd      # Nazwa katalogu bieżacego
2 /
3 # cd /home      # Przejdz do katalogu home
4 # mkdir katalog    # Utworzenie katalogu
5 # cd katalog      # Przejdz do katalogu
6 # mkdir podkatalog    # Utworzenie podkatalogu
7 # touch plik      # Utworzenie pustego pliku
8 # touch plik2     # Utworzenie pustego pliku
9 # rm plik2       # Usuniecie pustego pliku
10 # rmdir podkatalog   # Usuniecie podkatalogu
11 # cd ..        # Wyjscie do katalogu nadziednego
12 # rmdir katalog    # Proba usuniecia podkatalogu
13 katalog/: Directory not empty
14 # rm -Ri katalog    # Usuniecie rekursywne z potwierdzeniem
15 rm: remove katalog/plik? (y/N) y
16 rm: remove directory katalog? (y/N) y

```

**Przykład 1.2.6** [Przenoszenie i kopianie katalogów i plików]. Pliki i katalogi można kopować i przekopiować. Zestawienie najważniejszych komend przedstawiono w tabeli 5.

**Tabela 5:** Przenoszenie i kopianie katalogów i plików

Polecenie	Opis
mv [-if] zrodlo cel	Przenoszenie lub zmiana nazwy plików: -i - żądanie potwierdzenia, gdy plik docelowy może być nadpisany; -f - bezwarunkowe skopiowanie pliku
cp [-ifR] zrodlo cel	Kopianie plików: -i - żądanie potwierdzenia, gdy plik docelowy może być nadpisany; -f - bezwarunkowe skopiowanie pliku; -R - kopianie zawartości katalogu z podkatalogami

Sprawdzić działanie następujących komend:

```
1 # pwd
```

```
2 /home
3 # mkdir katalog
4 # cd katalog
5 # touch pliczek
6 # cd ..
7 # mv ./katalog/pliczek .      # Przeniesienie do katalogu
      biezacego
8 # mv pliczek plik.dat      # Zmiana nazwy pliku
9 # cp plik.dat katalog      # Skopiowanie pliku do katalogu
10 # cp -Ri katalog katalog2    # Skopiowanie katalogu do katalogu2
      razem z zawartoscia
```

**Przykład 1.2.7** [Zmiana atrybutów pliku]. Pliki w systemie mają określonych właścicieli, grupy użytkowników, a także zestawy atrybutów do nich przypisanych. System umożliwia dostęp do plików w trybie odczytu, zapisu lub wykonania. Symboliczne oznaczenia praw dostępu do pliku są następujące:

- r - prawo odczytu (ang. read)
- w - prawo zapisu (ang. write)
- x - prawo wykonania (ang. execute)

Prawa te mogą być zdefiniowane dla właściciela pliku, grupy, do której on należy i wszystkich innych użytkowników.

- u - właściciel pliku (ang. user)
- g - grupa (ang. group)
- o - inni użytkownicy (ang. other)

Aby obejrzeć właściciela pliku oraz atrybuty wykonajmy następujące polecenie:

```
1 # ls -l /home/plik.dat
2 -rw-rw-rw- 1 root root 0 Oct 22 11:41 /home/plik.dat
```

W terminalu wyświetcone zostały w kolejności atrybuty dla właściciela (rw-), grupy (rw-) oraz innych użytkowników (rw-); wskazano także liczbę dowiązań 1, nazwę właściciela pliku root, nazwę grupy root, rozmiar pliku 0, datę utworzenia Oct 22 11:41 oraz nazwę pliku home/plik.dat.

Atrybuty plików oraz ich właścicieli można zmieniać - zobacz tabelę 6.

**Tabela 6:** Tworzenie i usuwanie katalogów i plików

Polecenie	Opis
chmod	Zmiana atrybutów dla pliku, bądź katalogu
chown	Zmiana właściciela (lub opcjonalnie grupy) dla pliku, bądź katalogu
chgrp	Zmiana grupy dla pliku, bądź katalogu

Zmiana atrybutów pliku odbywa się wg następujące składni:

```
1 # chmod właściciel akcja atrybuty
```

gdzie `własciciel` jest jednym ze skrótów literowych (u, g, o, bądź a - dla wszystkich użytkowników), `atrybuty` dotyczą oznaczeń (r, w lub x). Możliwe do wykonania akcje opisano w tabeli 7.

**Tabela 7:** Zarządzanie prawami dostępu

Polecenie	Opis
+	Dodanie praw dostępu
-	Usunięcie praw dostępu
=	Jawne ustawienie praw dostępu

Wykonać następującą serię poleceń:

```
1 # pwd
2 /home
3 # ls -l
4 total 4
5 drwxrwxrwx  2 root      root          1024 Oct 22 11:43 katalog
6 drwxrwxrwx  2 root      root          1024 Oct 22 11:43
    katalog2
7 -rw-rw-rw-  1 root      root           0 Oct 22 11:41 plik.
     dat
8 # chmod a+rwx plik.dat      # Dodanie praw dostepu dla wszystkich
9 # ls -l plik.dat
10 -rwxrwxrwx  1 root      root           0 Oct 22 11:41 plik.
     dat
11 # chmod go-wx plik.dat      # Odebranie praw dostepu
12 # ls -l plik.dat
13 -rwxr--r--  1 root      root           0 Oct 22 11:41 plik.
     dat
```

### 1.3 Obsługa procesów

W systemie QNX każdy program jest uruchamiany jako proces. System zarządza procesami układając je w hierarchię rodzic-potomek. Proces, który uruchamia inny proces, nazywa się macierzystym, a proces uruchomiony - potomnym. Po wydaniu polecenia w konsoli, np. `ls`, proces powłoki powołuje do życia nowy proces `ls`. Powłoka jest więc w tej sytuacji procesem macierzystym, a proces `ls` jest procesem potomnym.

**Przykład 1.3.1** [Proces uruchomiony w tle]. Procesy mogą być uruchamiane w pierwszym planie (ang. foreground) i w tle (ang. background). Domyślnie, procesy są uruchamiane w pierwszym planie. Strumień wejścia do programu stanowi klawiatura, natomiast wyniki są wyprowadzane na ekran, np.

```
1 # ls
2 katalog katalog2 plik.dat
```

Procesy tła możemy uruchamiać poprzez dodanie znaku (&):

```

1 # ls &
2 [1] 1011730
3 # katalog katalog2 plik.dat
4
5 [1] + Done   ls

```

Po uruchomieniu programu, na ekranie pojawia się nr zadania [1] oraz numer identyfikujący proces PID (ang. process identification number). Po wciśnięciu klawisza Enter, program kończy zadanie, a powłoka wyświetla znak zachęty. Proces uruchomiony w pierwszym planie można przesunąć do tła (bez podłączenia do klawiatury) i na odwrót. Podstawowe komendy, służące kontrolowaniu zadań zestawione w tabeli 8.

**Tabela 8:** Kontrola zadań

Polecenie	Opis
polecenie &	Uruchomienie zadania w tle
jobs [-l]	Listowanie zadań pracujących w tle
Ctrl+Z	Wstrzymanie bieżącego zadania
Ctrl+C	Zakończenie bieżącego zadania
fg [PID]	Przeniesienie procesu działającego w tle na pierwszy plan na podstawie numeru procesu
fg [jobID]	Przeniesienie procesu działającego w tle na pierwszy plan na podstawie numeru zadania
bg [PID]	Uruchomienie w tle wstrzymanego zadania na podstawie numeru procesu
bg [jobID]	Uruchomienie w tle wstrzymanego zadania na podstawie numeru zadania

**Przykład 1.3.2** [Procesy tła i procesy pierwszoplanowe]. Proces top wyświetla statystyki wydajności systemu operacyjnego. Uruchomić proces top w pierwszym planie, a następnie nacisnąć kombinację Ctrl+Z, aby wstrzymać bieżący proces.

```

1 # top
2 22 processes; 64 threads;
3 CPU states: 99.9% idle, 0.0% user, 0.0% kernel
4 Memory: 0 total, 204M avail, page size 4K
5
6      PID      TID PRI STATE      HH:MM:SS      CPU      COMMAND
7 1040402        1 10 Rply      0:00:00  0.01% top
8     4110       2 21 Rcv      0:00:01  0.00% io-pkt-v4-hc
9     4106       1 10 Rcv      0:00:00  0.00% devc-con-hid
10    4101       1 10 Rcv      0:00:00  0.00% pci-bios
11    4102       2 21 Rcv      0:00:00  0.00% devb-eide
12          1     9 10 Rcv      0:00:00  0.00% kernel
13 ...
14
15                  Min          Max          Average
16 CPU idle:    99%        99%        99%
17 Mem Avail: 204MB      204MB      204MB

```

```

18 Processes:      22          22          22
19 Threads:       64          64          64
20
21 [1] + Stopped top      # Po wcisnieciu Ctrl+Z
22 # jobs -l
23 [1] + 1040402 Stopped top
24 # fg %1    # Przeniesienie procesy na pierwszy plan;
             alternatywnie fg 1040402
25 22 processes; 64 threads;
26 CPU states: 99.9% idle, 0.0% user, 0.0% kernel
27 Memory: 0 total, 204M avail, page size 4K
28
29     PID      TID PRI STATE      HH:MM:SS      CPU  COMMAND
30     1040402      1 10 Rply      0:00:00   0.01% top
31     4110       2 21 Rcv       0:00:01   0.00% io-pkt-v4-hc
32 ...
33
34 [1] + Stopped top      # Po wcisnieciu Ctrl+Z
35 # bg %1    # Przeniesienie do tła
36 # jobs -l
37 [1] + 1040402 Running top
38 # fg %1    # Przeniesienie do pierwszego planu
39 #       # Po wcisnieciu Ctrl+C

```

**Przykład 1.3.3** [Procesy tła i procesy pierwszoplanowe]. Uruchamiając i testując programy często zauważa się potrzeba zbierania informacji o stanie systemu. Statystyki dostarczają specjalizowane programy, których przegląd umieszczono w tabeli 9.

**Tabela 9:** Statystyki stanu systemu

Polecenie	Opis
ps [-f]	Wyświetla listę procesów i ich status
top	Wyświetla statystyki wydajnościowe systemu
pidin	Wyświetla statystyki systemowe
hogs	Wyświetla listę procesów, wg użycia procesora
showmem [-S]	Wyświetla informacje nt. użytej pamięci

Obejrzeć tablicę procesów wywołując następujące polecenie:

```

1 # ps -f
2   UID      PID      PPID   C STIME  TTY      TIME CMD
3   0      45068          1   - Oct22 ?
4   0      4109          1   - Oct22 ?
5   0     1118226        4119   - 17:29 ?
6   0      4116          1   - Oct22 ?
7   0      4118          1   - Oct22 ?
8   0      4119          1   - Oct22 ?

```

Znaczenie kolumn po wydaniu polecenie ps -f opisano w tabeli 10.

Wystosować zawartość statystyk systemowych za pomocą polecenia pidin, hogs oraz showmem.

```
1 # pidin | more
```

**Tabela 10:** Opis pól wyświetlanego przez polecenie ps

Oznaczenie	Opis
UID	Numer identyfikacyjny użytkownika, który uruchomił proces
PID	Numer identyfikacyjny procesu (potomnego)
PPID	Numer identyfikacyjny procesu nadzecnego (macierzystego)
C	Wykorzystanie procesora
STIME	Czas uruchomienia procesu
TTY	Nazwa terminala kontrolującego (nie-wspierana)
TIME	Czas działania procesu
CMD	Komenda, która uruchomiła proces

```

2 ...
3 # hogs
4 ...
5 # showmem -S
6 ...

```

**Przykład 1.3.4** [Usuwanie procesów]. Ważną grupą poleceń są komendy umożliwiające przerwanie działania procesów, bądź przesłanie im sygnału. Ich zestawienie przedstawiono w tabeli 11.

**Tabela 11:** Usuwanie procesów

Polecenie	Opis
kill [-nazwa_sygnalu  -nr_sygnalu] PID	Przesłanie sygnału do procesu o nr PID
slay [-nazwa_sygnalu  -nr_sygnalu] pro	Przesłanie sygnału do procesu o nazwie pro

Na działający proces można oddziaływać wysyłając do niego sygnały. Sygnały zostały stworzone z myślą o sytuacjach wyjątkowych, np. awariach systemu, czy błędach w pracy programu. Mechanizm sygnałów - ideę przypomina mechanizm przerwań. Proces po otrzymaniu sygnału może wykonać pewną procedurę, lub pozostawić obsługę sygnału systemowi. Nazwy sygnałów i ich liczbowe odpowiedniki możemy podejrzeć poleceniem (kill -l). Trzy często używane sygnały przedstawiono w tabeli 12.

**Tabela 12:** Usuwanie procesów

Nazwa sygnału	Numer sygnału	Opis
INT	2	Przerwanie z klawiatury
KILL	9	Żądanie natychmiastowego zakończenia procesu
TERM	15	Żądanie normalnego zakończenia procesu

W poniższym przykładzie powołuje się do życia proces cat, a następnie przesyła się do niego sygnał KILL. Przykład ten wymaga użycia dwóch konsol. Przełączenia pomiędzy konsolami dokonujemy kombinacją klawiszy Ctrl+Alt+1 oraz Ctrl+Alt+2.

#### Kod źródłowy 1: Konsola 1

```
1 # cat
2 Terminated
3 #
```

#### Kod źródłowy 2: Konsola 2

```
1 # ps -f
2     UID          PID      PPID   C STIME  TTY          TIME CMD
3       0        45068          1   - Oct23 ?          00:00:00 inetd
4       0        4109          1   - Oct23 ?          00:00:00 sh
5       0        4110          1   - Oct23 ?          00:00:00 sh
6       0        4111          1   - Oct23 ?          00:00:00 sh
7       0        4112          1   - Oct23 ?          00:00:00 sh
8       0        81943        4112   - Oct23 ?          00:00:00 cat
9       0       110616        4109   - 14:58 ?          00:00:00 ps -f
10  # kill -TERM 81943
```

## 1.4 Zmienne

Zmienne środowiskowe tworzą tzw. środowisko, w którym wykonują się polecenia. Każde polecenie uruchomione w systemie działa w pewnym „otoczeniu” zmiennych środowiskowych. Zmienne pozwalają na przechowywanie wartości, bądź zdefiniowanych przez użytkownika, bądź danych systemowych (zmienne środowiskowe). Powłoka umożliwia tworzenie zmiennych, przypisywanie wartości i modyfikacje oraz ich usuwanie. Podstawowe operacje na zmiennych przedstawiono w tabeli 13.

**Tabela 13:** Podstawowe operacje na zmiennych środowiskowych

Polecenie	Opis
ZMIENNA=wartosc	Przypisanie wartości do ZMIENNA. Jeśli zmienna nie istnieje, to jest tworzona
\$ZMIENNA	Użycie wartości zmiennej (substytucja)
set	Umożliwia m.in. wyświetlenie wszystkich zmiennych
unset ZMIENNA	Usuwa zmienną ZMIENNA z pamięci (ze środowiska)
export ZMIENNA[=wartosc]	Powoduje, że ZMIENNA jest przekazywana do środowiska każdego uruchamianego polecenia

**Przykład 1.4.1** [Podstawowe operacje na zmiennych]. Należy przetestować działanie poniższych poleceń.

```
1 # A="foo"
2 # echo $A
3 foo
4 # unset A
5 # echo $A
6
7 # set
8 ...
9 # echo $PATH
10 /proc/boot:/bin:/usr/bin:/sbin:/usr/sbin
11 #
```

**Przykład 1.4.2** [Zmienne i polecenia]. Należy przetestować działanie poniższych poleceń.

```
1 # A=1 echo "Zmienna A: " $A
2 Zmienna A:
3 # B=2
4 # echo "Zmienna B: " $B
5 Zmienna B: 2
6 #
```

Polecenia są wykonywane w swoim własnym środowisku. Środowisko to jest zainicjalizowane niektórymi wartościami otrzymanymi od środowiska macierzystego. Aby zmienna A przenosiła się do środowiska uruchamianych poleceń, należy na niej wykonać polecenie export A. W powyższym przykładzie wartość zmiennej A nie została wyświetlona, ponieważ jest ona definiowana tylko dla środowiska polecenia echo, a podstawienia wartości \$A dokonuje wcześniej powłoka. Zmienna B natomiast jest zdefiniowana w środowisku powłoki, zatem jej wartość \$B jest znana w chwili wywołania echo.

**Przykład 1.4.3** [Wyświetlanie kodu zakończenia]. Kod zakończenia ostatnio wykonanego polecenia zapamiętywany jest w specjalnej zmiennej \$. Wyświetl wartość tej zmiennej po wywołaniu polecenia, które zakończyło się sukcesem oraz po zakończeniu polecenia, które nie mogło się powieść.

```
1 # echo "dobre polecenie"
2 dobre polecenie
3 # echo $?
4 0
5 # zlepolecenie
6 sh: zlepolecenie: cannot execute - No such file or directory
7 # echo $?
8 126
```

**Przykład 1.4.4** [Skrypt i zmienne środowiskowe]. Skrypty po uruchomieniu działają w swoim własnym środowisku. Niektóre zmienne ze środowiska macierzystego (np. ze środowiska powłoki sh) są kopiowane do środowiska skryptu, inne nie. Poniższy przykład ilustruje oddziaływanie między skryptem a jego środowiskiem oraz środowiskiem macierzystym.

Przejść do katalogu /home i napisać skrypt piszA o treści:

```
1 echo "Zmienna A: " $A
```

W terminalu powłoki przeprowadź następujące eksperymenty:

```

1 # unset A
2 # sh piszA
3 Zmienna A:
4
5 # A=10
6 # sh piszA
7 Zmienna A:
8
9 # A=10 sh piszA
10 Zmienna A: 10
11
12 # export A=20
13 # sh piszA
14 Zmienna A: 20

```

## 1.5 Przetwarzanie tekstu

Powłoka zawiera specjalne skrypty, które służą do wyświetlania, modyfikacji i formatowania tekstów. Podstawowe komendy zestawiono w tabeli 14.

**Tabela 14:** Wyświetlanie tekstu

Polecenie	Opis
cat [plik]	Wyświetlanie całej treści pliku (lub standardowego wejścia)
more	Wyświetlanie ze stronicowaniem
less	Wyświetlanie ze stronicowaniem
head [-liczbalinii] [plik]	Wyświetlenie kilku pierwszych wierszy tekstu
tail [-liczbalinii] [plik]	Wyświetlenie kilku ostatnich wierszy tekstu

Jednocześnie możliwe jest filtrowanie tekstu za pomocą poleceń zaprezentowanych w tabeli 15.

**Tabela 15:** Filtrowanie tekstu

Polecenie	Opis
grep wzor [plik]	Wyświetlanie tylko linii tekstu pasujących do wzorca
sort [plik]	Wyświetlanie posortowanej treści
wc [plik]	Zliczanie wierszy, słów, znaków, itp.

## 1.6 Strumienie wejściowo-wyjściowe

Uruchomiony program ma do dyspozycji trzy strumienie danych. Każdy z tych plików jest reprezentowany przez małą liczbę całkowitą, zwaną deskryptorem pliku (uchwytem do pliku) - patrz tabela 16.

**Tabela 16:** *Strumienie wejście-wyjście*

Strumień	Deskryptor	Opis
stdin	0	Standardowy strumień wejściowy
stdout	1	Standardowy strumień wyjściowy
stderr	2	Standardowy strumień dla komunikatów o błędach

Wiele programów działa w ten sposób, że pobierają tekst ze strumienia `stdin`, przetwarzają go wysyłając rezultat do strumienia `stdout`, a komunikaty o błędach do `stderr`, tak, jak przedstawiono to na rysunku 2. Przy zwykłym uruchomieniu programu, strumień `stdin` przychodzi z klawiatury a strumienie `stdout` i `stderr` są wyświetlane w konsoli.



**Rysunek 2:** *Strumienie wejście-wyjście*

Źródło oraz wyjście danych można zmienić stosując tzw. przekierowanie strumieni. Jako strumień wejściowy do programu można wysłać zawartość pliku. Służy do tego zapis:

```
1 # polecenie < plik
```

**Przykład 1.6.1** [Przekierowanie strumienia wejściowego]. Polecenie `cat` wywołane bez żadnych argumentów kopiuje strumień `stdin` do strumienia `stdout`. Podaj zawartość pliku `piszA` jako strumień wejściowy do polecenia `cat`.

```
1 # cat < pisza
```

Strumień wyjściowy `stdout` możemy przekierować do pliku za pomocą zapisu:

```
1 # cat > mojplik
2 pierwszy wiersz
3 drugi wiersz
4 trzeci wiersz
5 #
      # Ctrl+C
6 # cat mojplik
```

W ten sposób wyjście z programu zostanie zapisane do pliku `plik`. Można również sprawić, aby wyjście zostało dopisane do pliku:

```

1 # cat >> mojplik
2 czwarty wiersz
3 piaty wiersz
4 #                                     # Ctrl+C
5 # cat mojplik

```

**Przykład 1.6.2** [Przekierowanie strumienia wyjściowego]. Wylistuj zawartość bieżącego katalogu i przekieruj listing do pliku mojplik2. Wyświetl zawartość pliku mojplik2.

```

1 # ls -la > mojplik2
2 # cat mojplik2

```

Wyjście stderr można przekierować do pliku w następujący sposób:

```

1 # polecenie 2> plikzbledem
2 # cat plikzbledem

```

lub

```

1 # polecenie 2>> plikzbledem
2 # cat plikzbledem

```

**Przykład 1.6.3** [Przekierowanie strumienia stderr]. Próba skopiowania nieistniejącego pliku spowoduje wyświetlenie komunikatu o błędzie. Przekieruj ten komunikat do pliku plikzbledem2 po czym wyświetl zawartość pliku

```

1 # cp pliknieist . 2> plikzbledem2
2 # cat plikzbledem2

```

**Przykład 1.6.4** [Jednoczesne przekierowanie strumienia stdin i stdout]. Podaj zawartość pliku piszA na wejście polecenia cat a wyjście przekieruj do pliku wyjscie. Wyświetl zawartość pliku wyjscie.

```

1 # cat < piszA > wyjscie
2 # cat wyjscie

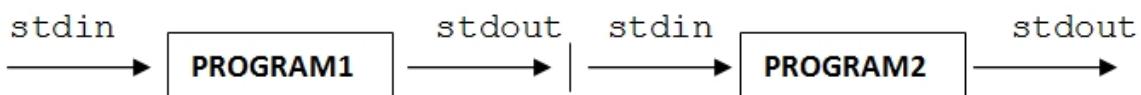
```

**Przykład 1.6.5** [Potoki i filtrowanie wyników]. Polecenia można łączyć w potoki (ang. pipes). Dokonuje się tego poprzez zapis:

```

1 # polecenie1 | polecenie 2 | ...

```



Rysunek 3: Zasada działania potoków (pipes)

Wyjście stdout programu1 jest podawane na wejście stdin programu2 itd. Kodem zakończenia потoku jest kod zwrócony przez ostatnie polecenie.

Polecenie grep filtry przepuszczając tylko linie tekstu pasujące do wzorca. Należy „przepompować” zawartość pliku piszA poleceniem cat do polecenia grep, aby wyszukać linie zawierające literę A.

```
1 # cat piszA | grep 'A'
```

**Przykład 1.6.6** [Sortowanie i obcinanie wyników]. Wyświetl listę pięciu największych plików w bieżącym katalogu:

```
1 # ls -s | sort -n | tail -5
```

**Przykład 1.6.7** [Prosta sekwencja poleceń]. Polecenia można grupować w tzw. sekwencje. Służą do tego operatory ;, && i || . Polecenia połączone operatorem ; wykonują się jedno po drugim bezwarunkowo.

W jednej linii zdefiniuj zmienną o nazwie FOO i wyświetl jej zawartość.

```
1 # FOO='To jest zmienna FOO'; echo $FOO
```

**Przykład 1.6.8** [Sekwencja warunkowa - koniunkcja]. Operator && umożliwia wykonanie następnego polecenia w sekwencji tylko, jeśli poprzednie wykonało się pomyślnie (kod wykonania =0). Polecenie grep zwraca 0 (sukces), jeśli poszukiwany wzór występuje w tekście. W przeciwnym przypadku zwraca wartość niezerową (błąd). Przeszukać zawartość pliku piszA w poszukiwaniu wzorca A i wyświetlić napis „Znaleziono”, jeśli wzorzec wystąpił. Wykonać podobny zabieg dla wzorca kawa.

```
1 # cat piszA | grep 'A' && echo "Znaleziono"
2 Znaleziono
3 # echo $?
4 0
5 # cat piszA | grep 'kawa' && echo "Znaleziono"
6 # echo $?
7 1
```

**Przykład 1.6.9** [Sekwencja warunkowa - alternatywa]. Powtórzyć poprzedni przykład używając operatora || , zamiast &&.

```
1 # cat piszA | grep 'A' || echo "Nie znaleziono"
2 echo "Zmienna A: " $A
3 # echo $?
4 0
5 # cat piszA | grep 'kawa' || echo "Nie znaleziono"
6 Nie znaleziono
7 # echo $?
8 0
```

## **1.7 Ćwiczenia**

1. W systemie pomocy znajdź opis programu do archiwizacji danych tar. Spakować wszystkie pliki znajdujące się w katalogu bieżącym (tj. /home) do pojedynczego archiwum o nazwie `programy.tar` (opcje `-cvf`). Utworzyć w katalogu domowym, katalog o nazwie `Dokumenty`. Skopiować archiwum do katalogu `/home/Dokumenty` i tam go rozpakować (opcje `-xvf`).

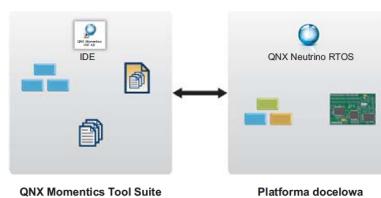


## Laboratorium 2

### Wprowadzenie do QNX Momentics

---

W celu opracowania programów pracujących pod kontrolą systemu operacyjnego czasu rzeczywistego (hard real time), będziemy potrzebowali Platformy Programistycznej QNX. W jej skład wchodzi pakiet QNX Momentics Tool Suite, składający się z elementów niezbędnych do rozwoju i uruchomienia oprogramowania pod QNX Neutrino - patrz rysunek 4. Do tej grupy należą kompilatory, linker, biblioteki i inne komponenty systemu operacyjnego, zbudowane dla wszystkich architektur wspieranych przez QNX Neutrino. Posługując się QNX Momentics w systemie operacyjnym Windows i Linux mamy do dyspozycji zintegrowane środowisko programistyczne na bazie projektu Eclipse.



Rysunek 4: Platforma rozwoju oprogramowania

Dzięki platformie programistycznej możemy tworzyć oprogramowanie w konfiguracji cross development (host-target). Na maszynie typu host (Windows) będziemy dysponować platformą programistyczną QNX Momentics, natomiast na maszynie docelowej typu target (QNX Neutrino na maszynie wirtualnej) będziemy uruchamiać nasze programy. Komunikacja pomiędzy komputerem host i target odbywa się przez sieć, a wspomaga go proces qconn.



Rysunek 5: Konfiguracja host-target

Niniejsze laboratorium będzie poswięcone kilka zagadnieniom:

1. Podstawy obsługi QNX Momentics
2. Zarządzanie projektami C/C++
3. Edycja kodu źródłowego, komplikacja i budowanie
4. Dostęp do platformy docelowej oraz uruchamianie aplikacji

## 2.1 Podstawy obsługi QNX Momentics

**Podstawy IDE**

- Tematy:
  - Podstawowe pojęcia IDE
  - Projekty
  - Przestrzeń robocza
  - Preferencje
  - Podsumowanie

**Podstawy QNX Momentics IDE**

**Metody Programowania Robotów**

**Podstawowe pojęcia – środowisko robocze**

**IDE**

**Metody Programowania Robotów**

**Podstawowe pojęcia IDE - Edytor**

- Edytor:
  - komponent IDE w którym edytujemy i przeglądamy zasób (taki jak plik źródłowy C)
- Obszar edytora:
  - część środowiska roboczego zarezerwowana dla edytora

**Metody Programowania Robotów**

**Podstawowe pojęcia IDE - Widoki**

**Metody Programowania Robotów**

**Podstawowe pojęcia IDE - Perspektywy**

- Widok:
  - obszar który dostarcza:
    - nawigację
    - informacje
    - kontrole
- Perspektywy
  - zawartość układ okna środowiska roboczego może się zmieniać
  - możemy to konfigurować wykorzystując perspektywy
  - Perspektywa jest:
    - zbiorem widoków, edytorów, elementów menu i przycisków paska narzędzi pomocnych do wykonania specyficznego zadania

**Metody Programowania Robotów**

**Podstawowe pojęcia IDE - Perspektywy**

**Metody Programowania Robotów**

**Podstawowe pojęcia IDE - Perspektywy**

- np. perspektywa programowania C/C++:
  - narzędzia do pisania, budowania i uruchamiania
- W celu otwarcia perspektywy:
  - LUB
    - przechodzimy do menu 'Window' i wybieramy 'Open Perspective'
    - przykłady:
      - przyciski debugowania i uruchamiania
      - lista symboli
      - lista błędów komplikacji
      - więcej...
  - klikamy przycisk 'Open a Perspective' i wybieramy z listy. Wyberamy 'Other...' w przypadku innego wyboru

**Metody Programowania Robotów**

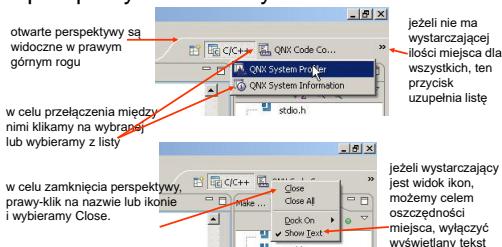
**7**

**Metody Programowania Robotów**

**8**

### Podstawowe pojęcia IDE - Perspektywy

- W celu przełączenia pomiędzy perspektywami i zamknięcia:



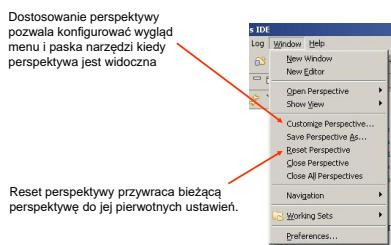
### Podstawowe pojęcia IDE - Perspektywy

- Inne perspektywy:

- CVS Repository Exploring
  - kontrolowanie wersji z wykorzystaniem CVS
- Debug
  - usuwanie błędów na poziomie źródła
- QNX Application Profiler
  - profilowanie aplikacji
- QNX Memory Analysis
  - stos, obiekty pamięci, analiza wyciekającej pamięci, analiza stosu
- QNX System Builder
  - tworzenie obrazu systemu, narzędzia dla systemów wbudowanych
- QNX System Profiler
  - profilowanie systemu

### Podstawowe pojęcia IDE – Opcje perspektyw

- Niektóre przydatne opcje perspektyw:



### Podstawy IDE

- Tematy:

- Podstawowe pojęcia IDE**
- Projekty**
- Przestrzeń robocza**
- Preferencje**
- Podsumowanie**

29 kwietnia 2007

### Projekty

- Projekt jest:

- podstawowym pojemnikiem w środowisku IDE dla:
  - kod źródłowy C/C++
  - informacje o platformie docelowej
  - obraz systemu
  - i inne
- katalogi/pliki

- Niektóre typy projektów to:

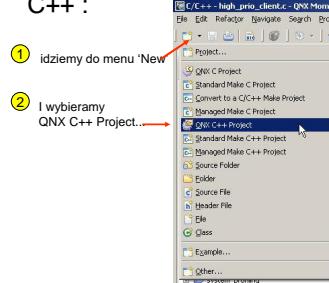
- Standard Make C Project
- Standard Make C++ Project
- QNX C Project
- QNX C++ Project
- QNX Target System Project

Przykłady użycia:

- jeżeli piszemy w i mamy własne drzewo katalogów, utwórzmy Standard Make C Project
- Kod C++ i mamy wiele maszyn docelowych, utwórzmy QNX C++ Project
- Aby połączyć się z platformą docelową - QNX Target System Project

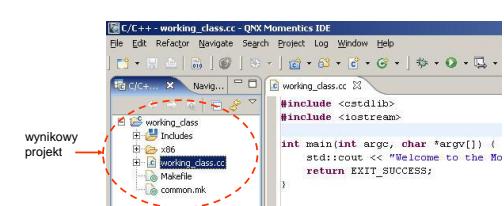
### Projekty – Tworzenie projektu

- Na przykład, aby utworzyć projekt QNX C++ :



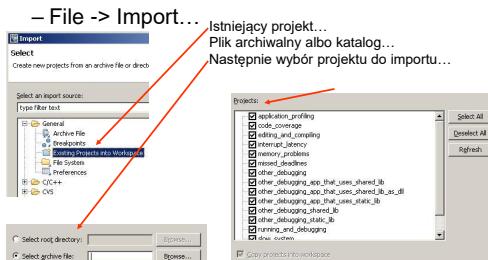
### Projekty - rezultat

- Rezultat końcowy:



### Projekty – Importowanie projektu

- Inny sposób tworzenia projektów to import projektu istniejącego:
  - File -> Import...



### Podstawy IDE

- Tematy:

**Podstawowe pojęcia IDE**

**Projekty**

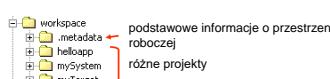
→ **Przestrzeń robocza**

**Preferencje**

**Podsumowanie**

### Przestrzeń robocza

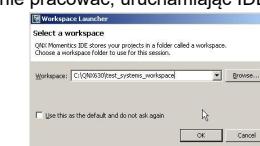
- Przestrzeń robocza to:
  - katalog/folder zawierający projekty lub linki do nich



### Przestrzeń robocza – Wiele przestrzeni roboczych

- Wiele przestrzeni roboczych:

- możemy pracować z jedną przestrzenią roboczą jednocześnie
- ale możemy wybrać z którą przestrzenią chcemy aktualnie pracować, uruchamiając IDE:



- lub z menu 'File':
- jeżeli przestrzeń robocza nie istnieje, zostanie utworzona

### Podstawy IDE

- Tematy:

**Podstawowe pojęcia IDE**

**Projekty**

**Przestrzeń robocza**

→ **Preferencje**

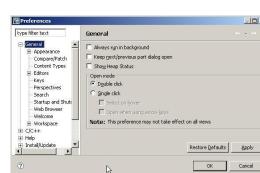
**Podsumowanie**

### Preferencje

- Ustawianie globalnych preferencji:



- okno preferencji zawiera wiele opcji



### Podsumowanie

- Tematy:

**Podstawowe pojęcia IDE**

**Projekty**

**Przestrzeń robocza**

**Dostęp do platformy docelowej**

**Preferencje**

→ **Podsumowanie**

### Podsumowanie

- Usłyszeliśmy:

- jak korzystać z IDE aby pracować efektywniej
- koncepcja projektu w IDE
- podstawowa struktura przestrzeni roboczej
- gdzie ustawiać preferencje

### Rozpowszechnianie bez zgody autorów zabronione

## 2.2 Zarządzanie projektami C/C++

### Wprowadzenie

- Problemy do omówienia:
  - jak utworzyć projekt dla kodu w C/C++
  - podstawowe mechanizmy zarządzania projektami (importowanie)

### Zarządzanie projektami C/C++

Przegląd	Przegląd - make i Makefiles
----------	-----------------------------

- Projekt C/C++:
  - pojemnik na kod źródłowy C/C++
    - różne typy projektów
  - zanim rozpoczęniemy pisanie kodu C/C++ w QNX® IDE
    - musimy utworzyć projekt który będzie zawierał kod źródłowy

- Budowanie z użyciem "make" i "Makefiles":
  - narzędzie linii poleceń **make** jest wykorzystywane przez IDE do kontroli procesu budowania plików wykonawczych i bibliotek
  - **make** wykorzystuje Makefile:
    - może zawierać komendy sterujące procesem budowania (np. wywołania kompilatora i linkera)
    - opisuje zależności w drzewie źródeł
      - np. xxx.o potrzebuje być przebudowany jeżeli xxx.c i xxx.h zmienią się change
      - wykorzystanie daty i czasu modyfikacji podczas budowania
      - np. jeśli data i czas utworzenia xxx.c lub xxx.h jest nowszy niż xxx.o wtedy buduj nowy xxx.o
  - w celu uzyskania informacji o Makefiles patrz **make** w pomocy 'Utilities Reference'

Przegląd – Typy projektów	Przegląd – Wybór typu projektu
---------------------------	--------------------------------

- Projekty dla C lub C++:
  - Standard Make C
  - Standard Make C++
  - QNX C Application
  - QNX C++ Application
  - QNX C Library
  - QNX C++ Library

Który z nich użyć? ...

- Projekty 'Standard Make' oraz 'QNX projects':
  - Projekty Standard Make C/C++:
    - bardziej elastyczne sterowanie procesem budowania
    - łatwiejsze importowanie istniejących, niebanalnych, drzew źródeł
    - możliwość wykorzystywania innych niż **make** narzędzi budowania
  - Projekty QNX C/C++ :
    - łatwość budowania na wiele platform procesorowych, np. rozwój oprogramowania dla PPC i x86
    - nie musimy pisać własnego skryptu Makefile

Zarządzanie projektami C/C++	Projekty 'Standard Make C/C++'
------------------------------	--------------------------------

- Tematy:
  - Przegląd
  - Projekty 'Standard Make C/C++'
  - Projekty 'QNX C/C++ Application'
  - Importowanie kodu źródłowego
  - Ćwiczenie
  - Podsumowanie

- Projekt 'Standard Make C/C++':
  - pusty projekt do którego możemy dodać cokolwiek chcemy
  - jest ogólnym projektem, nie jest specyficzny dla QNX
  - domyślnie, kiedy chcemy zbudować projekt wywoływana jest komenda:
 

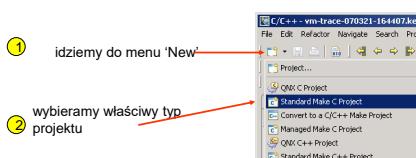
```
make -k all
```

 jak zobaczymy, może być zmieniona na dowolną komendę budowania
  - ten typ projektu jest użyteczne w sytuacji:
    - jeżeli posiadamy istniejącą strukturę budowania i/lub narzędzia i chcemy to zachować
    - chcemy budować projekt nie korzystając ze środowiska IDE

Metody programowania robotów	Metody programowania robotów
------------------------------	------------------------------

### Projekty – Tworzenie projektu

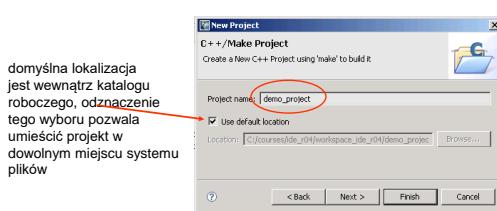
- Tworzenie projektu:
  - otwieramy perspektywę 'C/C++ Development'



– powinien pojawić się kreator tworzenia projektu

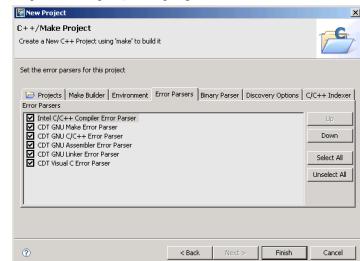
### Projekty 'Standard Make C/C++' – Tworzenie projektu

- Wpisujemy odpowiednią nazwę:



### Projekty 'Standard Make C/C++' – Tworzenie projektu

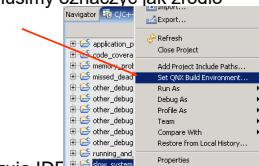
- Mamy do dyspozycji wiele ustawień:



– ale zwykle pozostawiamy wartości domyślne

### Projekty 'Standard Make C/C++'

- Projekt 'Standard Make C/C++':
  - w ogólności nie musi być przeznaczony dla QNX
  - dla poprawnej pracy musimy oznaczyć jak źródło "QNX":



- informacje te wskazują IDE:
  - gdzie znaleźć pliki nagłówkowe
  - jak przetwarzac kod źródłowy
  - zmienne środowiskowe przekazywane do make

### Projekty 'Standard Make C/C++' – Wypełnianie projektu

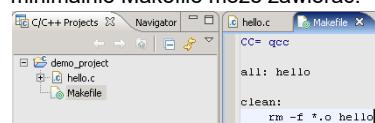
- Następnie tworzymy skrypt
- Makefile:
  - niedawno utworzony projekt
  - jeżeli to rozwinie się, zobaczymy że jest pusty
  - teraz możemy go wypełnić
    - jeżeli posiadamy istniejącą strukturę źródeł, możemy ją skopiować do wewnątrz projektu,
    - w przeciwnym razie, musimy utworzyć pliki, zaczynając od Makefile...



– wykonujemy wszystkie kroki w kreatorze celem utworzenia pliku  
– minimalna zawartość skryptu Makefile to...

### Projekty 'Standard Make C/C++' – Wypełnianie projektu

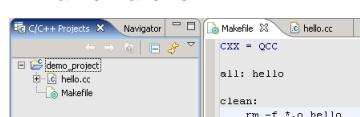
- Minimalna zawartość Makefile projektu C:
  - minimalnie Makefile może zawierać:



zapis ten zakłada, że będziemy edytować plik źródłowy hello.c

### Projekty 'Standard Make C/C++' – Wypełnianie projektu

- Minimalna zawartość Makefile projektu C++:
  - minimalnie Makefile może zawierać :

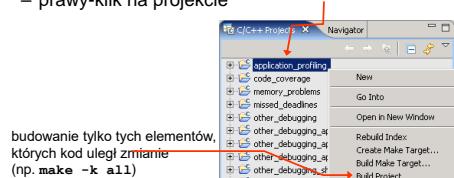


zapis ten zakłada, że będziemy edytować plik źródłowy hello.cc

– więcej informacji o skryptach Makefiles w make w Utilities Reference

### Kompilacja – Budowanie z poziomu IDE

- Budowanie pliku wykonawczego:
  - prawy-klik na projekcie



usunięcie: wykonawczych, obiektów, dzienników błędów, itp (np. make -k)

### Metody programowania robotów

15

Metody programowania robotów

16

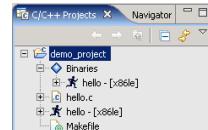
### Projekty – Budowanie z linii poleceń

- Budowanie pliku wykonawczego – poza środowiskiem IDE:
  - projekt jest po prostu katalogiem, zawierającym pliki
  - możemy zbudować projekt z linii poleceń:

