

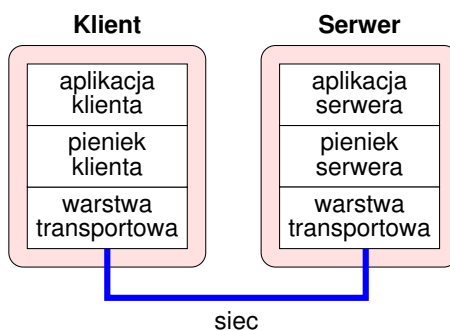
# 1

## Zdalne wywołanie procedur Sun RPC

### 1.1 Wprowadzenie

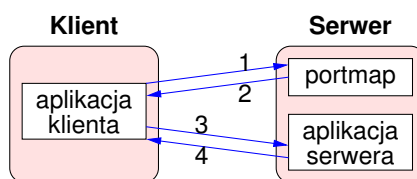
Mechanizm zdalnego wywołania procedur (ang. *remote procedure call*) umożliwia wywoływanie procedur udostępnianych przez zdalne serwery w sposób zapewniający dużą przezroczystość wywołań w stosunku do wywołań procedur lokalnych. Koncepcja wywołań zdalnych została zrealizowana w wielu różnych środowiskach programowych. W ramach ćwiczeń zapoznamy się z konkretną realizacją: Sun RPC. Sun RPC jest zestawem bibliotek programowych i narzędzi wspomagających umożliwiającymi tworzenie aplikacji rozproszonych w języku C.

Rys. 1.1 pokazuje warstwy oprogramowania zaangażowane w realizację zdalnego wywołania procedury. Transparentność (przezroczystość) wywołania zdalnego zapewniają pieńki (ang. *stubs*) po stronie klienta i serwera. Pieniek klienta dostarcza lokalnie interfejsu zdalnej procedury. Po wywołaniu procedury w aplikacji klienta następuje przygotowanie komunikatu zawierającego kod wywołania procedury z argumentami i wysłanie go do serwera. Pieniek serwera odbiera komunikat, odpakowuje argumenty i wywołuje lokalną procedurę zaimplementowaną w aplikacji serwera. Wynik wywołania procedury jest następnie pakowany do nowego komunikatu i przesyłany do klienta. Pieniek klienta odbiera komunikat i przekazuje wyniki do aplikacji. Kod pieńków jest generowany automatycznie, co umożliwia programiście skoncentrowanie się na właściwej funkcjonalności aplikacji.



Rys. 1.1: Warstwy oprogramowania realizujące zdalne wywołanie procedury RPC

Procedury Sun RPC identyfikowane są po numerach. W typowych usługach sieciowych (np. WWW czy FTP), identyfikacja usługi następuje poprzez wskazanie dwóch wartości: adresu IP serwera i numeru portu (TCP lub UDP). W przypadku usługi RPC zdalna procedura identyfikowana jest czwórką wartości: adres IP serwera, numer usługi RPC, numer wersji usługi oraz numer procedury w ramach usługi. Oczywiście w systemie rozproszonym komunikacja ostatecznie musi sprowadzić się do zastosowania protokołów warstwy transportowej, co oznacza, że aplikacja klienta musi mieć możliwość pozyskania w jakiś sposób numerów portów do komunikacji z serwerem. Funkcjonalność tą zapewnia dodatkowa usługa *portmapper* (implementowana jako program o nazwie *portmap*), która pracuje na ogólnie znanym numerze portu (111). Zadaniem usługi jest odwzorowywanie numerów usług RPC na numery portów i nazwy protokołów transportowych poprzez które można się dostać do serwera zdalnej procedury. Wywołanie zdalnej procedury realizowane jest więc zgodnie ze schematem przedstawionym na rys. 1.2. Aplikacja klienta (a dokładnie *pień*) wysyła zapytanie do usługi *portmapper* na port 111 pytając o usługę RPC o konkretnym numerze i wersji (1). Uzyskuje informacje o protokołach transportowych i numerach portów, poprzez które odpowiedni serwer RPC jest osiągalny (2). Następny komunikat zawiera już właściwe zlecenie wywołania zdalnej procedury i jest kierowany do właściwej aplikacji-serwera (3). Serwer RPC po wykonaniu zdalnej procedury odsyła wyniki (4). W kolejnych dowołaaniach procedur można pominąć etap odpytywania usługi *portmapper*, zakładając, że konfiguracja serwera RPC nie zmieniła się.