```
Command Prompt
C:\QNXsdk\workspace\hellostandard>make
gcc    hello.c   -o hello
C:\QNXsdk\workspace\hellostandard>
```

### Projekty - Folder plików binarnych

- Nasz zbudowany projekt:



– katalog 'Binaries' faktycznie nie istnieje

- Jest tutaj po to, żeby pokazać wszystkie pliki wykonawcze z tego projektu w jednym miejscu.
- Jeżeli mamy wiele plików wykonawczych lub skomplikowaną strukturę katalogów, może to być bardzo pomocne

### Projekty 'QNX C/C++ Application'

- Tematy:

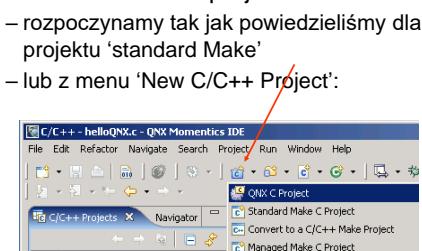
- Przegląd
- Projekty 'Standard Make C/C++'
- Projekty 'QNX C/C++ Application'
- Importowanie kodu źródłowego
- Ćwiczenie
- Podsumowanie

- Projekt 'QNX C/C++ Application':

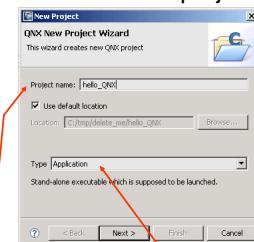
- jest to projekt:
  - ze strukturą Makefile przeznaczoną dla QNX
  - wspiera wiele wariantów/CPU/platform

### Projekty 'QNX C/C++ Application' – Tworzenie projektu

- W celu utworzenia projektu:



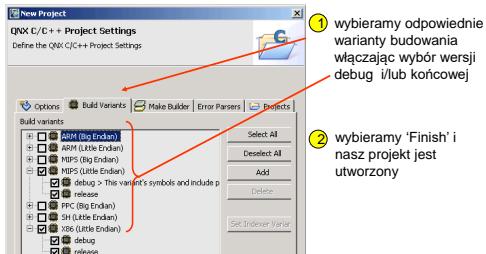
- W celu utworzenia projektu:



① wprowadzamy ② wybieramy 'Application', wśród innych możliwości, jak np. biblioteki ③ wybieramy 'Next'

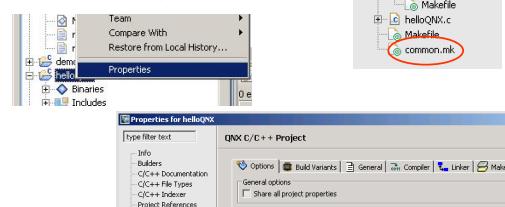
### Projekty 'QNX C/C++ Application' - Makefiles

- W celu utworzenia projektu:



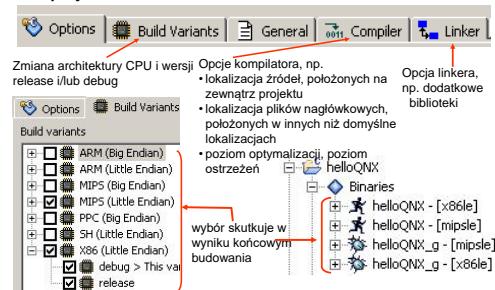
- Praca ze skryptami Makefiles:

- nie modyfikujemy pliku Makefiles
- Makefiles dołączka common.mk
- możemy modyfikować common.mk, upewniwszy się wcześniej czy jest ta opcja we właściwościach projektu



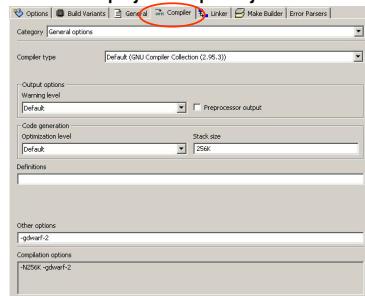
### Projekty 'QNX C/C++ Application' – Opcje budowania

- Opcje budowania:



### Opcje budowania – Opcje komplikacji

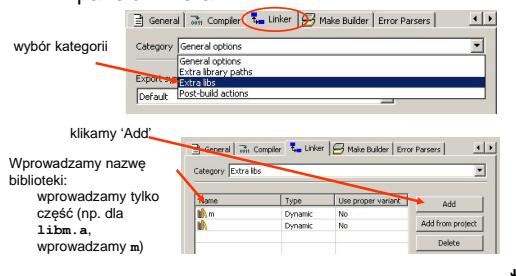
- Ustawianie opcji komplikacji:



### Opcje budowania – Dodatkowe biblioteki

- Dodanie biblioteki matematycznej:

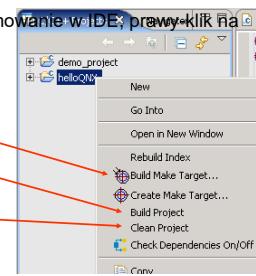
– w panelu linkera



### Budowanie z poziomu IDE

- Budowanie wykonawczego – z poziomu IDE:

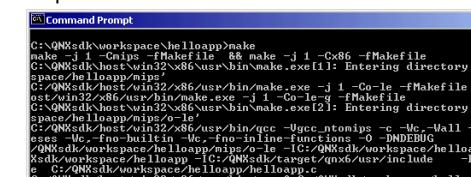
– aby zbudować oprogramowanie w IDE, prawy-klik na projektie...



### Budowanie z linii poleceń

- Budowanie wykonawczego - poza IDE:

- projekt jest po prostu katalogiem, zawierającym pliki
- możemy zbudować projekt z poziomu linii poleceń



### Importowanie kodu

- Istnieje kilka sposobów na umieszczenie kodu źródłowego w IDE:
  - system kontroli źródeł
  - kreator importu IDE
  - wykorzystanie kodu położonego poza środowiskiem roboczym
  - korzystanie z zewnętrznych narzędzi kopiowania

### Zarządzanie projektami C/C++

- Tematy:

#### Przegląd

#### Projekty 'Standard Make C/C++'

#### Projekty 'QNX C/C++ Application'

#### → Importowanie kodu źródłowego

#### Ćwiczenie

#### Podsumowanie

### Zarządzanie projektami C/C++

- Tematy:

#### Przegląd

#### Projekty 'Standard Make C/C++'

#### Projekty 'QNX C/C++ Application'

#### Importowanie kodu źródłowego

#### → Ćwiczenie

#### Podsumowanie

### Metody programowania robotów

### Metody programowania robotów

## Ćwiczenie

- Tworzenie projektu typu 'Standard Make C/C++':
    - utworzyć 'Standard Make C Project' lub 'Standard Make C++ Project'. Nazwać go **standard**
    - dodać dwa nowe pliki:
      - plik źródłowy z kodem funkcji *main()*
      - Makefile – wcześniejsze slajdy
    - zbudować projekt aby upewnić się, że wszystko jest dobrze
    - Tworzenie projektu typu 'QNX C/C++ Project'.

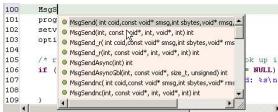
Podsumowanie

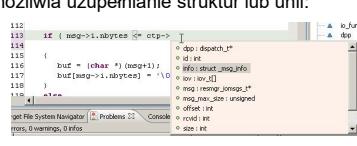
- Omawiane tematy:
    - tworzenie projektów C/C++
      - Standard Make C/C++ Projects
      - QNX C/C++ Application Projects
    - podstawy zarządzania projektami
    - umieszczanie kodu źródłowego wewnątrz IDE

## 2.3 Edycja kodu, komplikacja i budowanie

Wprowadzenie	Problemy do omówienia :
Edycja, komplikacja	<ul style="list-style-type: none"> <li>- kilka pozytycznych cech edytora C/C++</li> <li>- różne sposoby budowania kodu</li> <li>- jak używać IDE do poszukiwania błędów</li> </ul>

Wprowadzenie	Metody programowania robotów	Edycja kodu - Preferencje	Metody programowania robotów
<p><b>Tematy:</b></p> <ul style="list-style-type: none"> <li>→ Wykorzystanie edytora C/C++</li> <li>– Kodowanie</li> <li>Kompilacja</li> <li>Poprawa błędów</li> <li>Ćwiczenie</li> <li>Podsumowanie</li> </ul>	<p>Wygląd i zachowanie się edytora jest bardzo konfigurowalne:</p> <ul style="list-style-type: none"> <li>– rozpoczynamy od menu Windows → Preferences</li> <li>– następnie: C/C++ → Editor</li> <li>– fonty możemy zmienić poprzez: Workbench → Colors and Fonts</li> </ul>		

Kodowanie – Uzupełnianie kodu	Metody programowania robotów	Uzupełnianie kodu – Uzupełnianie funkcji	Metody programowania robotów
<p>IDE wspomaga uzupełnianie kodu:</p> <ul style="list-style-type: none"> <li>– trzy typy:             <ul style="list-style-type: none"> <li>• uzupełnianie funkcji</li> <li>• bloki kodu</li> <li>• uzupełnianie składowych struktury</li> </ul> </li> <li>– wywoływanie przez Ctrl-Space</li> <li>– uzupełnianie składowych struktury może być również wywołane po oczekaniu pewnego czasu             <ul style="list-style-type: none"> <li>• domyślnie jest to ½ sekundy</li> </ul> </li> </ul>	<p>Uzupełnianie funkcji:</p> <ul style="list-style-type: none"> <li>– wyświetli funkcje odpowiednio dopasowane</li> </ul> 	<p>3/ 4/</p>	
<p>3/ 4/</p>			

Uzupełnianie kodu – Uzupełnianie struktur	Metody programowania robotów	Kodowanie – system pomocy	Metody programowania robotów								
<p>Uzupełnianie struktur:</p> <ul style="list-style-type: none"> <li>– umożliwia uzupełnianie struktur lub unii:</li> </ul> 	<p>IDE dostarcza system bezpośredniej pomocy dla:</p> <ul style="list-style-type: none"> <li>– funkcji bibliotecznych:</li> </ul> <pre>printf ("%s\n", filenames[i]); Name: printf Prototype: int printf (const char *format, ...); Description: Write formatted output to stdout #include &lt;stdio.h&gt; int printf (const char *format, ...);</pre>	<p>5/ 6/</p>		Uzupełnianie kodu – Uzupełnianie struktur	Metody programowania robotów	Kodowanie – system pomocy	Metody programowania robotów	<p>– może być skonfigurowane do automatycznego wyczyszczenia po upływie zadanego czasu:</p> <ul style="list-style-type: none"> <li>• ustawiane w Window → Preferences → C/C++ → Editor → Content Assist</li> </ul> 	<ul style="list-style-type: none"> <li>– funkcji użytkownika:</li> </ul> <pre>display_filenames (filenames, nfilenames); void display_filenames (char **filenames, int nfilenames);</pre>	<p>7/ 8/</p>	
<p>5/ 6/</p>											
Uzupełnianie kodu – Uzupełnianie struktur	Metody programowania robotów	Kodowanie – system pomocy	Metody programowania robotów								
<p>– może być skonfigurowane do automatycznego wyczyszczenia po upływie zadanego czasu:</p> <ul style="list-style-type: none"> <li>• ustawiane w Window → Preferences → C/C++ → Editor → Content Assist</li> </ul> 	<ul style="list-style-type: none"> <li>– funkcji użytkownika:</li> </ul> <pre>display_filenames (filenames, nfilenames); void display_filenames (char **filenames, int nfilenames);</pre>										
<p>7/ 8/</p>											

### Kodowanie – Pliki nagłówkowe

IDE może dołączać systemowe pliki nagłówkowe:

- kliknij lub wybierz żądaną funkcję:

```
1#include <sys/types.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4
5int
6main(int argc, char **argv) {
7    int fd;
8
9    fd = open("./tmp/outfile", O_RDONLY|O_CREAT, 0_666 );
10}
```

- prawy-klik i wybieramy 'Add Include':

Content Assist      Ctrl+Space  
Add Include      Ctrl+Shift+F  
Format...      Ctrl+Shift+F

- i wszystkie niezbędne nagłówki zostaną dołączone:

```
1#include <sys/types.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4
5int
6main(int argc, char **argv) {
7    int fd;
8
9    fd = open("./tmp/outfile", O_RDONLY|O_CREAT, 0_666 );
10}
```

### Edycja i komplikacja

Tematy:

#### Wykorzystanie edytora C/C++

- Kodowanie
- Nawigacja
- Oops, co ja zrobiłem?

#### Kompilacja

#### Poprawa błędów

#### Ćwiczenie

#### Podsumowanie

### Kodowanie – Edycja blokowa

Możemy wykonywać niektóre proste czynności związane z edycją bloków oprogramowania:

- wybieramy:

- możemy:

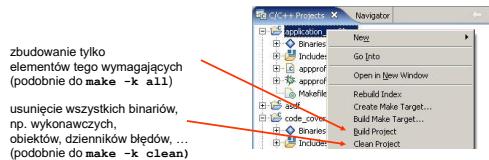
- zakomentować (w stylu C++)
- odkomentować
- przesunąć tab w prawo
- przesunąć tab w lewo

cały blok

### Kompilacja – Budowanie z poziomu IDE

W IDE komplikację nazywamy "budowaniem":

- aby zbudować z poziomu IDE, prawy-klik na wybranym projekcie...



### Kompilacja – Budowanie wszystkich projektów

Możemy zbudować wszystkie projekty:



- możemy zbudować wszystkie projekty wciskając skrót Ctrl-B
- kolejność budowania podana jest tutaj:  
Window→Preferences→General→Workspace→Build Order

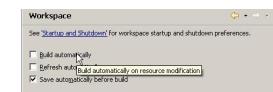


### Kompilacja – Automatyczne budowanie

Budowanie może nastąpić automatycznie, w sytuacji:

- zapisujemy (zmodyfikowany) zasób taki jak plik lub preferencje:

- Window→Preferences→General→Workspace:



- uruchamiania lub debugowania programu:

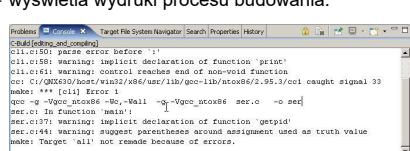
- Window→Preferences→Run/Debug→Launching:



### Poprawa błędów – Widok konsoli

Widok konsoli:

- wyświetla wydruki procesu budowania:

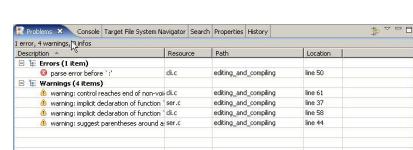


- wydruk jest przechowywany dla każdego projektu
- wybór projektu powoduje wyświetlenie wydruków dla tego projektu

### Poprawa błędów – Widok problemów

The Problems view:

- lista problemów ( ostrzeżeń i błędów )

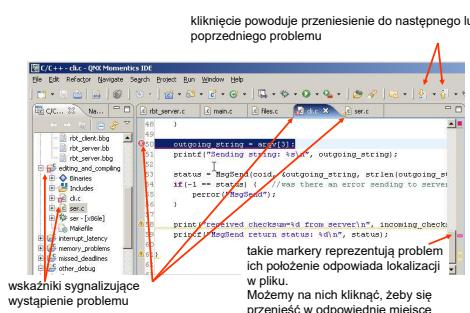


- komunikaty zawierające informację o lokalizacji (numer linii):

- dwu-klik na komunikacie otwiera edytor na właściwej linii
- jeżeli plik jest aktualnie edytowany, kliknięcie na komunikacie przeniósł go do odpowiedniej linii

### Poprawa błędów – Wskaźniki błędów

### Edycja i komplikacja



#### Tematy:

##### Wykorzystanie edytora C/C++

- Kodowanie
- Nawigacja
- Oops, co ja zrobiłem?

##### Kompilacja

##### Poprawa błędów

##### → Ćwiczenie

##### Podsumowanie

### Ćwiczenie

### Edycja i komplikacja

#### Ćwiczenie edycji i komplikacji:

- Ze strony przedmiotu ściągnąć a następnie zbudować projekt qadu. Proces budowania powinien się nie powieść ze względu na błędy w kodzie.
- znaleźć i poprawić błędy komplikacji

#### Tematy:

##### Wykorzystanie edytora C/C++

- Kodowanie
- Nawigacja
- Oops, co ja zrobiłem?

##### Kompilacja

##### Poprawa błędów

##### → Ćwiczenie

##### → Podsumowanie

### Podsumowanie

### Metody programowania robotów

20/

#### Omawiane tematy :

- IDE pomaga w pisaniu i nawigacji po kodzie źródłowym włączając:
  - wyzwalanie i używanie uzupełniania kodu
  - zwijanie plików źródłowych dla lepszego przeglądania
  - wyszukiwanie
- budowanie wybranych projektów oraz wszystkich
- wykorzystanie widoku 'Problems' i innych elementów IDE do wyszukiwania błędów

## 2.4 Dostęp do platformy docelowej oraz uruchamianie aplikacji

Wprowadzenie

### Problemy do omówienia :

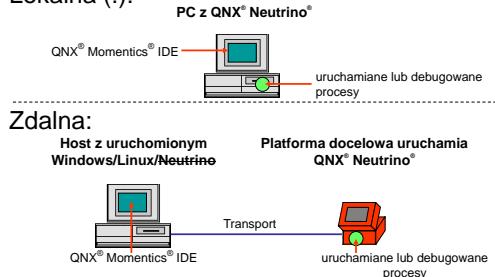
- Dostęp do platformy docelowej
  - Metody uruchamiania programów

## Dostęp do platformy docelowej oraz uruchamianie aplikacji



## Lokalna oraz Zdalna konfiguracja

Lokalna (!):



- `aconn` :

- proces który musi być uruchomiony na platformie docelowej
  - zapewnia wymianę informacji pomiędzy IDE a platformą docelową

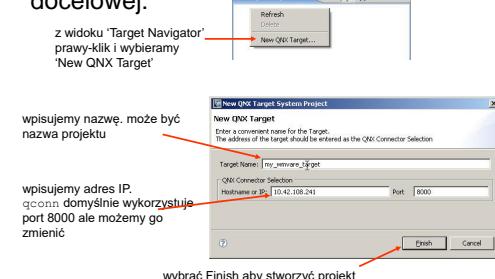


- Widok nawigatora platformy docelowej (Target Navigator):
    - z tego poziomu można:
      - podłączyć do platformy docelowej (utworzyć projekt 'Target System')
      - przejrzeć listę procesów uruchomionych na docelowym komputerze
      - zakończyć proces
      - uruchomić sesję telnet
      - ...
    - jednak najpierw należy utworzyć projekt typu 'QNX Target System Project'

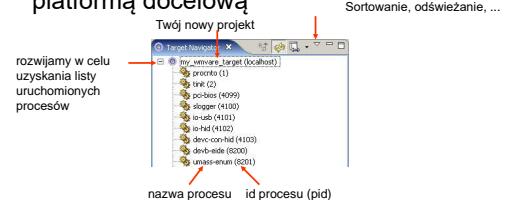
- Projekt 'Target System':
  - zawiera informacje konfiguracyjne dotyczące połączenia z platformą docelową
  - musi być utworzony aby IDE mogło nawiązać połączenie z platformą docelową
  - możliwe miejsca na tworzenie projektów typu 'Target System':
    - widok 'Target Navigator'
    - widok 'Target File System Navigator'
    - okno konfiguracji uruchamiania



- Tworzenie projektu dla platformy docelowej:



- Powinniśmy teraz być połączeni z platformą docelową

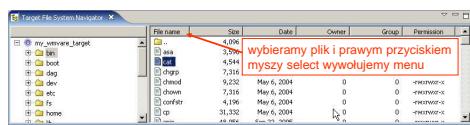


- jeżeli połączenie nie powiedzie się, należy sprawdzić adres IP, czy qconn jest uruchomiony, sprawdzić okablowanie

### Dostęp do platformy docelowej – dostęp do plików

- Dostęp do systemu plików na platformie docelowej:

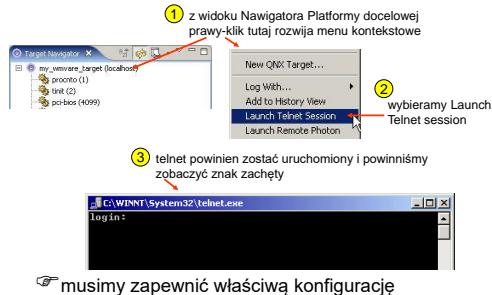
– używamy widoku Nawigatora Systemu Plików



- możemy przenosić pliki do/z widoku projektu C/C++
- w ten sposób kopujemy pliki
- możemy również kopiować pliki za pomocą narzędzi zewnętrznych
- dru-klik na pliku wykonawczym uruchamia go

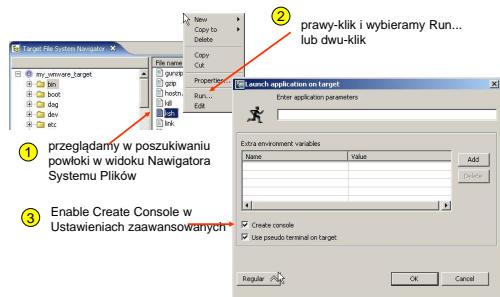
### Dostęp do platformy docelowej – dostęp do linii poleceń

- 1. Uruchomienie sesji telnet:



### Dostęp do platformy docelowej – dostęp do linii poleceń

- 3. Uruchomienie powłoki:



### Wprowadzenie

#### Co to jest konfiguracja uruchamiania?

- dla przypadku zdalnego, IDE potrzebuje informacji:
  - gdzie uruchamiać lub debugować nasz program
  - jak umieścić tam nasz program
- we wszystkich przypadkach, IDE również potrzebuje:
  - jaki program uruchamiać
  - argumenty wywołania
  - zmienne środowiskowe
  - specjalne narzędzia lub ustawienia
  - ip...

### Dostęp do platformy docelowej – dostęp do linii poleceń

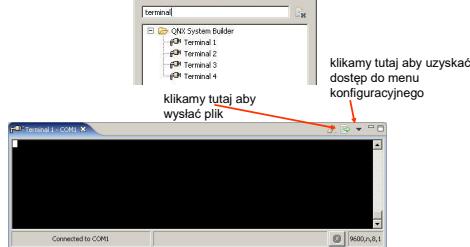
- Sposoby na dostęp do linii poleceń platformy docelowej:

- uruchomienie sesji telnet
- wykorzystanie terminala szeregowego
- uruchomienie powłoki z konsolą w widoku Nawigatora Systemu Plików

### Dostęp do platformy docelowej – dostęp do linii poleceń

- 2. Wykorzystanie terminala szeregowego:

- wykorzystujemy jeden z widoków o nazwie Terminal 1 do Terminal 4:

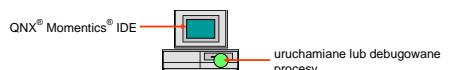


### Wprowadzenie

#### Lokalna oraz Zdalna konfiguracja:

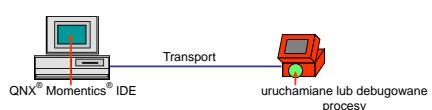
##### Lokalna (!):

PC z QNX® Neutrino®



##### Zdalna:

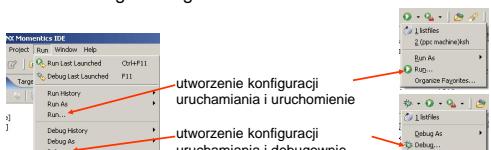
Host z uruchomionym Windows/Linux/Neutrino®



### Ustawienia – Łącuch uruchamiania

#### Zanim uruchomimy lub debugujemy potrzebujemy konfiguracji uruchamiania:

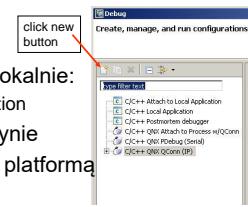
- ustawienie (kreator) jest podobny w obu przypadkach
- konfiguracja, raz utworzona, może być wykorzystana do innego debugowania lub uruchamiania



### Konfiguracja uruchamiania - Typy

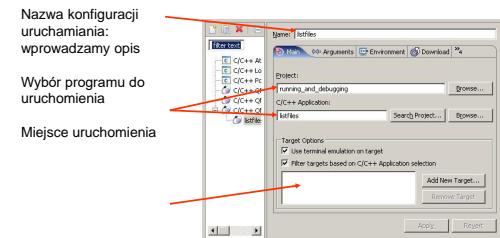
Launch configuration – wybór typu:

- typowy przypadek:  
• C/C++ QConn (IP)
- jeżeli uruchamiamy lokalnie:  
• C/C++ Local Application
- jeżeli posiadamy jedynie lokalne połączenie z platformą docelową:  
• C/C++ Pdebug(Serial)



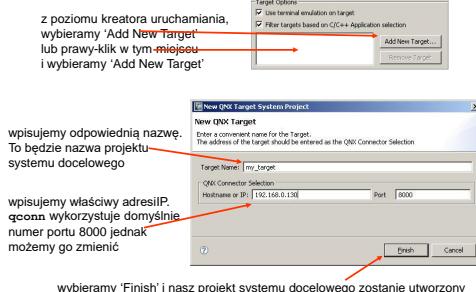
### Konfiguracja uruchamiania - Minimum

Minimum którego potrzebujemy:



### Launch Configuration - Creating a Target System project

Tworzenie projektu systemu docelowego:



### Uruchamianie – Ostatni krok

Kiedy jesteśmy gotowi, klikamy:

- 'Run' aby uruchomić nasz program:

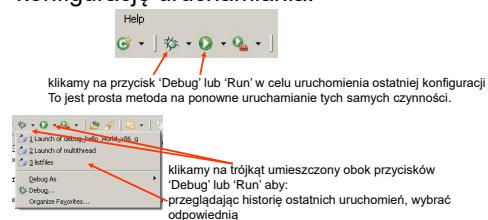


- 'Debug' aby debugować nasz program:



### Uruchamianie i debugowanie – Ponowne użycie

Jeżeli już mamy utworzoną i używaną konfigurację uruchamiania:



### Ćwiczenie

Ćwiczenie uruchamiania aplikacji:

- zbudować projekt typu „hello world”.
- Uruchomić terminal poprzez telnet
- Uruchomić widok Target File System Navigator oraz widok Target Navigator
- Uruchomić program na maszynie docelowej.

### Podsumowanie

Omawiane tematy:

- Jak uzyskać dostęp do platformy docelowej
- Jak uruchamiać program z wykorzystaniem konfiguracji uruchamiania

21/

Metody programowania robotów

22/



## Laboratorium 3

### Wprowadzenie do programowania w języku C

---

### 3.1 Wstęp

Treść laboratorium zawiera krótkie wprowadzenie do języka programowania C jako tego, który będzie wiodący w dalszych etapach zajęć. Wprowadzenie to pozwoli zacząć programować w języku C, tak szybko, jak to możliwe. Nie należy jednak traktować tej części laboratorium, jako substytutu kursów języka C, prezentowanych na innych wykładach i laboratoriach oraz w podręcznikach poświęconych językowi C. Pozostałe podrozdziały omawiają również podstawy użytkowania kompilatora oraz narzędzi make obecnych w systemie QNX.

### 3.2 Kompilowanie i uruchamianie programów

#### 3.2.1 Kompilator qcc

Kody źródłowe, napisane w języku C muszą zostać wstępnie przetworzone (preprocessing), skompilowane (compiling) i skonsolidowane (linking), aby utworzyć plik wykonywalny. Używając linii poleceń, napisany program można skompilować w systemie QNX za pomocą następujących komend:

Dla języka C:

```
1 qcc [opcje] [operandy]
```

Dla języka C++:

```
1 QCC [opcje] [operandy]
```

Wybrane opcje kompilatora qcc zestawiono w tabeli , natomiast operandy stanowią pliki źródłowe (\*.c) oraz pliki typu (\*.o).

**Przykład 3.2.1.** [Konfiguracja środowiska pracy] W trakcie tego laboratorium będziemy głównie pracować w linii poleceń. Zanim przejdziemy do właściwych przykładów, musimy skonfigurować środowisko pracy.

1. Ustawiamy zmienne środowiskowe w systemie Windows. W tym celu należy:

- Otworzyć linię poleceń w systemie Windows, tj. nacisnąć przycisk Start, a w okienku Wyszukaj programy i pliki wpisać nazwę cmd.
- W linii poleceń, uruchomić skrypt C:\qnx660\qnx660-env.bat.

**Tabela 17:** Wybrane opcje kompilatora gcc

Opcje	Opis
<code>-c</code>	Tylko komplikacja
<code>-E</code>	Preprocessing na standardowe wyjście
<code>-g</code>	Kompilacja z debugowaniem
<code>-I path [:path ...]</code>	Ustawia ścieżkę przeszukiwania dla dyrektyw <code>#include</code>
<code>-L path [:path ...]</code>	Ustawia ścieżkę przeszukiwania dla bibliotek
<code>-lplik</code>	Dołącza bibliotekę o nazwie <code>libplik.a</code> lub <code>libplik.so</code> .
<code>-O1</code>	Kompilowanie z optymalizacją 01.
<code>-O2</code>	Kompilowanie z optymalizacją 02.
<code>-O3</code>	Kompilowanie z optymalizacją 03.
<code>-o outfile</code>	Ustala nazwę pliku wyjściowego (wykonywalnego). Domyślnie <code>a.out</code> .
<code>-Wall</code>	Wyświetla wszystkie ostrzeżenia kompilatora.
<code>-pedantic</code>	Wyświetla wszystkie ostrzeżenia kompilatora wymagane przez ANSI C.

2. Na pulpicie tworzymy katalog o nazwie `tmp`.
3. W linii poleceń przechodzimy do katalogu `tmp` wpisując komendę `cd C:\Users\prog_N\Desktop\tmp`, gdzie `N` jest numerem komputera, do którego zalogowany jest użytkownik.
4. Uruchamiamy maszynę wirtualną z systemem operacyjnym QNX oraz sprawdzamy IP maszyny za pomocą polecenia `ifconfig`.
5. Uruchamiamy środowisko do programowania aplikacji QNX Momentics oraz konfigurujemy dostęp do platformy docelowej w kartach `Target Navigator` oraz `Target File System Navigator`.

### Przykład 3.2.2. [Pierwszy program...]

Otworzyć edytor tekstu Notatnik. Wpisać treść poniższego kodu i zapisać plik pod nazwą `hello.c` w katalogu `tmp` na Pulpicie.

**Kod źródłowy 3:** Pierwszy program...

```

1 #include <stdio.h> /* wlaczenie informacji o bibliotece standard
. */
2 /* komentarz */
3 int main() /* deklaracja glownej funkcji */
4 { /* klamra otwierajaca blok */
5     printf("\nHello world !!!\n"); /* wypisz tekst na stdout */
6     return 0; /* zwrocenie kodu wyjscia */
7 } /* klamra zamykajaca blok */

```

W linii poleceń (Windows) skompilować program `hello.c` wydając polecenie:

```
1 > qcc -Wall hello.c
2 > ls
3 ...
```

Proces tworzenia pliku wykonywalnego z nadaniem nazw plików wyjściowych można podzielić na dwa etapy: komplikacja i budowanie.

```
1 > qcc -Wall -c hello.c
2 > qcc hello.o -o hello
3 > ls
4 ...
```

Kod źródłowy `hello.c` możemy jednocześnie skompilować i zbudować z nadaniem nazwy plikowi wyjściowemu.

```
1 > qcc -Wall hello.c -o hello2
2 > ls
3 ...
```

Ostatnim etapem tego przykładu będzie uruchomienie programu `hello` na maszynie z systemem operacyjnym QNX. Poprzez widok Target File System Navigator w środowisku QNX Momentics przekopiować program wykonywalny `hello` z katalogu `tmp` do katalogu `/home` na maszynie wirtualnej. Uruchomić program poprzez polecenie:

```
1 # ./hello
2
3 Hello world !!!
4
5 #
```

### 3.2.2 Sterowanie procesem budowania programów

Kompilacja i uruchamianie projektu, składającego się z wielu plików źródłowych, w których zależności są złożone, może być uciążliwa. Istnieją programy narzędziowe, które ułatwiają zarządzanie złożonymi programami. Jednym z nich jest narzędzie `make`, które przetwarza specjalny plik `Makefile`.

W pliku `Makefile` występują tzw. reguły (ang. rules) mające następującą formę:

```
1 target: prerequisite [prerequisites]
2 <tab> commands
```

Cel (ang. target) - jest zazwyczaj plikiem, który chcemy utworzyć. Zależność (ang. prerequisite) - do utworzenia celu wymagane są zazwyczaj pliki źródłowe; nazywamy je zależnościami. Polecenia (ang. commands) - są krokami (np. wywołania kompilatora lub polecenia powłoki), które należy wykonać, aby

utworzyć cel.

**Przykład 3.2.3.** [Pierwszy plik Makefile] W Notatniku napisać skrypt `Makefile` do uruchomienia programu zapisanego w `hello.c` oraz zapisać go w katalogu `tmp`.

**Kod źródłowy 4:** Pierwszy skrypt `Makefile`

```

1 all: hello
2 hello: hello.o
3   @echo "buduje..."
4   gcc hello.o -o hello
5 hello.o: hello.c
6   @echo "kompiluje..."
7   gcc -Wall -c hello.c
8 clean:
9   @echo "usuwam..."
10  rm -f hello hello.o

```

Wpisać w wiersz poleceń (Windows) następujące wywołania:

**Kod źródłowy 5:** Pierwszy plik `Makefile`

```

1 > make -n    # Wyswietla tylko polecenia, ale ich nie wykonuje
2 > make all
3 > ls
4 ...
5 > make clean
6 > ls
7 ...

```

**Przykład 3.2.4.** [Rozbudowany plik Makefile] W bardziej rozbudowanych projektach stosuje się różnego typu zmienne, które ułatwiają proces konstrukcji pliku `Makefile`. Należą do nich zmienne definiowane przez użytkownika, zmienne standardowe (predefiniowane), np. dotyczące nazw kompilatorów i flag wywołań oraz zmienne automatyczne, których wartości są obliczane, gdy reguła jest wykonywana. Wybrane zmienne standardowe i automatyczne przedstawiono w tabelach 18 oraz 19.

**Tabela 18:** Zmienne standardowe

Argumenty	Opis
CC	Nazwa kompilatora języka C
CXX	Nazwa kompilatora języka C++
CFLAGS	Opcje kompilatora języka C
CXXFLAGS	Opcje kompilatora języka C++
LFLAGS	Opcje dla linkera

Zapisać w Notatniku i uruchomić skrypt `Makefile` ze zmiennymi standardowymi i automatycznymi.

**Kod źródłowy 6:** Rozbudowany plik `Makefile`

```

1 CC=gcc
2 CFLAGS=-Wall
3 LFLAGS=-lm
4 SRC=hello.c

```

**Tabela 19:** Zmienne automatyczne

Argumenty	Opis
<	Nazwa pliku pierwszej zależności
@	Nazwa pliku docelowego
^	Lista wszystkich zależności

```

5 OBJS=hello.o
6 BINS=hello
7
8 all: $(BINS)
9 $(BINS): $(OBJS)
10    @echo "buduje..."
11    $(CC) $(LFLAGS) $^ -o $@
12 $(OBJS): $(SRC)
13    @echo "kompiluje..."
14    $(CC) $(CFLAGS) -c $< -o $@
15 clean:
16    @echo "usuwam..."
17    rm -f $(OBJS) $(BINS)
18 .PHONY: all clean

```

W przykładzie zastosowano zmienne definiowane przez użytkownika, zmienne standardowe oraz zmienne automatyczne. Dodatkowo użyto reguły .PHONY, służącej do poinstruowania narzędzia make, że reguły all oraz clean są regułami specjalnymi, a nie nazwami plików.

**Przykład 3.2.5.** [Makefile w środowisku QNX Momentics] Oczywiście, możemy wykorzystać plik Makefile utworzony w przykładzie 3.2.4 oraz kod źródłowy 3 zapisany w przykładzie 3.2.2 w środowisku QNX Momentics. W tym celu należy:

1. W środowisku QNX Momentics utworzyć pusty projekt C Project, który należy nazwać jako hello.
2. W kolejnym oknie z katalogu Executable jako typ projektu wybrać Empty project, a jako kompilator QNX QCC. Pozostawić domyślną konfigurację.
3. Przekopiować zawartość skryptu 3.2.4 oraz kodu źródłowego 3 do bieżącego projektu. Zbudować projekt klikając prawym przyciskiem na projekt i wciskając opcję Build Project.
4. Skonfigurować środowisko do uruchamiania programu i uruchomić zbudowany program na maszynie QNX.
5. Usunąć pliki tymczasowe z projektu klikając prawym przyciskiem na projekt i wciskając opcję Clean Project.

### 3.3 Podstawy języka C

#### 3.3.1 Typy zmiennych

Niektóre wbudowane typy zmiennych przedstawiono w tabeli 20.

Tabela 20: Typy zmiennych

Typ	Opis
int	integer
short	short integer
long	long integer
float	single precision real (floating point) variable
double	double precision real (floating point) variable
char	character variable (single byte)

### 3.3.2 Pętle

Większość programów zawiera pętle, umożliwiające powtarzanie określonych czynności. W języku C istnieje kilka różnych sposobów tworzenia pętli. Dwie najbardziej rozpowszechnione to pętla **while** i **for**. Składnia poleceń podana jest poniżej.

```

1 while (expression)
2 {
3     ...block of statements to execute...
4 }
```

```

1 for (expression_1; expression_2; expression_3)
2 {
3     ...block of statements to execute...
4 }
```

Pętla **while** jest kontynuowana do momentu, w którym wyrażenie logiczne jest prawdą, przy czym warunek ten jest sprawdzany po wejściu do pętli. Pętla **for** jest równoważna następującej pętli **while**.

```

1 expression_1;
2 while (expression_2)
3 {
4     ...block of statements...
5     expression_3;
6 }
```

Przykłady zastosowania pętli.

- Pętla **while**.

```

1 i = initial_i;
2
3 while (i < i_max)
4 {
5     ...block of statements...
6     i = i + i_increment;
7 }
```

- Pętla `for`.

**Kod źródłowy 7: Pętla for - przykład**

```

1  for (i = initial_i; i <= i_max; i = i + i_increment)
2  {
3      ...block of statements...
4  }

```

**3.3.3 Konstrukcje warunkowe**

Składnia wyrażeń warunkowych w konstrukcji `if` wygląda następująco:

```

1  if (conditional_1)
2  {
3      ...block of statements executed if conditional_1 is true...
4  }
5  else if (conditional_2)
6  {
7      ...block of statements executed if conditional_1 was false
8          and conditional_2 is true...
9  }
10 else
11 {
12     ...block of statements executed otherwise...
13 }

```

Innego typu często używaną konstrukcją warunkową jest `switch`:

```

1  switch (expression)
2  {
3      case const_expression_1:
4          ...block of statements...
5      break;
6      case const_expression_2:
7          ...block of statements...
8      break;
9      default:
10         ...block of statements..
11         break;
12 }

```

**Przykład 3.3.1.** [Makefile z dołączoną biblioteką] Poniższy kod źródłowy oblicza wartości funkcji *sinus* dla kątów od  $0^\circ$  –  $360^\circ$ .

**Kod źródłowy 8: Drugi program...**

```

1  #include <math.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      int angle_degree;
7      double angle_radian, pi, value;
8

```

```

9     /* Wydrukuj komentarz */
10    printf ("\nOblicz tablice funkcji sinus...\\n\\n");
11
12    /* oblicz pi */
13    pi = 4.0*atan(1.0);
14    printf ( " wartosc PI = %f \\n\\n", pi );
15
16    printf ( " kat \\n" );
17    /* poczatkowa wartosc kata */
18    angle_degree=0.0;
19    /* petla po katach */
20    while( angle_degree <= 360.0 )
21    {
22        angle_radian = pi * angle_degree/180.0;
23        value = sin(angle_radian);
24        printf(" %3d %f \\n", angle_degree, value);
25        /* inkrementuj indeks */
26        angle_degree = angle_degree + 10.0;
27    }
28    return 0;
29 }
```

- Napisać odpowiedni skrypt `Makefile` w Notatniku oraz skompilować i zbudować program z linii poleceń. Za pomocą QNX Momentics przekopiować program wykonywalny na maszynę docelową. Sprawdzić poprawność działania programu uruchamiając go z linii poleceń pod QNX-em.  
Uwaga: w skrypcie `Makefile` należy zapisać informację o dołączeniu biblioteki matematycznej `libm.a`. Biblioteka ta zawiera m.in. kody funkcji trygonometrycznych.
- Alternatywnie zadanie można wykonać w środowisku QNX Momentics. Przekopiować kod źródłowy oraz skrypt `Makefile` do IDE. Z pliku `Makefile` usunąć wpis o dołączeniu biblioteki matematycznej. Zamiast informacji w skrypcie, bibliotekę matematyczną należy dołączyć poprzez własności projektu.

### 3.3.4 Operatory relacji

Wyrażenia warunkowe zawierają operacje logiczne, dzięki którym można porównać różnego typu wielkości. Lista operatorów porównania jest podana w tabeli 21.

**Tabela 21:** Operatory relacji

Typ	Opis
<	smaller than
<=	smaller than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

### 3.3.5 Operatory logiczne

Często w instrukcjach warunkowych stosowane są operatory logiczne przedstawione w tabeli 22. Te z kolei dzielą się na jednoargumentowe (unarne) i dwuargumentowe (binarne).

**Tabela 22:** Operatory logiczne

Typ	Opis
&&	and
	or
!	not

### 3.3.6 Wskaźniki

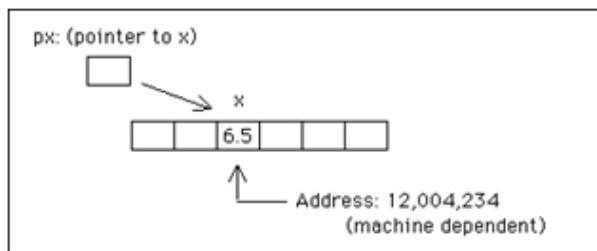
Język programowania C pozwala na odwoływanie do zmiennych poprzez ich adres w pamięci. Daje to dużą elastyczność przy programowaniu, ale powoduje również trudności dla nowicjuszy programujących w C. Wszystkie zmienne w programie rezydują w pamięci. Rozważmy prosty przykład:

```
1 float x;
2 x = 6.5;
```

Czasami chcemy uzyskać informację, gdzie zmienna rezyduje w pamięci. Uzyskujemy to poprzez umieszczenie operatora adresu & przed nazwą zmiennej. Język C pozwala na definiowanie wskaźników (ang. pointers), które przechowują adresy zmiennych. Tę sytuację ilustruje kolejny przykład.