Rys. 1.2: Schemat wywołania zdalnej procedury w Sun RPC

Szerszy opis RPC można znaleźć w [?]. Dokładna specyfikacja znajduje się w RFC 1050 [?].

## 1.2 Proste wywołanie zdalnej procedury

Wywołanie zdalnej procedury może być realizowane na różnych poziomach szczegółowości. Najwyższy poziom zapewnia maksimum przezroczystości i faktycznie nie różni się od wywołania lokalnej procedury. Zejście niżej ujawnia pewne szczegóły realizacji zdalnego wywołania, dając jednakże możliwość zoptymalizowania jego przebiegu. Najniższy poziom pozwala na uzyskanie niestandardowych schematów realizacji zdalnego wywołania. Kosztem jest tu jednakże konieczność zapoznania się ze szczegółami implementacyjnymi *pieńków* oraz ich ręczna modyfikacja.

Na najwyższym poziomie wywołanie zdalnej procedury RPC można zrealizować z wykorzystaniem funkcji:

```

callrpc(char* hostname, u_long prognum,
        u_long versnum, u_long procnum,
        xdrproc_t inproc, char* in,
        xdrproc_t outproc, char* out)
  
```

Przykładowy klient usługi *rusers* został przedstawiony w przykładzie 1.1.

Kompilację programu można przeprowadzić następującym wywołaniem:

### Przykład 1.1: Klient usługi rusers

---

```
1 #include <stdio.h>
2 #include <rpc/rpc.h>
3 #include <rpcsvc/rusers.h>
4
5 int main(int argc, char **argv)
6 {
7     long nusers;
8     enum clnt_stat stat;
9
10    if (argc <= 1)
11    {
12        printf("Wywołanie: %s <serwer>\n", argv[0]);
13        exit(1);
14    }
15
16    stat=callrpc(argv[1], RUSERSPROG, RUSERSVERS_3, RUSERSPROC_NUM,
17                xdr_void, NULL, xdr_u_long, &nusers);
18    if(stat != RPC_SUCCESS)
19    {
20        clnt_perrno(stat);
21        printf("\n");
22        exit(1);
23    }
24    printf("W systemie %s pracuje %d uzytkownikow.\n",
25          argv[1], nusers);
26 }
```

---

```
# gcc -o rusers rusers.c
rusers.c: In function 'main':
rusers.c:19: warning: passing arg 5 of 'callrpc' from incompatible pointer type
rusers.c:19: warning: passing arg 7 of 'callrpc' from incompatible pointer type
rusers.c:19: warning: passing arg 8 of 'callrpc' from incompatible pointer type
```

Kompilacja kończy się ostrzeżeniami związanymi z niezgodnością typów danych niektórych argumentów. Dla uproszczenia zapisu przykładowego programu zrezygnowano bowiem z rzutowania argumentów do wymaganych przez funkcję `callrpc()` argumentów. Powstały program `rusers` można wykonać podając jako argument nazwę serwera:

```
# rusers localhost
RPC: Program not registered
```

Zgłoszony błąd wynika z nieobecności serwera usługi `rusers`. Każdy program RPC przed właściwym wywołaniem metody zdalnej musi bowiem dokonać odwzorowania numeru usługi RPC na numer portu w odpowiednim protokole transportowym. W powyższym przykładzie następuje odwołanie do usługi identyfikowanej jako `RUSERSPROG`. W pliku `/usr/include/rpcsvc/rusers.h` można znaleźć deklarację identyfikatora `RUSERSPROG` wskazującą na wartość 100002. Aktualnie zarejestrowane usługi można wyświetlić komendą `rpcinfo`:

```
# /usr/sbin/rpcinfo -p
  program vers proto  port
  100000    2  tcp   111  portmapper
  100000    2  udp   111  portmapper
  100007    2  udp   932  ypbind
  100007    1  udp   932  ypbind
  100007    2  tcp   935  ypbind
  100007    1  tcp   935  ypbind
```

W kolumnie `program` wyświetlany jest numer usługi RPC, kolumna `vers` opisuje wersję danej usługi, kolumna `proto` to nazwa protokołu transportowego, a kolumna `port` zawiera numer portu, na którym nasłuchuje odpowiedni serwer. W przykładowym systemie dostępne są więc dwie usługi: `portmapper` i `ypbind`. Serwer usługi `rusers` można uruchomić komendą:

```
# /usr/sbin/rpc.rusersd
```

której wykonanie powoduje uruchomienie w tle procesu o tej samej nazwie. Nowy serwer rejestruje swoją obecność w usłudze `portmapper`:

```
# /usr/sbin/rpcinfo -p
  program vers proto  port
  ...
  100002    3  udp  32769  rusersd
  100002    2  udp  32769  rusersd
```

Program `rusers` wyświetla teraz poprawnie liczbę procesów:

```
# rusers localhost
W systemie localhost pracuje 4 użytkowników.
```

## 1.3 Przykład aplikacji RPC

Poniższa kompletna aplikacja `rkill` ma za zadanie umożliwić zatrzymywanie zdalnych procesów poprzez wysłanie do nich sygnału KILL.

### 1.3.1 Definicja usługi

Definicja jest zapisana w pliku `rkill.x`:

```
program RKILL_PRG {
    version RKILL_VERSION_1 {
        int rkill(int pid) = 1;
    } = 1;
} = 0x21000001;
```

W przykładzie zdefiniowano usługę `RKILL_PRG` o numerze `0x21000001` (dziesiętnie: `553648129`), która występuje w jednej wersji (identyfikowanej jako `RKILL_VERSION_1`). Usługa definiuje jedną funkcję o następującym nagłówku:

```
int rkill(int pid);
```

Wymagany jest więc jeden argument będący identyfikatorem zatrzymywanego procesu. Funkcja zwraca status zakończenia operacji wysłania sygnału.

### 1.3.2 Generator kodu `rpcgen`

Na podstawie przygotowanego pliku definicyjnego można automatycznie wygenerować pliki źródłowe przykładowej aplikacji:

```
# rpcgen -a -N rkill.x
# ls
Makefile.rkill  rkill.x          rkill_clnt.c    rkill_svc.c
rkill.h         rkill_client.c  rkill_server.c
```

Powstałe pliki reprezentują odpowiednio:

<code>rkill.h</code>	plik nagłówkowy z definicjami poszczególnych identyfikatorów,
<code>rkill_client.c</code>	szkielet aplikacji klienckiej,
<code>rkill_server.c</code>	szkielet serwera,
<code>rkill_clnt.c</code>	pień klienta,
<code>rkill_svc.c</code>	pień serwera,
<code>Makefile.rkill</code>	plik <code>Makefile</code> zarządzający kompilacją projektu.

Kompilację najprościej jest przeprowadzić z wykorzystaniem programu `make`<sup>1</sup>:

```
# make -f Makefile.rkill
cc -g -c -o rkill_clnt.o rkill_clnt.c
cc -g -c -o rkill_client.o rkill_client.c
cc -g -o rkill_client rkill_clnt.o rkill_client.o -lnsl
cc -g -c -o rkill_svc.o rkill_svc.c
cc -g -c -o rkill_server.o rkill_server.c
cc -g -o rkill_server rkill_svc.o rkill_server.o -lnsl
```

<sup>1</sup>Zadaniem programu `make` jest przeprowadzenie wszystkich niezbędnych kompilacji cząstkowych projektu w odpowiedniej kolejności z zachowaniem zależności pomiędzy plikami źródłowymi.