```
1 float x;
2 float* px;
3 x = 6.5;
4 px = &x;
```

W powyższym przykładzie zdefiniowano wskaźnik px wskazujący na typ **float** i przypisano mu wartość adresu zmiennej x. Zawartość pamięci wskazywanej przez wskaźnik można uzyskać poprzez operator dereferencji \*. Tak więc \*px zawiera wartość zmiennej x. Opisaną sytuację ilustruje również rysunek 6.



**Rysunek 6:** Idea wskaźnika

**Przykład 3.3.2.** [Zastosowania wskaźników] W poniższym przykładzie pokazano kilka możliwości zastosowania wskaźników. Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 9: Zastosowania wskaźników**

```

1 #include <stdio.h>
2
3 int main()
4 {
5     float x, y;      /* deklaracja zmiennych */
6     float *fp, *fp2;  /* fp fp2 wskazniki do typu float */
7
8     x = 6.5;
9
10    /* wydrukuj zawartosc i adres zmiennej x */
11    printf("Zawartosc x: %f, adres x: %ld\n", x, &x);
12
13    fp = &x;          /* fp wskazuje na lokacje zmiennej x */
14
15    /* wydrukuj zawartosc fp */
16    printf("Wartosc komorki pamieci wskazanej przez fp: %f\n", *fp)
17        ;
18
19    /* zmien zawartosc komorki pamieci wskazanej przez fp */
20    *fp = 9.2;
21    printf("Nowa wartosc x %f = %f \n", *fp, x);
22
23    /* Obliczenia arytmetyczne */
24    *fp = *fp + 1.5;
25    printf("Wartosc koncowa x is %f = %f \n", *fp, x);
26    /* Zamiana wartosci x i y */
27    y = *fp;
28    fp2 = fp;
29    printf("Wartosc y = %f i fp2 = %f \n", y, *fp2);
30    return 0;
}

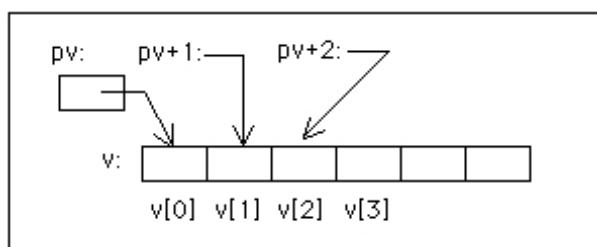
```

### 3.3.7 Tablice

W języku C typy danych można umieszczać w tablicach. Składnia jest przy tym następująca:

```
1 typ nazwatablicy[wymiar];
```

W języku C tablice zaczynają się od indeksu 0. Pozostałe elementy zajmują sąsiednie komórki w pamięci. Język C traktuje nazwę tablicy jako wskaźnik do jej pierwszego elementu. Tak więc, jeśli *v* jest tablicą, *\*v* ma tą samą wartość co element tablicy *v[0]*, *\*(v+1)* ma tą samą wartość, co element tablicy *v[1]*. Sytuację tę ilustruje rysunek 7.



Rysunek 7: Tablica a wskaźnika

Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 10:** Zastosowania wskaźników i tablic

```

1 #include <stdio.h>
2
3 #define SIZE 3
4 int main()
5 {
6     float x[SIZE];
7     float *fp;
8     int i;
9     /* inicjalizacja tablicy x */
10    /* rzutowanie i na float */
11    for (i = 0; i < SIZE; i++)
12        x[i] = 0.5*(float)i;
13    /* drukuj x */
14    for (i = 0; i < SIZE; i++)
15        printf(" %d %f \n", i, x[i]);
16    /* fp wskazuje na x */
17    fp = x;
18    /* drukuj poprzez wskaźnik */
19    /* *(fp+i)=x[i] */
20    for (i = 0; i < SIZE; i++)
21        printf(" %d %f \n", i, *(fp+i));
22    return 0;
23 }
```

### 3.3.8 Funkcje

Funkcje w języku C pozwalają na znaczne uproszczenie kodów źródłowych. W trakcie laboratorium zetknęliśmy się już z funkcją główną `main`, a także z funkcjami matematycznymi i funkcjami wejścia-/wyjścia. Oprócz zastosowań funkcji bibliotecznych, programista może implementować własne funkcje. Nagłówek i ciało funkcji mogą przyjąć następującą postać:

```

1 typ nazwa_funkcji ( lista_argumentow )
2 {
3     ... deklaracja lokalne...
4
5     ... operacje...
6
7     return zwrocana_wartosc;
8 }
```

Argumenty w wywołaniach funkcji można przekazywać przez wartość. Oznacza to, że w ciele funkcji istnieje kopia argumentu wywołania. Jakakolwiek zmiana zmiennej przekazanej do funkcji nie powoduje zmiany wartości zmiennej poza ciałem funkcji. Aby zmienić wartość zmiennej przekazanej w wywołaniu w ciele funkcji, należy przekazać ją poprzez wskaźnik. Problem ten ilustrują poniższe przykłady.

**Przykład 3.3.3.** [Przekazywanie argumentów przez wartość] Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 11:** Przekazywanie argumentów przez wartość

```
1 #include <stdio.h>
2
3 void exchange(int a, int b);
4
5 int main()
6 {
7     int a, b;
8
9     a = 5;
10    b = 7;
11    printf("przed zamiana w main: a = %d, b = %d\n", a, b);
12
13    exchange(a, b);
14    printf("po zamianie w main: ");
15    printf("a = %d, b = %d\n", a, b);
16    return 0;
17 }
18
19 void exchange(int a, int b)
20 {
21     int temp;
22
23     temp = a;
24     a = b;
25     b = temp;
26     printf(" Z funkcji exchange: ");
27     printf("a = %d, b = %d\n", a, b);
28 }
```

**Przykład 3.3.4.** [Przekazywanie argumentów przez wskaźnik] Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 12: Przekazywanie argumentów przez wskaźnik**

```
1 #include <stdio.h>
2
3 void exchange ( int *a, int *b );
4
5 int main()
6 {
7     int a, b;
8
9     a = 5;
10    b = 7;
11    printf("przed zamiana w main: a = %d, b = %d\n", a, b);
12
13    exchange(&a, &b);
14    printf("po zamianie w main: ");
15    printf("a = %d, b = %d\n", a, b);
16    return 0;
17 }
18
19 void exchange ( int *a, int *b )
20 {
21     int temp;
22
23     temp = *a;
24     *a = *b;
25     *b = temp;
26     printf(" Z funkcji exchange: ");
27     printf("a = %d, b = %d\n", *a, *b);
```

```
28 }
```

### 3.3.9 Przekazywanie argumentów z wiersza poleceń

Często zdarza się (szczególnie w systemach UNIX–owych), że argumenty przetwarzane w programie są przekazywane poprzez wiersz poleceń. Typowe przykłady to `ls -la`, czy też `tail -20`. Argumenty z wiersza poleceń pozwalają na uzyskanie większej elastyczności naszych programów. Można je przekazywać do programu poprzez funkcję główną `main()`. Ogólna składnia funkcji ma następującą postać:

```
1 main(int argc, char* argv[]);
```

gdzie `argc`, określa liczbę argumentów wywołania (łącznie z nazwą wywoływanego programu), a `argv` jest tablicą ciągów znakowych `char*`, w której przechowywane są argumenty wywołania. Sposób użycia ilustruje poniższy przykład.

**Przykład 3.3.5.** [Argumenty wiersza poleceń] Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 13:** Argumenty wiersza poleceń

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     int i;
6
7     printf("argc = %d\n", argc);
8
9     for (i = 0; i < argc; i++)
10        printf("argv[%d] = \"%s\"\n", i, argv[i]);
11    return 0;
12 }
```

### 3.3.10 Operacje I/O (wejścia/wyjścia)

Język C, poprzez swoje biblioteki, dostarcza różnorodnych funkcji, pozwalających na obsługę wejścia/wyjścia. Na poziomie liter, funkcją która pobiera znak ze standardowego wejścia `stdin`, jest `getchar()`, podczas gdy funkcja `putchar()` zapisuje jeden znak do standardowego wyjścia `stdout`. Użycie funkcji ilustruje poniższy kod źródłowy.

**Przykład 3.3.6.** [Operacje wejścia i wyjścia] Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 14:** Użycie funkcji `getchar()` i `putchar()`

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i, nc;
```

```

6      nc = 0;
7      i = getchar();
8      while (i != EOF) {
9          nc++;
10         i = getchar();
11     }
12 }
13 printf("\nLiczba wczytanych liter = %d\n", nc);
14 return 0;
15 }
```

Znak EOF jest wartością końca pliku, zdefiniowaną w nagłówku biblioteki standardowej. Zakończenie danych (koniec pliku) uzyskujemy poprzez kombinację klawiszy Ctrl+D. Bardziej zaawansowaną funkcją pozwalającą pisać sformatowany tekst do standardowego wyjścia jest poznana wcześniej funkcja `printf`. Funkcją, która umożliwia czytanie ze standardowego wejścia jest `scanf`. Składnie obu poleceń przedstawiono poniżej.

```

1 printf("format", zmienne);
2 scanf("format", &zmienne);
```

Odpowiednikiem powyższych wyrażeń, pozwalających pisać do tablic zmiennych typu `char` są poniższe funkcje.

```

1 sprintf(string, "format", zmienne);
2 sscanf(string, "format", &zmienne);
```

String w składni oznacza nazwę tablicy zmiennych typu `char`, bądź wskaźnik do jej pierwszego elementu.

### 3.3.11 Operacje I/O na plikach

Podobne instrukcje wejścia/wyjścia istnieją w przypadku obsługi plików. Ogólną składnię I/O w przypadku plików podano poniżej.

```

1 FILE *fp; /* wskaznik do pliku */
2 fp = fopen(nazwa, tryb); /* otworzenie pliku */
3 fscanf(fp, "format", lista_zmiennych); /* czytanie z pliku */
4 fprintf(fp, "format", lista_zmiennych); /* pisanie do pliku */
5 fclose(fp); /* zamknięcie pliku */
```

Tryb w instrukcji `fopen` określa cel otwarcia pliku. Dozwolone tryby przedstawiono w tabeli 23.

**Przykład 3.3.7.** [Operacje IO na plikach] Skompilować, zbudować oraz uruchomić przykład.

**Kod źródłowy 15:** Zastosowanie funkcji `fopen()` i `fclose()`

```

1 #include <stdio.h>
2
3 int main()
4 {
```

Tabela 23: Operacje na plikach

Typ	Opis
r	czytanie z pliku
w	pisanie do pliku
a	dodanie zawartości do pliku

```

5   FILE *fp;
6   int i;
7
8   fp = fopen("foo.dat", "w"); /* otworz foo.dat do pisania */
9
10  fprintf(fp, "\nAla ma kota\n\n"); /* tekst */
11  for (i = 1; i <= 10 ; i++)
12    fprintf(fp, "i = %d\n", i);
13
14  fclose(fp); /* zamknij plik */
15  return 0;
16 }
```

## 3.4 Ćwiczenia

- Zmodyfikować program obliczający tablicę funkcji sinus, tak, aby zawierał instrukcję sterującą `for`. Utworzyć funkcję `sinus` i przenieść jej kod do oddzielnego pliku źródłowego `sinus.c`. Dołączyć plik nagłówkowy `sinus.h` z deklaracją funkcji. Napisać odpowiedni skrypt `Makefile`. Wyniki zapisać do pliku `sinus.dat`.
- Napisz funkcję sprawdzającą ile par liczb całkowitych z przedziału  $\langle a, b \rangle$  jest zawartych w kole o średnicy 8. ( $x^2 + y^2 \leq 8$ ). Wartości `a` i `b` powinny być zadawane z klawiatury i przekazywane jako parametry funkcji.
- Napisać bibliotekę operacji wektorowych, wykorzystując struktury. Zdefiniować wektor jako strukturę:

```

1 struct Wektor
2 {
3     double x, y, z;
4 };
```

Biblioteka operacji powinna pozwalać na dodawanie, odejmowanie, mnożenie przez liczbę wektora, liczenie iloczynów skalarnych i wektorowych. Uzupełnić program o odpowiednie funkcje testowe.

- Uzupełnić poprzedni program o operacje transformacji wektorów z układu do układu. Napisać funkcje odpowiadające za elementarne obroty wokół osi `x`, `y`, `z`.



## Laboratorium 4

### Procesy i zarządzanie procesami

---

#### 4.1 Wstęp

Przykłady procesów w życiu codziennym: proces fizyczny, chemiczny, technologiczny, proces sądowy, administracyjny. Proces to przebieg powiązanych ze sobą zmian. Większość procesów zachodzi w określonym środowisku (np. w urzędzie), podlega pewnym ograniczeniom (np. procedurom prawnym) i wymaga pewnych zasobów (np. pieniędzy podatników). Pewne zdarzenia w procesie mogą występować sekwencyjnie, inne mogą nakładać się na siebie w czasie. Jeśli jesteśmy w stanie wyodrębnić unikalny ciąg występujących po sobie zdarzeń powiązanych tak, że ich przedstawienie czy uniezależnienie od siebie jest niemożliwe, bądź przeczyłoby logicznie, to taką sekwencję często określa się mianem wątku. Proces, podobnie jak fabuła powieści, może obejmować wiele wątków biegących równocześnie.

Niektóre z wymienionych faktów mają przeniesienie na procesy w systemach operacyjnych. Proces działa w środowisku systemu operacyjnego, podlega restrykcjom narzuconym przez system operacyjny (np. ograniczenia dostępu do cudzych plików). Procesowi przydzielane są zasoby takie jak czas procesora, obszar pamięci operacyjnej, pliki, urządzenia peryferyjne. W pamięci komputera zwykle znajduje się wiele procesów, które są na różnych etapach wykonania i współzawodniczą o zasoby. W obrębie jednego procesu występuje jeden lub więcej wątków (w uproszczeniu wątek obrazuje ciąg wykonywanych po sobie instrukcji), wątki mogą być wykonywane równolegle.

Proces, jak większość obiektów w informatyce, posiada pewne atrybuty, można na nim dokonywać pewnych operacji (funkcje) oraz podlega regułom określającym jego czas życia. W niniejszym laboratorium omówimy następujące zagadnienia:

- wyświetlanie i modyfikowanie atrybutów procesów (atributy),
- funkcje umożliwiające tworzenie i usuwanie procesów (czas życia).

Często proces jest utożsamiany z programem uruchomionym w systemie operacyjnym. Rzeczywiście aplikacje mogą się jednak składać z wielu procesów. W niniejszym laboratorium zostaną pokazane przykłady prostych aplikacji wielo-procesowych zapisanych w języku C.

#### 4.2 Wyświetlanie i modyfikowanie atrybutów procesów

Atrybuty to pewne właściwości przypisane procesowi. Każdy proces w systemie QNX ma np. przypisany unikalny identyfikator PID (process ID), który jest liczbą całkowitą. Każdy proces (oprócz procesu „głównego” procnto) ma również swój proces macierzysty. Identyfikator PPID (parent process ID) jest

identyfikatorem PID procesu macierzystego. Proces procnto ma identyfikator PID=1. Wybrane atrybuty i funkcje systemowe umożliwiające dostęp do nich przedstawiono w tabeli 24.

**Tabela 24:** Niektóre atrybuty procesów oraz funkcje biblioteki systemowej umożliwiające dostęp do nich

Atrybuty procesu	Wyświetlanie	Modyfikacja
Identyfikator PID	<code>getpid()</code>	–
Identyfikator PPID	<code>getppid()</code>	–
Priorytet i strategia szeregowania	<code>sched_getparam()</code>	<code>sched_setparam()</code>

Na ogół mamy do czynienia z sytuacją, kiedy procesów gotowych do wykonania jest więcej niż umożliwiają to dostępne w danej chwili zasoby. Procedura szeregująca (scheduler) rozstrzyga, któremu z procesów (wątków) zostanie w danej chwili przydzielony czas procesora. Jednym z istotnych parametrów rzutujących na kolejność przydzielania czasu procesora jest priorytet – każdy z procesów (i wątków – patrz następne laboratoria) ma przyporządkowany priorytet (process priority). Jest on miarą pilności wykonania danego procesu względem innych. W systemie QNX Neutrino 6 priorytet jest liczbą z zakresu od 0 (najniższy) do 255 (najwyższy). System nakłada ograniczenia na dopuszczalne zakresy priorytetów dla programów uruchamianych przez poszczególnych użytkowników (tabela 25):

**Tabela 25:** Priorytety w QNX Neutrino 6

Priorytet	Użytkownik
0	Proces jałowy
1 – 63	Wątki zwykłego użytkownika
1 – 255	Wątki użytkownika root

W systemie QNX są dostępne trzy strategie szeregowania:

- szeregowanie FIFO (FIFO scheduling),
- szeregowanie karuzelowe (round robin scheduling),
- szeregowanie sporadyczne (sporadic scheduling).

Są one omówione szczegółowo w dokumentacji QNX [9]. W tabeli 26 przedstawiono ich symbole, wykorzystywane w wywołaniach systemowych (symbole te zdefiniowane są w pliku nagłówkowym `sched.h`).

**Tabela 26:** Strategie szeregowania w QNX Neutrino 6

Symbol	Opis
<code>SCHED_FIFO</code>	Szeregowanie FIFO
<code>SCHED_RR</code>	Szeregowanie karuzelowe
<code>SCHED_SPORADIC</code>	Szeregowanie sporadyczne

**Przykład 4.2.1.** [Lista procesów] W terminalu wykonać polecenie `ps -1`. Sprawdzić w dokumentacji, co oznaczają poszczególne kolumny wyświetlnego raportu.

**Przykład 4.2.2.** [Wyświetlanie i modyfikacja wybranych atrybutów procesu]

```

1 #include <stdio.h>
2 #include <stdlib.h>
```

```

3 #include <sched.h>
4 #include <process.h>
5
6 int main(int argc, char *argv[]) {
7
8     pid_t pid, ppid;
9     struct sched_param param; /* struktura opisujaca parametry
10        szeregowania */
11    int sched;
12
13    pid = getpid();
14    ppid = getppid();
15    printf("Proces potomny PID: %d\n", pid);
16    printf("Proces macierzysty PPID: %d\n", ppid);
17    /* pobranie parametrow */
18    sched_getparam(pid, &param);
19
20    printf("Priorytet ustawiony przy starcie procesu %d.\n", param.
21           sched_priority);
22    printf("Biezacy priorytet procesu %d.\n", param.
23           sched_curpriority);
24
25    sched = sched_getscheduler(pid);
26    printf("Strategia szeregowania: %d\n", sched);
27
28    param.sched_priority = 9;
29    sched_setparam(pid, &param);
30    sched_setscheduler(pid, SCHED_FIFO, &param);
31
32    sched_getparam(pid, &param);
33
34    printf("Nowy priorytet procesu %d.\n", param.sched_priority);
35    printf("Biezacy priorytet procesu %d.\n", param.
36           sched_curpriority);
37
38    sched = sched_getscheduler(pid);
39    printf("Nowa strategia szeregowania: %d\n", sched);
40
41    return EXIT_SUCCESS;
42 }
```

## 4.3 Tworzenie procesów

W systemie QNX Neutrino, modułem odpowiedzialnym za dynamiczne tworzenie, usuwanie oraz zarządzanie procesami jest zawarty w mikrojądrze (procnto) manager procesów. W systemie QNX istnieją różne metody tworzenia procesów. Część z nich pochodzi wprost od systemów UNIX-owych, opartych o standard POSIX, inne funkcje do tworzenia procesów są charakterystyczne tylko dla systemu QNX. Podstawowe funkcje do tworzenia procesów przedstawiono w tabeli 27. W ramach laboratorium omówimy cztery funkcje służące tworzeniu procesów, tj. `system()`, `fork()`, `exec()`, `spawn()`.

Jeden ze sposobów na utworzenie nowego procesu polega na uruchomieniu programu zapisanego w pliku wykonywalnym. Z poziomu języka C można tego dokonać używając funkcji `system()` (przykład 4.3.1). Poleciением tym można uruchomić program w sposób podobny, jak to się czyni wprost z powłoki. Funkcja zwraca status zakończenia uruchomionego programu.

**Tabela 27:** Metody tworzenia procesów w systemie QNX

Funkcja	Opis
<code>system()</code>	Wywołanie programów, poleceń systemowych, bądź skryptów
<code>fork()</code>	Utworzenie kopii procesu bieżącego
<code>exec()</code>	Zastąpienie bieżącego procesu nowym procesem – rodzina funkcji
<code>spawn()</code>	Utworzenie procesu potomnego – rodzina funkcji
<code>vfork()</code>	Utworzenie procesu potomnego i zablokowanie procesu macierzystego
<code>forkpty()</code>	Utworzenie procesu potomnego w oknie pseudoterminala
<code>popen()</code>	Uruchomienie programu jako procesu potomnego z jednoczesnym utworzeniem łącza pomiędzy procesem bieżącym, a potomnym

**Przykład 4.3.1.** [Wywołanie programu za pomocą polecenia system()]

```

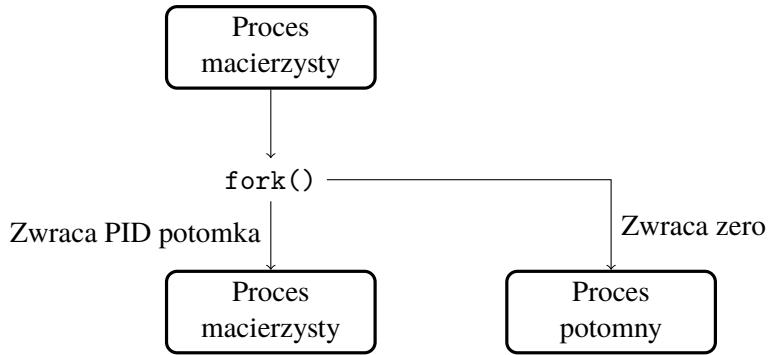
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int rc;
7
8     rc = system( "pwd; ls" );
9
10    if( rc == -1 )
11    {
12        printf( "Nie mozna uruchomic polecenia\n" );
13    }
14    else
15    {
16        printf( "Status zakonczenia %d\n", rc );
17    }
18    return EXIT_SUCCESS;
19 }
```

Funkcja `fork()` tworzy kopię bieżącego procesu i uruchamia ją jako proces potomny. Utworzony proces potomny wykonuje się współbieżnie z procesem tworzącym, posiada własny nr PID, a jego PPID wskazuje na proces tworzący. Funkcja `fork()` tworzy deskryptor nowego procesu oraz kopię segmentu kodu, danych i stosu. Modyfikacje zmiennych w procesie macierzystym nie są widoczne w procesie potomnym i odwrotnie. Jeżeli operacja utworzenia procesu potomnego zakończy się powodzeniem, to funkcja w procesie macierzystym zwraca identyfikator (PID) procesu potomnego (wartość większa od 1), a w procesie potomnym wartość 0. W przypadku niepowodzenia, funkcja `fork()` zwraca wartość -1. Działanie funkcji przedstawiono na rysunku 8, a użycie w przykładzie 4.3.2.

**Przykład 4.3.2.** [Wywołanie funkcji fork()]

```

1 #include <stdio.h>
2 #include <process.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define CHILD 4
7 #define PARENT 8
8
9 int main()
```

Rysunek 8: Idea działania funkcji *fork()*

```

10  {
11      int fork_pid;
12      int i, j;
13
14      fork_pid = fork();
15
16      if (fork_pid == -1)
17      {
18          perror("M: Nie udalo sie utworzyc procesu potomnego...\n");
19      }
20      else if (fork_pid == 0)
21      {
22          /* proces potomny */
23          printf("P: Proces potomny PID = %d\n", getpid());
24          for(j = 0; j < CHILD; j++)
25          {
26              printf("P: pracuje %d sek. ...\n", j);
27              sleep(1);
28          }
29          printf("P: Koniec petli\n");
30      }
31      else
32      {
33          /* Proces macierzysty, fork_pid = PID procesu potomnego */
34          printf("M: Proces macierzysty PID = %d\n", getpid());
35          printf("M: Moj proces potomny ma PID = %d\n", fork_pid);
36          for(i = 0; i < PARENT; i++)
37          {
38              printf("M: pracuje %d sek. ...\n", i);
39              sleep(1);
40          }
41          printf("M: Koniec petli\n");
42      }
43      printf("Proces PID=%d konczy prace\n", getpid());
44      return 0;
45  }

```

Sprawdzić działanie programu, gdy zmienna CHILD = 8, a zmienna PARENT = 4.

## 4.4 Obsługa zakończenia procesów

Procesy na ogół współdziałyają ze sobą (np. komunikują się), mogą też być w relacji pokrewieństwa (dany proces może być np. procesem macierzystym innych procesów). Proces może, dla przykładu, oczekiwając

na określone zdarzenia, generowane przez inne procesy. Przedwczesne lub nieprawidłowe zakończenie procesu, będącego częścią aplikacji wielo-procesowej, może doprowadzić do błędów w działaniu aplikacji lub niepotrzebnego zużycia zasobów systemowych. Przed zakończeniem procesu należy zwolnić zajęte przezeń zasoby, zakończyć z innymi procesami scenariusze komunikacyjne i synchronizacyjne, a w przypadku procesów macierzystych, zaczekać na zakończenie procesów potomnych.

Zakończenie procesu jest inicjowane w następujących sytuacjach:

- wywołaniem funkcji `exit()` z poziomu procesu,
- poprzez powrót z funkcji `main()`, np. instrukcją `return`,
- proces zakończony/unicestwiony przez system operacyjny lub inny proces.

W dowolnym miejscu programu można zainicjować zakończenie procesu funkcją `exit()`

```
1 #include <stdlib.h>
2 void exit( int status );
```

Funkcja ta powoduje zakończenie procesu bieżącego oraz przekazanie do procesu macierzystego wartości `status`. Przyjęto się przekazywać wartość `EXIT_SUCCESS`, gdy proces zakończył się poprawnie, bądź `EXIT_FAILURE`, gdy wystąpił błąd (to samo dotyczy instrukcji `return` w funkcji `main`).

Jeśli dany proces posiada potomków, należy zapobiegać tzw. „osieracaniu procesów”. Ma to miejsce gdy proces macierzysty kończy się wcześniej niż jego procesy potomne. Proces macierzysty powinien zaczekać na zakończenie wszystkich swoich procesów potomnych. Można do tego użyć funkcji `wait()`:

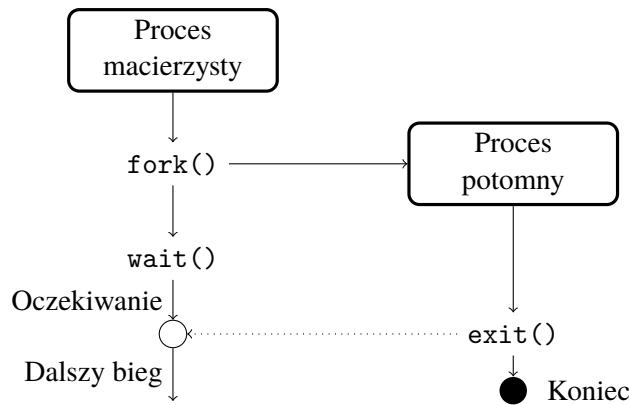
```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 pid_t wait( int * status );
```

Funkcja `wait()` wstrzymuje bieżący proces do momentu zakończenia któregokolwiek z jego procesów potomnych. Funkcja zwraca PID zakończonego procesu (bądź `-1`, gdy brak procesów potomnych). Pod adres wskazany przez argument `status` wpisywany jest status zakończenia procesu. Ten `status` zawiera liczbę `status` podaną w procesie potomnym do funkcji `exit()` bądź instrukcji `return`, ale wartość ta może być w pewien sposób „zakodowana”. Do jej „odekodowania” używa się makra `WEXITSTATUS()`.

Ilustrację omawianych zagadnień stanowi przykład 4.4.1. Zastosowanie funkcji `wait()` w przykładzie 4.4.1 powoduje, że proces macierzysty nie zakończy się przed procesem potomnym.

### Przykład 4.4.1. [Wywołanie funkcji `fork()` wraz z obsługą zakończenia procesów]

```
1 #include <stdio.h>
2 #include <process.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define CHILD 8
7 #define PARENT 4
8
9 int main()
```



Rysunek 9: Schemat poprawnego zakończenia procesu

```

10  {
11      int fork_pid;
12      int i, j;
13      int status;
14
15      fork_pid = fork();
16
17      if (fork_pid == -1)
18      {
19          perror("M: Nie udalo sie utworzyc procesu potomnego...\n");
20      }
21      else if (fork_pid == 0)
22      {
23          /* proces potomny */
24          printf("P: Proces potomny PID = %d\n", getpid());
25          for(j = 0; j < CHILD; j++)
26          {
27              printf("P: pracuje %d sek. ...\n", j);
28              sleep(1);
29          }
30          printf("P: Koniec petli\n");
31      }
32      else
33      {
34          /* Proces macierzysty, fork_pid = PID procesu potomnego */
35          printf("M: Proces macierzysty PID = %d\n", getpid());
36          for(i = 0; i < PARENT; i++)
37          {
38              printf("M: pracuje %d sek. ...\n", i);
39              sleep(1);
40          }
41          printf("M: Koniec petli\n");
42          printf("M: Czekam na potomny...\n");
43          fork_pid = wait(&status);
44          printf("M: Proces potomny PID=%d zakonczony; status: %d\n",
45                 fork_pid, WEXITSTATUS(status));
46      }
47      printf("Proces PID=%d konczy prace\n", getpid());
48      return 0;
49  }
    
```

Możliwy jest scenariusz, w którym proces potomny zainicjuje swoje zakończenie zanim proces macierzysty „dobiegnie” do funkcji `wait()`. Zwalniane są wtedy wtedy wszystkie zasoby procesu potomnego,

z wyjątkiem deskryptora procesu, czyli miejsca w pamięci operacyjnej, gdzie przechowywane są informacje potrzebne do zarządzania procesem. Tam zapamiętywany jest m.in. status zakończenia procesu potomnego, który ma być przekazany do procesu macierzystego. Proces oczekuje w rekordach systemowych na zwolnienie deskryptora, które może być dokonane dopiero po wywołaniu `wait()` przez proces macierzysty. Taki stan procesu potomnego nazywa się stanem „zombie”.

Do oczekiwania na zakończenie *konkretnego* procesu można użyć funkcji `waitpid()`.

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3
4 pid_t waitpid( pid_t pid, int *status, int options );
```

Funkcja zwraca PID zakończonego procesu, bądź -1, w przypadku, gdy brak jest procesów potomnych. Funkcja zwraca również status zakończonego procesu poprzez argument `status`. Jedną z użytecznych opcji (`options`) jest flaga `WNOHANG`, która powoduje, że proces macierzysty, w przypadku braku oczekujących procesów potomnych, natychmiast wraca z funkcji i kontynuuje swoje działanie i w tym przypadku możemy użyć takiej kombinacji do cyklicznego sprawdzania, czy proces potomny się zakończył.

## 4.5 Zastąpienie procesu bieżącego innymi procesami

Funkcje z rodziny `exec()` zastępują bieżący proces nowym procesem, powstały poprzez uruchomienie wskazanego pliku wykonywalnego. W systemie QNX zdefiniowano następującą rodzinę funkcji: `execl()`, `execle()`, `execlp()`, `execlepe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`. Działanie tych funkcji jest podobne, natomiast różnią się listą parametrów formalnych. W trakcie laboratorium będziemy używać najprostszych funkcji `execl()` oraz `execv()`, o następujących sygnaturach:

```
1 #include <process.h>
2
3 int execl( const char * path,
4             const char * arg0,
5             const char * arg1,
6             ...
7             const char * argn,
8             NULL );
9 int execv( const char * path,
10            char * const argv[] );
```

gdzie `path` jest ścieżką do pliku wykonywalnego, a argumenty `arg0`...`argn` (`argv[0]`...`argv[n]`) – argumentami przekazanymi do programu. Na ostatnim miejscu podaje się wartość `NULL`, na oznaczenie zakończenia listy parametrów wywoływanego programu. Funkcja `execl()` nadaje się w sytuacji, kiedy długość listy argumentów przekazywanych do uruchamianego programu jest znana już na etapie programowania aplikacji. Funkcja `execv()` nadaje się w sytuacji gdy długość listy argumentów jest ustalana dopiero w trakcie działania aplikacji, np. jest zadana przez użytkownika aplikacji.

**Przykład 4.5.1.** [Wywołanie funkcji `execl()`]

```

1 #include <stdio.h>
2 #include <process.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <sys/wait.h>
7
8 int main()
9 {
10     printf("Poczatek\n");
11     switch(fork())
12     {
13         case -1:
14             printf("Blad utworzenia nowego procesu potomnego\n");
15             break;
16         case 0: /* proces potomny */
17             execl("/bin/ls", "ls", "-a", NULL);
18             printf("Blad uruchomienia procesu\n");
19             exit(EXIT_FAILURE);
20         default: /* proces macierzysty */
21             if (wait(NULL) == -1)
22                 printf("Blad w oczekiwaniu na zakoñczenie procesu
23                         potomnego\n");
24     }
25     printf("Koniec\n");
26 }

```

## 4.6 Tworzenie procesów funkcją spawn()

Funkcje z rodziny `spawn()`, służą do tworzenia procesów potomnych poprzez uruchomienie wskazanego pliku wykonywalnego. Do tej grupy zaliczamy następujące funkcje: `spawn()`, `spawnl()`, `spawnv()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnve()`, `spawnvp()`, `spawnvpe()`. Omówimy tylko funkcje `spawnl()` i `spawnv()`, których deklaracje wyglądają następująco:

```

1 #include <process.h>
2
3 int spawnl( int mode,
4             const char * path,
5             const char * arg0,
6             const char * arg1,
7             ...
8             const char * argn,
9             NULL );
10 int spawnv( int mode,
11             const char * path,
12             char * const argv[] );

```

Argument `mode` jest trybem wykonania procesu, który określa sposób wywołania procesu potomnego i zachowanie procesu macierzystego, gdy proces potomny zostanie zainicjowany (tabela 28). Pozostałe argumenty są interpretowane dokładnie tak samo jak w przypadku funkcji `execl()` i `execv()`.

Ilustrację wywołania funkcji `spawnl()` stanowi przykład 4.6.1:

**Tabela 28:** Tryby wywołania funkcji *spawn()*

Tryb	Opis
P_WAIT	Proces macierzysty czeka na zakończenie procesu potomnego i później jest kontynuowany.
P_NOWAIT	Proces macierzysty i potomny są wykonywane współbieżnie. Można używać funkcji <i>wait()</i> .
P_NOWAITO	Proces macierzysty i potomny są wykonywane współbieżnie. Nie wolno używać funkcji <i>wait()</i> do uzyskania kodu wyjścia. Relacja pokrewieństwa między nimi zostaje przerwana.
P_OVERLAY	Proces macierzysty jest zastępowany przez proces potomny. Wywołanie w tym trybie jest równoważne wywołaniu funkcji <i>exec()</i> .

**Przykład 4.6.1.** [Wywołanie funkcji *spawnl()*] Utworzyć dwa programy (oddzielne projekty w środowisku IDE) o nazwie **macierzysty** i **potomny**, następnie wywołać program **macierzysty**.

```

1  /* macierzysty.c */
2  #include <stdio.h>
3  #include <process.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/wait.h>
8  #include <signal.h>
9
10 int main()
11 {
12     int pid, i, res, status;
13     res = spawnl(P_NOWAIT, "potomny", "potomny", "10", NULL);
14     if (res<0)
15     {
16         printf("blad wywolania procesu 'potomny'\n");
17         exit(0);
18     }
19     for(i=1;i<=10;i++)
20     {
21         printf("Macierzysty - krok %d\n",i);
22         sleep(1);
23     }
24     pid = wait(&status);
25     printf("Proces %d zakonczony, status %d\n", pid, status);
26     return 0;
27 }
```

```

1  /* potomny.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6
7  int main(int argc, char* argv[])
8  {
9     int i;
10    for(i=1; i<=atoi(argv[1]); i++)
11    {
12        printf("Potomny krok: %d\n",i);
13        sleep(1);
14    }

```

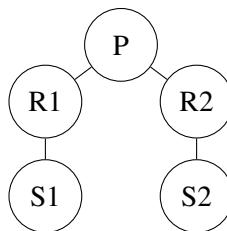
```

15
16     exit(EXIT_SUCCESS);
17 }

```

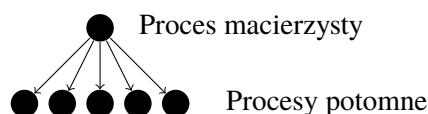
## 4.7 Ćwiczenia

- Napisać program, w którym proces macierzysty P tworzy dwa procesy potomne R1, R2. Każdy z procesów potomnych powinien utworzyć swój własny proces potomny: R1 → S1, R2 → S2. Każdy z procesów (z wyjątkiem P) niech wyświetli swój PID oraz PPID.



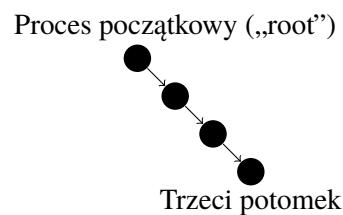
Rysunek 10: Hierarchia procesów w ćwiczeniu 1

- Napisać program, w którym proces macierzysty P tworzy jeden proces potomny R (`fork()`). Proces R niech wyświetli swój PID i uruchomi wskazany plik wykonywalny przy użyciu funkcji `execv()`. Ścieżkę do pliku wykonywalnego oraz argumenty jego wywołania przekazać z wiersza poleceń.
- Utworzyć za pomocą funkcji `fork()` pięć procesów potomnych, ponumerowanych (numer kolejny *i*) od 0 do 4. Zaczekać na ich zakończenie. Niech każdy z procesów potomnych wyświetla cyklicznie co jedną sekundę swój PID, numer kolejny *i*, oraz numer cyklu. Liczba cykli do wykonania przez poszczególne procesy powinny pochodzić z tablicy zdefiniowanej wewnętrz funkcji `main()`, np. `int t[] = {20, 5, 3, 8, 9}`. Na zakończenie procesu potomnego o nr *i* wywołać funkcję `exit(i)`. Proces macierzysty powinien czekać na zakończenie potomnych i wyświetlić PID zakończonego procesu oraz wartość podaną przez proces potomny do funkcji `exit()`.



Rysunek 11: Hierarchia procesów w ćwiczeniu 3

- Zadanie jest analogiczne do poprzedniego, z tym, że struktura procesów ma wyglądać jak drzewo przedstawione na rysunku 12. Każdy z procesów, oprócz ostatniego tworzy jeden proces potomny.



Rysunek 12: Hierarchia procesów w ćwiczeniu 4 dla trzech potomków

## Laboratorium 5

### Zarządzanie wątkami – tworzenie, kończenie, atrybuty wątków

---

#### 5.1 Wprowadzenie

Podczas rozwoju oprogramowania, np. czasu rzeczywistego, wbudowanego, graficznego, często zachodzi potrzeba przetwarzania współbieżnego, bądź równoległego, w przypadku implementacji na komputerach równoległych z pamięcią wspólną. Współbieżność (równoległość) tę można osiągnąć poprzez używanie wątków POSIX (biblioteka `pthread`), zaimplementowanych w systemie QNX i opartych na standardzie IEEE POSIX 1003.1c.

Dotyczczas zajmowaliśmy się procesami mającymi po jednym wątku. Pojęcie procesu można rozszerzyć do istnienia wielu wątków sterowania i rozpatrywać proces jako zbiór wątków i zasobów. Wątek w tym przypadku będzie traktowany jako elementarna, niezależna jednostka, podlegająca szeregowaniu (scheduling). Część zasobów procesu jest wspólna dla jego wszystkich wątków. Należą do nich:

- Wspólna przestrzeń adresowa, w szczególności zmienne statyczne i sterta.
- Pliki i urządzenia wejścia-wyjścia.
- Kanały, kolejki i połączenia.

Wątki mają również swoje prywatne atrybuty i zasoby. Spośród ważniejszych można wymienić:

- Identyfikator wątku TID (thread identifier).
- Priorytet.
- Stos.
- Zestaw rejestrów.
- Atrybuty służące szeregowaniu wątków.
- Maska sygnałów.
- Lokalne zmienne wątku TLS (thread local storage).
- Procedura zakończenia wątku.

Zalety stosowania wielowątkowego modelu programowania (multithreaded programming model):

- Mniejszy koszt tworzenia, kończenia, w porównaniu z procesami.
- Zwykle szybszy czas przełączenia wątków niż procesów.
- Wszystkie wątki dzielą tę samą przestrzeń adresową. W wielu przypadkach, komunikacja między wątkami jest łatwiejsza i wydajniejsza, niż komunikacja międzyprocesorowa.

- Korzyści wydajnościowe, w przypadku przetwarzania równoległego na maszynach z pamięcią wspólną SMP (symmetric multiprocessing) - cecha szczególnie istotna, ze względu na fakt istnienia relatywnie tanich procesorów wielordzeniowych (multicore processors).

Biblioteka pthread zawiera wiele funkcji, które umożliwiają zarządzanie wątkami. Ich prototypy zostały zdefiniowane w pliku nagłówkowym `pthread.h`. Procedury można podzielić zgrubnie na trzy grupy:

- Zarządzanie wątkami – tworzenie, anulowanie, odłączanie, dołączanie wątków, operowanie na atrybutach wątków.
- Mechanizmy synchronizacji wątków (np. mutexy, zmienne warunkowe, bariery).
- Mechanizmy komunikacji wątków.

Biblioteka pthread zawiera ponad 60 procedur, zdefiniowanych dla języka C. W trakcie niniejszego laboratorium skupimy się na zbiorze funkcji, które są na ogół najczęściej używane i będą użyteczne dla początkującego programisty wątków POSIX.

## 5.2 Zarządzanie wątkami

### 5.2.1 Tworzenie wątków

Funkcja `main()` posiada jeden wątek (domyślny). Dodatkowe wątki w obrębie procesu muszą być jawnie utworzone przez programistę. Funkcja `pthread_create()` służy utworzeniu nowego wątku. Prototyp funkcji wygląda następująco:

```
1 int pthread_create( pthread_t* thread,
2                     const pthread_attr_t* attr,
3                     void* (*start_routine)(void*),
4                     void* arg );
```

gdzie

- `thread` jest unikalnym identyfikatorem wątku (TID), nadawanym przez system operacyjny.
- `attr` - atrybuty utworzonego wątku; `NULL`, jeśli przyjęte są wartości domyślne.
- `start_routine` - funkcja, która będzie wykonywana przez utworzony wątek, o sygnaturze w prototypie.
- `arg` - argument przekazywany jako parametr do wątku (typu `void*`); bądź `NULL`, jeśli nie przekazujemy żadnego argumentu.

Funkcja zwraca 0, gdy sukces i -1, gdy wystąpił błąd. Maksymalna liczba możliwych do utworzenia wątków jest zależna od implementacji biblioteki. Utworzone wątki mogą tworzyć nowe wątki, bez ograniczeń, związanych z hierarchią wątków.

Do pobierania identyfikatora wątku służy funkcja:

```
1 pthread_t pthread_self( void )
```

Obie funkcje `pthread_create`, `pthread_self` użyjemy w poniższym przykładzie.

**Przykład 5.2.1.** [Utworzenie wątku z pobraniem identyfikatora] Należy skompilować, zbudować i uruchomić poniższy kod.

**Kod źródłowy 16:** *Utworzenie wątku z pobraniem identyfikatora*

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define STEP 10
7
8 /* Funkcja wykonywana przez watek */
9 void* start_routine (void* arg)
10 {
11     int i;
12     pthread_t id;
13
14     id = pthread_self();
15
16     /* Wyswietl nr wątku i krok */
17     for(i = 0; i < STEP; i++)
18     {
19         printf("\tWatek: %d, step = %d\n", id, i);
20         sleep(1);
21     }
22     return NULL;
23 }
24
25 /* Program główny */
26 int main ()
27 {
28     int i, rc;
29     pthread_t id, thread_id;
30
31     /* Utworz nowy watek, który będzie wykonywał funkcje
32      start_routine. */
33     printf("Tworze watek\n");
34     rc = pthread_create (&thread_id, NULL, &start_routine, NULL);
35     if (rc)
36     {
37         printf("Blad. Kod powrotu z funkcji pthread_create(): %d\n"
38               , rc);
39         exit(1);
40     }
41
42     id = pthread_self();
43     /* Wyswietl nr wątku i krok */
44     for(i = 0; i < STEP; i++)
45     {
46         printf("Watek: %d, step = %d\n", id, i);
47         sleep(1);
48     }
49     return 0;
50 }
```

Po uruchomieniu programu, wątek główny i wątek utworzony wykonują wspólnie (równolegle) swoje zadania, wyświetlając nr aktualnie wykonywanego wątku i nr kroku. W trakcie wykonywania programu pobrać informację o identyfikatorach wątków za pomocą polecenia:

```
1 # pidin -p watek
```

gdzie `watek` jest nazwą wykonywanego procesu. Wynik działania programu powinien wyglądać następująco:

	pid	tid	name	prio	STATE	Blocked
1	237594	1	./watek	10r	NANOSLEEP	
2	237594	2	./watek	10r	NANOSLEEP	

### 5.2.2 Kończenie wątków

Wątek może być zakończony w następujący sposób:

- Następuje powrót z funkcji wykonywanej przez wątek.
- Wątek wykonuje funkcję `pthread_exit()`.
- Wątek jest anulowany przez inny wątek, za pomocą funkcji `pthread_cancel()`.
- Następuje zakończenie procesu macierzystego, poprzez wykonanie np. funkcji `exec`, bądź `exit`.

Aby jawnie zakończyć wątek, używamy funkcji:

```
1 void pthread_exit( void* value_ptr );
```

gdzie wartość `value_ptr` jest kodem powrotu wątku. W przypadku, gdy funkcja `main()` zakończy swoje wykonywanie wywołaniem `pthread_exit()`, to wykonywanie wątków będzie kontynuowane do ich zakończenia, w przeciwnym przypadku, wątki będą automatycznie zakończone, wraz z zakończeniem procesu macierzystego. Gdy wątek jest dołączalny (atribut `PTHREAD_CREATE_JOINABLE` ustawiony), to wraz z wywołaniem funkcji `pthread_exit()` wątek kończy swoje działanie, ale zwalnia zasoby dopiero po wywołaniu funkcji `pthread_join()` przez inny wątek. Gdy atrybut jest niedołączalny, to zwalnia wszystkie zasoby, tuż po wywołaniu funkcji `pthread_exit()`.

**Przykład 5.2.2.** [Utworzenie i zakończenie wątku wraz z przekazaniem argumentów] Należy przeanalizować i przetestować działanie programu.

**Kod źródłowy 17:** Utworzenie i zakończenie wątku wraz z przekazaniem argumentów

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define STEP 5
7 #define NUM_THREADS 4
```

```

8  /* Funkcja wykonywana przez wątek */
9  void* start_routine (void* arg)
10 {
11     int i, id;
12
13     id = (int)arg;
14     /* Wyświetl nr przekazanego wątku i krok */
15     for(i = 0; i < STEP; i++)
16     {
17         printf("\tWątek: %d, step = %d\n", id, i);
18         sleep(1);
19     }
20     pthread_exit(NULL);
21 }
22
23
24 /* Program główny */
25 int main ()
26 {
27     int i, rc;
28     pthread_t thread_id[NUM_THREADS];
29
30     for(i = 0; i < NUM_THREADS; i++)
31     {
32         /* Utworz nowy wątek, który będzie wykonywał funkcje
33          start_routine. */
34         /* Przekaz argument i do wątku */
35         printf("Tworzę wątek nr: %d\n", i);
36         rc = pthread_create (&thread_id[i], NULL, &start_routine, (
37             void *)i);
38         /* W przypadku, gdy nie uda się utworzyć wątku */
39         if (rc)
40         {
41             printf("Blad. Kod powrotu z funkcji pthread_create(): %d\n"
42                 , rc);
43             exit(1);
44         }
45     }
46     printf("Wątek główny zakończony. Wyjście\n");
47     pthread_exit(NULL);
48 }
```

Wyświetlić informacje o wykonywanych wątkach poleciением:

	pid	tid	name	prio	STATE	
1	# pidin -p watek					Blocked
2						
3	282650	1	./watek	10r	DEAD	
4	282650	2	./watek	10r	NANOSLEEP	
5	282650	3	./watek	10r	NANOSLEEP	
6	282650	4	./watek	10r	NANOSLEEP	
7	282650	5	./watek	10r	NANOSLEEP	

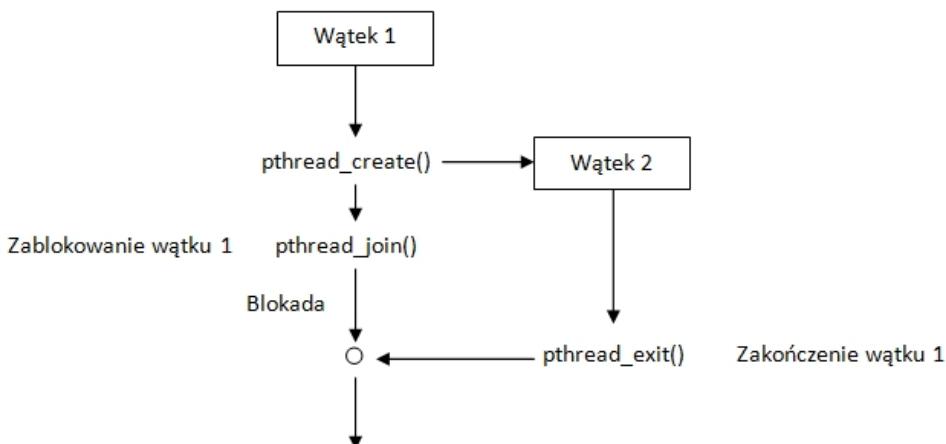
Pomimo, iż proces macierzysty (wątek główny) się zakończył, to utworzone wątki wciąż pracują, co zapewnia funkcja `pthread_exit()`.

### 5.2.3 Łączenie wątków

Funkcja `pthread_exit()` powoduje zakończenie wątku, który ją wywołał, jednak nie zwalnia automatycznie zasobów zajętych przez wątek, jak deskryptory plików, mutexy, itd. W przypadku wątków dołączalnych, zasoby te są zwalniane w momencie dołączenia wątku bieżącego do innego wątku poprzez wywołanie funkcji:

```
1 int pthread_join( pthread_t thread, void** value_ptr );
```

gdzie `thread` jest numerem wątku, na którego zakończenie czekamy, a `value_ptr` kodem powrotu, zwracanym przez zakończony wątek (wartość przekazana do `pthread_exit()` lub `PTHREAD_CANCELED`, jeśli wątek został anulowany), bądź `NULL`. Funkcja zwraca `0`, gdy sukces i `-1`, jeśli wystąpi błąd. Funkcja `pthread_join()` blokuje wątek, który wywołał funkcję, do momentu, gdy wątek o identyfikatorze `thread` zakończy swoje działanie. Gdy wskazany wątek już się zakończył, wątek bieżący nie jest blokowany i odbiera status w funkcji `pthread_join()`. Sytuacje te przedstawiają rysunki 13 oraz 14.



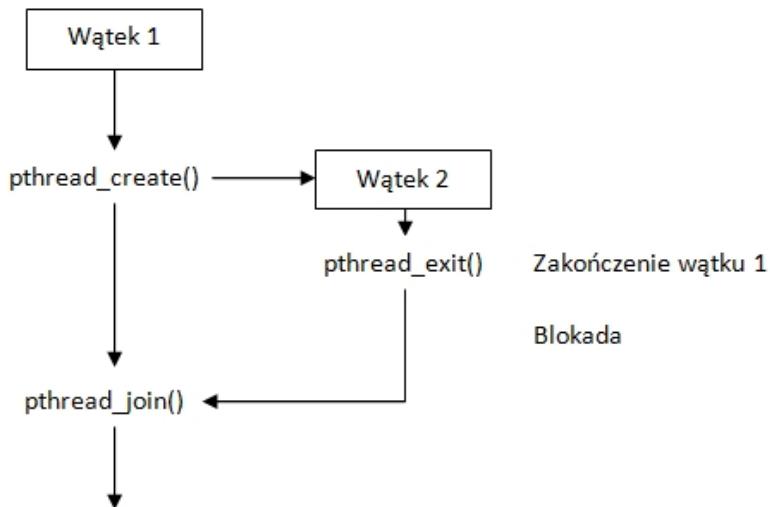
Rysunek 13: Łączenie wątków. Wątek 1 czeka na wątek 2

**Przykład 5.2.3.** [Łączenie wątków] Przykład ilustruje omawiane zagadnienia łączenia wątków, wraz z przekazaniem kodu powrotu z utworzonych wątków do wątku głównego.

Kod źródłowy 18: Łączenie wątków

```

1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define STEP 5
7 #define NUM_THREADS 4
8
9 /* Funkcja wykonywana przez wątek */
10 void* start_routine (void* arg)
11 {
12     int i, id;
13     id = (int)arg;
14 }
```



Rysunek 14: Łączenie wątków. Wątek 2 czeka na wątek 1

```

15  /* Wyświetl nr przekazanego wątku i krok */
16  for(i = 0; i < STEP; i++)
17  {
18      printf("\tWątek: %d, step = %d\n", id, i);
19      sleep(1);
20  }
21  pthread_exit((void *)id);
22 }
23
24 /* The main program. */
25 int main ()
26 {
27     int i, rc;
28     pthread_t thread_id[NUM_THREADS];
29     void* status;
30
31     for(i = 0; i < NUM_THREADS; i++)
32     {
33         /* Utworz nowy wątek, który będzie wykonywał funkcje
            start_routine. */
34         /* Przekaz argument i do wątku */
35         printf("Tworzę wątek nr: %d\n", i);
36         rc = pthread_create (&thread_id[i], NULL, &start_routine, (
            void *)i);
37         /* W przypadku, gdy nie uda się utworzyć wątku */
38         if (rc)
39         {
40             printf("Błąd. Kod powrotu z funkcji pthread_create(): %d\n"
                   , rc);
41             exit(1);
42         }
43     }
44     /* Oczekiwanie na wątki potomne */
45     for(i = 0; i < NUM_THREADS; i++)
46     {
47         rc = pthread_join(thread_id[i], &status);
48         if (rc)
49         {
50             printf("Błąd. Kod powrotu z funkcji pthread_join(): %d\n",
                   rc);
51             exit(1);
52         }
  
```

```

53     printf("Watek glowny dolaczyl watek %d o statusie %d\n", i, (
54         int)status);
55
56     printf("Watek glowny zakonczony. Wyjscie.\n");
57     pthread_exit(NULL);
58 }
```

### 5.2.4 Atrybuty wątków

Biblioteka pthread dostarcza mechanizmu do regulowania własności wątków. Atrybuty wątków są przekazywane jako parametry w strukturze `pthread_attr_t*` `attr` funkcji `pthread_create()`, w chwili tworzenia wątku. W przypadku, gdy przekazujemy wskaźnik `NULL`, atrybuty są ustawiane domyślnie. Jeśli chcemy utworzyć wątek, z atrybutami innymi, niż domyślne, jawnie dobranymi przez użytkownika należy:

- Utworzyć obiekt `attr` typu `pthread_attr_t`.
- Zainicjować zmienną `attr` wywołaniem funkcji `pthread_attr_init(&attr)`.
- Zmodyfikować obiekt z atrybutami, tak, aby zawierał pożądane wartości.
- Przekazać wskaźnik do struktury `attr`, w trakcie tworzenia wątku za pomocą `pthread_create()`.
- Zwolnić zasoby wykorzystywane przez atrybut, przez wywołanie funkcji `pthread_attr_destroy(&attr)`.

Wybrane atrybuty wątku przedstawiono w tabeli 29, a funkcje służące do ich ustawiania w tabeli 30.

**Tabela 29:** Wybrane atrybuty wątku i ich wartości domyślne

Atrybut	Wartość domyślna
Dołączalność ( <code>detachstate</code> )	<code>PTHREAD_CREATE_JOINABLE</code>
Dziedziczenie atrybutów ( <code>inherit-scheduling</code> )	<code>PTHREAD_INHERIT_SCHED</code>
Strategia szeregowania ( <code>schedpolicy</code> )	<code>PTHREAD_INHERIT_SCHED</code>
Parametry szeregowania ( <code>schedparam</code> )	Dziedziczone z procesu macierzystego
Rywaliczka wątku o zasoby ( <code>contentionscope</code> )	<code>PTHREAD_SCOPE_SYSTEM</code>
Rozmiar stosu ( <code>stacksize</code> )	4kB
Adres stosu ( <code>stackaddr</code> )	<code>NULL</code>

Objaśnienia:

- Dołączalność - informacja, czy wątek ma zwolnić zasoby natychmiast (`PTHREAD_CREATE_DETACHED`), czy po wywołaniu przez proces macierzysty funkcji `pthread_join` (`PTHREAD_CREATE_JOINABLE`).
- Dziedziczenie atrybutów - domyślnie są dziedziczone z wątku macierzystego.

- Strategia szeregowania: SCHED\_FIFO, SCHED\_RR, SCHED\_SPORADIC, SCHED\_OTHER - szczegółowe wyjaśnienia w dokumentacji.
- Parametry szeregowania - informacje, dot. szeregowania wątków. Modyfikowana jest struktura `const struct sched_param * param`.
- Rywalizacja wątku o zasoby - wątki mogą rywalizować o zasoby z wątkami systemowymi z innych procesów (ang. system contention scope). Wątki mogą również rywalizować o zasoby tylko w obrębie procesu, który je utworzył (process contention scope). Domyślnie ustawiana jest pierwsza opcja.
- Rozmiar stosu - informacja o rozmiarze stosu do przechowywania zmiennych lokalnych.
- Adres stosu - gdy adres stosu jest ustawiony na NULL, to będzie ustalany i zwalniany automatycznie przez system operacyjny. Pamięć na stos może być też przydzielona przez programistę i wtedy jest on odpowiedzialny za jej zwolnienie.

Po inicjalizacji struktury z atrybutami, możemy pobierać i ustawiać atrybuty związane z wątkami.

**Tabela 30:** Funkcje od pobierania i ustawiania atrybutów

Atrybut	Pobierz atrybut	Ustaw atrybut
Dołączalność	<code>pthread_attr_getdetachstate()</code>	<code>pthread_attr_setdetachstate()</code>
Dziedziczenie atrybutów	<code>pthread_attr_getinheritsched()</code>	<code>pthread_attr_setinheritsched()</code>
Strategia szeregowania	<code>pthread_attr_getschedpolicy()</code>	<code>pthread_attr_setschedpolicy()</code>
Parametry szeregowania	<code>pthread_attr_getschedparam()</code>	<code>pthread_attr_setschedparam()</code>
Rywalizacja o zasoby	<code>pthread_attr_getscope()</code>	<code>pthread_attr_setscope()</code>
Rozmiar stosu	<code>pthread_attr_getstacksize()</code>	<code>pthread_attr_setstacksize()</code>
Adres stosu	<code>pthread_attr_getstackaddr()</code>	<code>pthread_attr_setstackaddr()</code>

**Przykład 5.2.4.** [Atrybuty wątków] Przykład ilustruje omawiane zagadnienia łączenia wątków, wraz z przekazaniem kodu powrotu z utworzonych wątków do wątku głównego.

**Kod źródłowy 19:** Ustawianie atrybutu dołączalności

```

1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define STEP 5
7 #define NUM_THREADS 4
8
9 /* Funkcja wykonywana przez watek */
10 void* start_routine (void* arg)
11 {
12     int i, id;
13
14     id = (int)arg;
15     /* Wyswietl nr przekazanego wątku i krok */
16     for(i = 0; i < STEP; i++)
17     {
18         printf("\tWatek: %d, step = %d\n", id, i);
19         sleep(1);

```

```

20     }
21     pthread_exit((void *)id);
22 }
23
24 /* Program glowny */
25 int main ()
26 {
27     int i, rc;
28     pthread_t thread_id[NUM_THREADS];
29     void* status;
30     pthread_attr_t attr;
31
32     /* Zainicjalizowanie struktury atrybutow */
33     pthread_attr_init(&attr);
34     /* Ustawienie atrybut dolaczalnosci */
35     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
36
37     for(i = 0; i < NUM_THREADS; i++)
38     {
39         /* Utworz nowy watek, ktory bedzie wykonywal funkcje
39          start_routine. */
40         /* Przekaz argument i do watku */
41         printf("Tworze watek nr: %d\n", i);
42         rc = pthread_create (&thread_id[i], &attr, &start_routine, (
43             void *)i);
44         /* W przypadku, gdy nie uda sie utworzyc watku */
45         if (rc)
46         {
47             printf("Blad. Kod powrotu z funkcji pthread_create(): %d\n"
48                   , rc);
49             exit(1);
50         }
51         /* Zwolnienie atrybutow */
52         pthread_attr_destroy(&attr);
53         /* Oczekiwanie na watki potomne */
54         for(i = 0; i < NUM_THREADS; i++)
55         {
56             rc = pthread_join(thread_id[i], &status);
57             if (rc)
58             {
59                 printf("Blad. Kod powrotu z funkcji pthread_join(): %d\n",
60                       rc);
61                 exit(1);
62             }
63             printf("Watek glowny dolaczyl watek %d o statusie %d\n", i, (
64                 int)status);
65         }
66     }
67
68     printf("Watek glowny zakonczony. Wyjscie.\n");
69     pthread_exit(NULL);
70 }
```

## 5.2.5 Ustalanie priorytetu, strategii i parametrów szeregowania wątków

### Dziedziczenie atrybutów

**Ustawianie atrybutów.** Wątki potomne domyślnie dziedziczą priorytet i własności dotyczące szeregowania z wątku macierzystego. Funkcja `pthread_attr_setinheritsched` umożliwia zmianę tego stanu:

```
1 int pthread_attr_setinheritsched(pthread_attr_t * attr, int
    inheritsched );
```

gdzie attr jest wskaźnikiem na strukturę definiującą atrybuty wątku, a wartość inheritsched jest jedną z dwóch wartości:

- PTHREAD\_INHERIT\_SCHED - wątek potomny dziedziczy strategię szeregowania od wątku rodzica - wartość domyślna.
- PTHREAD\_EXPLICIT\_SCHED - strategia szeregowania jest ustawiona przez strukturę attr.

**Pobieranie atrybutów.** Funkcją, która umożliwia pobieranie parametrów dot. dziedziczenia własności z wątku macierzystego ma następującą sygnaturę:

```
1 int pthread_attr_getinheritsched(const pthread_attr_t* attr, int*
    inheritsched );
```

gdzie wartość inheritsched jest wskaźnikiem do miejsca, gdzie funkcja przechowa atrybut.

#### Strategia szeregowania

**Ustawianie atrybutów.** Ustawianie strategii szeregowania realizujemy funkcją:

```
1 int pthread_attr_setschedpolicy(pthread_attr_t* attr, int policy)
    ;
```

gdzie attr jest wskaźnikiem na strukturę definiującą atrybuty wątku, a wartość policy jest jedną z wartości zdefiniowanych w tabeli 31.

**Tabela 31:** Strategie szeregowania w QNX Neutrino

Numer	Symbol	Opis
0	SCHED_NOCHANGE	Brak zmiany strategii szeregowania
1	SCHED_FIFO	Szeregowanie FIFO
2	SCHED_RR	Szeregowanie karuzelowe
3	SCHED_OTHER	wskazuje wartość SCHED_RR
4	SCHED_SPORADIC	Szeregowanie sporadyczne

**Pobieranie atrybutów.** Pobieranie strategii szeregowania można zrealizować funkcją:

```
1 int pthread_attr_getschedpolicy(const pthread_attr_t* attr, int*
    policy );
```

gdzie policy jest wskaźnikiem do miejsca przechowywania ustawionej wartości strategii szeregowania.

#### Parametry szeregowania

**Ustawianie atrybutów.** Aby ustawić atrybuty z tego zbioru należy zastosować funkcję:

```
1 int pthread_attr_setschedparam(pthread_attr_t * attr, const struct
    sched_param * param );
```

gdzie attr jest wskaźnikiem na strukturę definiującą atrybuty wątku, a zmienna param jest wskaźnikiem na strukturę sched\_param, która definiuje parametry szeregowania wątków.

**Pobieranie atrybutów.** Tę funkcję realizujemy następująco:

```
1 int pthread_attr_getschedparam(const pthread_attr_t * attr,
    struct sched_param * param );
```

gdzie param jest wskaźnikiem na strukturę sched\_param.

**Przykład 5.2.5.** [Zarządzanie atrybutami szeregowania wątków] Przykład pokazuje w jaki sposób używać atrybutów, w celu utworzenia wątku ze zdefiniowanymi przez programistę parametrami i strategią szeregowania.

### Kod źródłowy 20: Pobieranie i ustawianie atrybutów szeregowania wątków

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 #define STEP 5
7
8 /* Funkcja wykonywana przez wątek */
9 void* start_routine (void* arg)
10 {
11     int i;
12
13     /* Wyświetl nr przekazanego wątku i krok */
14     for(i = 0; i < STEP; i++)
15     {
16         printf("\tStep = %d\n", i);
17         sleep(1);
18     }
19     pthread_exit(NULL);
20 }
21
22 /* The main program. */
23 int main ()
24 {
25     int rc;
26     pthread_t thread_id;
27     void* status;
28     pthread_attr_t attr;
29     struct sched_param param; /* struktura opisująca parametry
        szeregowania */
30     int policy;
31
32     /* pobranie parametrów procesu macierzystego */
33     sched_getparam(0, &param);
34
35     printf("Priorytet ustawiony przy starcie procesu (wątku
        głównego) %d.\n", param.sched_priority);
```

```

36  /* Zainicjalizowanie struktury atrybutow */
37  pthread_attr_init(&attr);
38  /* Ustawienie parametru PTHREAD_EXPLICIT_SCHED */
39  pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
40  /* Pobranie strategii szeregowania */
41  pthread_attr_getschedpolicy(&attr, &policy);
42  /* Pobranie parametrow szeregowania */
43  pthread_attr_getschedparam(&attr, &param);
44
45  /* Wyswietl atrybuty */
46  printf("Domyslna strategia %s, priorytet: %d\n", policy ==
47      SCHED_FIFO ? "FIFO" :
48      ( policy == SCHED_RR ? "ROUND ROBIN" :
49      ( policy == SCHED_SPORADIC ? "SPORADIC" :
50      (policy == SCHED_OTHER ? "OTHER" :
51      "unknown"))), param.sched_priority);
52
53  /* Ustawienie strategii szeregowania */
54  pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
55  param.sched_priority = 8;
56  /* Ustawienie parametrow szeregowania */
57  pthread_attr_setschedparam(&attr, &param);
58
59  /* Pobranie strategii szeregowania */
60  pthread_attr_getschedpolicy(&attr, &policy);
61  /* Pobranie parametrow szeregowania */
62  pthread_attr_getschedparam(&attr, &param);
63
64  /* Wyswietl atrybuty */
65  printf("Domyslna strategia %s, priorytet: %d\n", policy ==
66      SCHED_FIFO ? "FIFO" :
67      ( policy == SCHED_RR ? "ROUND ROBIN" :
68      ( policy == SCHED_SPORADIC ? "SPORADIC" :
69      (policy == SCHED_OTHER ? "OTHER" :
70      "unknown"))), param.sched_priority);
71
72  /* Utworz nowy watek, z nowymi atrybutami. */
73  printf("Tworze watek z nowymi atrybutami: \n");
74  rc = pthread_create (&thread_id, &attr, &start_routine, NULL);
75  /* W przypadku, gdy nie uda sie utworzyc watku */
76  if (rc)
77  {
78      printf("Blad. Kod powrotu z funkcji pthread_create(): %d\n",
79             rc);
80      exit(1);
81  }
82
83  /* Zwolnienie atrybutow */
84  pthread_attr_destroy(&attr);
85
86  /* Oczekiwanie na watek potomny */
87  rc = pthread_join(thread_id, &status);
88  if (rc)
89  {
90      printf("Blad. Kod powrotu z funkcji pthread_join(): %d\n", rc
91             );
92      exit(1);
93  }
94
95  printf("Watek glowny dolaczyl watek potomny.\n");
96  printf("Watek glowny zakonczony. Wyjscie.\n");
97  pthread_exit(NULL);
98 }
```

Przykładowy wynik działania programu powinien wyglądać następująco:

```

1 Priorytet ustawiony przy starcie procesu (wątku głównego) 10.
2 Domyslna strategia ROUND ROBIN, priorytet: 10
3 Domyslna strategia FIFO, priorytet: 8
4 Tworze watek z nowymi atrybutami:
5   Step = 0
6   Step = 1
7   Step = 2
8   Step = 3
9   Step = 4
10 Watek glowny dolaczyl watek potomny.
11 Watek glowny zakonczony. Wyjscie.

```

### 5.3 Ćwiczenia

- Napisać program, który tworzy 8 wątków i przekazuje do niego strukturę następującego typu:

```

1 struct thread_data
2 {
3     int cnt;           /* numer wątku */
4     int sum;          /* suma kontrolna */
5     const char *msg; /* wiadomość */
6 }

```

gdzie

- cnt - numer wątku (nadajemy sami wg kolejności tworzenia: `cnt=0,1,2,...`).
- sum - „suma kontrolna”, będąca sumą numerów wątków (np. dla pierwszego wątku `sum=0`, dla drugiego `sum=0+1`, dla trzeciego `sum=0+1+2`, itd.).
- msg - wskaźnik do wiadomości otrzymanej z programu głównego.

W programie głównym utworzyć tablicę o następującej zawartości:

```

1 messages [0] = "English: Hello World!";
2 messages [1] = "French: Bonjour, le monde!";
3 messages [2] = "Spanish: Hola al mundo";
4 messages [3] = "Polski: Witaj swiecie!";
5 messages [4] = "German: Guten Tag, Welt!";
6 messages [5] = "Russian: Zdravstvyyte, mir!";
7 messages [6] = "Japan: Sekai e konnichiwa!";
8 messages [7] = "Latin: Orbis, te saluto!";

```

Odpowiednie pola tablicy będą przekazywane do wątków. Każdy z wątków powinien wyświetlać otrzymaną strukturę danych. Należy zadbać o poprawne zakończenie wątków.

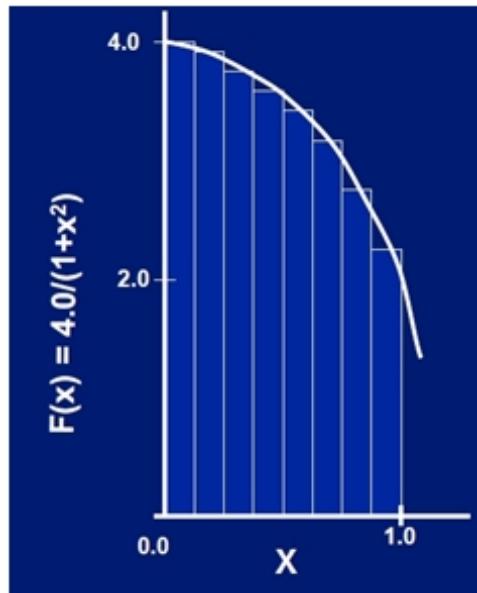
- Zaproponować współbieżną (równoległą - gdy dysponujemy komputerem wieloprocesorowym) wersję programu do obliczania liczby  $\pi$  za pomocą wątków. Matematycznie wiadomo, że:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

Całkę oznaczoną możemy aproksymować jako sumę:

$$\sum_{i=0}^N \frac{4}{1+x_i^2} \Delta x \approx \pi$$

Ilustrację graficzną aproksymacji liczby  $\pi$  można znaleźć na rysunku 15.



Rysunek 15: Idea aproksymacji liczby  $\pi$

**Kod źródłowy 21:** Kod źródłowy sekwencyjnej wersji do obliczania liczby  $\pi$

```

1 #include <stdio.h>
2
3 int num_steps = 100000;
4
5
6 int main ()
7 {
8     int i;
9     double x, pi, step, sum = 0.0;
10
11    step = 1.0 / (double) num_steps;
12    for (i=0; i < num_steps; i++)
13    {
14        x = (double)(i + 0.5) * step;
15        sum = sum + 4.0 / (1.0 + x * x);
16    }
17    pi = step * sum;
18
19    printf("PI = %.15f\n", pi);
20
21    return 0;
22 }
```

3. Rozszerzyć poprzedni przykład o możliwość zmiany priorytetów i strategii szeregowania wątków.



## Laboratorium 6

### Mechanizmy synchronizacji wątków – mutexy, zmienne warunkowe, bariery

---

#### 6.1 Wprowadzenie

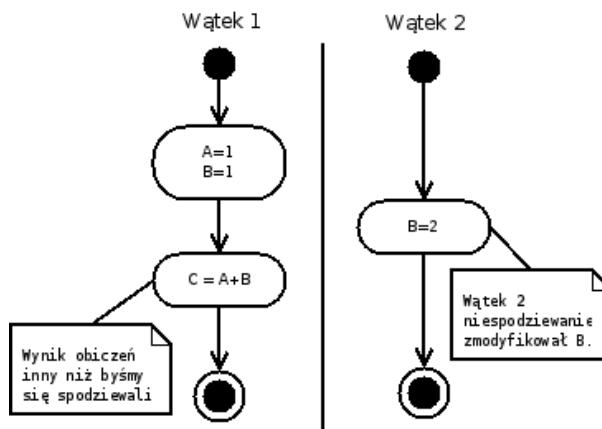
Zastosowanie obliczeń współbieżnych na ogólnie pozwala na osiągnięcie większej wydajności przetwarzania, w szczególności, gdy obliczenia są przeprowadzane na komputerach wieloprocesorowych. Pisanie poprawne działających programów wielowątkowych wymaga specyfikacji i rozwiązania problemów, które w typowych obliczeniach sekwencyjnych nie występują. Problemy w programowaniu współbieżnym biorą się z potrzeby synchronizowania wykonania wątków i zapewnienia im metod komunikowania się między sobą. Przy współbieżnym programowaniu należy zwrócić uwagę między innymi na następujące zagadnienia:

- Wyścigi – do sytuacji wyścigu (ang. race condition) dochodzi wówczas, gdy dwa lub więcej procesów (wątków) wykonuje operacje na zasobach wspólnie dostępujących, a ostateczny wynik tej operacji w sposób przypadkowy zależy od czasu realizacji ciągów instrukcji.
- Spójność danych (ang. data consistency) – współpracujące wątki mogą jednocześnie odczytywać i zapisywać te same obszary pamięci. Dostęp do wspólnego zasobu musi być synchronizowany, w celu zapewnienia spójności danych, w przeciwnym przypadku, rezultat przetwarzania może być inny, niż oczekiwany.

Rozwiązaniem wymienionych problemów jest zastosowanie synchronizacji, czyli kontroli przepływu sterowania pomiędzy wątkami (procesami), tak, żeby dopuszczalne były tylko ciągi instrukcji, które są zgodne z intencjami. System operacyjny czasu rzeczywistego QNX Neutrino dostarcza wielu metod synchronizacji pracy wątków. Jedne dostarczają jawnych metod synchronizacji, takich jak mutexy (mutex), zmienne warunkowe (conditional variables), bariery (barriers), zamki wstrzymujące (sleepon lock), zamki czytelników i pisarzy (reader writer lock), semafory (semaphores). Drugą grupę stanowią metody niejawne, które gwarantują synchronizację, poprzez wbudowane mechanizmy. Należą do nich szeregowanie FIFO (FIFO scheduling), wysyłanie komunikatów (messages), operacje atomowe (atomic operations). Oprócz barier i blokad wstrzymujących wszystkie przytoczone mechanizmy pozwalają synchronizować nie tylko wątki, ale również procesy. Dodatkowym aspektem, który może mieć znaczenie jest możliwość synchronizacji wątków (procesów) w sieci. Spośród wymienionych metod semafory (nazwane) oraz komunikaty spełniają to wymaganie. W trakcie laboratorium omówimy podstawowe operacje synchronizacji wątków (mutexy, zmienne warunkowe, bariery) oraz pojęcia z nimi związane (wzajemne wykluczanie, sekcja krytyczna, zakleszczenia, inwersja priorytetów). Rozważania będą ilustrowane przykładami pracy programów współbieżnych z zastosowaniem mechanizmów synchronizacji pracy wątków.

## 6.2 Wyścigi

Scenariusz, w którym występuje wyścig o współdzielony zasób przedstawia przykład. Wątek główny (wątek 1) wykonuje się współbieżnie z wątkiem 2. Oba wątki współdzielą zmienną `b`. Problem pojawia się w sytuacji, gdy dwa wątki próbują "jednocześnie" zmodyfikować dane. W rezultacie wynik operacji będzie nieprzewidywalny i zależny od kolejności wykonania ciągów instrukcji przez poszczególne wątki. Jeden ze scenariuszy przedstawiono na rysunku 16.



Rysunek 16: Ilustracja wyścigu (race condition) pomiędzy dwoma wątkami

**Przykład 6.2.1.** [Brak synchronizacji pracy wątków] Skompilować, uruchomić kilkakrotnie program oraz zaobserwować różne scenariusze wykonania programu w zależności od czasu pracy poszczególnych wątków.

Kod źródłowy 22: Ilustracja braku synchronizacji pracy wątków

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <time.h>
6
7 /* Dane wspolne */
8 int b;
9
10 void* start(void* arg)
11 {
12     /* zainicjalizuj generator liczb losowych */
13     srand((unsigned int)(time(NULL)+20));
14     /* wygeneruj liczbe z zakresu 1 do 3 */
15     int number = rand() % 3 + 1;
16     printf("\t\t\t\t2: Pracuje %d sek. ...\n",number);
17     sleep(number);
18     printf("\t\t\t\t2: Modyfikuje b\n");
19     b = 2;
20     printf("\t\t\t\t2: b = %d\n", b);
21     return NULL;
22 }
23
24 int main(int argc, char* argv[])
25 {
26     pthread_t start_id; /* Identyfikator tworzonego wątku */
27     pthread_attr_t attr; /* Atrybuty wątku */

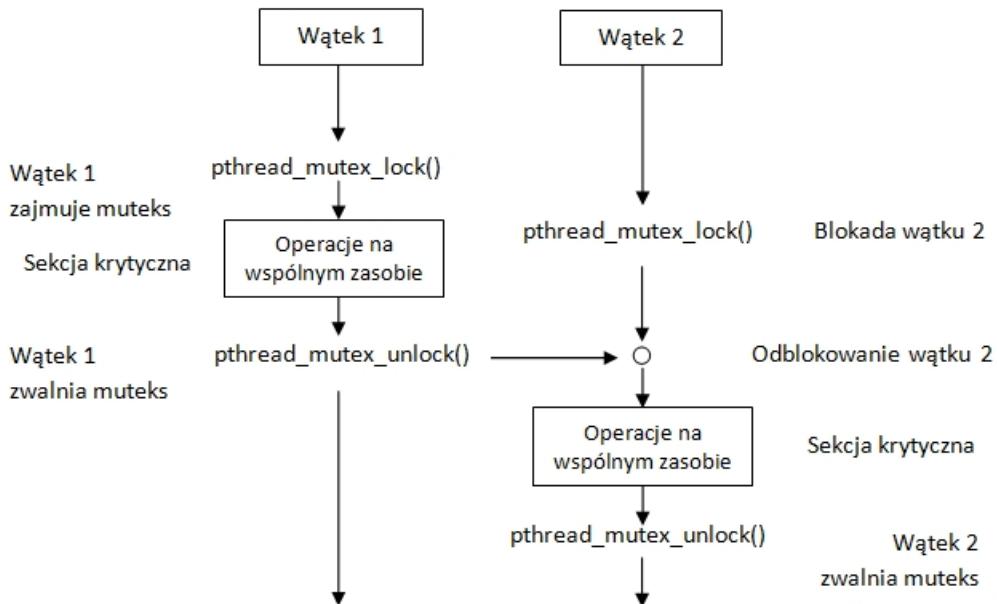
```

```

28     int a, c; /* Dane lokalne wątku głównego */
29     int number; /* Przechowuje liczbę losową */
30
31     /* zainicjalizuj generator liczb losowych */
32     srand((unsigned int)time(NULL));
33     /* wygeneruj liczbę z zakresu 1 do 3 */
34     number = rand() % 3 + 1;
35
36     /* Inicjalizuj strukturę z atrybutami*/
37     pthread_attr_init( &attr );
38     /* Utworz nowy wątek (nazwijmy go "start") */
39     pthread_create(&start_id, &attr, start, NULL);
40
41     /* Inicjalizuj "a" i "b" */
42     printf("1: Modyfikuje b\n");
43     a = 1; b = 1;
44     printf("1: a = %d, b = %d\n",a, b);
45
46     printf("1: Pracuje %d sek. ... \n",number);
47     sleep(number);
48
49     /* Oblicz "c" i wydrukuj wynik */
50     c = a + b;
51     printf("1: c = a + b = %d\n",c);
52
53     if (c == 2)
54     {
55         printf("1: Obliczenia przeprowadzone prawidłowo.\n");
56     }
57     else
58     {
59         printf("1: Niepoprawne obliczenia!\n");
60     }
61     /* Czekaj na zakończenie wątku "start" */
62     pthread_join(start_id, NULL);
63
64     return 0;
65 }
```

### 6.3 Muteksy

Popularną i ogólną metodą synchronizowania pracy wątków jest zapewnienie dostępu do współdzielonych zasobów (np. pamięci) tylko dla jednego z potencjalnie wielu wykonywanych wątków lub procesów. Wymaganie to nazywa się wzajemnym wykluczaniem (ang. mutual exclusion). System QNX RTOS dostarcza zgodnego ze standardem POSIX mechanizmu wzajemnego wykluczania zwanego muteksem, który jest specjalną formą bardziej ogólnego mechanizmu semafora. Muteks jest rodzajem blokady, którą może zająć tylko jeden wątek jednocześnie. Jeśli jeden wątek zajął muteks i drugi w tym czasie próbuje zająć muteks, to drugi wątek zostanie zablokowany (ang. locked). Muteks może być zwolniony tylko przez ten wątek, który go zajął. W tej sytuacji, wątek drugi zostaje odblokowany (ang. unlocked). System operacyjny gwarantuje, że nie pojawią się wyścigi pomiędzy wątkami o zajęcie muteksu. Tylko jeden wątek może zająć muteks, pozostałe będą zablokowane. Ciąg instrukcji, który jest wykonywany tylko przez jeden, z potencjalnie wielu wątków (procesów) nazywamy sekcją krytyczną (ang. critical section). Zasadę działania muteksu, w przypadku dwóch wątków, pracujących wspólnie przedstawia rysunek 17.



Rysunek 17: Zasada działania mutexów

Typowy sposób użycia mutexu składa się z następujących operacji:

1. Utworzenie mutexu (zmiennej typu `pthread_mutex_t`) i inicjalizacja.
2. Kilka wątków próbuje zająć mutex, tylko jednemu się udaje.
3. Wykonanie operacji na zasobie w sekcji krytycznej, przez wątek, który zajął mutex.
4. Zwolnienie mutexu.
5. Zajęcie mutexu i powtórzenie procesu.
6. Skasowanie mutexu.

### 6.3.1 Tworzenie i kasowanie mutexów

Mutex jest reprezentowany w programie za pomocą zmiennej `pthread_mutex_t`. Funkcja `pthread_mutex_init()` służy do inicjalizacji tego obiektu:

```

1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3 int pthread_mutex_init(
4     pthread_mutex_t * mutex,
5     const pthread_mutexattr_t * attr );
  
```

gdzie

- `mutex` – jest wskaźnikiem do obiektu, który chcemy zainicjalizować.
- `attr` – jest albo NULL (atrybuty domyślne) albo wskaźnikiem do struktury definiującej atrybuty mutexu.

Muteksy możemy inicjalizować nie tylko dynamicznie, używając struktury `attr`, ale również statycznie, poprzez zastosowanie makra `PTHREAD_MUTEX_INITIALIZER`, które zapewnia przypisanie domyślnych wartości do muteksu. Domyślnie muteks jest w stanie odblokowanym. Do kasowania muteksu służy funkcja:

```
1 int pthread_mutex_destroy( pthread_mutex_t* mutex );
```

Funkcja `pthread_mutex_destroy()` kasuje zasoby zajęte przez muteks. Kasowany muteks powinien być w stanie odblokowanym. Zajęty muteks może być skasowany przez właściciela muteksu. W takim przypadku wątki czekające na muteksie zostają odblokowane.

### 6.3.2 Zajmowanie i zwalnianie muteksów

Funkcja `pthread_mutex_lock()` służy do zajęcia muteksu przez wątek. Jeśli muteks jest zajęty przez inny wątek, to pozostałe będą w stanie zablokowanym, aż do momentu, gdy muteks zostanie odblokowany. Gdy muteks jest wolny, to następuje jego zajęcie przez wątek bieżący.

```
1 int pthread_mutex_lock( pthread_mutex_t* mutex );
```

gdzie `mutex` jest wskaźnikiem do obiektu typu `pthread_mutex_t`. Zablokowane wątki mogą czekać dowolnie długo na odblokowanie muteksu. Funkcją, która umożliwia zajęcie muteksu na określony czas jest `pthread_mutex_timedlock()` o sygnaturze

```
1 int pthread_mutex_timedlock(
2                     pthread_mutex_t * mutex,
3                     const struct timespec * abs_timeout );
```

gdzie `mutex` jest wskaźnikiem do obiektu typu `pthread_mutex_t`, `abs_timeout` wskaźnikiem do struktury:

```
1 struct timespec {
2     time_t      tv_sec;
3     long        tv_nsec;
4 }
```

opisującej końcową chwilę czasową, do której wątek ma czekać na odblokowanie muteksu (czas absolutny). Używanie funkcji `pthread_mutex_lock()`, czy `pthread_mutex_timedlock()` do zajęcia muteksu powoduje zablokowanie wątków wywołujących, gdy muteks jest zajęty. Na ogół taki stan jest pożądaný. Zdarzają się jednak sytuacje, kiedy wątek zamiast czekać na zwolnienie muteksu, może wykonać użyteczne operacje. Biblioteka pthread dostarcza użytecznego mechanizmu `pthread_mutex_trylock()`, który jest formą nieblokującego zajęcia muteksu.

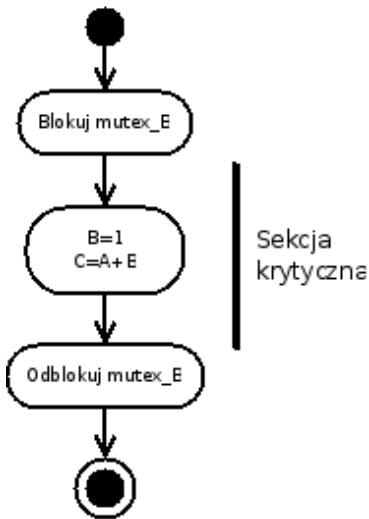
```
1 int pthread_mutex_trylock( pthread_mutex_t* mutex );
```

Funkcja `pthread_mutex_trylock()` próbuje zająć muteks, ale nie blokuje wątków wywołujących, gdy

mutex jest zajęty. Należy pamiętać, że aby odblokowywać mutex wówczas, kiedy funkcja `pthread_mutex_trylock` zwróci kod sukcesu. Funkcja ta jest użyteczna, w przypadku zakleszczeń i sytuacji związanych z inwersją priorytetów. Funkcją, która pozwala na zwolnienie mutexu jest `pthread_mutex_unlock()`:

```
1 int pthread_mutex_unlock( pthread_mutex_t * mutex );
```

Funkcja `pthread_mutex_unlock()` zwalnia mutex. Gdy istnieją wątki, które są zablokowane na mutexie, to zostaje odblokowany wątek, z najwyższym priorytetem i on staje się właścicielem mutexu. Gdy brak wątków zajmujących mutex, to stan mutexu zostaje ustawiony na wolny.



Rysunek 18: Przykład użycia mutexów

**Przykład 6.3.1.** [Synchronizacja pracy wątków] Przykład jest modyfikacją przykładu 6.2.1 i pokazuje sposób użycia mutex do ochrony wspólnych zasobów. Skompilować, uruchomić program oraz prześledzić jego działanie.

Kod źródłowy 23: Przykład użycia mutexów

```

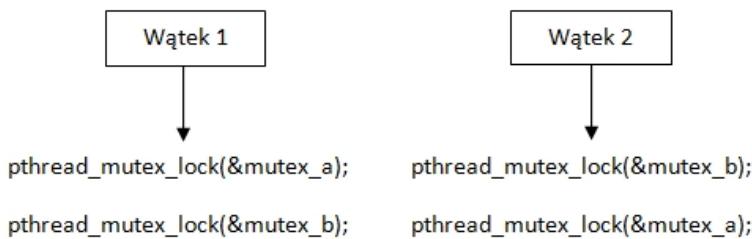
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <time.h>
6
7 /* Dane wspolne */
8 int b;
9 /* Statyczna inicjalizacja mutexu chroniacego dane wspolne */
10 pthread_mutex_t mutex_b = PTHREAD_MUTEX_INITIALIZER;
11
12
13 void* start(void* arg)
14 {
15     /* zainicjalizuj generator liczb losowych */
16     srand((unsigned int)(time(NULL)+20));
17     /* wygeneruj liczbe z zakresu 1 do 3 */
18     int number = rand() % 3 + 1;
19     printf("\t\t\t\tt2: Pracuje %d sek. ... \n", number);
20     sleep(number);
  
```

```

21     pthread_mutex_lock(&mutex_b);      /* Blokuj mutex */
22     printf("\t\t\t\t2: Modyfikuje b\n");
23     b = 2;
24     printf("\t\t\t\t2: b = %d\n", b);
25     pthread_mutex_unlock(&mutex_b);      /* Odblokuj mutex */
26
27     return NULL;
28 }
29
30 int main(int argc, char* argv[])
31 {
32     pthread_t start_id;    /* Identyfikator tworzonego wątku */
33     pthread_attr_t attr;   /* Atrybuty wątku */
34     int a, c; /* Dane lokalne wątku głównego */
35     int number; /* Przechowuje liczbę losową */
36
37     /* Zainicjalizuj generator liczb losowych */
38     srand((unsigned int)time(NULL));
39     /* Wygeneruj liczbę z zakresu 1 do 3 */
40     number = rand() % 3 + 1;
41
42     /* Inicjalizuj strukturę z atrybutami */
43     pthread_attr_init( &attr );
44     /* Utwórz nowy wątek (nazwijmy go "start") */
45     pthread_create(&start_id, &attr, start, NULL);
46
47     /* Inicjalizuj "a" i "b" */
48     printf("1: Modyfikuje b\n");
49     a = 1;
50
51     pthread_mutex_lock(&mutex_b);      /* Blokuj mutex */
52     b = 1;
53     printf("1: a = %d, b = %d\n", a, b);
54
55     printf("1: Pracuje %d sek. ... \n", number);
56     sleep(number);
57
58     /* Oblicz "c" i wydrukuj wynik */
59     c = a + b;
60     pthread_mutex_unlock(&mutex_b);      /* Odblokuj mutex */
61     printf("1: c = a + b = %d\n", c);
62
63     if (c == 2)
64     {
65         printf("1: Obliczenia przeprowadzone prawidłowo.\n");
66     }
67     else
68     {
69         printf("1: Niepoprawne obliczenia!\n");
70     }
71     pthread_mutex_destroy(&mutex_b); /* Skasowanie mutexu */
72
73     /* Czekaj na zakończenie wątku "start" */
74     pthread_join(start_id, NULL);
75
76     return 0;
77 }
```

### 6.3.3 Problemy przy stosowaniu muteksów

**Zakleszczenia.** Czasami użycie jednego muteksu do ochrony wspólnych danych jest niewystarczające. W przypadku jednoczesnego stosowania więcej niż jednego muteksu pojawiają się problemy, które nie występują, gdy stosujemy jeden muteks. Jednym z nich jest zakleszczenie (ang. deadlock), czyli sytuacja, gdy dwa (lub więcej wątków) czekają na siebie nawzajem, zajmując muteks (zasób) i jednocześnie czekają na zwolnienie innego muteksu (zasobu), zajętego przez inne wątki, potrzebnego do kontynuacji swojego działania. Klasyczną sytuacją jest scenariusz przedstawiony na rysunku 19.



Rysunek 19: Sytuacja zakleszczenia wątków (deadlock)

Oba wątki mogą ukończyć pierwszy etap scenariusza (zablokowanie na muteksach `mutex_a` oraz `mutex_b`) orientacyjnie w tym samym czasie. Jeśli ciąg instrukcji będzie taki, że np. wątek 2 najpierw zablokuje `mutex_b`, to wątek 1 zostanie zablokowany na muteksie, który jest już zajęty przez wątek 2, a następnie dojdzie do zablokowania wątku 2. Istnieją dwa typowe rozwiązania tego problemu:

- Zajmowanie muteksów w ścisłe określonej kolejności. (np. najpierw `mutex_a`, a później `mutex_b`).
- Zastosowanie nieblokującej funkcji do zajmowania muteksu. Po zablokowaniu np. `mutex_a`, używamy nieblokującej funkcji `pthread_mutex_trylock()`, aby sprawdzić, czy `mutex_b` jest zajęty; jeśli tak, to zwalniamy `mutex_a` i `mutex_b` i powtarzamy całą procedurę.

**Przykład 6.3.2.** [Zakleszczenie wątków] Przykład pokazuje zakleszczenie wątków. Skompilować i uruchomić program.

Kod źródłowy 24: Deadlock

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6
7 /* zainicjalizuj tablice muteksow */
8 pthread_mutex_t mutex_a = PTHREAD_MUTEX_INITIALIZER;
9 pthread_mutex_t mutex_b = PTHREAD_MUTEX_INITIALIZER;
10
11 /* Drukuje info o rezultacie operacji (sukces/blad).
12 * Jesli status!=0 (blad) konczy dzialanie programu */
13 int report(int status, const char* caller, const char* operation)
14 {
15     if(status != 0)
16     {

```

```

17     printf("%s: %s: error: %s\n", caller, operation, strerror(
18         status));
19     exit(EXIT_FAILURE);
20 }
21 printf("%s: %s: success!\n", caller, operation);
22 return status;
23 }

24 #define REPORT(cmd) report(cmd, __func__, #cmd)

25 /* Watek zamyka muteksy w kolejnosci mutex_a, mutex_b */
26 void* lock_forward(void* unused)
27 {
28     (void)unused; /* nie uzywamy 'arg' w funkcji */
29     REPORT(pthread_mutex_lock(&mutex_a));
30     sleep(1); /* dajemy czas watkowi backward na lock(mutex_b)
31     */
32     REPORT(pthread_mutex_lock(&mutex_b));

33     printf("lock_forward: zablokowalem mutex_a i mutex_b\n");
34
35     REPORT(pthread_mutex_unlock(&mutex_b));
36     REPORT(pthread_mutex_unlock(&mutex_a));
37     return NULL;
38 }

39 /* Watek zamyka muteksy w kolejnosci mutex_b, mutex_a */
40 void* lock_backward(void* unused)
41 {
42     (void)unused; /* nie uzywamy 'arg' w funkcji */
43     REPORT(pthread_mutex_lock(&mutex_b));
44     sleep(1); /* dajemy czas watkowi forward na lock(mutex_a)
45     */
46     REPORT(pthread_mutex_lock(&mutex_a));

47     printf("lock_backward: zablokowalem mutex_b i mutex_a\n");
48
49     REPORT(pthread_mutex_unlock(&mutex_a));
50     REPORT(pthread_mutex_unlock(&mutex_b));
51     return NULL;
52 }

53 int main(int argc, char* argv[])
54 {
55     pthread_t forward, backward;

56     REPORT(pthread_create(&backward, NULL, lock_backward, NULL));
57     REPORT(pthread_create(&forward, NULL, lock_forward, NULL));
58     printf("main: utworzono watki.\n");
59     printf("main: czekam na zakonczenie watkow.\n");
60     REPORT(pthread_join(backward, NULL));
61     REPORT(pthread_join(forward, NULL));
62     printf("main: koniec programu.\n");

63     return EXIT_SUCCESS;
64 }

```

**Przykład 6.3.3.** [Użycie `pthread_mutex_trylock()` dla uniknięcia zakleszczenia wątków] Przykład pokazuje użycie funkcji `pthread_mutex_trylock()`. Skompilować i uruchomić program.

**Kod źródłowy 25:** *Nieblokujące próbkowanie mutexów (`thread_mutex_trylock()`)*

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <errno.h>
7
8 /* zainicjalizuj tablice mutexow */
9 pthread_mutex_t mutex_a = PTHREAD_MUTEX_INITIALIZER;
10 pthread_mutex_t mutex_b = PTHREAD_MUTEX_INITIALIZER;
11
12 /* Drukuje info o rezultacie operacji (sukces/blad).
13 * Jesli status!=0 (blad) konczy dzialanie programu */
14 int report(int status, const char* caller, const char* operation)
15 {
16     if(status == 0)
17         printf("%s: %s: success!\n", caller, operation);
18     else if(status == EBUSY)
19         printf("%s: %s: busy!\n", caller, operation);
20     else
21     {
22         printf("%s: %s: error: %s\n", caller, operation, strerror(
23             status));
24         exit(EXIT_FAILURE);
25     }
26     return status;
27 }
28
#define REPORT(cmd) report(cmd, __func__, #cmd)
29
30 /* Watek zamyka mutexy w kolejnosci mutex_a, mutex_b */
31 void* lock_forward(void* unused)
32 {
33     (void)unused; /* nie uzywamy 'arg' w funkcji */
34     REPORT(pthread_mutex_lock(&mutex_a));
35     sleep(1); /* dajemy czas watkowi backward na lock(mutex_b)
36     */
37     if(REPORT(pthread_mutex_trylock(&mutex_b)) == 0) {
38         printf("lock_forward: zablokowalem mutex_a i mutex_b\n");
39         REPORT(pthread_mutex_unlock(&mutex_b));
40     }
41     REPORT(pthread_mutex_unlock(&mutex_a));
42     return NULL;
43 }
44
45 /* Watek zamyka mutexy w kolejnosci mutex_b, mutex_a */
46 void* lock_backward(void* unused)
47 {
48     (void)unused; /* nie uzywamy 'arg' w funkcji */
49     REPORT(pthread_mutex_lock(&mutex_b));
50     sleep(1); /* dajemy czas watkowi forward na lock(mutex_a)
51     */
52     if(REPORT(pthread_mutex_trylock(&mutex_a)) == 0) {
53         printf("lock_backward: zablokowalem mutex_b i mutex_a\n");
54         REPORT(pthread_mutex_unlock(&mutex_a));
55     }
56     REPORT(pthread_mutex_unlock(&mutex_b));
57     return NULL;
58 }
59
60 int main(int argc, char* argv[])
61 {
62     pthread_t forward, backward;

```

```

62     REPORT(pthread_create(&backward, NULL, lock_backward, NULL));
63     REPORT(pthread_create(&forward, NULL, lock_forward, NULL));
64     printf("main: utworzono wątki.\n");
65     printf("main: czekam na zakończenie wątków.\n");
66     REPORT(pthread_join(backward, NULL));
67     REPORT(pthread_join(forward, NULL));
68     printf("main: koniec programu.\n");
69
70     return EXIT_SUCCESS;
71 }
```

**Inwersja priorytetów.** Inwersja priorytetów jest problemem, który pojawia się w sytuacji, gdy występują conajmniej trzy wątki o różnych priorytetach. Istnieje wtedy konflikt pomiędzy wymaganiami nakładanymi przez metody synchronizacji (np. muteksy), a wymaganiami, dotyczącymi szeregowania wątków. Wykonywany jest wątek o niższym priorytecie, w sytuacji, gdy wątek o wyższym priorytecie został zablokowany na operacji synchronizacyjnej.

Przypuśćmy, że wątek 1, o najniższym priorytecie zajął mutex. Wątek 3, o najwyższym priorytecie, używa wspólnego zasobu z wątkiem 1 i próbuje zająć ten sam mutex poprzez wywołanie funkcji `pthread_mutex_lock()`. Operacja nie udaje się i wątek 3 zostaje zablokowany na operacji synchronizacyjnej, mimo, iż ma wyższy priorytet, niż wątek 1. Jeśli pojawi się wątek 2 (np. na skutek aktywności innych wątków), o pośrednim priorytecie i wywalczy wątek 1, to wątek 1 nie będzie mógł zwolnić mutexu. Wówczas, wątek 3 będzie zablokowany na operacji synchronizacyjnej, a wątek 2 będzie się wykonywał, mimo, iż ma mniejszy priorytet, niż wątek 3.

W systemie QNX Neutrino są stosowane dwie strategie zapobiegania inwersji priorytetów:

1. Dziedziczenie priorytetów polega na tymczasowym zwiększeniu priorytetu wątku, który posiada zasób (mutex) do najwyższego priorytetu wątku, który ubiega się o zasób. Priorytet wątku posiadającego zasób zostanie przywrócony do pierwotnego, po jego zwolnieniu. Schemat ten zapewnia, że wątki o najwyższych priorytetach będą zablokowane na najkrótszy możliwy czas, jednocześnie rozwiązuje problem inwersji priorytetów.
2. Zastosowanie protokołu wykorzystującego pułap priorytetów – polega na przydzieleniu zasobowi (mutexowi) priorytetu statycznego, większego od najwyższego priorytetu wątków, które o dany zasób będą konkurowały. Gdy jakiś wątek będzie próbował zająć zasób, to zostanie mu przydzielony tymczasowo priorytet związany z zasobem. Po zwolnieniu zasobu priorytet wątku wraca do wartości pierwotnej.

## 6.4 Zmienne warunkowe

### 6.4.1 Wstęp

Zmienne warunkowe dostarczają alternatywnego sposobu synchronizacji wątków. Podczas gdy muteksy synchronizują dostęp poszczególnych wątków do danych, zmienne warunkowe umożliwiają wątkom uzy-

skiwać informację (sygnalizować) o aktualnym stanie współdzielonych zasobów. Bez mechanizmu sygnalizowania zmiennych warunkowych, jeden z wątków musiałby nieustannie badać w pętli (potencjalnie w sekcji krytycznej) spełnienie pewnego warunku. Taki zabieg prowadziłby do niekorzystnego zużycia czasu procesora. Zmienna warunkowa pozwala osiągnąć ten sam cel bez użycia nieskończonej pętli. Zmienna warunkowa jest mechanizmem sygnalizowania informacji i powinna być skojarzona z mutessem, który z kolei chroni dostęp do wspólnego zasobu (danych). Trzy podstawowe operacje dokonywane na zmiennych są następujące:

- Oczekiwanie na zmienną warunkową (wait) – zwalnia muteks i zawiesza wątek aż do momentu zasygnalizowania zmiennej warunkowej.
- Sygnalizowanie zmiennej warunkowej (signal) – sygnalizuje zmienną warunkową i wznowia zawieszony wątek.
- Sygnalizowanie zmiennej warunkowej (broadcast) – sygnalizuje zmienną warunkową i wznowia wszystkie wątki zawieszone w kolejce zmiennej warunkowej.

Typowy scenariusz synchronizacji zmiennymi warunkowymi jest następujący:

- Zadeklarować i zainicjalizować zmienną warunkową.
- Zadeklarować i zainicjalizować stwarzyszony ze zmienną warunkową muteks.
- Zablokować muteks stwarzyszony ze zmienną warunkową.
- Kilka wątków zostaje zawieszonych w kolejce na zmiennej warunkowej i czekają aż jakiś inny wątek zasygnalizuje sytuację sprzyjającą dalszej pracy.
- Sygnalizacja zmiennej warunkowej.
- Jeden z zawieszonych wątków (pierwszy w kolejce) wznowia swoje działanie. Pozostałe wątki czekają nadal nad zmienną warunkową, aż do kolejnego zasygnalizowania zmiennej.
- Odblokować muteks stwarzyszony ze zmienną warunkową.
- Skasowanie mutesku.
- Skasowanie zmiennej warunkowej.

### 6.4.2 Tworzenie i kasowanie zmiennych warunkowych

Zmienna warunkowa jest reprezentowana w programie, jako zmienna typu `pthread_cond_t` i musi być zainicjalizowana przed użyciem. Podobnie jak w przypadku mutesków istnieją dwa sposoby inicjalizacji statycznie, bądź dynamicznie. `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`

```
1 int pthread_cond_init( pthread_cond_t* cond,
2                         pthread_condattr_t* attr );
```

gdzie

- `cond` – jest wskaźnikiem do zmiennej warunkowej `pthread_cond_t`.

- attr – jest albo NULL (atrybuty domyślne) albo wskaźnikiem do struktury definiującej atrybuty muteksu.

Zmienne warunkowe możemy inicjalizować nie tylko dynamicznie, używając struktury attr, ale również statycznie, poprzez zastosowanie makra PTHREAD\_COND\_INITIALIZER, które zapewnia przypisanie domyślnych wartości do zmiennej warunkowej. Do kasowania zmiennej warunkowej służy funkcja:

```
1 int pthread_cond_destroy( pthread_cond_t* cond );
```

Funkcja `pthread_cond_destroy` kasuje zasoby zajęte przez zmienną warunkową.

#### 6.4.3 Czekanie i sygnalizowanie zmiennej warunkowej

Funkcja `pthread_cond_wait()` pozwala na oczekiwanie na zmienną warunkową:

```
1 int pthread_cond_wait( pthread_cond_t* cond ,  
2                         pthread_mutex_t* mutex );
```

gdzie

- cond – jest wskaźnikiem do zmiennej warunkowej `pthread_cond_t`.
- mutex – stwarzyszony muteks.

Funkcja `pthread_cond_wait()` zawiesza wątek na zmiennej warunkowej cond, ustawia go w kolejce wątków oczekujących i jednocześnie odblokowuje stwarzyszony muteks. Funkcja ta powinna być wywoływana, wówczas kiedy muteks jest zajęty i automatycznie ponownie zamknie muteks po wyjściu z funkcji.

Istnieje odmiana funkcji, która pozwala czekać na zmiennej warunkowej przez określony czas:

```
1 int pthread_cond_timedwait(  
2                             pthread_cond_t* cond ,  
3                             pthread_mutex_t* mutex ,  
4                             const struct timespec* abstime );
```

gdzie

- abstime – jest wskaźnikiem do struktury `timespec`, która definiuje maksymalny absolutny czas blokowania wątku.

Wątek w przypadku funkcji `pthread_cond_wait()` lub `pthread_cond_timedwait()` jest zablokowany do czasu zasygnalizowania zmiennej warunkowej przez inny z wątków przez funkcję:

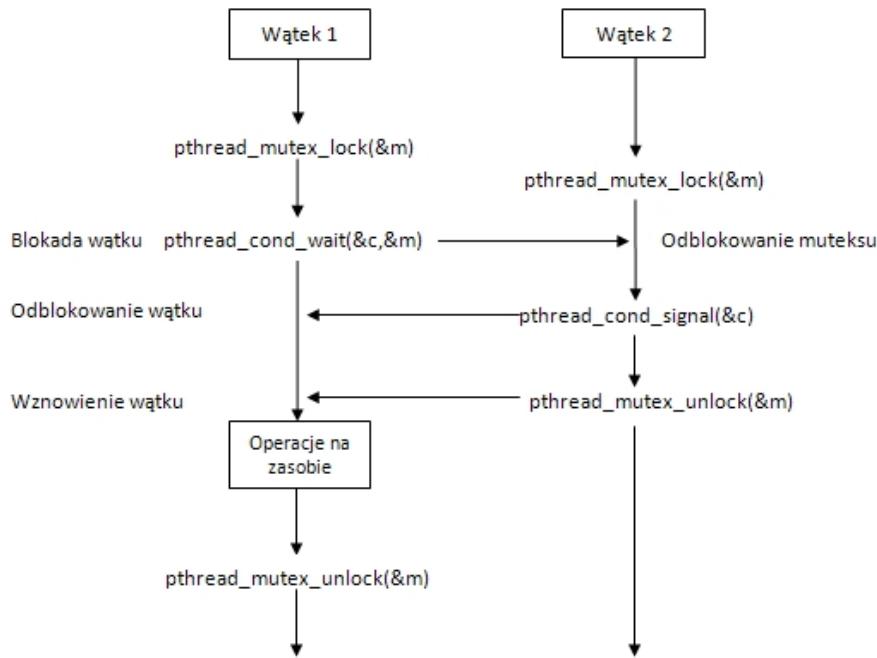
```
1 int pthread_cond_signal( pthread_cond_t* cond );
```

Funkcja odblokowuje wątek o najwyższym priorytecie, który czeka na zmiennej warunkowej cond. Gdy wątków o najwyższym priorytecie jest więcej, odblokowany będzie ten, który czeka najdłużej.

Odblokowanie wszystkich wątków, które czekają na zmiennej warunkowej można zrealizować poprzez wywołanie funkcji:

```
1 int pthread_cond_broadcast( pthread_cond_t* cond );
```

Wątki są odblokowane w zależności od wysokości priorytetów. W przypadku, gdy istnieje kilka wątków, o takich samych priorytetach, to wątki będą odblokowywane wg strategii FIFO. Działanie zmiennej warunkowej przedstawiono na rysunku 20.



Rysunek 20: Idea działania zmiennej warunkowej

**Przykład 6.4.1.** [Zmienne warunkowe] Program tworzy trzy wątki. Wątek główny pracuje jako zarządcza (manager) i co 2 sekundy deleguje do pracy jeden z dwóch wątków wykonawczych. Realizacja zadania zajmuje każdemu z wątków wykonawców 3 sekundy. Skompilować program, uruchomić i prześledzić jego współbieżną pracę.

Kod źródłowy 26: Zastosowanie zmiennej warunkowej

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4
5 /* Zmienna warunkowa */
6 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
7 /* Mutex */
8 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
9
10 /* Wątek – zarządcza */
11 void* manager(void *arg)
12 {
13     for(;;) /* Petla nieskończona */
14     {
15         sleep(2);

```

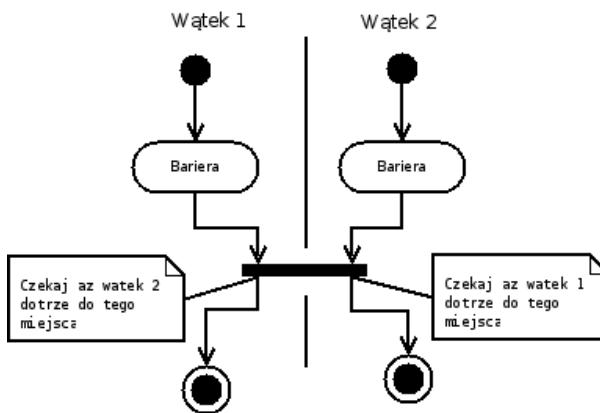
```

16     pthread_mutex_lock(&mutex);
17     printf("M: do pracy!\n");
18     pthread_cond_signal(&cond);
19     pthread_mutex_unlock(&mutex);
20 }
21     return NULL;
22 }
23
24 /* Watek - wykonawca */
25 void* worker(void* arg)
26 {
27     const char* prefix = (const char*)arg;
28     int i;
29
30     for(;;) /* Petla nieskonczona */
31     {
32         pthread_mutex_lock(&mutex);
33         pthread_cond_wait(&cond, &mutex); /* Czekaj na sygnal od
34             zarzadcy */
35         pthread_mutex_unlock(&mutex);
36
37         /* Wykonaj swoja ciezka prace */
38         printf("%s: pracuje ...\\n", prefix);
39         for(i=0; i<3; i++)
40         {
41             printf("%s:    |\\n", prefix);
42             sleep(1);
43         }
44         printf("%s: skonczylem\\n", prefix);
45     }
46     return NULL;
47 }
48
49 int main(int argc, char* argv[])
50 {
51     /* Identyfikatory watkow - wykonawcow */
52     pthread_t workers[2];
53     /* Atrybuty watku */
54     pthread_attr_t attr;
55
56     /* Inicjalizuj strukture z atrybutami*/
57     pthread_attr_init(&attr);
58
59     /* Tworzymy waki-wykonawcow */
60     pthread_create(&workers[0], &attr, worker, "\\t\\tW[1]");
61     pthread_create(&workers[1], &attr, worker, "\\t\\t\\t\\t\\t\\tW[2]");
62
63     /* Sam realizuj zadanie zarzadcy */
64     manager(NULL);
65
66     /* Czekaj na zakonczenie watkow - wykonawcow */
67     pthread_join(workers[0], NULL);
68     pthread_join(workers[1], NULL);
69
70     return 0;
71 }
```

## 6.5 Bariery

### 6.5.1 Tworzenie i kasowanie bariery

Synchronizacja przy użyciu bariery umożliwia koordynację pracy algorytmów. Bariera definiuje punkt w programie, do którego muszą dotrzeć wszystkie wątki zanim rozpoczną dalsze wykonanie. Liczba wątków potrzebnych do przełamania bariery jest definiowana w chwili tworzenia bariery.



Rysunek 21: Idea działania bariery

Bariera jest reprezentowana w programie, jako zmienna typu `pthread_barrier_t` i musi być zainicjalizowana przed użyciem. Inicjalizacja bariery odbywa się poprzez wywołanie następującej funkcji:

```

1 int pthread_barrier_init(
2                     pthread_barrier_t * barrier,
3                     const pthread_barrierattr_t * attr
4                     unsigned int count );
  
```

gdzie

- `barrier` – wskaźnik do obiektu typu `pthread_barrier_t`, który chcemy zainicjalizować.
- `attr` – jest albo `NULL` (atrybuty domyślne) albo wskaźnikiem do struktury definiującej atrybuty bariery `pthread_barrierattr_t`.
- `count` – liczba wątków, które muszą dojść do momentu wywołania funkcji `pthread_barrier_wait()`, aby wszystkie mogły kontynuować dalej swoje działanie.

Bariery możemy inicjalizować nie tylko dynamicznie, używając struktury `attr`, ale również statycznie, poprzez zastosowanie makra `PTHREAD_BARRIER_INITIALIZER(count)`, które zapewnia przypisanie domyślnych wartości do bariery oraz liczby `count`, omówionej poprzednio.

Do kasowania bariery służy funkcja:

```

1 int pthread_barrier_destroy(
2                     pthread_barrier_t * barrier );
  
```

Funkcja `pthread_barrier_destroy()` kasuje zasoby zajęte przez bariere.

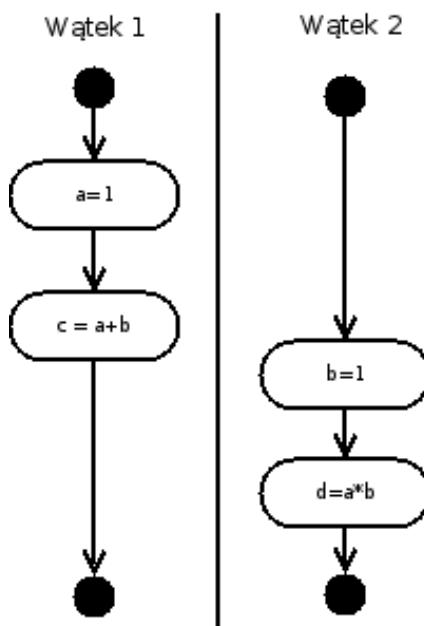
### 6.5.2 Czekanie na barierze

Funkcja `pthread_barrier_wait()` pozwala na synchronizację wątków na barierze:

```
1 int pthread_barrier_wait( pthread_barrier_t * barrier );
```

Wątki, które wywołują funkcję są blokowane, aż do momentu, gdy wymagana liczba wątków (count) wywołała funkcję `pthread_barrier_wait()`, wówczas bariera jest przełamywana, jednemu z wątków (nieokreślonemu) jest zwracana wartość `BARRIER_SERIAL_THREAD`, natomiast innym jest zwracana wartość zero. Po tym etapie stan bariery jest ustawiany do wartości po wywołaniu inicjalizacji bariery.

**Przykład 6.5.1.** [Brak synchronizacji] Program zawiera dwa wątki. Wątek 1 oblicza pewną wartość  $a$ , wątek 2 oblicza wartość  $b$ . Następnie wątek 1 powinien obliczyć  $c = a + b$ , a wątek 2 powinien obliczyć  $d = a \cdot b$  – zobacz rysunek 22. Przykład ilustruje brak synchronizacji pracy wątków.



Rysunek 22: Przykładowy scenariusz przy braku synchronizacji wątków

Kod źródłowy 27: Brak synchronizacji pracy wątków

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4
5 int a = -1234, b = 4321; /* Wartosci przed obliczeniami */
6
7 /* Procedura wykonywana przez watek */
8 void* start(void* arg)
9 {
10     int d;
11
12     printf("\t\t\tt2: Obliczam b ... \n");
13     sleep(2); /* Obliczenia zmudne, trwaja 2 sekundy */
14     b = 1;
15     printf("\t\t\tt2: b = %d\n", b);
16     printf("\t\t\tt2: Obliczam d = a * b\n");

```

```

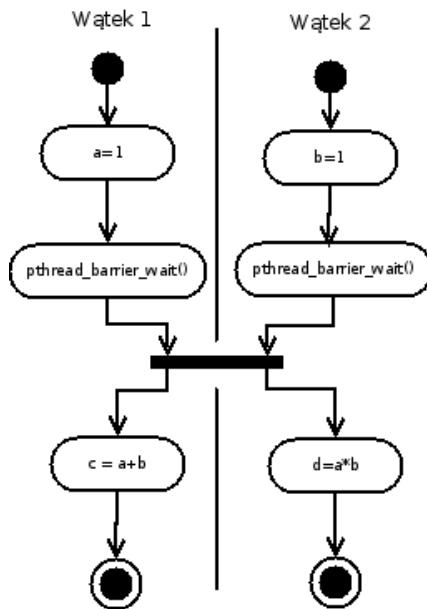
17     d = a * b;
18     printf("\t\t\tt2: d = %d\n", d);
19
20     /* Spodziewamy sie a = 1 i b = 1, wiec d = a * b = 1 * 1 = 1 */
21     if(d == 1)
22     {
23         printf("\t\t\tt2: Obliczenia przeprowadzone prawidlowo.\n");
24     }
25     else
26     {
27         printf("\t\t\tt2: Rety, nie wiem co sie stalo!\n");
28     }
29     return NULL;
30 }
31
32 int main(int argc, char* argv[])
33 {
34     /* identyfikator tworzonego wątku */
35     pthread_t start_id;
36     /* Atrybuty wątku */
37     pthread_attr_t attr;
38
39     int c; /* Dane lokalne wątku głównego */
40
41     /* Inicjalizuj strukturę z atrybutami*/
42     pthread_attr_init( &attr );
43
44     /* Utworz nowy wątek (nazwijmy go "start") */
45     pthread_create(&start_id, &attr, start, NULL);
46
47     printf("1: Obliczam a ... \n");
48     sleep(1); /* Obliczenia zmudne, trwaja 1 sekunde */
49     a = 1;
50     printf("1: a = %d\n", a);
51     printf("1: Obliczam c = a + b\n");
52     c = a + b;
53     printf("1: c = %d\n", c);
54
55     /* Spodziewamy się a = 1 i b = 1, wiec c = a + b = 1 + 1 = 2 */
56     if (c == 2)
57     {
58         printf("1: Obliczenia przeprowadzone prawidlowo.\n");
59     }
60     else
61     {
62         printf("1: Rety, nie wiem co sie stalo!\n");
63     }
64
65     /* Czekaj na zakonczenie wątku "start" */
66     pthread_join(start_id, NULL);
67
68     return 0;
69 }
```

**Przykład 6.5.2.** [Użycie bariery] Poniższy przykład jest modyfikacją poprzedniego. Obliczenia zsynchronizowano używając bariery – zobacz rysunek 23.

**Kod źródłowy 28:** *Synchronizacja przy pomocy bariery*

```

1
2 #include <stdio.h>
3 #include <pthread.h>
```



Rysunek 23: Idea działania bariery

```

4 #include <unistd.h>
5
6 int a = -1234, b = 4321; /* Wartosci przed obliczeniami */
7 /* Obiekt reprezentujacy bariere */
8 pthread_barrier_t barrier;
9 /* Inicjalizacja statyczna */
10 /* pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(2); */
11
12 /* Procedura wykonywana przez watek */
13 void* start(void* arg)
14 {
15     int d;
16
17     printf("\t\t\t2: Obliczam b ... \n");
18     sleep(2); /* Obliczenia zmudne, trwaja 2 sekundy */
19     b = 1;
20     printf("\t\t\t2: b = %d\n", b);
21     printf("\t\t\t2: Obliczam d = a * b\n");
22
23     pthread_barrier_wait(&barrier); /* Czekaj na barierze*/
24     d = a * b;
25     printf("\t\t\t2: d = %d\n", d);
26
27     /* Spodziewamy sie a=1 i b=1, wiec d = a * b = 1 * 1 = 1 */
28     if (d == 1)
29     {
30         printf("\t\t\t2: Obliczenia przeprowadzone prawidlowo.\n");
31     }
32     else
33     {
34         printf("\t\t\t2: Rety, nie wiem co sie stalo!\n");
35     }
36     return NULL;
37 }
38
39 int main(int argc, char* argv[])
40 {
41     pthread_t start_id; /* identyfikator tworzonego watku */
42     pthread_attr_t attr; /* Atrybuty watku */
43 }
```

```

44     int c; /* Dane lokalne wątku głównego */
45
46     /* Inicjalizuj strukturę z atrybutami*/
47     pthread_attr_init( &attr );
48
49     /* Inicjalizuj barierę na dwa wątki */
50     pthread_barrier_init(&barrier, NULL, 2);
51
52     /* Utworz nowy wątek (nazwijmy go "start") */
53     pthread_create(&start_id, &attr, start, NULL);
54
55     printf("1: Obliczam a ...\\n");
56     sleep(1); /* Obliczenia zmudne, trwaja 1 sekunde */
57     a = 1;
58     printf("1: a = %d\\n", a);
59     printf("1: Obliczam c = a + b\\n");
60
61     pthread_barrier_wait(&barrier);      /* Czekaj na barierze*/
62     c = a + b;
63     printf("1: c = %d\\n", c);
64
65     /* Spodziewamy się a=1 i b=1, więc c = a + b = 1 + 1 = 2 */
66     if (c == 2)
67     {
68         printf("1: Obliczenia przeprowadzone prawidłowo.\\n");
69     }
70     else
71     {
72         printf("1: Rety, nie wiem co się stało!\\n");
73     }
74
75     /* Czekaj na zakończenie wątku "start" */
76     pthread_join(start_id, NULL);
77
78     return 0;
79 }
```

## 6.6 Ćwiczenia

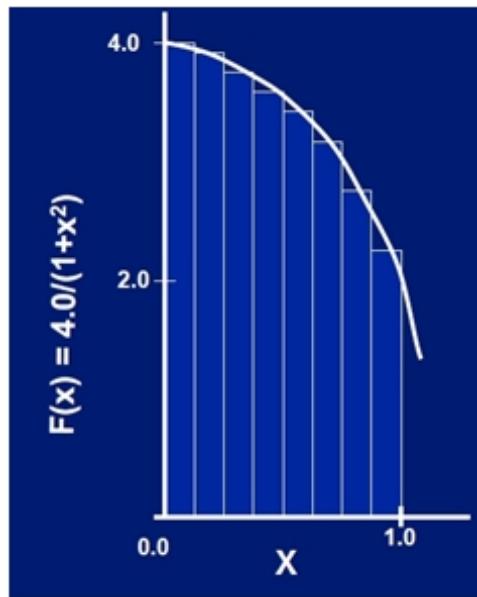
1. Zaproponować współbieżną (równoległą – gdy dysponujemy komputerem wieloprocesorowym) wersję programu do obliczania liczby  $\pi$  za pomocą wątków i mechanizmów synchronizacji. Matematycznie wiadomo, że:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

Całkę oznaczoną możemy aproksymować jako sumę:

$$\sum_{i=0}^N \frac{4}{1+x_i^2} \Delta x \approx \pi$$

Ilustrację graficzną aproksymacji liczby  $\pi$  można znaleźć na rysunku 24. Należy zadeklarować sumę, jak zmienną globalną oraz obliczać ją w sekcji krytycznej. Synchronizację dostępu do zmiennej globalnej zapewnić poprzez zastosowanie muteksów.



Rysunek 24: Idea aproksymacji liczby  $\pi$

**Kod źródłowy 29:** Kod źródłowy sekwencyjnej wersji do obliczania liczby  $\pi$

```

1 #include <stdio.h>
2
3 int num_steps = 100000;
4
5
6 int main ()
7 {
8     int i;
9     double x, pi, step, sum = 0.0;
10
11    step = 1.0 / (double) num_steps;
12    for (i=0; i < num_steps; i++)
13    {
14        x = (double)(i + 0.5) * step;
15        sum = sum + 4.0 / (1.0 + x * x);
16    }
17    pi = step * sum;
18
19    printf("PI = %.15f\n", pi);
20
21    return 0;
22 }
```

2. Należy przeczytać tekst dot. inwersji prioryterów w oprogramowaniu sondy marsjańskiej Pathfinder [http://research.microsoft.com/en-us/um/people/mbj/mars\\_pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/Mars_Pathfinder.html).



## Laboratorium 7

### Potoki nienazwane i nazwane

---

Współpracujące procesy i wątki mogą komunikować się ze sobą poprzez zastosowanie mechanizmów komunikacji międzyprocesowej IPC (ang. Inter-Process Communication). System operacyjny QNX Neutrino oferuje następujące mechanizmy komunikacji:

- potoki (łącza) nienazwane (pipes) i nazwane (kolejki FIFO),
- komunikaty (message passing) i impulsy (pulses) – specyficzne dla QNX,
- sygnały (signals),
- kolejki wiadomości (message queues),
- pamięć współdzielona (shared memory),
- gniazdka (sockets).

Niektóre z metod komunikacji są wprost implementowane w jądrze systemu operacyjnego QNX (komunikaty i sygnały), inne stanowią zewnętrzne procesy, bądź są częścią menadżera procesów. Projektant aplikacji może wybrać odpowiednią metodę komunikacji, po uwzględnieniu wymagań dotyczących działania aplikacji, takich jak:

- Czy wymagane jest stosowanie standardu POSIX?
- Jaka jest częstotliwość i wielkość wysyłanych wiadomości?
- Czy niezbędna jest komunikacja sieciowa?
- Czy wolno używać komunikacji blokującej?

W tym laboratorium omówimy jeden z podstawowych rodzajów komunikacji: potoki.

#### 7.1 Czym jest potok?

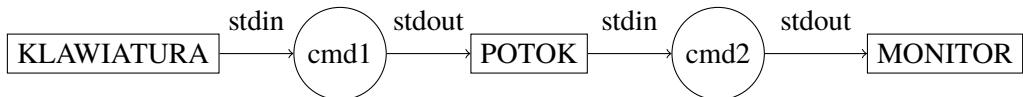
Mechanizm potoków wywodzi się z systemu UNIX i jest jedną z najstarszych metod komunikacji między procesami. Użytkownicy popularnych systemów operacyjnych na ogół spotykają się z potokami na poziomie wykonywania komend w wierszu poleceń. Kiedy wpisujemy sekwencję poleceń:

```
1 # cmd1 | cmd2
```

strumień wyjściowy procesu cmd1 jest przekazywany na wejście procesu cmd2. Przykładem użycia potoków może być wywołanie w wierszu poleceń następującej sekwencji:

```
1 # ls -l | wc -w
```

Zasada działania potoku przy „sprzęganiu” komend cmd1 i cmd2, została pokazana na rysunku 25. Symbole `stdin` i `stdout` oznaczają odpowiednio wejściowy i wyjściowy strumień danych procesu. W ty-



Rysunek 25: Użycie potoku do przekazywania strumieni danych

powej sytuacji `stdin` pobiera dane z klawiatury a `stdout` wyprowadza tekst na konsolę (monitor). Użycie potoku umożliwia skierowanie strumienia `stdout` jednego procesu do strumienia `stdin` innego procesu.

Potok jest nienazwanym plikiem, który stosuje się jako kanał komunikacyjny dla dwóch lub więcej współpracujących procesów. Jeden z procesów pisze do potoku, inny z niego czyta. Pojemność potoku jest ograniczona. Kiedy na przykład proces cmd1 pisze do potoku szybciej niż proces cmd2 czyta, potok może się napełnić i wtedy proces cmd1 jest blokowany, aż do momentu, gdy potok zostanie (przynajmniej częściowo) opróżniony przez cmd2. Podobnie, gdy proces cmd2 próbuje odebrać dane, które nie nadeszły, to zostaje zablokowany, aż do momentu, gdy będą one dostępne.

W trakcie laboratorium poznamy niskopoziomowe funkcje dostępu do plików, dowiemy się jaki jest wewnętrzny mechanizm przekazywania danych pomiędzy procesami na bazie potoków, a także w jaki sposób możemy zastosować potoki do komunikacji pomiędzy wieloma procesami.

## 7.2 Niskopoziomowe funkcje dostępu do plików

Do obsługi potoków z poziomu języka C można użyć niskopoziomowych funkcji dostępu do plików. Cztery podstawowe funkcje z tej rodziny przedstawiono w tabeli 32. Typowy schemat postępowania z pli-

Tabela 32: Ważniejsze funkcje dostępu do plików

<code>open()</code>	Otwarcie lub utworzenie pliku
<code>close()</code>	Zamknięcie pliku
<code>read()</code>	Odczyt z pliku
<code>write()</code>	Zapis do pliku

kiem sprowadza się do otwarcia pliku (`open()`), dokonania serii odczytów/zapisów (`read()`, `write()`) a następnie zamknięcia pliku (`close()`).

Do otwarcia pliku można użyć funkcji `open()`.

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4
5 int open(const char *path, int oflags, mode_t mode);

```

Funkcja `open()` przyjmuje następujące argumenty:

- `path` – ścieżka do pliku, który chcemy otworzyć,

- `oflags` – flagi określające m.in. tryb dostępu do otwieranego pliku (tabela 33),
- `mode` – jeśli w argumencie `oflags` wybierzemy flagę `O_CREAT`, musimy dostarczyć również zestaw parametrów specyfikujących prawa dostępu do nowo tworzonego pliku (tabela 34).

Funkcja `open()` zwraca deskryptor otwartego pliku (nieujemna liczba całkowita) lub `-1` w przypadku niepowodzenia.

Flagi, których można użyć w argumencie `oflags` przedstawiono w tabeli 33.

**Tabela 33:** Wybrane flagi, które można przekazać przez argument `oflags`

Flaga	Opis
<code>O_RDONLY</code>	otwarcie tylko do odczytu
<code>O_RDWR</code>	otwarcie dla odczytu i zapisu
<code>O_WRONLY</code>	otwarcie tylko do zapisu
<code>O_CREAT</code>	utworzenie pliku, gdy nie istnieje
<code>O_APPEND</code>	dopisywanie na końcu istniejącego pliku
<code>O_NONBLOCK</code>	zapis i odczyt będą działać nieblokująco

Flagi, których można użyć w argumencie `mode` przedstawiono w tabeli 34.

**Tabela 34:** Wybrane flagi, które można przekazać przez argument `mode`. Umożliwiają one zdefiniowanie praw dostępu do nowo tworzonego pliku (`O_CREAT`) odpowiednio dla właściciela pliku, członków wyróżnionej grupy oraz pozostałych użytkowników

User	Group	Others	Permission
<code>S_IRUSR</code>	<code>S_IRGRP</code>	<code>S_IROTH</code>	<code>r</code>
<code>S_IWUSR</code>	<code>S_IWGRP</code>	<code>S_IWOTH</code>	<code>w</code>
<code>S_IXUSR</code>	<code>S_IXGRP</code>	<code>S_IXOTH</code>	<code>x</code>
<code>S_IRWXU</code>	<code>S_IRWXG</code>	<code>S_IRWXO</code>	<code>rwx</code>

Aby zamknąć plik, używamy funkcji `close()`:

```

1 #include <unistd.h>
2
3 int close(int filedes);
```

Funkcja `close()` przyjmuje jeden argument:

- `filedes` – deskryptor pliku do zamknięcia, ten sam, który był zwrócony przez funkcję `open()`.

Odczyt danych z pliku realizujemy przez wywołanie funkcji `read()`

```

1 #include <unistd.h>
2
3 ssize_t read(int filedes, void *buf, size_t nbytes);
```

Argumenty przyjmowane przez funkcję `read()` to:

- `filedes` – deskryptor otwartego pliku, z którego chcemy czytać,

- `buf` – wskaźnik do bufora, do którego funkcja ma zapisać przeczytane dane,
- `nbyte` – liczba bajtów, którą chcemy przeczytać.

Funkcja `read()` odczytuje `nbyte` bajtów z pliku określonego przez deskryptor `filedes` i umieszcza dane w buforze wskazywanym przez `buf`. Gdy wywołanie funkcji się powiedzie, funkcja zwraca liczbę przeczytanych bajtów umieszczonych w buforze. W przypadku błędu, funkcja zwraca `-1`. Liczba bajtów zwracana przez funkcję może być mniejsza od `nbyte` w następujących przypadkach

- liczba bajtów w pliku jest mniejsza niż `nbyte`,
- działanie funkcji `read()` zostało przerwane przez sygnał,
- plik jest potokiem i posiada w danym momencie mniej bajtów danych niż `nbyte`.

Zapis danych do pliku realizujemy poprzez wywołanie funkcji `write()`

```
1 #include <unistd.h>
2
3 ssize_t read(int filedes, const void *buf, size_t nbyte);
```

Argumenty przyjmowane przez funkcję `write()` to:

- `filedes` – deskryptor otwartego pliku, do którego chcemy pisać,
- `buf` – wskaźnik do bufora w pamięci, z którego funkcja czerpie dane przeznaczone do zapisu,
- `nbyte` – liczba bajtów, którą chcemy zapisać.

Funkcja `write()` zapisuje `nbyte` bajtów do pliku określonego przez deskryptor `filedes` z bufora wskazywanego przez `buf`. Jeśli flaga `O_APPEND` była ustawiona podczas otwierania pliku, to dane są dopisywane na końcu istniejącego pliku. W przeciwnym przypadku zapis do pliku następuje od miejsca wskazanego przez bieżącą pozycję pliku. Po dokonaniu zapisu, wskaźnik bieżącej pozycji jest przesuwany o liczbę zapisanych bajtów. Liczba zapisanych bajtów może być mniejsza niż `nbyte` gdy:

- brakuje miejsca na nośniku,
- działanie funkcji `write()` zostało przerwane przez sygnał.

**Przykład 7.2.1.** [Utworzenie nowego pliku i pisanie do pliku] Uruchomić program, który tworzy nowy plik i zapisuje do niego dane. Jako argument wywołania podać nazwę pliku.

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <sys/stat.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6 #include <time.h>
7 #include <string.h>
8 #include <stdlib.h>
9
10 #define SIZE 5
11
12 char* timenow(void);
```

```

13 char* timenow(void)
14 {
15     /* Pobierz bieżący czas */
16     time_t now = time(NULL);
17     /* Przekonwertuj na string */
18     return asctime(localtime(&now));
19 }
20
21 int main (int argc, char* argv[])
22 {
23     int i, fd, offset = 0;
24     const char* mytime;
25     const char* path;
26     ssize_t bytes_written;
27     /* Atrybuty dostępu do pliku. */
28     mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;
29
30     if (argc < 2)
31     {
32         printf("Podaj nazwę pliku jako argument wywołania.\n");
33         return EXIT_FAILURE;
34     }
35     path = argv[1];
36     printf("Tworzę plik o nazwie: %s\n", path);
37     /* Utworz plik */
38     fd = open (path, O_WRONLY | O_CREAT, mode);
39     /* fd = creat( path, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
40      S_IROTH ); */
41
42     printf("Zapisuje dane do pliku");
43     /* Zapisz do pliku */
44     for(i = 0; i < SIZE; i++)
45     {
46         /* Pobierz czas */
47         mytime = timenow();
48         bytes_written = write (fd, mytime, strlen (mytime));
49         offset = offset + bytes_written;
50         printf(".\n");
51         sleep(1);
52     }
53
54     printf("Zapisano %d bajtów\n", offset);
55     printf("Zamknienie pliku.\n");
56     /* Zamknij plik. */
57     close(fd);
58     return EXIT_SUCCESS;
59 }
```

**Przykład 7.2.2.** [Kopiowanie plików] Uruchomić program. Jako argumenty wywołania podać nazwę pliku do skopiowania, nazwę pliku docelowego i liczbę bajtów do skopiowania.

```

1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <sys/stat.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6 #include <time.h>
7 #include <stdlib.h>
8 #include <memory.h>
9
10 int main (int argc, char* argv[])
11 {
```

```

12     int buf_size, offset = 0;
13     int fd1, fd2;
14     char* buffer;
15     ssize_t bytes_read, bytes_written;
16     /* Sciezki dostepu i nazwy plikow */
17     char const *path1, *path2;
18     /* Atrybuty dostepu do pliku. */
19     mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;
20
21     if (argc < 4)
22     {
23         printf("Podaj nazwy plikow jako argument wywolania oraz
24             rozmiar bufora.\n");
25         printf("Przyklad wywolania:\n\t./program plik1 plik2 150\n"
26             );
27         return EXIT_FAILURE;
28     }
29
30     path1 = argv[1];
31     path2 = argv[2];
32     buf_size = atoi(argv[3]);
33     buffer = (char*)malloc(buf_size * sizeof(char));
34
35     /* Otworz plik 1 */
36     printf("Otwieram plik o nazwie: %s\n", path1);
37     fd1 = open(path1, O_RDONLY, mode);
38     if (fd1 < 0)
39     {
40         printf("Nie moge otworzyc pliku.\n");
41         return EXIT_FAILURE;
42     }
43
44     /* Utworz plik 2 */
45     printf("Tworze plik o nazwie: %s\n", path2);
46     /* fd2 = open(path2, O_WRONLY | O_CREAT | O_TRUNC, mode); */
47     fd2 = creat(path2, mode);
48     if (fd2 < 0)
49     {
50         printf("Nie moge otworzyc pliku.\n");
51         return EXIT_FAILURE;
52     }
53
54     printf("Czytam dane z pliku %s i zapisuje do pliku %s\n",
55           path1, path2);
56     do
57     {
58         bytes_read = read(fd1, buffer, buf_size);
59         bytes_written = write(fd2, buffer, bytes_read);
60         offset = offset + bytes_written;
61     }
62     while (bytes_read == buf_size);
63
64     printf("Zapisano %d bajtow\n", offset);
65     printf("Zamknienie pliku.\n");
66     /* Zamknij plik. */
67     close(fd1);
68     close(fd2);
69     /* Zwolnij pamiec bufora */
70     free(buffer);
71     return EXIT_SUCCESS;
72 }
```

### 7.3 Potoki nienazwane

Komunikacja za pomocą potoków nienazwanych ma zastosowanie do procesów pozostających w relacji pokrewieństwa (np. proces potomny i macierzysty). Przy tworzeniu potoku nienazwanego tworzone są deskryptory plików reprezentujących potok, które są dziedziczone przez procesy potomne. Komunikacja jest jednokierunkowa. Deskryptator, który jest nieużywany przez dany proces, powinien być w przez ten proces zamknięty.

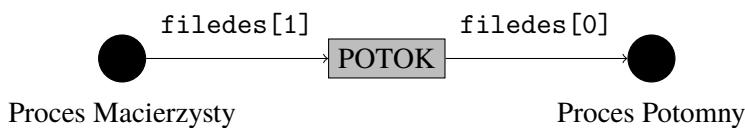
Utworzenie potoku wymaga wywołania funkcji `pipe()`:

```
1 #include <unistd.h>
2 int pipe(int filedes[2]);
```

Funkcja `pipe` przyjmuje jeden argument

- `filedes` – tablica przechowująca deskryptory plików do odczytu i zapisu (końcówki potoku).

Funkcja zwraca zero, gdy operacja się powiedzie, bądź  $-1$  w przeciwnym przypadku. Element `filedes[0]` jest deskryptorem „pliku” do odczytu z potoku, element `filedes[1]` jest deskryptorem „pliku” do zapisu. Operacje odczytu i zapisu można przeprowadzić przy użyciu opisanych wcześniej funkcji `read()` i `write()`. Zasadę działania potoku nienazwanego przedstawiono na rysunku 26.



Rysunek 26: Użycie potoku do przekazywania strumieni danych

System QNX zarządza potokami poprzez menadżera potoków. Menadżer potoków przydziela ograniczoną pamięć dla łącza danych. Bufor ten domyślnie ma rozmiar 5120 bajtów. Funkcją umożliwiającą pobranie informacji o zmiennych konfiguracyjnych dotyczących plików (w tym potoków) jest `fpathconf(int filedes, int name)`, gdzie `filedes` jest deskryptorem pliku, a `name` jest kodem numerycznym stanowiącym nazwę odpytywanej zmiennej. W naszym przypadku `name = _PC_PIPE_BUF`.

**Przykład 7.3.1.** [Potoki nienazwane w obrębie jednego procesu] Uruchomić program. Przetestować rozmiar bufora `BUFSIZE` równy 8. Zwrócić uwagę na fakt, że komunikaty czytane są w kolejności, w jakiej zostały zapisane (na bazie FIFO).

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <string.h>
5
6 #define BUFSIZE 32
7 #define MIN(a, b) ((a < b) ? a : b)
8
9 char *msg1 = "Hello!";
10 char *msg2 = "Wiadomosc z potoku...";
```

```

11 char *msg3 = "Koniec.";
12
13 int main(void)
14 {
15     int size1 = strlen(msg1);
16     int size2 = strlen(msg2);
17     int size3 = strlen(msg3);
18     int fildes[2];
19     int pipe_status, size;
20     char buffer[BUFSIZE] = { '\x0' };
21     ssize_t bytes_written, bytes_read;
22     ssize_t offset_w = 0;
23     ssize_t offset_r = 0;
24
25     printf("Tworze potok nienazwany...\n");
26     pipe_status = pipe(fildes);
27     /* Utworzenie potoku nie powiodlo sie */
28     if (pipe_status == -1)
29     {
30         perror("Nie moze utworzyc potoku\n");
31         return EXIT_FAILURE;
32     }
33     /* Utworzenie potoku powiodlo sie */
34     printf("Utworzylem potok.\n");
35     size = fpathconf(fildes[1], _PC_PIPE_BUF);
36     printf("Rozmiar bufora: %d B\n", size);
37     printf("Zapisuje dane do potoku...\n");
38     printf("\t%s\n", msg1);
39     bytes_written = write(fildes[1], msg1, size1);
40     offset_w += bytes_written;
41     printf("\t%s\n", msg2);
42     bytes_written = write(fildes[1], msg2, size2);
43     offset_w += bytes_written;
44     printf("\t%s\n", msg3);
45     bytes_written = write(fildes[1], msg3, size3);
46     offset_w += bytes_written;
47     printf("Zapisalem do potoku %d B\n", (int)offset_w);
48     printf("Odczytuje dane z potoku...\n");
49     bytes_read = read(fildes[0], buffer, MIN(size1, BUFSIZE));
50     offset_r += bytes_read;
51     printf("\t%.*s\n", MIN(size1, BUFSIZE), buffer);
52     bytes_read = read(fildes[0], buffer, MIN(size2, BUFSIZE));
53     offset_r += bytes_read;
54     printf("\t%.*s\n", MIN(size2, BUFSIZE), buffer);
55     bytes_read = read(fildes[0], buffer, MIN(size3, BUFSIZE));
56     offset_r += bytes_read;
57     printf("\t%.*s\n", MIN(size3, BUFSIZE), buffer);
58     printf("Odczytalem z potoku %d B\n", (int)offset_r);
59     printf("Koniec programu\n");
60     return EXIT_SUCCESS;
61 }
```

**Przykład 7.3.2.** [Potoki nienazwane w obrębie procesu macierzystego i potomnego (fork)] Uruchomić program, w którym proces macierzysty pisze do potoku, natomiast proces potomny odczytuje dane z potoku.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
```

```

6  #include <unistd.h>
7
8  #define BUFSIZE 32
9  #define MIN(a,b) ( (a < b) ? a : b)
10
11 char *msg1 = "Hello!";
12 char *msg2 = "Wiadomosc z potoku...";
13 char *msg3 = "Koniec.";
14
15 int main(void)
16 {
17     int size1 = strlen(msg1);
18     int size2 = strlen(msg2);
19     int size3 = strlen(msg3);
20     int fildes[2];
21     int pipe_status, size;
22     char buffer[BUFSIZE];
23     ssize_t bytes_written, bytes_read;
24     ssize_t offset_w = 0;
25     ssize_t offset_r = 0;
26     pid_t child_pid;
27     int status;
28
29     printf("M: Tworze potok nienazwany...\n");
30     pipe_status = pipe(fildes);
31     /* Utworzenie potoku nie powiodlo sie */
32     if (pipe_status == -1)
33     {
34         perror("M: Nie moge utworzyc potoku.\n");
35         return EXIT_FAILURE;
36     }
37
38     child_pid = fork();
39
40     if (child_pid == -1)
41     {
42         perror("M: Nie udalo sie utworzyc procesu potomnego...\n");
43     }
44     else if (child_pid == 0)
45     {
46         /* proces potomny */
47         printf("\tP: Proces potomny PID = %d.\n",getpid());
48         printf("\tP: Pracuje...\n");
49         printf("\tP: Zamykam deskryptor zapisu pliku.\n");
50         close(fildes[1]);
51         printf("\tP: Odczytuje dane z potoku...\n");
52         bytes_read = read(fildes[0],buffer,MIN(size1,BUFSIZE));
53         offset_r += bytes_read;
54         printf("\tP: %.*s\n",MIN(size1,BUFSIZE),buffer);
55         bytes_read = read(fildes[0],buffer,MIN(size2,BUFSIZE));
56         offset_r += bytes_read;
57         printf("\tP: %.*s\n",MIN(size2,BUFSIZE),buffer);
58         bytes_read = read(fildes[0],buffer,MIN(size3,BUFSIZE));
59         offset_r += bytes_read;
60         printf("\tP: %.*s\n",MIN(size3,BUFSIZE),buffer);
61         printf("\tP: Odczytalem z potoku %d B.\n", (int)offset_r);
62         printf("P: Zakonczenie procesu potomnego.\n");
63         exit(EXIT_SUCCESS);
64     }
65     else
66     {
67         /* Proces macierzysty */
68         printf("M: Proces macierzysty PID = %d.\n",getpid());
69         printf("M: Pracuje ...\n");
70         /* Utworzenie potoku powiodlo sie */

```

```

71     printf("M: Utworzylem potok.\n");
72     size = fpathconf(fildes[1], _PC_PIPE_BUF);
73     printf("M: Rozmiar bufora: %d B\n",size);
74     printf("M: Zamykam deskryptor odczytu pliku.\n");
75     close(fildes[0]);
76     printf("M: Zapisuje dane do potoku...\\n");
77     printf("M: %s\\n",msg1);
78     bytes_written = write(fildes[1],msg1,size1);
79     offset_w += bytes_written;
80     printf("M: %s\\n",msg2);
81     bytes_written = write(fildes[1],msg2,size2);
82     offset_w += bytes_written;
83     printf("M: %s\\n",msg3);
84     bytes_written = write(fildes[1],msg3,size3);
85     offset_w += bytes_written;
86     printf("M: Zapisalem do potoku %d B.\n", (int)offset_w);
87     printf("M: Czekam na potomny...\\n");
88     child_pid = wait(&status);
89     printf("M: Proces potomny o PID=%d zakoñczony; status: %d.\\n",
90           child_pid, WEXITSTATUS(status));
91   }
92 
93   printf("M: Koniec programu.\n");
94 }
```

Proces potomny może wywołać funkcję z rodziny `exec()`, która zastępuje bieżący proces innym procesem. Możliwa jest wtedy komunikacja pomiędzy procesem macierzystym, a nowo wywołanym procesem. Jednak wywołany proces nie dziedziczy od macierzystego deskryptorów plików, zatem musimy mu je przekazać jako parametry wywołania procesu. Sytuację tę ilustruje przykład 7.3.3.

**Przykład 7.3.3.** [Potoki nienazwane i funkcja `exec()`] Skompilować i uruchomić program **konsument.c** i **producent.c**.

```

1  /* konsument.c */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <string.h>
6
7  #define BUFSIZE 1024
8
9  int main(int argc, char *argv[])
10 {
11   int offset_r = 0;
12   int bytes_read, fildes;
13   char buffer[BUFSIZE];
14
15   printf("\\tK: Konsument...\\n");
16   if (argc < 2)
17   {
18     printf("\\tK: Podaj deskryptor pliku jako argument wywolania
19             .\\n");
20     return EXIT_FAILURE;
21   }
22   /* Pobierz deskryptor pliku */
23   sscanf(argv[1], "%d", &fildes);
24   printf("\\tK: Proces PID = %d.\\n", getpid());
25   printf("\\tK: Deskryptor pliku do odczytu: %d\\n", fildes);
```

```

25     printf("\tK: Odczytuje dane z potoku...\n");
26     memset(buffer, '\0', BUFSIZE);
27     bytes_read = read(fildes, buffer, BUFSIZE);
28     offset_r += bytes_read;
29     printf("\tK: *** %s ***\n", buffer);
30     printf("\tK: Odczytalem z potoku %d B.\n", offset_r);
31     printf("\tK: Zakonczenie procesu konsumenta.\n");
32
33     return EXIT_SUCCESS;
34 }
```

```

1  /* producent.c */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <unistd.h>
7
8  #define BUFSIZE 1024
9  #define MIN(a, b) ((a < b) ? a : b)
10
11 const char *msg = "Hello! Wiadomosc z potoku... Koniec.";
12
13 int main(void)
14 {
15     int msg_size = strlen(msg);
16     int fildes[2];
17     int pipe_status, size;
18     ssize_t bytes_written;
19     ssize_t offset_w = 0;
20     pid_t child_pid;
21     char fd[BUFSIZE];
22
23     printf("M: Tworze potok nienazwany...\n");
24     pipe_status = pipe(fildes);
25     if(pipe_status == -1)
26     {
27         /* Utworzenie potoku nie powiodlo sie */
28         perror("M: Nie moge utworzyc potoku.\n");
29         return EXIT_FAILURE;
30     }
31
32     child_pid = fork();
33
34     if (child_pid == -1)
35     {
36         perror("M: Nie udalo sie utworzyc procesu potomnego...\n");
37         exit(EXIT_FAILURE);
38     }
39     else if(child_pid == 0)
40     {
41         /* proces potomny */
42         printf("\tP: Proces potomny PID = %d.\n", getpid());
43         printf("\tP: Zamykam deskryptor zapisu pliku.\n");
44         close(fildes[1]);
45         printf("\tP: Deskryptor pliku do odczytu: %d\n", fildes[0])
46         ;
47         memset(fd, '\0', sizeof(fd));
48         sprintf(fd, "%d", fildes[0]);
49         execl("konsument", "konsument", fd, (char *)0);
50         printf("\tP: Blad uruchomienia procesu\n");
51         exit(EXIT_FAILURE);
52     }

```

```

52     else
53     {
54         /* Proces macierzysty */
55         printf("M: Proces macierzysty PID = %d.\n", getpid());
56         /* Utworzenie potoku powiodlo sie */
57         printf("M: Utworzylem potok.\n");
58         size = fpathconf(fildes[1], _PC_PIPE_BUF);
59         printf("M: Rozmiar bufora: %d B.\n", size);
60         printf("M: Zamykam deskryptor odczytu pliku.\n");
61         close(fildes[0]);
62         printf("M: Zapisuje dane do potoku...\\n");
63         printf("M: %s\\n", msg);
64         bytes_written = write(fildes[1], msg, msg_size);
65         offset_w += bytes_written;
66         printf("M: Zapisalem do potoku %d B.\n", (int)offset_w);
67     }
68
69     printf("M: Koniec programu.\n");
70     return EXIT_SUCCESS;
71 }
```

Do tej pory mieliśmy do czynienia z sytuacją, gdy komunikujące się procesy wymieniały informacje bez udziału standardowych strumieni. Często wygodnie jest połączyć deskryptory plików, uzyskane z funkcji `pipe()` ze standardowym wejściem 0 oraz standardowym wyjściem 1. Aby to zrobić, potrzebujemy dwóch spokrewnionych ze sobą funkcji, które umożliwiają duplikowanie deskryptorów plików:

```

1 #include <unistd.h>
2 int dup( int filedes );
3 int dup2( int filedes, int filedes2 );
```

gdzie

- `filedes` – deskryptor pliku, który chcemy zduplikować,
- `filedes2` – nowy deskryptor pliku, który chcemy przypisać do `filedes`.

Nowy deskryptor wskazuje na obiekt `filedes`, przekazany jako argument wywołania funkcji. W szczególności ma te same atrybuty i tryby dostępu, jak również wskazuje na tę samą pozycję w pliku. Funkcja `dup()` automatycznie generuje wartość nowego deskryptora pliku (duplikatu). Funkcja `dup2()` pozostawia to zadanie programistie, tj. programista dostarcza nowy deskryptor jako `filedes2`. Gdy deskryptor `filedes2` jest już otwarty, wskazywany przez niego plik zostanie najpierw zamknięty.

W dalszych przykładach skojarzymy końce potoku nienazwanego ze strumieniami `stdin` i `stdout`. Zabieg taki umożliwia realizację z poziomu języka C wywołania sekwencji procesów w postaci

```
1 # program1 | program2
```

**Przykład 7.3.4.** [Potoki nienazwane i przekierowania strumieni] Skompilować, uruchomić i przeanalizować program. Porównać wynik programu z wywołaniem: `ls -l | wc -w`

```

1 #include <stdlib.h>
2 #include <stdio.h>
```

```

3 #include <sys/wait.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 int join(char* const com1[], char* const com2[]);
8
9 int join(char* const com1[], char* const com2[])
10 {
11     int fd[2], status;
12
13     /* utworz proces Potomny */
14     switch(fork())
15     {
16         case -1: /* blad */
17             perror("Blad przy tworzeniu procesu potomnego 1.\n");
18             return EXIT_FAILURE;
19         case 0: /* Potomek */
20             break;
21         default: /* Rodzic */
22             wait(&status);
23             return status;
24     }
25     /* Potomek kontynuuje swoje dzialanie */
26     if (pipe(fd) == -1)
27     {
28         /* Utworzenie potoku nie powiodlo sie */
29         perror("M: Nie moze utworzyc potoku.\n");
30         return EXIT_FAILURE;
31     }
32     /* utworz proces Wnuczek */
33     switch(fork())
34     {
35         case -1: /* blad */
36             perror("Blad przy tworzeniu procesu potomnego 2.\n");
37             return EXIT_FAILURE;
38         case 0: /* Wnuczek pisze do stdout */
39             /* podlacz stdout z wnuczka do wejscia do potoku */
40             dup2(fd[1], 1);
41             /* zamknij deskryptory */
42             close(fd[0]);
43             execvp(com1[0], com1);
44             perror("Blad uruchomienia procesu\n");
45             exit(EXIT_FAILURE);
46         default: /* Potomek czyta z stdin */
47             /* podlacz stdin z potomka do wyjscia z potoku */
48             dup2(fd[0], 0);
49             /* zamknij deskryptory */
50             close(fd[1]);
51             execvp(com2[0], com2);
52             perror("Blad uruchomienia procesu\n");
53             exit(EXIT_FAILURE);
54     }
55 }
56
57
58 int main(void)
59 {
60     char* const one[] = {"/bin/ls", "-l", "./", (char *)0 };
61     char* const two[] = {"usr/bin/wc", "-w", (char *)0 };
62
63     join(one, two);
64
65     return EXIT_SUCCESS;
66 }
```

Mechanizm potoków nienazwanych jest dość skomplikowany. Programista musi dbać o tworzenie potoków (`pipe()`), zarządzanie procesami (`fork()`, `exec()`), czy przekierowanie strumieni standardowych (`dup2()`). Istnieją funkcje, które ułatwiają ten proces. Należą do nich `popen()` i `pclose()`.

```
1 FILE* popen( const char *command, const char *mode );
```

gdzie

- `command` – komenda, którą chcemy uruchomić jako proces potomny,
- `mode` – tryb otwierania potoku ("r", bądź "w").

Funkcja `popen()` uruchamia proces wskazany przez `command` i tworzy potok między procesem wywołującym a procesem `command`. W zależności od ustawionego trybu, zwracany wskaźnik na strukturę `FILE` może być użyty do odczytu bądź zapisu. W przypadku błędu, funkcja zwraca `NULL`.

Aby zamknąć potok stwarzyszony ze strumieniem `stream` używamy funkcji `pclose()`:

```
1 FILE* pclose( FILE* stream );
```

gdzie

- `stream` – wskaźnik na strumień otwarty wcześniej funkcją `popen()`.

Funkcja `pclose()` zamyka potok i jednocześnie czeka na zakończenie podprocesów utworzonych przez `popen()`. Funkcja zwraca status zakończenia, bądź `-1`, w przypadku błędu.

**Przykład 7.3.5.** [Przykład pisania do potoku nienazwanego `popen()`, `pclose()`] Skompilować, uruchomić program.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main(void)
4 {
5     /* Utworz potok, uruchom polecenie "wc -w", badz "sort" */
6     /* FILE* fp = popen ("wc -w", "w"); */
7     FILE* fp = popen("sort", "w");
8     if(fp != NULL)
9     {
10         /* pisz do potoku */
11         fprintf(fp, "To jest test.\n");
12         fprintf(fp, "Hello, world.\n");
13         fprintf(fp, "aaa.\n");
14         fprintf(fp, "Wspanialy program...\\n");
15         fprintf(fp, "Koniec programu.\n");
16         pclose(fp);
17     }
18
19     return 0;
20 }
```

## 7.4 Potoki nazwane

Potoki nazwane, określane również mianem plików FIFO, pozwalają na uniknięcie ograniczeń, którymi obarczone są potoki nienazwane. Pliki FIFO istnieją w przestrzeni nazw plików i mają atrybuty, prawa dostępu, właścicieli, takie, jak zwykłe pliki. Do obsługi potoków nazwanych używa się głównie funkcji plikowych, jedyna różnica, to inicjalizacja potoku:

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3
4 int mkfifo( const char* path, mode_t mode );
```

Funkcja `mkfifo()` przyjmuje następujące argumenty

- `path` – nazwa pliku FIFO,
- `mode` – prawa dostępu do pliku.

Funkcja `mkfifo()` tworzy nowy plik specjalny FIFO o nazwie podanej przez parametr `path`. Plik FIFO będzie miał uprawnienia określone przez argument `mode`. Funkcja zwraca 0 w przypadku powodzenia i -1 w przypadku błędu.

Otwieranie (`open()`) pliku FIFO do zapisu blokuje proces bieżący do czasu, gdy inny proces nie otworzy tego pliku do odczytu. Podobna sytuacja ma miejsce, gdy proces otwiera plik do odczytu – jest on blokowany dotąd, aż inny proces otworzy plik do zapisu. Możliwe jest także nieblokujące wywołanie funkcji `open()` dla pliku FIFO (należy użyć flagi `O_NONBLOCK`).

**Przykład 7.4.1.** [Tworzenie i używanie plików FIFO z wiersza poleceń]

```
1 # mkfifo /tmp/fifo
2 # ls -l /tmp/fifo
3 prw-rw-r-- 1 root root 0 Sep 12 12:24 fifo
```

Litera **p** w atrybutach (`prw-rw-r`) oznacza, że jest to potok nazwany (named pipe). Plik FIFO może być odczytywany i zapisywany za pomocą standardowych poleceń.

**Przykład 7.4.2.** [Zapis/odczyt pliku FIFO z wiersza poleceń] Używając dwu konsoli wpisać następujące polecenia:

Konsola 1:

```
1 # cat < /tmp/fifo
```

Konsola 2:

```
1 # cat > /tmp/fifo
```

Wpisać w konsoli 2 kilka linii tekstu i zakończyć pisanie (Ctrl+D). Przełączyć się na konsolę 1 i obejrzeć

rezultat. Zakończyć polecenie na konsoli 1 (Ctl+C). Usunąć plik FIFO jak zwykły plik.

Pliki FIFO można oczywiście obsługiwać z poziomu języka C. Poniższy przykład ilustruje sposób tworzenia, otwierania i zamykania pliku FIFO.

**Przykład 7.4.3.** [Tworzenie i używanie potoków nazwanych w C] Uruchomić program z argumentami wywołania O\_RDONLY lub O\_WRONLY. Zaobserwować zachowanie się procesu. Sprawdzić obecność pliku FIFO w systemie plików po wywołaniu programu. Uruchomić program w dwóch konsolach, w pierwszej z flagą O\_RDONLY, w drugiej z flagą O\_WRONLY. Sprawdzić co się stanie, gdy do flag dodamy opcję O\_NONBLOCK.

```

1  /* fifo.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <fcntl.h>
5  #include <string.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8
9  #define FIFO_FILE "/tmp/my_fifo"
10
11 int main(int argc, char *argv[])
12 {
13     mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
14         S_IWOTH;
15     int res, open_mode = 0;
16
17     if (argc < 2 )
18     {
19         printf("Jako argument wywolania podaj flagi:\nO_RDONLY ,\n"
20               "O_WRONLY lub O_RDWR.\n");
21         printf("Np.: ./fifo O_RDONLY\n");
22         printf("Np.: ./fifo O_WRONLY\n");
23         exit(EXIT_FAILURE);
24     }
25
26     if (strncmp(argv[1], "O_RDONLY", 8) == 0)
27         open_mode = O_RDONLY;
28     else if (strncmp(argv[1], "O_WRONLY", 8) == 0)
29         open_mode = O_WRONLY;
30     else if (strncmp(argv[1], "O_RDWR", 6) == 0)
31         open_mode = O_RDWR;
32     else
33     {
34         fprintf(stderr, "error: nieprawidlowa wartosc argv[1]: %s\n"
35                 ", argv[1]);
36         exit(EXIT_FAILURE);
37     }
38     /* open_mode |= O_NONBLOCK; */
39
40     /* Jesli plik FIFO nie istnieje, to go utworz */
41     if (access(FIFO_FILE, F_OK) == -1)
42     {
43         res = mkfifo(FIFO_FILE, mode);
44         /* Nie udalo sie utworzyc pliku FIFO */
45         if (res == -1)
46         {
47             fprintf(stderr, "error: nie udalo sie utworzyc pliku "
48                     FIFO.\n");
49             exit(EXIT_FAILURE);
50     }
51 }
```

```

46         }
47     }
48     printf("Proces %d: otwiera plik FIFO w trybie %s.\n", getpid(), argv[1]);
49     res = open(FIFO_FILE, open_mode);
50     printf("Proces %d: deskryptor pliku: %d.\n", getpid(), res);
51     printf("Proces %d: pracuje...\n", getpid());
52     sleep(5);
53     if (res != -1)
54         close(res);
55     printf("Proces %d: zakończył swoje działanie.\n", getpid());
56
57     return EXIT_SUCCESS;
58 }
```

Następny przykład jest ilustracją zastosowania potoków nazwanych do implementacji bardzo prostego jednokierunkowego komunikatora. W przykładzie mamy proces-serwer, który odbiera komunikaty od klienta i wyprowadza je na standardowe wyjście. Programy tworzą szkielet prostego systemu buforowania.

**Przykład 7.4.4.** [Prosty system buforowania na bazie potoków nazwanych] Skompilować, uruchomić i przeanalizować działanie klienta i serwera.

```

1  /* klient.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/types.h>
7  #include <unistd.h>
8
9 #define FIFO_FILE "/tmp/my_fifo_com"
10#define BUFSIZE 256
11
12 int main(void)
13 {
14     int fd;
15     int bytes_written;
16     char buffer[BUFSIZE] = { '\x0' };
17     pid_t pid = getpid();
18
19     printf("Klient (%d): otwieram plik FIFO w trybie %s.\n", pid, "O_WRONLY");
20     fd = open(FIFO_FILE, O_WRONLY);
21
22     if(fd == -1)
23     {
24         printf("Klient (%d): nie udało się otworzyć pliku FIFO.\n", pid);
25         exit(EXIT_FAILURE);
26     }
27     printf("Klient (%d): deskryptor pliku: %d.\n", pid, fd);
28     printf("Klient (%d): wpisz komunikat\n", pid);
29
30     while(feof(stdin) == 0)
31     {
32         printf("Klient (%d): ", pid);
33         fgets(buffer, BUFSIZE - 1, stdin);
34         /* fprintf(stdout, "K: %s\n", buffer); */
```

```

35     bytes_written = write(fd, buffer, strlen(buffer));
36     if (bytes_written == -1)
37     {
38         printf("Klient (%d): nie udalo sie zapisac do pliku
39             FIFO.\n", pid);
40         exit(EXIT_FAILURE);
41     }
42
43     return EXIT_SUCCESS;
44 }
```

```

1  /* serwer.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <fcntl.h>
5  #include <string.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <unistd.h>
9
10 #define FIFO_FILE "/tmp/my-fifo.com"
11 #define BUFSIZE 256
12
13 int main(void)
14 {
15     int fd, res;
16     int bytes_read;
17     char buffer[BUFSIZE] = { '\x0' };
18     pid_t pid = getpid();
19
20     /* Jesli plik FIFO nie istnieje, to utworz */
21     if (access(FIFO_FILE, F_OK) == -1)
22     {
23         printf("Serwer (%d): Tworze plik FIFO.\n", pid);
24         res = mkfifo(FIFO_FILE, S_IRUSR | S_IWUSR | S_IRGRP |
25             S_IWGRP | S_IROTH | S_IWOTH);
26         /* Nie udalo sie utworzyc pliku FIFO */
27         if (res == -1)
28         {
29             printf("Serwer (%d): Nie udalo sie utworzyc pliku FIFO
30                 .\n", pid);
31             exit(EXIT_FAILURE);
32         }
33     }
34     printf("Serwer (%d): otwieram plik FIFO w trybie %s.\n", pid, "O_RDONLY");
35     fd = open(FIFO_FILE, O_RDONLY);
36     if (fd == -1)
37     {
38         printf("Serwer (%d): nie udalo sie otworzyc pliku FIFO.\n",
39             pid);
40         exit(EXIT_FAILURE);
41     }
42
43     do
44     {
45         printf("\tSerwer (%d): ", pid);
46         bytes_read = read(fd, buffer, sizeof(buffer) - 1);
47         if (bytes_read == -1)
```

```

48         {
49             printf("Serwer (%d): nie udalo sie odczytac z pliku
50                 FIFO.\n", pid);
51             exit(EXIT_FAILURE);
52         } /* Przetworzenie wiadomosci */
53         printf("%s", buffer);
54     }
55     while(bytes_read > 0);
56
57     printf("Serwer (%d): koniec strumienia.\n", pid);
58
59     return EXIT_SUCCESS;
60 }
```

## 7.5 Ćwiczenia

1. Napisać program, który otwiera plik podany jako argument wywołania i zlicza liczbę znaków występujących w pliku. Przetestować program na małym pliku z kilkoma liniami tekstu.
2. Napisać następujące programy:
  - (a) Program tylko pisze do potoku w pętli nieskończonej po jednym znaku. Dane nie są nigdy odczytywane z potoku. Pokazać całkowitą liczbę bajtów zapisaną do potoku. Co się dzieje ze stanem procesu?
  - (b) Program tylko odczytuje z potoku w pętli nieskończonej po jednym znaku. Dane nie są nigdy zapisywane do potoku. Co się dzieje ze stanem procesu?
  - (c) O zachowaniu się procesów w sytuacjach a. i b. decyduje flaga `O_NONBLOCK`, którą można kontrolować za pomocą funkcji `fcntl()`. Domyślnie flaga jest wyzerowana, informację o jej stanie można pobrać przez wywołanie funkcji: `fcntl(fd, F_GETFL, O_NONBLOCK)`; Stan procesów w przypadku a. i b. można zmienić poprzez ustawienie flagi `O_NONBLOCK` wywołaniem: `fcntl(fd, F_SETFL, O_NONBLOCK)`; Jak się zachowują oba programy ?
3. Napisać dwa programy p1 i p2. Pierwszy z nich ma wypisywać na standardowe wyjście napis (funkcja `fprintf (stdout, "Tekst1")`). Drugi program ma czytać ze standardowego wejścia (funkcja `fscanf (stdin, &zmienna)`) i wyświetlać przetworzony tekst. Wywołać i przetestować programy pojedynczo. Wywołać programy przez potok (p1 — p2). Zastosować funkcję `join()` do połączenia obu programów.
4. Programy **klient.c** i **serwer.c** tworzą prosty system wymiany komunikatów. Zaimplementować program, w którym klient wysyła do serwera nazwę pliku do utworzenia oraz dane, a serwer tworzy plik i zapisuje do niego przekazane informacje.

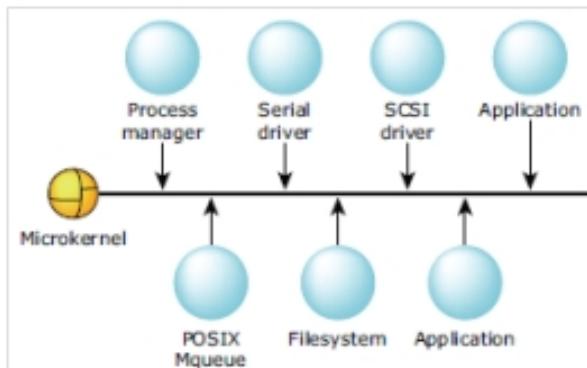


## Laboratorium 8

### Mechanizmy komunikacji w systemie QNX Neutrino - komunikaty

#### 8.1 Wprowadzenie

W systemie QNX Neutrino istnieje wiele różnych metod komunikacji międzyprocesorowej. Przesyłanie komunikatów (message passing) jest podstawową formą IPC (interprocess communication) w systemie QNX. Ten unikalny dla Neutrino mechanizm został zaimplementowany wprost w mikrojądrze systemu operacyjnego i stanowi prymityw do komunikacji dwukierunkowej modułów (np. menadżera procesów, modułu systemu plików) z mikrojądrzem. Pozwala to na odseparowanie procesów od mikrojądra. W przypadku awarii oprogramowania, procesy modułów nie mają bezpośredniego wpływu na działanie mikrojądra. Taka architektura (patrz rysunek 27) zapewnia wysoką wydajność, konfigurowalność i skalowalność systemu, dostosowaną do ograniczeń nakładanych na system. Inne mechanizmy IPC (np. omawiane wcześniej potoki i pliki FIFO) są nadbudową tej formy komunikacji międzyprocesorowej.

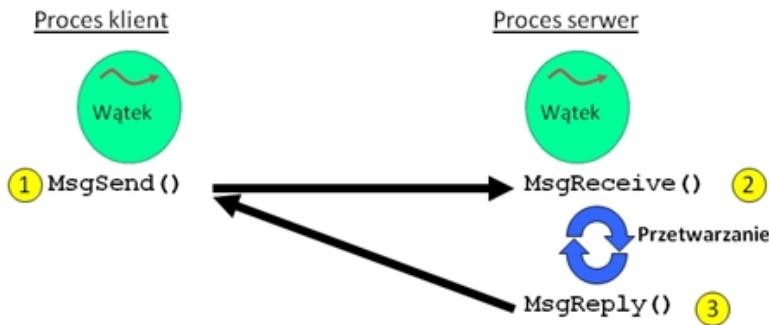


Rysunek 27: Modularna architektura QNX Neutrino

W zastosowaniach, często spotyka się aplikacje oparte o model klient-serwer, jak na rysunku 28. Początkowo serwer czeka na wiadomości od procesów-klientów. Procesy klienta wysyłają (1) dane do serwera, a następnie zostają zablokowane w oczekiwaniu na odpowiedź. W tym czasie serwer otrzymuje wiadomości (2), przetwarza je i odpowiada klientom (3), które kontynuują swoje działanie.

Ten model wymiany informacji może być zrealizowany poprzez przesyłanie komunikatów (message passing), na który składają się następujące mechanizmy:

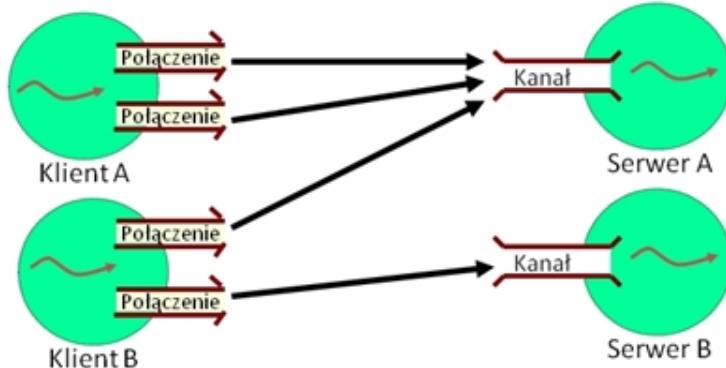
- Tworzenie kanałów i połączeń.
- Wysyłanie, odbieranie i potwierdzanie komunikatów.
- Impulsy (krótkie, nieblokujące nadawcy komunikaty).
- Przesyłanie komunikatów poprzez sieć oraz rejestrowanie nazw procesów, w ramach usługi GNS.



Rysunek 28: Model aplikacji typu klient-serwer

## 8.2 Tworzenie kanałów i połączeń

W systemie QNX Neutrino przesyłanie komunikatów nie następuje bezpośrednio pomiędzy procesami, czy wątkami. Medium pośredniczącym w przekazywaniu komunikatów jest kanał komunikacyjny. Sytuację pokazano na rysunku 29. Proces serwer, który odbiera komunikaty, tworzy kanał i oczekuje w nim wiadomości od klientów. Prosesy klient dołączają się do kanału poprzez połączenia i przez nie wysyła do serwera komunikaty. Klient może mieć wiele połączeń, do różnych serwerów, natomiast serwer używa wyłącznie jednego kanału komunikacyjnego do odbierania wiadomości od klientów.



Rysunek 29: Kanały i połączenia

Funkcje służące obsłudze kanałów i połączeń przedstawiono w tabeli 35.

Tabela 35: Obsługa kanałów i połączeń

Funkcja	Opis
<code>ChannelCreate()</code>	Tworzenie kanału do odbioru wiadomości
<code>ChannelDestroy()</code>	Kasowanie kanału
<code>ChannelAttach()</code>	Tworzenie połączenia do wysyłania wiadomości
<code>ChannelDetach()</code>	Kasowanie połączenia

Pierwszą czynnością, którą należy wykonać przy implementacji serwera jest utworzenie kanału ko-

munikacyjnego. Kanał komunikacyjny jest własnością procesu, który go utworzył. Wątki, które chcą się skomunikować z danym kanałem mogą być zawarte w tym samym procesie, mogą należeć do innego procesu w obrębie tego samego węzła, bądź innego węzła w sieci. Każdorazowo jednak muszą one utworzyć z kanałem połączenie. Kanał komunikacyjny tworzymy za pomocą funkcji:

```
1 #include <sys/neutrino.h>
2 int ChannelCreate( unsigned flags );
3 int ChannelCreate_r( unsigned flags );
```

gdzie `flags` stanowią opcje, które mogą być używane do zmiany własności kanału (tabela 36).

**Tabela 36:** Opcje tworzenia kanałów

Opcje	Opis
<code>_NTO_CHF_COID_DISCONNECT</code>	Dostarcz impuls, kiedy dowolne połączenie do kanału jest zamknięte.
<code>_NTO_CHF_DISCONNECT</code>	Dostarcz impuls, kiedy wszystkie połączenia do kanału są zamknięte.
<code>_NTO_CHF_FIXED_PRIORITY</code>	Nie stosuje dziedziczenia priorytetów.
<code>_NTO_CHF_NET_MSG</code>	Zarezerwowane dla menadżera zasobów (ionet)
<code>_NTO_CHF_REPLY_LEN</code>	Długość odpowiedzi ma być zawarta w strukturze <code>_msg_info</code> , którą wypełnia funkcja <code>MsgReceivev()</code> .
<code>_NTO_CHF_SENDER_LEN</code>	Długość komunikatu ma być zawarta w strukturze <code>_msg_info</code> , którą wypełnia funkcja <code>MsgReceivev()</code> .
<code>_NTO_CHF_THREAD_DEATH</code>	Dostarcz do kanału impuls, gdy zakończy się dowolny wątek posiadający kanał.
<code>_NTO_CHF_UNBLOCK</code>	Dostarcz do kanału impuls, gdy wątek wysyłający będący w stanie <code>REPLY_BLOCKED</code> odblokuje się przed wysłaniem mu odpowiedzi funkcją <code>MsgReply()</code> .

Funkcje `ChannelCreate()` i `ChannelCreate_r()` tworzą kanał komunikacyjny, który może być użyty do odbierania komunikatów lub impulsów. Zwracają identyfikator kanału CHID (channel ID) lub `-1` i ustawiają kod błędu. Komunikaty i impulsy są odbierane w kanale komunikacyjnym i ustawiane w kolejkę, zgodnie z wartościami priorytetów. Domyślnie, kiedy proces (wątek) odbiera komunikat z kanału, jego priorytet jest ustawiany w ten sposób, aby był równy priorytetowi procesu (wątku) wysyłającego komunikat. Metoda ta, zwana dziedziczeniem priorytetów (priority inheritance) zapobiega inwersji priorytów. Po otrzymaniu komunikatu, wątek może odłączyć się od kanału poprzez wywołanie `MsgReceive()` z kanałem `-1`. Dziedziczenie priorytetów można wyłączyć, poprzez ustawienie flagi `_NTO_CHF_FIXED_PRIORITY`. Różnica między funkcjami `ChannelCreate()` i `ChannelCreate_r()` polega na tym, że pierwsza z nich ustawia globalną zmienną `errno`, umożliwiającą odczytanie kodu błędu, a także samego opisu błędu przez funkcję `char* strerror( int errno );`.

Aby skasować kanał należy wywołać funkcję:

```
1 #include <sys/neutrino.h>
2 int ChannelDestroy( int chid );
```

```
3 int ChannelDestroy_r( int chid );
```

gdzie `chid` - jest numerem kanału zwróconym przez funkcję `ChannelCreate()`, bądź `ChannelCreate_r()`.

Kiedy kanał jest kasowany, to wszystkie wątki, które są w stanie zablokowany na operacjach `MsgSend()` i `MsgReceive()` zostaną odblokowane. Odbieranie wiadomości, bądź impulsów z kanałów po jego zamknięciu zakończy się niepowodzeniem.

**Przykład 8.2.1.** [Utworzenie kanału komunikacyjnego] Skompilować, zbudować oraz uruchomić przykład.

### Kod źródłowy 30: Kanał komunikacyjny

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/neutrino.h>
4 #include <string.h>
5 #include <errno.h>
6
7 int main(int argc, char *argv[])
8 {
9     int chid;
10
11     chid = ChannelCreate(0);
12     if (chid == -1)
13     {
14         printf("Nie moze utworzyc kanalu: %s\n", strerror(errno));
15         return EXIT_FAILURE;
16     }
17     /* ... */
18     printf("Utworzylem kanał o nr CHID=%d\n", chid);
19     ChannelDestroy(chid);
20     return EXIT_SUCCESS;
21 }
```

Ustanowienie połączenia między procesem (wątkiem), a kanałem wymaga wyłania funkcji `ConnectAttach()`:

```
1 #include <sys/neutrino.h>
2 int ConnectAttach( uint32_t nd,
3                     pid_t pid,
4                     int chid,
5                     unsigned index,
6                     int flags );
7
8 int ConnectAttach_r( uint32_t nd,
9                      pid_t pid,
10                     int chid,
11                     unsigned index,
12                     int flags );
```

gdzie

`nd` - jest numerem węzła, na którym uruchomiony jest proces posiadający kanał lub 0, bądź `ND_LOCAL_NODE`, w przypadku węzła bieżącego.

`pid` - numer PID procesu serwera, tzn. zawierającego kanał komunikacyjny.

`chid` - jest numerem kanału zwróconym przez funkcję `ChannelCreate()`.

`index` - najmniejszy akceptowalny numer połączenia; przekazanie flagi `_NTO_SIDE_CHANNEL` powoduje wybranie numeru połączenia z innego zakresu, niż deskryptory plików. Rekomenduje się używanie tej flagi podczas wywołania funkcji.

`flags` - flagi modyfikujące działanie; jeśli zawierają `_NTO_COF_CLOEXEC`, to połączenie jest zamknięte, kiedy proces wywołuje funkcję z rodziny `exec()`.

Funkcja zwraca identyfikator połączenia `COID` (connection ID), bądź `-1` w przypadku niepowodzenia. Podobnie, jak poprzednio, funkcje `ConnectAttach()` i `ConnectAttach_r()` różnią się ustawianiem zmiennej globalnej `errno`.

Zamykanie połączenia realizujemy funkcjami:

```
1 #include <sys/neutrino.h>
2 int ConnectDetach( int coid );
3 int ConnectDetach_r( int coid );
```

gdzie

`coid` - jest numerem kanału, który chcemy skasować.

Funkcja zwraca wartość dodatnią, w przypadku sukcesu i `-1`, w przypadku niepowodzenia. W przypadku wywołanai funkcji, wątki, które są zablokowane na połączeniu, zostaną odblokowane, a funkcja `MsgSend()` zwróci kod błędu.

**Przykład 8.2.2.** [Utworzenie połączenia] Skompilować, zbudować oraz uruchomić przykład. Dlaczego wynikiem działania programu jest błąd?

#### Kod źródłowy 31: Połączenie

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/neutrino.h>
4 #include <string.h>
5 #include <errno.h>
6
7 int main(int argc, char *argv[])
8 {
9     int coid, nd, chid;
10    pid_t pid;
11
12    nd = 0;
13    coid = ConnectAttach(nd, pid, chid, _NTO_SIDE_CHANNEL, 0);
14    if (coid == -1)
15    {
16        printf("Nie moge utworzyc polaczenia: %s\n", strerror(errno))
17    }
18}
```

```

17     return EXIT_FAILURE;
18 }
19 /* ... */
20 printf("Utworzylem polaczenie o nr connect=%d\n", coid);
21 ConnectDetach(coid);
22 return EXIT_SUCCESS;
23 }

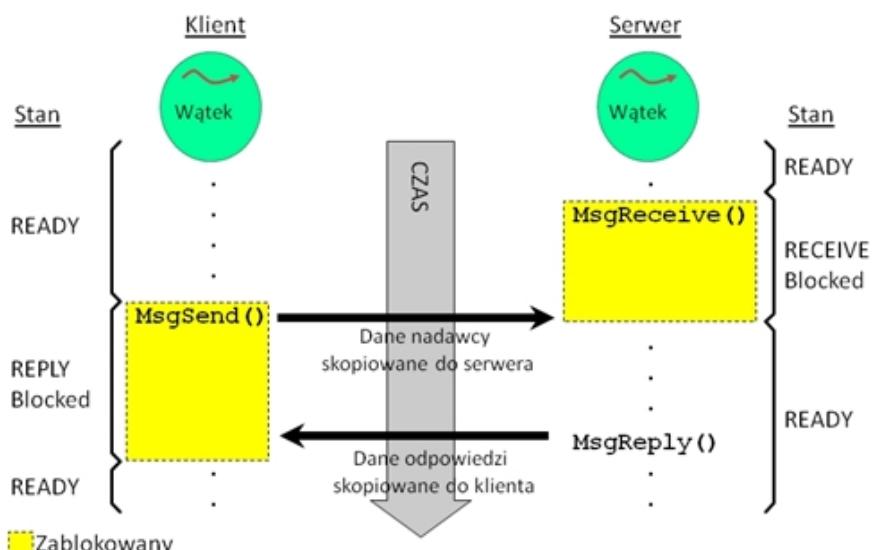
```

### 8.3 Wysyłanie, odbieranie i odpowiadanie na komunikaty

Komunikacja pomiędzy klientem a serwerem za pomocą przesyłania komunikatów (spotkań) jest komunikacją dwukierunkową, synchroniczną i składa się z trzech etapów: wysyłanie komunikatu przez klienta do serwera, odbiór komunikatu przez serwer i przesłanie przez serwer odpowiedzi do klienta. System operacyjny QNX Neutrino dostarcza całej grupy funkcji, służących do obsługi opisanych trzech etapów komunikacji. Można jednak napisać w pełni funkcjonalne aplikacje typu klient-serwer posługując się minimalnym zestawem funkcji takich, jak: `ChannelCreate()`, `ChannelDestroy()`, `ConnectAttach()`, `ConnectDetach()`, `MsgReply()`, `MsgSend()`, `MsgReceive()`.

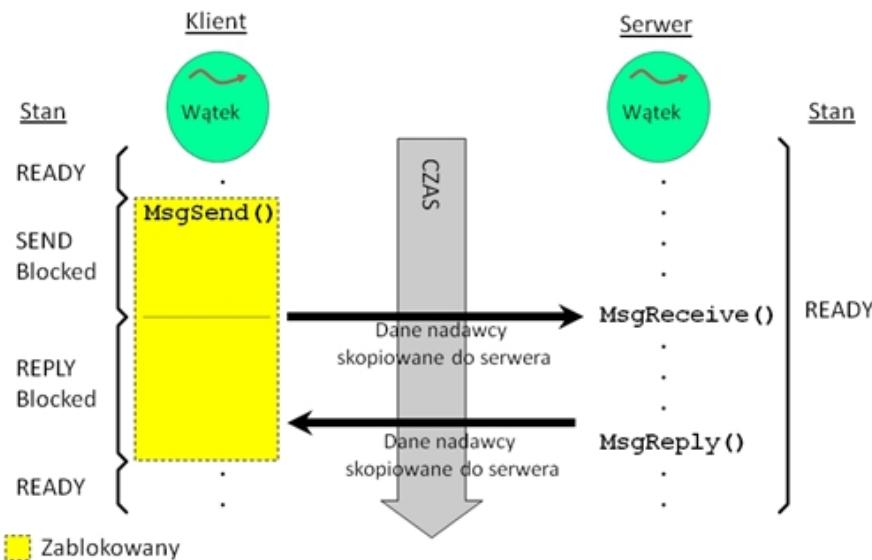
W trakcie komunikacji za pomocą przesyłania komunikatów mogą się zdarzyć dwa scenariusze:

1. Wywołanie funkcji `MsgSend()` następuje po wywołaniu funkcji `MsgReceive()`. Serwer jest blokowany do momentu nadania wiadomości i pozostaje w stanie RECEIVE blocked do momentu odebrania wiadomości wysłanej przez klienta. W trakcie przetwarzania danych przez serwer, klient pozostaje w stanie REPLY blocked, aż do czasu, gdy otrzyma odpowiedź od serwera. Poza omówionymi stanami, procesy klienta i serwera pozostają w stanie READY. Sytuację tę ilustruje rysunek 30.



Rysunek 30: Ilustracja przesyłania komunikatów - przypadek 1

2. Wywołanie funkcji `MsgSend()` następuje przed wywołaniem funkcji `MsgReceive()`. Klient w stanie SEND blocked pozostaje zablokowany do momentu kiedy serwer rozpoczęte odbieranie komunikatu. Jak poprzedni, w trakcie przetwarzania danych przez serwer, klient pozostaje w stanie REPLY blocked, aż do czasu, gdy otrzyma odpowiedź od serwera. Poza omówionymi stanami, procesy klienta i serwera pozostają w stanie READY. Sytuację tę ilustruje rysunek 31.



Rysunek 31: Ilustracja przesyłania komunikatów - przypadek 2

Wysyłanie komunikatu realizujemy poprzez następujące wywołanie:

```

1 #include <sys/neutrino.h>
2
3 int MsgSend( int coid ,
4               const void* smsg ,
5               int sbytes ,
6               void* rmsg ,
7               int rbytes );
8
9 int MsgSend_r( int coid ,
10                 const void* smsg ,
11                 int sbytes ,
12                 void* rmsg ,
13                 int rbytes );

```

gdzie

`coid` - jest numerem kanału, do którego chcemy wysyłać wiadomości.

`smsg` - wskaźnik do bufora zawierającego wiadomość do wysłania.

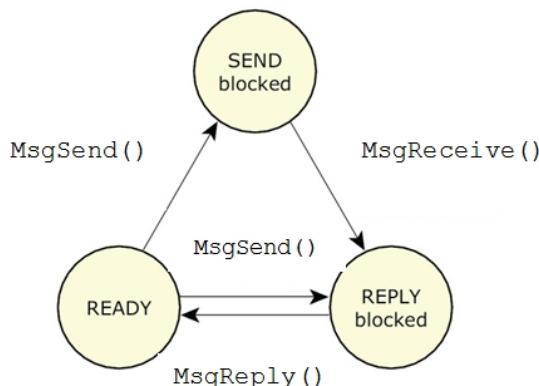
`sbytes` - liczba bajtów określająca pojemność bufora do wysłania.

`rmsg` - wskaźnik do bufora zawierającego odpowiedź.

`rbytes` - liczba bajtów określająca pojemność bufora na odpowiedź.

Funkcja `MsgSend()` wysyła komunikat do kanału określonego przez `coid`. Funkcja ta służy zarówno

do wysyłania komunikatu, jak i odbierania wiadomości do serwera. `MsgSend()` zwraca status, który jest przekazywany przez `MsgReply()` lub -1 w przypadku błędu. Działanie funkcji zależy od stanu procesu serwera. Przejścia stanów procesu-klienta zilustrowano na rysunku 32.



Rysunek 32: Przejścia stanów procesu klienta

Serwer może odebrać komunikat z kanału poprzez wywołanie następującej funkcji:

```

1 #include <sys/neutrino.h>
2
3 int MsgReceive( int chid,
4                  void * msg,
5                  int bytes,
6                  struct _msg_info * info );
7
8 int MsgReceive_r( int chid,
9                  void * msg,
10                 int bytes,
11                 struct _msg_info * info );

```

gdzie

`chid` - numer kanału zwrócony przez funkcję `ChannelCreate()`.

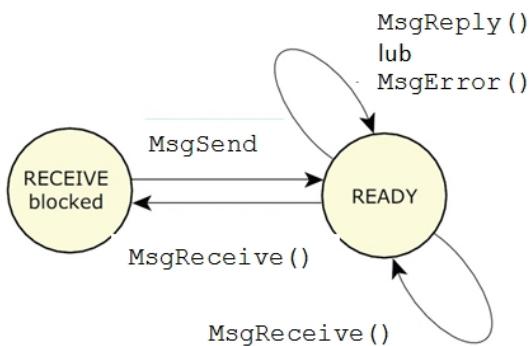
`msg` - wskaźnik do bufora zawierającego odebraną wiadomość.

`bytes` - liczba bajtów określająca pojemność bufora do odbioru komunikatu.

`info` - wskaźnik NULL, bądź wskaźnik do struktury `_msg_info`, zawierającej dodatkowe informacje o komunikacie.

Funkcja odbiera wiadomość, bądź impuls od klienta. W przypadku komunikatu, funkcja zwraca identyfikator nadawcy `rcvid > 0` (receive identifier), w której zakodowano identyfikator wątku (TID) wysyłającego i identyfikator połączenia (COID). Identyfikator `rcvid` będzie użyty w funkcji `MsgReply()`, wysyłającej do klienta odpowiedź. W przypadku, gdy funkcja odebrała impuls, identyfikator `rcvid = 0`; struktura `_msg_info` nie jest modyfikowana. Przejścia stanów procesu-klienta zilustrowano na rysunku 33

Przesłanie odpowiedzi na komunikat można zrealizować funkcją:



Rysunek 33: Przejścia stanów procesu serwera

```

1 #include <sys/neutrino.h>
2
3 int MsgReply( int rcvid,
4                 int status,
5                 const void* msg,
6                 int size );
7
8 int MsgReply_r( int rcvid,
9                  int status,
10                 const void* msg,
11                 int size );

```

gdzie

`rcvid` - identyfikator nadawcy, zwrocony przez funkcję `MsgReceive()`.

`status` - wartość zwracana przez `MsgSend()`, na której jest zablokowany wątek, którego ID jest zakodowane w `rcvid`.

`msg` - wskaźnik do bufora zawierającego wysyłaną wiadomość.

`info` - wskaźnik NULL, bądź wskaźnik do struktury `_msg_info`, zawierającej dodatkowe informacje o komunikacie. `size` - liczba bajtów określająca pojemność bufora do odpowiedzi.

Funkcja zwraca 0, gdy wykonała się poprawnie i -1, gdy wystąpił błąd. Zanim przejdziemy do napisania prostego szkieletu aplikacji typu klient-serwer spójrzmy na rysunek 34 prezentujący zależności pomiędzy danymi w procesie Send/Receive/Reply. Oczywiście proces-serwer może odpowiedzieć klientowi, nie wysyłając danych. Scenariusz taki może być użyty w celu odblokowania klienta, kiedy przekazujemy tylko status poprawnego zakończenia do funkcji `MsgSend()`. Realizujemy to przez wywołanie `MsgReply(rcvid, EOK, NULL, 0)`.

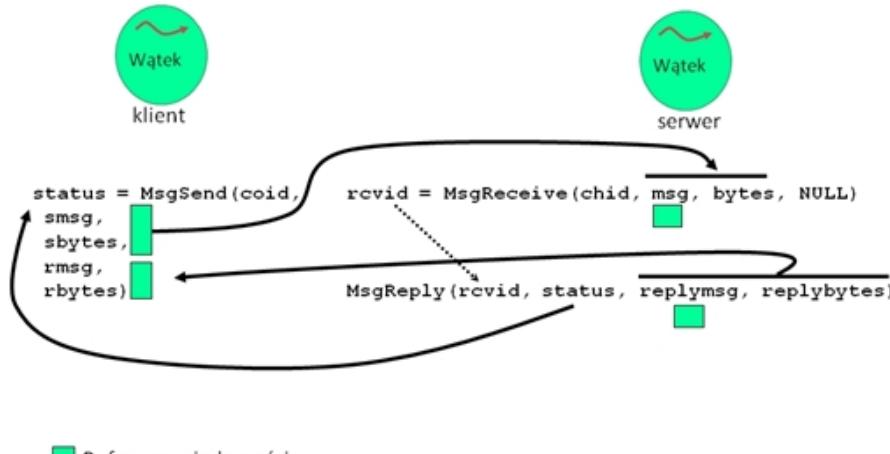
**Przykład 8.3.1.** [Komunikator typu klient-serwer] Skompilować oddzielnie program klienta i program serwera. Najpierw uruchomić serwer, a w następnej kolejności proces klienta z odpowiednimi parametrami.

Kod źródłowy 32: Szkielekt aplikacji typu klient

```

1 /* client.c */
2 #include <stdlib.h>

```



Rysunek 34: Zależności pomiędzy danymi w mechanizmie komunikatów

```

3 #include <stdio.h>
4 #include <sys/neutrino.h>
5 #include <sys/netmgr.h>
6 #include <string.h>
7 #include <errno.h>
8
9 int main(int argc, char *argv[])
10 {
11     int server_pid; // PID serwera
12     int server_chid; // CHID serwera
13     int coid; // COID
14     char smsg[256]; // bufor na wiadomosci do wyslania
15     int status; // status z funkcji MsgSend()
16     int checksum; // suma kontrolna – liczba liter w stringu
17
18     if(argc != 4) {
19         printf("K: Program musi byc wywolany z 3 argumentami, np:\n\n");
20         printf("K: ./client 482834 1 abcdefghi \n\n");
21         printf("K: - pierwszy arg(482834): pid serwera\n");
22         printf("K: - drugi arg(1): chid serwera\n");
23         printf("K: - trzeci arg(abcdefghi): string wysylany do
24             serwera\n");
25         exit(EXIT_FAILURE);
26     }
27     server_pid = atoi(argv[1]);
28     server_chid = atoi(argv[2]);
29
30     printf("K: Probuje nawiazac polaczenie z serwerem pid: %d, chid
31           %d\n", server_pid, server_chid);
32     /* Tworzenie polaczenia */
33     coid = ConnectAttach(ND_LOCAL_NODE, server_pid, server_chid,
34                         _NTO_SIDE_CHANNEL, 0);
35     if (coid == -1)
36     {
37         printf("K: Nie moge utworzyc polaczenia: %s\n", strerror(
38             errno));
39         exit(EXIT_FAILURE);
40     }
41
42     strcpy(smsg, argv[3]);
43     printf("K: Wysylam wiadomosc: %s\n", smsg);

```

```

41     status = MsgSend(coid, smsg, sizeof(smsg), &checksum, sizeof(
42         checksum));
43
44     if (status == -1)
45     {
46         printf("K: Nie moge wyslac wiadomosci: %s\n", strerror(errno
47             ));
48         exit(EXIT_FAILURE);
49     }
50
51     printf("K: Otrzymana z serwera suma kontrolna: %d\n", checksum)
52     ;
53     printf("K: Funkcja MsgSend zwrocila status: %d\n", status);
54
55     return EXIT_SUCCESS;
56 }
```

Kod źródłowy 33: Szkielet aplikacji typu serwer

```

1  /* server.c */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/neutrino.h>
5  #include <string.h>
6  #include <errno.h>
7  #include <process.h>
8
9  int main(int argc, char *argv[])
10 {
11     int chid; // Kanal komunikacyjny
12     pid_t pid; // PID serwera
13     int rcvid; // identifikator nadawcy
14     struct _msg_info info;
15     char rmsg[256]; // bufor na wiadomosci odpowiedzi
16     int checksum; // liczba znakow w odebranej wiadomosci
17     int status = 0;
18
19     chid = ChannelCreate(0);
20
21     if (chid == -1)
22     {
23         printf("\tS: Nie moge utworzyc kanalu: %s\n", strerror(errno
24             ));
25         return EXIT_FAILURE;
26     }
27
28     pid = getpid();
29     printf("\tS: PID serwera: %d, CHID: %d\n", pid, chid);
30
31     while(1)
32     {
33         /* Odebranie komunikatu */
34         rcvid = MsgReceive(chid, &rmsg, sizeof(rmsg), &info);
35         printf("\tS: Odebrana wiadomosc: %s\n", rmsg);
36         printf("\tS: Zakodowany identyfikator nadawcy i polaczenia: %
37             d\n", rcvid);
38         /* Jesli nie udalo sie pobrac wiadomosci */
39         if(rcvid == -1)
40         {
41             printf("\tS: Nie moge odebrac wiadomosci: %s\n", strerror(
42                 errno));
43             break; // sprobuj odebrac inna wiadomosc
44         }
45     }
46 }
```

```

42     checksum = strlen(rmsg);
43
44     /* odpowiedz klientowi, wysylajac sume kontrolna */
45     MsgReply(rcvid, status, &checksum, sizeof(checksum));
46     printf("\tS: Status serwera: %d, suma kontrolna: %d\n", status
47           , checksum);
48
49     if(status == -1)
50     {
51         printf("\tS: Nie moze odpowiedziec: %s\n", strerror(errno))
52         ;
53     }
54
55     return EXIT_SUCCESS;
56 }
```

## 8.4 Impulsy

W aplikacjach często występuje potrzeba powiadomienia procesu o wystąpieniu zdarzenia, ale bez blokady procesu powiadamiającego. System operacyjny QNX Neutrino oprócz synchronicznej komunikacji Send/Receive/Reply, które blokują proces klienta, do czasu, gdy serwer odpowie, wspiera szybką komunikację asynchroniczną. Narzędziem umożliwiającym taką sygnalizację są impulsy (pulses). Impulsy to małe wiadomości, które nie powodują zablokowania procesu wysyłającego. Impuls jest 40-bitowym komunikatem, który zawiera 8-bitowy kod z zakresu od 0 do 127, zdefiniowane w <sys/neutrino.h> i 32-bitową wartość, która może być dowolnie wykorzystana przez programistę. Impulsy mogą być odbierane tak, jak inne wiadomości za pomocą funkcji `MsgReceive()`. Jeśli funkcja odbierze impuls, to zwraca identyfikator nadawcy (receive identifier) równy 0. We wspomnianym pliku nagłówkowym zdefiniowano impuls, jako następującą strukturę:

```

1 struct _pulse {
2     _uint16 type;
3     _uint16 subtype;
4     _int8 code;
5     _uint8 zero [3];
6     union sigval value;
7     _int32 scoid;
8 };
9 union sigval {
10     int sival_int;
11     void *sival_ptr;
12 };
```

Najważniejsze pola tej struktury dotyczą elementów `code` oraz `value`. Pole `code` identyfikuje typ impulsu, natomiast element `value` jest uzupełnieniem pola `code` i może być dowolnie ustawione, w przeciwieństwie do innych pól tej struktury. Impulsy można wysyłać za pomocą funkcji `MsgSendPulse()`:

```

1 #include <sys/neutrino.h>
2
3 int MsgSendPulse ( int coid,
```

```

4     int priority,
5     int code,
6     int value );

```

gdzie

coid - identyfikator połączenia, ustanowiony przez funkcję `ConnectAttach()`.

priority - priorytet impulsu.

code - 8-bitowy pole kodu wysyłanego impulsu, określający jego typ. Bezpieczny zakres kodu mieści się między wartością `_PULSE_CODE_MINAVAIL`, a wartością `_PULSE_CODE_MAXAVAIL`.

value - 32-bitowy komunikat.

Funkcja `MsgSendPulse()` umożliwia nieblokujące przekazanie 32-bitowego komunikatu do kanału komunikacyjnego, identyfikowanego przez `coid`. Funkcja zwraca `-1` w przypadku błędu oraz inną wartość, gdy wykona się prawidłowo. Impulsy można odbierać za pomocą funkcji `MsgReceive()`. Jeśli jednak proces ma odbierać tylko impulsy, to można zastosować funkcję `MsgReceivePulse()`. Opis funkcji można znaleźć w dokumentacji systemu.

**Przykład 8.4.1.** [Komunikator typu klient-serwer-impulsy] Skompilować oddzielnie program klienta i program serwera. Najpierw uruchomić serwer, a w następnej kolejności proces klienta z odpowiednimi parametrami.

Kod źródłowy 34: Szkielet aplikacji typu klient

```

1  /* client.c */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/neutrino.h>
5  #include <sys/netmgr.h>
6  #include <string.h>
7  #include <errno.h>
8  #include <sched.h>
9
10 int main(int argc, char *argv[])
11 {
12     int server_pid;    // PID serwera
13     int server_chid;  // CHID serwera
14     int coid; // COID
15     int value1; // wysyłana liczba1
16     int value2 = 38; // wysyłana liczba2
17     int status; // status z funkcji MsgSend()
18
19     if(argc != 4) {
20         printf("K: Program musi byc wywolany z 3 argumentami, np:\n\n");
21         printf("K: client 482834 1 abcdefghi\n");
22         printf("K: - pierwszy arg(482834): pid serwera\n");
23         printf("K: - drugi arg(1): chid serwera\n");
24         printf("K: - trzeci arg(5): int wysylany do serwera\n");
25         exit(EXIT_FAILURE);
26     }
27     server_pid = atoi(argv[1]);
28     server_chid = atoi(argv[2]);
29

```

```

30     printf("K: Probuje nawiazac polaczenie z serwerem pid: %d, chid
31         %d\n", server_pid, server_chid);
32     /* Tworzenie polaczenia */
33     coid = ConnectAttach(ND_LOCAL_NODE, server_pid, server_chid,
34         _NTO_SIDE_CHANNEL, 0);
35     if (coid == -1)
36     {
37         printf("K: Nie moge utworzyc polaczenia: %s\n", strerror(
38             errno));
39         exit(EXIT_FAILURE);
40     }
41
42     value1 = atoi(argv[3]);
43     printf("K: Wysylam wiadomosc: %d\n", value1);
44     // getprio(0) pobiera priorytet biezacego procesu.
45     status = MsgSendPulse(coid, getprio(0), _PULSE_CODE_MINAVAIL,
46         value1);
47
48     if (status == -1)
49     {
50         printf("K: Nie moge wyslac impulsu 1: %s\n", strerror(errno))
51         ;
52         exit(EXIT_FAILURE);
53     }
54     printf("K: Wysylam wiadomosc: %d\n", value2);
55     status = MsgSendPulse(coid, getprio(0), _PULSE_CODE_MINAVAIL+5,
56         value2);
57     if (status == -1)
58     {
59         printf("K: Nie moge wyslac impulsu 2: %s\n", strerror(errno))
60         ;
61         exit(EXIT_FAILURE);
62     }
63
64     return EXIT_SUCCESS;
65 }
```

**Kod źródłowy 35:** Szkiclet aplikacji typu serwer

```

1  /* server.c */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/neutrino.h>
5  #include <string.h>
6  #include <errno.h>
7  #include <process.h>
8
9  int main(int argc, char *argv[])
10 {
11     int chid; // Kanał komunikacyjny
12     pid_t pid; // PID serwera
13     int rcvid; // identifikator nadawcy
14     struct _msg_info info;
15     int checksum; // liczba znakow w odebranej wiadomosci
16     int status = 0;
17     struct _pulse msg;
18
19     chid = ChannelCreate(0);
20
21     if (chid == -1)
22     {
23         printf("\tS: Nie moge utworzyc kanalu: %s\n", strerror(errno))
24         );
```

```

24     return EXIT_FAILURE;
25 }
26
27 pid = getpid();
28 printf("\tS: PID serwera: %d, CHID: %d\n", pid, chid);
29
30 while(1)
31 {
32     /* Odebranie impulsu */
33     rcvid = MsgReceive(chid, &msg, sizeof(msg), &info);
34     /* Jesli nie udalo sie pobrać wiadomosci */
35     if(rcvid == -1)
36     {
37         printf("\tS: Nie moge odebrac wiadomosci: %s\n", strerror(
38             errno));
39         break;                                // sprobuj odebrac inna wiadomosc
40     }
41     else if (rcvid == 0)
42     {
43         printf("\tS: Odebralem impuls.\n\tS: Kod impulsu: %d,
44             wartosc %d\n", msg.code, msg.value.sival_int);
45         continue;
46     }
47
48     /* odpowiedz klientowi, jesli komunikat */
49     MsgReply(rcvid, status, &checksum, sizeof(checksum));
50
51     if(status == -1)
52     {
53         printf("\tS: Nie moge odpowiedziec: %s\n", strerror(errno))
54         ;
55     }
56
57     return EXIT_SUCCESS;
58 }

```

## 8.5 W jaki sposób klient znajduje serwer?

Kiedy proces klienta komunikuje się z procesem serwera potrzebuje trzech informacji: identyfikator węzła ND, numer PID procesu serwera oraz identyfikator kanału CHID. W dotychczasowych przykładach, komunikacja występowała na lokalnym węźle, a numery PID i CHID były przekazywane z linii poleceń. W jaki sposób, w ogólnym przypadku, klient ma uzyskać od serwera informacje o numerach ND/PID/CHID? Istnieje wiele rozwiązań tego problemu. Wybrane sposoby, wg wzrastającego stopnia ogólności podano poniżej.

1. W ustalonym i znanym miejscu serwer tworzy plik tekstowy, w którym będą przechowywane numery ND/PID/CHID. Plik ten może być następnie przeczytany przez procesy klientów. Metoda ta jest często stosowana w systemach UNIX-owych.
2. Użycie zmiennych globalnych do przechowywania informacji o zmiennych ND/PID/CHID. Sytuacja ta jest typowa dla przypadku procesów macierzystych i potomnych, a także serwerów wielo-

wątkowych.

3. Użycie mechanizmu globalnych nazw GNS (Global Name Service), takich jak `name_attach()`, `name_detach()`, `name_open()` i `name_close()`.
4. Utworzyć serwer, jako menadżer zasobów. Do identyfikacji procesu serwera można użyć przestrzeni nazw plików.

Pierwsza metoda, pomimo, że dość prosta w implementacji, posiada istotne wady. Utworzzone przez serwer pliki z informacjami ND/PID/CHID pozostają w pamięci nawet w przypadku, gdy proces serwera zostanie zakończony, a dane stracą swoją ważność. Może się również zdarzyć sytuacja, że system operacyjny utworzy inny proces o takich samych danych ND/PID/CHID, jak w pliku. W tym przypadku, komunikaty mogą trafić do niewłaściwego adresata i spowodować niepoprawne działanie systemu.

Drugie podejście wymaga zastosowania zmiennych globalnych do przekazywania informacji o ND/PID/CHID. Rozwiązanie to działa w zakresie pamięci wspólnej, wykluczona jest komunikacja sieciowa. Metoda ta może być stosowana w przypadku aplikacji wieloprocesorowych i wielowątkowych. Proces (wątek) serwera tworzy kanał komunikacyjny i umieszcza w zmiennych globalnych PID/CHID, które następnie mogą być odczytane przez procesy (wątki) klientów.

Trzecie podejście polega na zastosowaniu mechanizmu globalnych nazw (Global Name Service). Mechanizm ten działa dobrze w przypadku prostych aplikacji typu klient-serwer.

Ostatnie rozwiązanie jest najbardziej ogólne spośród wymienionych. Proces serwera staje się menadżerem zasobów. Serwer rejestruje unikalną nazwę ścieżki dostępu, natomiast klient może wtedy wykonać prostą operację `open()` na tej ścieżce. Opis menadżerów zasobów można znaleźć w dokumentacji QNX.

## 8.6 Ćwiczenia

1. Wywołać w wierszu poleceń funkcję `pidin | more`. Obejrzeć wyniki, zwrócić uwagę na kolumnę STATE i BLOCKED. Pierwsza z nich wskazuje stan procesu. Druga z nich, w przypadku stanu REPLY pokazuje PID procesu serwera, od którego czekamy na odpowiedź, bądź w przypadku stanu RECEIVE pokazuje nr kanału, w którym oczekujemy wiadomości. Otworzyć perspektywę QNX System Information Perspective. Obejrzeć widoki Process Information, Connection Information i System Blocking Graph.
2. Na bazie szkieletu aplikacji klient-serwer, napisać program klienta, który będzie w pętli pobierał od użytkownika linie tekstu ze standardowego wejścia (użyć funkcji: `char *fgets(char *str, int size, FILE *stream)`) ; a następnie będzie przekazywał komunikat do serwera. Serwer w odpowiedzi ma wyświetlać na ekranie komunikat i wysyłać do klienta przerobiony komunikat, opatrzony nagłówkiem (dowolnym napisem, np. \*\*\*). Niech informacje o numerach ND/PID/CHID będą przekazywane poprzez plik tekstowy.

## **Laboratorium 9**

**Czas w systemie QNX Neutrino. Oprogramowanie timerów i zdarzeń**

---



## Laboratorium 10

### Tworzenie oprogramowania dla urządzeń wbudowanych

#### Plan prezentacji

## Tworzenie oprogramowania dla urządzeń wbudowanych

Paweł Małczyk

Zakład Teorii Maszyn i Robotów  
Instytut Techniki Lotniczej i Mechaniki Stosowanej  
Wydział Mechaniczny Energetyki i Lotnictwa  
Politechnika Warszawska

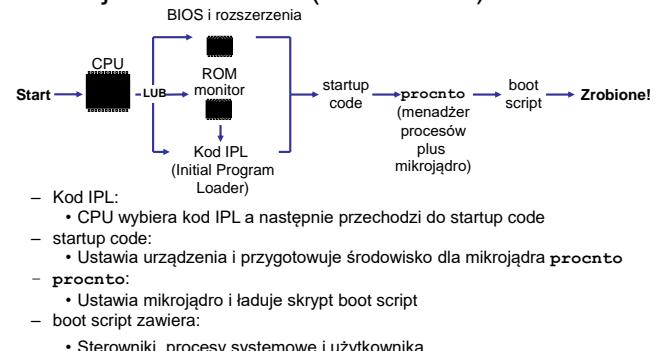
- Proces rozruchu i ładowania systemu operacyjnego QNX Neutrino
- Budowanie obrazu systemu QNX Neutrino na platformie typu BeagleBone Black
- Komputer BeagleBone Black
- Budowanie obrazu systemu z QNX Momentics

Metody programowania robotów

Metody programowania robotów

#### Kolejność rozruchu

- Kolejność rozruchu (bootowania):

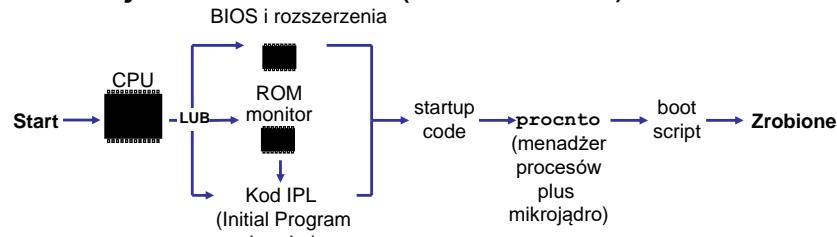


Uwaga: BBB nie posiada BIOS-u. Do ładowania systemu używane są bootloader-y.

Metody programowania robotów

Metody programowania robotów

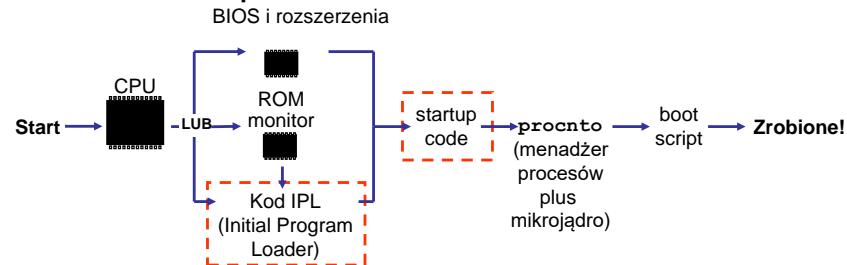
- Kolejność rozruchu (bootowania):



- Kod IPL:
  - CPU wybiera kod IPL a następnie przechodzi do startup code
- startup code:
  - Ustawia urządzenia i przygotowuje środowisko dla mikrojądra procnto
- procnto:
  - Ustawia mikrojądro i ładowa skrypt boot script
- boot script zawiera:
  - Sterowniki, procesy systemowe i użytkownika

Uwaga: BBB nie posiada BIOS-u. Do ładowania systemu używane są bootloader-y.

- IPL i startup code:



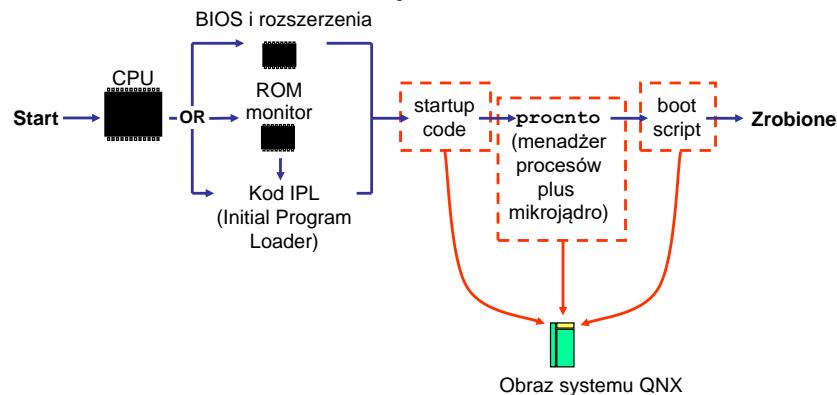
- Skrypty rozruchowe IPL i inicjujące (startup code):

- Są specyficzne dla określonych urządzeń wbudowanych
- Są częścią pakietów Board Support Package (BSP)

## Metody programowania robotów

### Co znajduje się w pliku obrazu systemu?

- Zawartość obrazu systemu:



## Metody programowania robotów

### Struktura pliku obrazu systemu

- Plik obrazu systemu operacyjnego (SO)

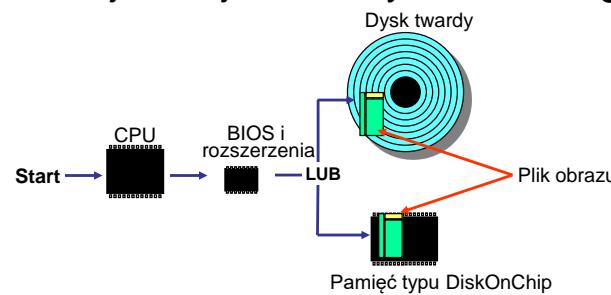
- Zawiera wiele komponentów niezbędnych do inicjalizacji
- Struktura pliku obrazu:



- Przykładowe pliki:
- Sterowniki sieci,
  - Stos TCP/IP,
  - Sterowniki dysku twardego,
  - Pliki konfiguracyjne

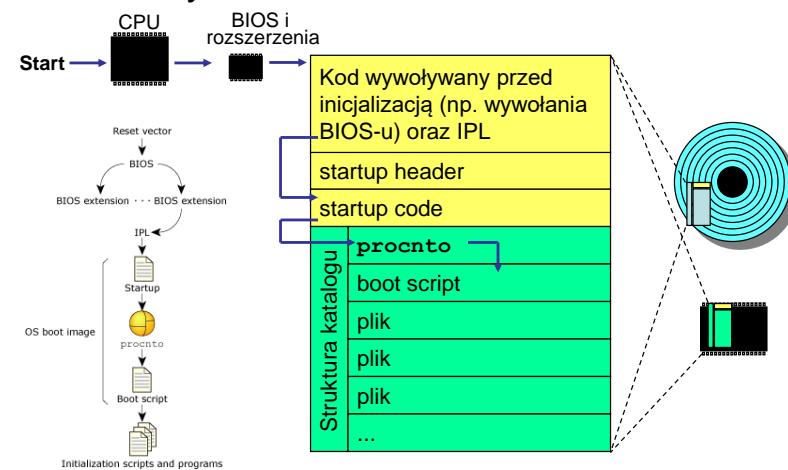
- Możliwośćinicjalizacji SO:
  1. ROM/flash
  2. Dysk twardy
  3. Sieć/połączenie szeregowe

- Inicjalizacja SO z dysku twardego:



- Obraz SO jest ładowany do RAM-u za pomocą różnych komponentów....

- Obraz systemu:



## Budowanie obrazu systemu QNX Neutrino

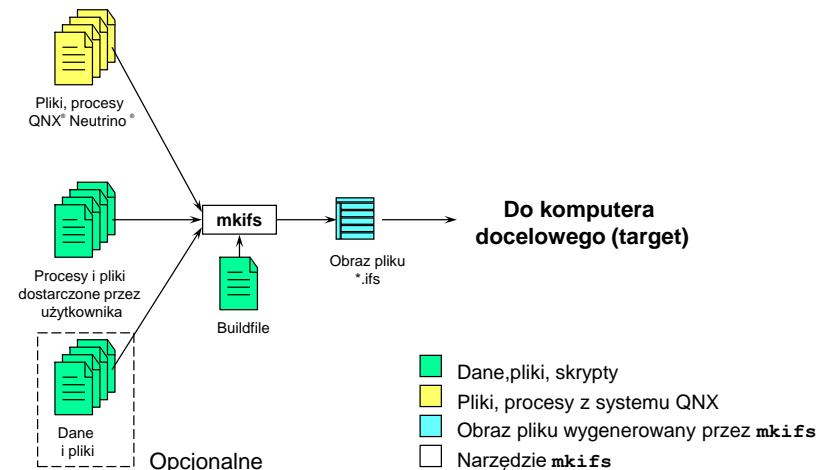
## Struktura pliku obrazu systemu

- Plik obrazu systemu operacyjnego (SO)
  - Zawiera wiele komponentów niezbędnych do inicjalizacji
  - Struktura pliku obrazu:



Po inicjalizacji pliki znajdują się w katalogu: `/proc/boot`

## Schemat tworzenia obrazu systemu



## Co zawiera obraz systemu?

- Skrypt startowy oraz mikrojądro z menadżerem procesów:  
`startup-*`  
`procnto`
- Pozostałe komponenty:  
sterowniki and menadżery, e.g.: `io-net`,  
`devc-ser*`, `devb-eide`  
`esh` (embedded shell), `ksh`  
pliki wykonywalne użytkownika i dane

## Skrypt buildfile

- Czym jest plik buildfile?
  - Określa pliki/polecenia, które będą dostępne w obrazie,
  - Porządek wywołania podczas rozruchu,
  - Procesy linii wywoływane z wiersza poleceń, zmienne środowiskowe
  - Plik zawiera opcje ładowania plików wykonywalnych

## Plik buildfile dla minimalnego systemu `min_image.ifs`

```
# Plik Buildfile dla minimalnego systemu
[virtual=x86,bios +compress] boot={
    startup-bios
    PATH=: /proc/boot:/bin:/sbin:/usr/bin:/usr/sbin
    LD_LIBRARY_PATH=: /proc/boot:/lib:/usr/lib:/lib/dll
    procnto -vv
}

[+script] startup-script={
    display_msg "Moj obraz systemu QNX..."
    devc-con &
    reopen /dev/con1
    hello
}

libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
devc-con
hello
```

bootscript

## Skrypt buildfile – format

- Format buildfile:  
**atrybut nazwa\_pliku zawartość**
- Skrypt może zawierać puste linie i komentarze zaznaczane za pomocą znaku „#”
- Dwa typy atrybutów:
  - boolowski, np. [+atrybut] lub [-atrybut]
  - wartość, np. [atrybut=wartość]
- Möżliwość utworzenia pliku inline  
`readme = {  
 W systemie plików zostanie utworzony plik  
 dostępny w /proc/boot/readme.  
}`

## Skrypt buildfile – komendy wbudowane

- Komendy wbudowane:
  - display\_msg** wyświetla tekst
  - procmgr\_symlink** link symboliczny, odpowiednik wywołania `ln -s`
  - reopen** strumienie stdin, stdout, and stderr są przekierowane do określonego pliku
  - waitfor** czeka aż powstanie określony plik
- Przykłady wywołań **display\_msg**, **reopen** można znaleźć w skrypcie.

## Plik buildfile dla minimalnego systemu `min_image.ifs`

```
# Plik Buildfile dla minimalnego systemu
[virtual=x86,bios +compress] boot={
    startup-bios
    PATH=: /proc/boot:/bin:/sbin:/usr/bin:/usr/sbin
    LD_LIBRARY_PATH=: /proc/boot:/lib:/usr/lib:/lib/dll
    procnto -vv
}

[+script] startup-script={
    display_msg "Moj obraz systemu QNX..."
    devc-con &
    reopen /dev/con1
    hello
}

libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
devc-con
hello
```

bootscript

## Plik Bootstrap file (skrypt inicjujący)

```
# Plik Buildfile dla minimalnego systemu
[virtual=x86,bios +compress] boot={
    startup-bios
    PATH=/proc/boot:/bin:/sbin:/usr/bin:/usr/sbin
    LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll
procnto -vv
}

Bootstrap file zawiera:
• atrybut [virtual=x86,bios +compress] Nazwa pliku Kompresja obrazu *.ifs
• nazwa pliku boot
• zawartość startup-bios Program startowy. Inicjalizuje sprzęt i wywołuje procnto
    PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot procnto -vv
    Mikrojadro
    Zmienne środowiskowe
    Adresowanie pamięci
    Architektura CPU
    Dla układów z BIOS-em
```

## Reszta pliku buildfile

```
[+script] startup-script={
    display_msg "Moj obraz systemu QNX..."
    devc-con &
    reopen /dev/con1
    hello
}
Uruchom hello Uruchom menadżer konsoli

libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
devc-con
hello
Dowiązanie symboliczne
Pliki zawarte w systemie
```

Startup script (skrypt ładowający procesy, usługi, urządzenia)

Reszta pliku buildfile (biblioteki, programy, ustawienia)

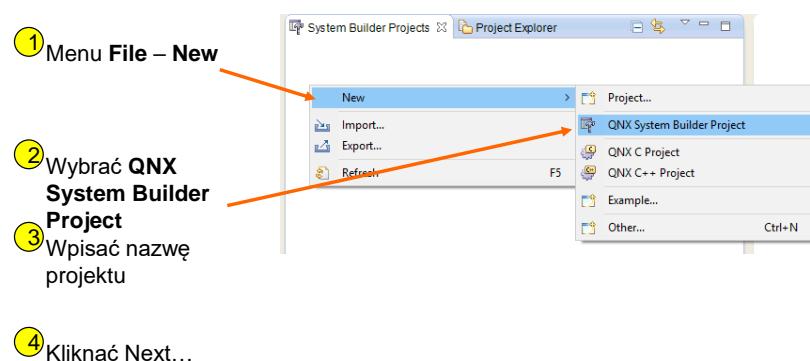
Narzędzie `mkifs` szuka plików w następujących katalogach:

```
 ${QNX_TARGET}\${PROCESSOR}/sbin oraz
 .../usr/sbin, .../boot/sys, .../bin, .../usr/bin,
 .../lib, .../lib/dll, .../usr/lib, .../usr/photon/bin
```

## Budowanie obrazu systemu z QNX Momentics

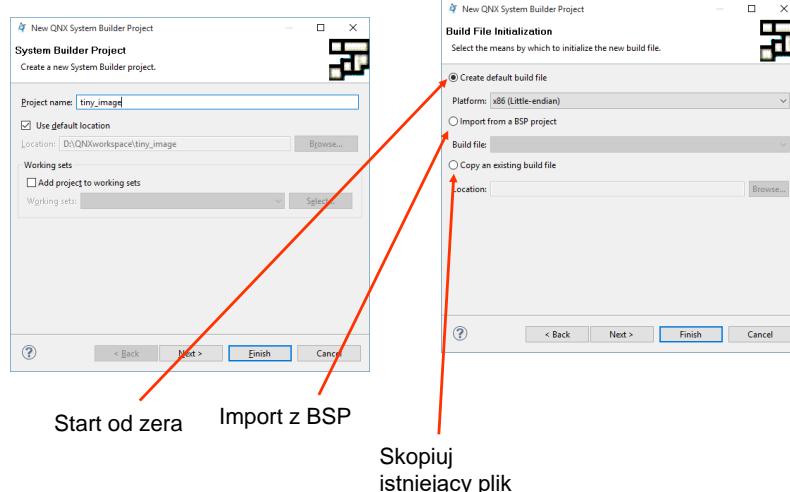
### Utworzenie obrazu w IDE

- Otwieramy IDE QNX Momentics i tworzymy projekt QNX System Builder Project



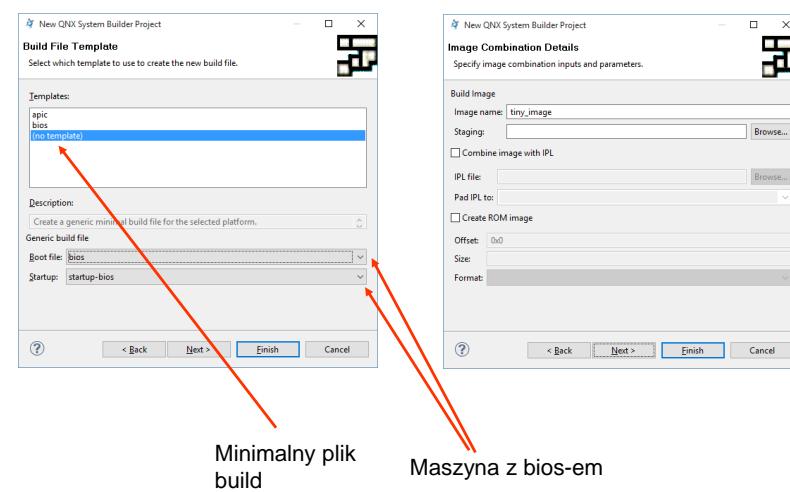
## Utworzenie obrazu w IDE Momentics

- Wybrać stosowną opcję:

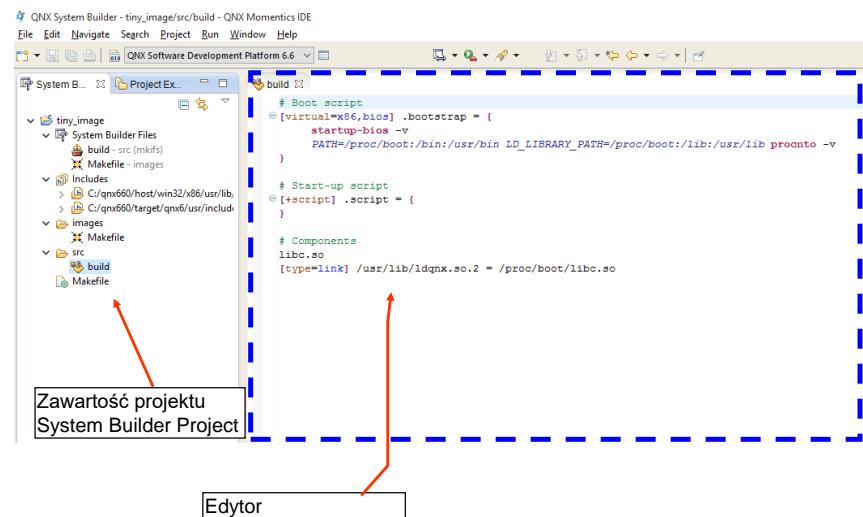


## Utworzenie obrazu w IDE Momentics

- Wybrać stosowną opcję:

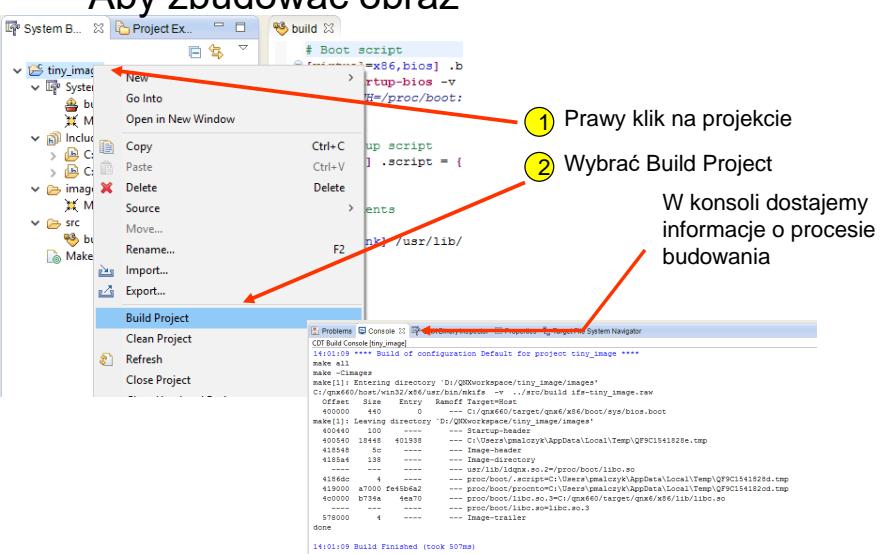


## Projekt System Builder



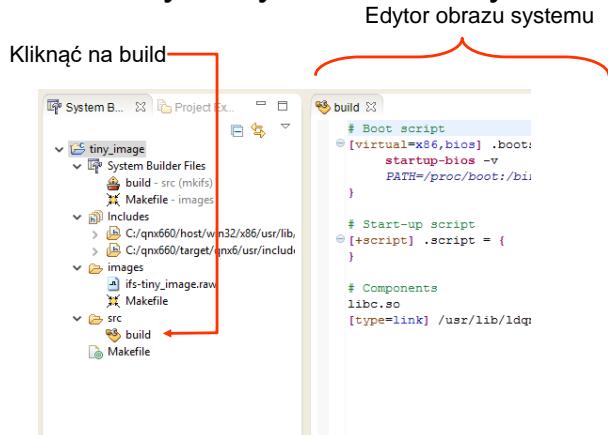
## Budowanie obrazu

- Aby zbudować obraz



W konsoli dostajemy informacje o procesie budowania

- Otworzyć edytor obrazu systemu:



- Zalogować się do systemu QNX. Wylistować zawartość katalogu `.boot` zawierającą obrazy systemu.
- Sprawdzić zawartość załadowanego obrazu systemu znajdującego się w katalogu `/proc/boot`.
- Napisać i zbudować w IDE program typu „Hello!”. Plik wykonywalny skopiować do katalogu `C:\qnx660\target\qnx6\x86\bin`
- Utworzyć nowy projekt QNX Builder System Project o nazwie `min_image`; wybrać platformę x86 oraz opcje: no template, boot file bios oraz startup-bios.
- Napisać skrypt `build` w katalogu `src` wg poniższej instrukcji.
- Zbudować obraz systemu.
- Przekopiować powstały obraz SO do katalogu `.boot` na maszynie wirtualnej. Obejrzeć zawartość obrazu `min_image.raw` wpisując w konsoli polecenie:  
`dumpifs min_image.raw`
- Zrestartować maszynę poleciem `shutdown` i wybrać obraz systemu `min_image.raw` z menu.

Metody programowania robotów  
Plik buildfile dla minimalnego systemu min\_image.rawMetody programowania robotów  
Ćwiczenie 2 – tworzenie obrazu tiny\_image.raw

```
# Plik build dla minimalnego systemu
[virtual=x86,bios +compress] boot={
    startup-bios
    PATH=:/proc/boot:/bin:/sbin:/usr/bin:/usr/sbin
    LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll
    procnto -vv
}

[+script] startup-script={
    display_msg "Moj obraz systemu QNX..."
    devc-con &
    reopen /dev/con1
    hello
}

libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
devc-con
hello
```

**bootscript**

Bootstrap file  
(skrypt inicjujący)

Startup script  
(skrypt ładujący procesy, usługi, urządzenia)

Reszta pliku  
buildfile  
(biblioteki, programy, ustawienia)

- Należy zbudować bardziej skomplikowany obraz systemu operacyjnego QNX o nazwie `tiny_image.raw`.

- Przetestować działanie wirtualnych konsoli i dołączonych poleceń.

```
[virtual=x86,bios +compress] boot={
    startup-bios
    PATH=:/proc/boot:/bin:/sbin:/usr/bin:/usr/sbin
    LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll procnto -vv
}
[+script] startup-script{
    display_msg "Moj obraz systemu QNX..."
    hello &
    display_msg "Uruchamiam sterownik dysku..."
    devb-eide blk auto=partition cam quiet &
    display_msg "Otwieram wirtualne konsole..."
    # Ctrl+Alt+1 oraz Ctrl+Alt+2
    devc-con -n2 &
    reopen /dev/con1
    [+session] HOME=/ ksh &
    reopen /dev/con2
    [+session] HOME=/ ksh &
}

# katalog /tmp w pamięci dzielonej
[type=link] /tmp=/dev/shmem
libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
# biblioteki do obsługi dysku
io-blk.so
cam-disk.so
libcam.so
```

## Ćwiczenie 2 - Plik build dla systemu tiny\_image.raw

```
[virtual=x86,bios +compress] boot=
    startup-bios
    PATH=/proc/boot:/bin:/sbin:/usr/bin:/usr/sbin

LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll
procinfo -vv
}
[+script] startup-script={
    display_msg "Moj obraz systemu QNX..."
    hello &
    display_msg "Uruchamiam sterownik dysku..."
    devb-eide blk auto=partition cam quiet &
    # zamontuj partycje
    waitfor /dev/hd0t79
    mount /dev/hd0t79 /pliki
    display_msg "Otwieram wirtualne konsole..."
    # Ctrl+Alt+1 oraz Ctrl+Alt+2
    devc-con -n2 &
    reopen /dev/con1
    [+session] HOME=/ ksh &
    reopen /dev/con2
    [+session] HOME=/ ksh &
}
# katalog /tmp w pamięci dzielonej
[type=link] /tmp=/dev/shmem
libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
# biblioteki do obsługi dysku
io-blk.so
cam-disk.so
libcam.so
```

```
# Dowiązanie symboliczne
[type=link]
/usr/lib/libcam.so.2=/proc/boot/libcam.so

# system plikow QNX
fs-qnx4.so

[code=uip data=copy perms=+r,+x]
kill
cat
ls
ksh
devc-con
less
ps
pidin
hogs
uname
mkdir
devb-eide
touch
hello
mount
shutdown
```

## Systemy wbudowane

**System wbudowany** jest to system komputerowy będący częścią większego systemu i wykonujący istotną część jego funkcji.

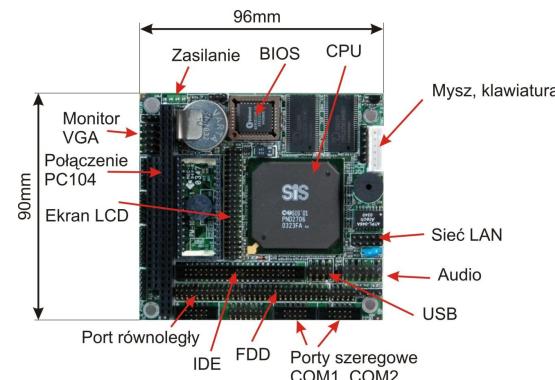
Wymagania na komputery przeznaczone do sterowania i zastosowań wbudowanych.

1. Wymagana jest odporność na pracę w trudnych warunkach otoczenia (wibracje, zapalenie, wilgoć), dopuszczalny jest szeroki zakres temperatur otoczenia.
2. Przeznaczone są do pracy ciągłej - brak jest elementów ruchomych (dyski obrotowe, wentylatory, napędy dyskietek), wymagana jest trwałość, łatwość serwisowania.
3. Oprogramowanie umieszczone jest w pamięci nieulotnej – ROM, flash, EPROM lub podobnej.

Stosowane jest wsparcie sprzętowe dla osiągania niezawodnej pracy – budzik (ang. *watchdog*), pamięci ECC, magistrala z kontrolą parzystości, poszerzona diagnostyka.

Przykład: standard PC104 ...

## Komputer BeagleBone Black

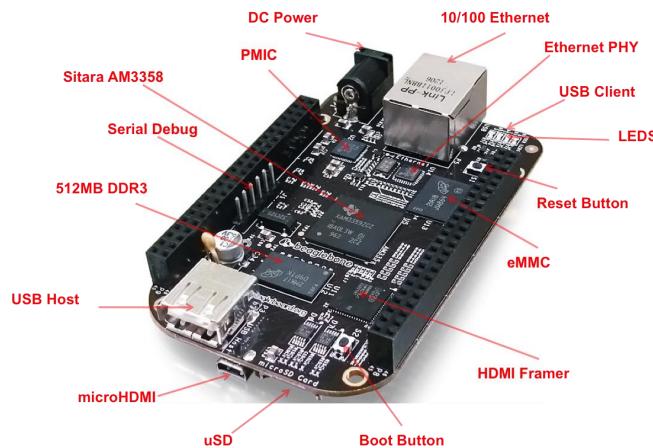


Standard PC104:

1. Standard mechaniczny (np. konstrukcja i wymiary, materiały)
2. Standard elektryczny (końcówki, własności elektryczne)
3. Standard logiczny (adresy, dane, magistrala, przerwania, DMA).

## BeagleBone Black

## BeagleBone Black - specyfikacja



Komponent	Specyfikacja
Procesor	AM335x 1GHz ARM Cortex-A8 + 2x32bitPRU (Programmable Real-time Units)
Pamięć RAM	512MB DDR3 RAM
Grafika	microHDMI (do 1280x1024@60Hz) + audio
Dysk	eMMC + karta microSD
Ethernet	LAN8710A, wsparcie dla DHCP, złącze RJ45
USB	1xminiUSB2.0, 1xUSB2.0
Debug	UART, JTAG
Porty WY-WE	2x46 dostępnych pinów dla wyjść i wejść analogowych, timer, I <sup>2</sup> C, UART, CAN, SPI, LCD, ....
Zasilanie	5VDC

## Sprzęt ...

Metody programowania robotów

Metody programowania robotów

## Praca z pakietami BSP

### Pakiety wsparcia BSP (Board Support Packages)

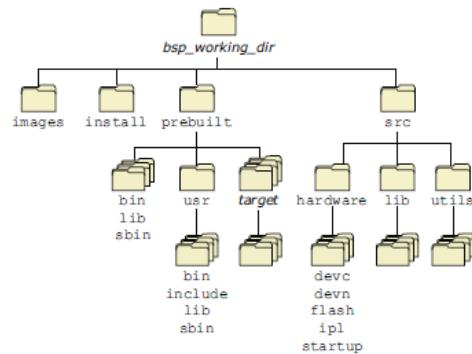
- **IPL** - minimalnie konfiguruje sprzęt, aby zbudować środowisko pozwalające na uruchomienie dalszych komponentów SO.
- **startup** – kopiowanie, dekompresja obrazu SO, konfiguracja sprzętu, start mikrojądra.
- **buildfile** – określa pliki, komendy, zmienne wewnętrz obrazu SO.
- wsparcie dla urządzeń na pokładzie urządzenia wbudowanego, np. dostęp do sieci.
- sterowniki urządzeń.



Metody programowania robotów

Metody programowania robotów

## Pakiety wsparcia BSP (Board Support Packages)



<http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/TiAm335Beaglebone>

1. Pobrać pakiet wsparcia BSP oraz bootloader-y (**MLO** i **u-boot.img**) dla QNX SDP 6.6.0 ze strony:  
<http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/TiAm335Beaglebone>
2. W środowisku IDE zimportować BSP do przestrzeni roboczej poprzez opcję Import oraz wybór **QNX->QNX Source Package and BSP**.
3. Obejrzeć plik **build**, który znajduje się w drzewie **System Builder Files**. Zidentyfikować skrypt inicjujący, skrypt ładowający usługi/procesy oraz listę dołączonych do obrazu plików.
4. Zbudować projekt.
5. Przygotować kartę SD poprzez utworzenie partycji FAT32 oraz ustawienie partycji jako aktywnej.
6. Skopiować na kartę pamięci MLO, u-boot.img oraz obraz systemu QNX **ifs-ti-am335x-beaglebone.bin**
7. Przełożyć kartę SD do urządzenia BBB, a następnie uruchomić BBB.
8. Za pomocą terminala szeregowego sprawdzić poprawność zainicjowania i załadowania QNX-a.



## Spis treści

---

<b>1 Podstawy obsługi systemu operacyjnego QNX RTOS</b>	<b>3</b>
1.1 Powłoka . . . . .	3
1.2 System plików . . . . .	8
1.3 Obsługa procesów . . . . .	13
1.4 Zmienne . . . . .	17
1.5 Przetwarzanie tekstu . . . . .	19
1.6 Strumienie wejściowo-wyjściowe . . . . .	20
1.7 Ćwiczenia . . . . .	23
<b>2 Wprowadzenie do QNX Momentics</b>	<b>25</b>
2.1 Podstawy obsługi QNX Momentics . . . . .	26
2.2 Zarządzanie projektami C/C++ . . . . .	29
2.3 Edycja kodu, komplikacja i budowanie . . . . .	34
2.4 Dostęp do platformy docelowej oraz uruchamianie aplikacji . . . . .	37
<b>3 Wprowadzenie do programowania w języku C</b>	<b>41</b>
3.1 Wstęp . . . . .	41
3.2 Kompilowanie i uruchamianie programów . . . . .	41
3.2.1 Kompilator qcc . . . . .	41
3.2.2 Sterowanie procesem budowania programów . . . . .	43
3.3 Podstawy języka C . . . . .	45
3.3.1 Typy zmiennych . . . . .	45
3.3.2 Pętle . . . . .	46
3.3.3 Konstrukcje warunkowe . . . . .	47
3.3.4 Operatory relacji . . . . .	48
3.3.5 Operatory logiczne . . . . .	49
3.3.6 Wskaźniki . . . . .	49
3.3.7 Tablice . . . . .	50

3.3.8 Funkcje . . . . .	51
3.3.9 Przekazywanie argumentów z wiersza poleceń . . . . .	53
3.3.10 Operacje I/O (wejścia/wyjścia) . . . . .	53
3.3.11 Operacje I/O na plikach . . . . .	54
3.4 Ćwiczenia . . . . .	55
<b>4 Procesy i zarządzanie procesami</b>	<b>57</b>
4.1 Wstęp . . . . .	57
4.2 Wyświetlanie i modyfikowanie atrybutów procesów . . . . .	57
4.3 Tworzenie procesów . . . . .	59
4.4 Obsługa zakończenia procesów . . . . .	61
4.5 Zastąpienie procesu bieżącego innymi procesami . . . . .	64
4.6 Tworzenie procesów funkcją spawn() . . . . .	65
4.7 Ćwiczenia . . . . .	67
<b>5 Zarządzanie wątkami – tworzenie, kończenie, atrybuty wątków</b>	<b>69</b>
5.1 Wprowadzenie . . . . .	69
5.2 Zarządzanie wątkami . . . . .	70
5.2.1 Tworzenie wątków . . . . .	70
5.2.2 Kończenie wątków . . . . .	72
5.2.3 Łączenie wątków . . . . .	74
5.2.4 Atrybuty wątków . . . . .	76
5.2.5 Ustalanie priorytetu, strategii i parametrów szeregowania wątków . . . . .	78
5.3 Ćwiczenia . . . . .	82
<b>6 Mechanizmy synchronizacji wątków – muteksy, zmienne warunkowe, bariery</b>	<b>85</b>
6.1 Wprowadzenie . . . . .	85
6.2 Wyścigi . . . . .	86
6.3 Muteksy . . . . .	87
6.3.1 Tworzenie i kasowanie muteksów . . . . .	88
6.3.2 Zajmowanie i zwalnianie muteksów . . . . .	89

6.3.3	Problemy przy stosowaniu muteksów . . . . .	92
6.4	Zmienne warunkowe . . . . .	95
6.4.1	Wstęp . . . . .	95
6.4.2	Tworzenie i kasowanie zmiennych warunkowych . . . . .	96
6.4.3	Czekanie i sygnalizowanie zmiennej warunkowej . . . . .	97
6.5	Bariery . . . . .	100
6.5.1	Tworzenie i kasowanie bariery . . . . .	100
6.5.2	Czekanie na barierze . . . . .	101
6.6	Ćwiczenia . . . . .	104
<b>7</b>	<b>Potoki nienazwane i nazwane</b>	<b>107</b>
7.1	Czym jest potok? . . . . .	107
7.2	Niskopoziomowe funkcje dostępu do plików . . . . .	108
7.3	Potoki nienazwane . . . . .	113
7.4	Potoki nazwane . . . . .	121
7.5	Ćwiczenia . . . . .	125
<b>8</b>	<b>Mechanizmy komunikacji w systemie QNX Neutrino - komunikaty</b>	<b>127</b>
8.1	Wprowadzenie . . . . .	127
8.2	Tworzenie kanałów i połączeń . . . . .	128
8.3	Wysyłanie, odbieranie i odpowiadanie na komunikaty . . . . .	132
8.4	Impulsy . . . . .	138
8.5	W jaki sposób klient znajduje serwer? . . . . .	141
8.6	Ćwiczenia . . . . .	142
<b>9</b>	<b>Czas w systemie QNX Neutrino. Oprogramowanie timerów i zdarzeń</b>	<b>143</b>
<b>10</b>	<b>Tworzenie oprogramowania dla urządzeń wbudowanych</b>	<b>145</b>
<b>Spis przykładów</b>		<b>160</b>
<b>Spis rysunków</b>		<b>163</b>

<b>Spis tabel</b>	<b>164</b>
<b>Piśmiennictwo</b>	<b>167</b>

---

## Spis przykładów

---

1.1.1 Przykład (Przykład prostej komendy) . . . . .	3
1.1.2 Przykład (Przykład komendy z argumentami) . . . . .	3
1.1.3 Przykład (Przykład złożonej komendy) . . . . .	3
1.1.4 Przykład (Wejście i wyjście z powłoki) . . . . .	3
1.1.5 Przykład (Obsługa edytora tekstu vi) . . . . .	4
1.1.6 Przykład (Utworzenie i uruchomienie skryptu) . . . . .	7
1.1.7 Przykład (Prosty skrypt powłoki) . . . . .	7
1.1.8 Przykład (Prosty skrypt powłoki) . . . . .	7
1.2.1 Przykład (Wylistowanie zawartości katalogu) . . . . .	9
1.2.2 Przykład (Obejrzenie zawartości pliku) . . . . .	9
1.2.3 Przykład (Wyświetlanie liczby wierszy, słów i bajtów zawartych w pliku) . . . . .	9
1.2.4 Przykład (Poruszanie się po systemie plików) . . . . .	10
1.2.5 Przykład (Utworzenie i kasowanie katalogów) . . . . .	10
1.2.6 Przykład (Przenoszenie i kopianie katalogów i plików) . . . . .	11
1.2.7 Przykład (Zmiana atrybutów pliku) . . . . .	12
1.3.1 Przykład (Proces uruchomiony w tle) . . . . .	13
1.3.2 Przykład (Procesy tła i procesy pierwszoplanowe) . . . . .	14
1.3.3 Przykład (Procesy tła i procesy pierwszoplanowe) . . . . .	15
1.3.4 Przykład (Usuwanie procesów) . . . . .	16
1.4.1 Przykład (Podstawowe operacje na zmiennych) . . . . .	17
1.4.2 Przykład (Zmienne i polecenia) . . . . .	18
1.4.3 Przykład (Wyświetlanie kodu zakończenia) . . . . .	18
1.4.4 Przykład (Skrypt i zmienne środowiskowe) . . . . .	18
1.6.1 Przykład (Przekierowanie strumienia wejściowego) . . . . .	20

1.6.2 Przykład (Przekierowanie strumienia wyjściowego) . . . . .	21
1.6.3 Przykład (Przekierowanie strumienia stderr) . . . . .	21
1.6.4 Przykład (Jednoczesne przekierowanie strumienia stdin i stdout) . . . . .	21
1.6.5 Przykład (Potoki i filtrowanie wyników) . . . . .	21
1.6.6 Przykład (Sortowanie i obcinanie wyników) . . . . .	22
1.6.7 Przykład (Prosta sekwencja poleceń) . . . . .	22
1.6.8 Przykład (Sekwencja warunkowa - koniunkcja) . . . . .	22
1.6.9 Przykład (Sekwencja warunkowa - alternatywa) . . . . .	22
3.2.1 Przykład . . . . .	41
3.2.2 Przykład . . . . .	42
3.2.3 Przykład . . . . .	44
3.2.4 Przykład . . . . .	44
3.2.5 Przykład . . . . .	45
3.3.1 Przykład . . . . .	47
3.3.2 Przykład . . . . .	49
3.3.3 Przykład . . . . .	51
3.3.4 Przykład . . . . .	52
3.3.5 Przykład . . . . .	53
3.3.6 Przykład . . . . .	53
3.3.7 Przykład . . . . .	54
4.2.1 Przykład . . . . .	58
4.2.2 Przykład . . . . .	58
4.3.1 Przykład . . . . .	59
4.3.2 Przykład . . . . .	60
4.4.1 Przykład . . . . .	62
4.5.1 Przykład . . . . .	64
4.6.1 Przykład . . . . .	65
5.2.1 Przykład . . . . .	71
5.2.2 Przykład . . . . .	72
5.2.3 Przykład . . . . .	74

## Spis przykładów

---

5.2.4 Przykład . . . . .	77
5.2.5 Przykład . . . . .	80
6.2.1 Przykład . . . . .	86
6.3.1 Przykład . . . . .	90
6.3.2 Przykład . . . . .	92
6.3.3 Przykład . . . . .	93
6.4.1 Przykład . . . . .	98
6.5.1 Przykład . . . . .	101
6.5.2 Przykład . . . . .	102
7.2.1 Przykład . . . . .	110
7.2.2 Przykład . . . . .	111
7.3.1 Przykład . . . . .	113
7.3.2 Przykład . . . . .	114
7.3.3 Przykład . . . . .	116
7.3.4 Przykład . . . . .	118
7.3.5 Przykład . . . . .	120
7.4.1 Przykład . . . . .	121
7.4.2 Przykład . . . . .	121
7.4.3 Przykład . . . . .	122
7.4.4 Przykład . . . . .	123
8.2.1 Przykład . . . . .	130
8.2.2 Przykład . . . . .	131
8.3.1 Przykład . . . . .	135
8.4.1 Przykład . . . . .	139

## **Spis rysunków**

---

1	Drzewo plików w systemie QNX . . . . .	8
2	Strumienie wejście-wyjście . . . . .	20
3	Zasada działania potoków (pipes) . . . . .	21
4	Platforma rozwoju oprogramowania . . . . .	25
5	Konfiguracja host-target . . . . .	25
6	Idea wskaźnika . . . . .	49
7	Tablica a wskaźnika . . . . .	50
8	Idea działania funkcji <code>fork()</code> . . . . .	61
9	Schemat poprawnego zakończenia procesu . . . . .	63
10	Hierarchia procesów w ćwiczeniu 1 . . . . .	67
11	Hierarchia procesów w ćwiczeniu 3 . . . . .	67
12	Hierarchia procesów w ćwiczeniu 4 dla trzech potomków . . . . .	68
13	Łączenie wątków. Wątek 1 czeka na wątek 2 . . . . .	74
14	Łączenie wątków. Wątek 2 czeka na wątek 1 . . . . .	75
15	Idea aproksymacji liczby $\pi$ . . . . .	83
16	Ilustracja wyścigu (race condition) pomiędzy dwoma wątkami . . . . .	86
17	Zasada działania muteksów . . . . .	88
18	Przykład użycia muteksów . . . . .	90
19	Sytuacja zakleszczenia wątków (deadlock) . . . . .	92
20	Idea działania zmiennej warunkowej . . . . .	98
21	Idea działania bariery . . . . .	100
22	Przykładowy scenariusz przy braku synchronizacji wątków . . . . .	101
23	Idea działania bariery . . . . .	103
24	Idea aproksymacji liczby $\pi$ . . . . .	105
25	Użycie potoku do przekazywania strumieni danych . . . . .	108
26	Użycie potoku do przekazywania strumieni danych . . . . .	113
27	Modularna architektura QNX Neutrino . . . . .	127

---

## Spis tabel

---

28	Model aplikacji typu klient-serwer . . . . .	128
29	Kanały i połączenia . . . . .	128
30	Ilustracja przesyłania komunikatów - przypadek 1 . . . . .	132
31	Ilustracja przesyłania komunikatów - przypadek 2 . . . . .	133
32	Przejścia stanów procesu klienta . . . . .	134
33	Przejścia stanów procesu serwera . . . . .	135
34	Zależności pomiędzy danymi w mechanizmie komunikatów . . . . .	136

---

## Spis tabel

---

1	Typy plików w systemie QNX . . . . .	9
2	Opcje polecenia wc . . . . .	10
3	Poruszanie się po systemie plików . . . . .	10
4	Tworzenie i usuwanie katalogów i plików . . . . .	11
5	Przenoszenie i kopiowanie katalogów i plików . . . . .	11
6	Tworzenie i usuwanie katalogów i plików . . . . .	12
7	Zarządzanie prawami dostępu . . . . .	13
8	Kontrola zadań . . . . .	14
9	Statystyki stanu systemu . . . . .	15
10	Opis pól wyświetlany przez polecenie ps . . . . .	16
11	Usuwanie procesów . . . . .	16
12	Usuwanie procesów . . . . .	16
13	Podstawowe operacje na zmiennych środowiskowych . . . . .	17
14	Wyświetlanie tekstu . . . . .	19
15	Filtrowanie tekstu . . . . .	19
16	Strumienie wejście-wyjście . . . . .	20
17	Wybrane opcje kompilatora qcc . . . . .	42
18	Zmienne standardowe . . . . .	44
19	Zmienne automatyczne . . . . .	45
20	Typy zmiennych . . . . .	46

21	Operatory relacji . . . . .	48
22	Operatory logiczne . . . . .	49
23	Operacje na plikach . . . . .	55
24	Niektóre atrybuty procesów oraz funkcje biblioteki systemowej umożliwiające dostęp do nich . . . . .	58
25	Priorytety w QNX Neutrino 6 . . . . .	58
26	Strategie szeregowania w QNX Neutrino 6 . . . . .	58
27	Metody tworzenia procesów w systemie QNX . . . . .	60
28	Tryby wywołania funkcji <code>spawn()</code> . . . . .	66
29	Wybrane atrybuty wątku i ich wartości domyślne . . . . .	76
30	Funkcje od pobierania i ustawiania atrybutów . . . . .	77
31	Strategie szeregowania w QNX Neutrino . . . . .	79
32	Ważniejsze funkcje dostępu do plików . . . . .	108
33	Wybrane flagi, które można przekazać przez argument <code>oflags</code> . . . . .	109
34	Wybrane flagi, które można przekazać przez argument <code>mode</code> . Umożliwiają one zdefiniowanie praw dostępu do nowo tworzonego pliku ( <code>O_CREAT</code> ) odpowiednio dla właściciela pliku, członków wyróżnionej grupy oraz pozostałych użytkowników . . . . .	109
35	Obsługa kanałów i połączeń . . . . .	128
36	Opcje tworzenia kanałów . . . . .	129

## **Spis tabel**

---

## Piśmiennictwo

---

- [1] M. Ben-Ari. *Podstawy programowania współbieżnego i rozproszonego*. WNT, 2009.
- [2] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
- [3] B. Gough. *An Introduction to GCC*. Addison-Wesley, 2005.
- [4] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- [5] K. Haviland, D. Gray, B. Salama. *Unix Programowanie systemowe*. Wydawnictwo RM, 1999.
- [6] B. Kernigham, D. Ritchie. *Język ANSI C*. WNT, 2004.
- [7] P. Laplante. *Real-Time Systems Design and Analysis*. IEEE Press, Wiley-Interscience, 2004.
- [8] M. Mitchel, J. Oldham, A. Samuel. *Advanced Linux Programming*. New Riders Publishing, 2001.
- [9] QNX. *QNX Documentation Library – podręczniki System Architecture, User's Guide, Getting Started with QNX Neutrino, Programmers Guide* dostępne na stronie [www.qnx.com](http://www.qnx.com). QNX.
- [10] K. Sacha. *Projektowanie oprogramowania systemów sterujących*. Oficyna Wydawnicza Politechniki Warszawskiej, 1999.
- [11] K. Sacha. *Laboratorium systemu QNX*. Oficyna Wydawnicza Politechniki Warszawskiej, 2001.
- [12] R. Stones, N. Matthew. *Beginning Linux Programming*. Wros Press, 2001.
- [13] J. Ułasiewicz. *Systemy czasu rzeczywistego QNX6 Neutrino*. BTC, Warszawa, 2007.