W wyniku kompilacji powinny powstać m.in. dwa programy: `rkill_client` i `rkill_server`. Pozwala to na testowe uruchomienie serwera i klienta (np. w różnych oknach środowiska graficznego). Domyślna implementacja serwera nie powoduje jednak wykonania żadnego przetwarzania, stąd trudno zauważyć efekty pracy programu. Obserwację działania serwera umożliwi modyfikacja kodu serwera zawarta w pliku `rkill_server.c`:

Dodanie linii 8 powoduje wyświetlanie komunikatu na ekranie przy każdorazowym wywołaniu zdalnej procedury. Dalsza modyfikacja kodu serwera umożliwia realizację powierzonego mu zadania:

```
result = kill(pid, 9);
```

Linia ta powinna zostać dopisana na pozycji 9. Argument `pid` będzie dostarczony przez bibliotekę RPC, a wartość zwrótna funkcji `kill()` powinna zostać zapisana do statycznej zmiennej `result`, w celu późniejszego przesłania jej do klienta.

Implementacja kodu klienta wymaga wprowadzenia większej ilości zmian. Przykład 1.3 przedstawia zmodyfikowaną wersję klienta:

W linii 39 dodano odwołanie do drugiego argumentu wywołania programu wraz z konwersją do liczby dziesiętnej (funkcja `atoi()`). Dodatkowy argument funkcji `rkill_prg_1()` pojawił się również w nagłówku tej funkcji (linia 5). Identyfikator procesu przekazywany jest dalej do pieńka klienta w linii 19. Po każdorazowej zmianie kodu należy oczywiście wykonać ponownie komendę `make`. W obecnej postaci możliwe jest więc zatrzymanie dowolnego procesu na komputerze z uruchomionym serwerem `rkill`:

```
# rkill_client unixlab 1845
```

gdzie `unixlab` jest przykładową nazwą docelowego serwera.

Dalsze modyfikacje kodu aplikacji klienta powinny umożliwić uzyskanie informacji o przebiegu wykonania wysłania sygnału. Należy w tym celu odwołać się do wartości zwracanej przez funkcję `rkill_1` będącą implementacją pieńka klienta. Wartość ta jest wskaźnikiem na liczbę, która przechowuje zwróconą przez serwer liczbę.

**Dodatkowy argument funkcji `rkill()`.** Jeżeli w wyniku zmiany specyfikacji wymagań należy dodać nowy argument wywołania funkcji `rkill()`, to możliwe są dwa rozwiązania: ręczne korygowanie odpowiednich pieńków i kodów klienta/serwera (niezalecane), lub ponowne wygenerowanie kodu aplikacji (zalecane). Zmodyfikowane wcześniej fragmenty kodu trzeba jednak w drugim przypadku ręcznie wprowadzić do nowej wersji aplikacji. Jako ćwiczenie należy więc zaimplementować usługę o następującym interfejsie:

#### Przykład 1.2: Przykładowa implementacja serwera

---

```
1 #include "rkill.h"
2
3 int *
4 rkill_1_svc(int pid, struct svc_req *rqstp)
5 {
6     static int result;
7
8     printf("Zdalna procedura\n");
9
10    return &result;
11 }
```

---

### Przykład 1.3: Zmodyfikowany klient usługi rkill

---

```
1  #include "rkill.h"
2
3
4  int
5  rkill_prg_1(char *host, int pid)
6  {
7      CLIENT *clnt;
8      int *result_1;
9      int rkill_1_pid;
10
11     #ifndef DEBUG
12     clnt = clnt_create (host, RKILL_PRG, RKILL_VERSION_1, "udp");
13     if (clnt == NULL) {
14         clnt_pcreateerror (host);
15         exit (1);
16     }
17     #endif /* DEBUG */
18
19     result_1 = rkill_1(pid, clnt);
20     if (result_1 == (int *) NULL) {
21         clnt_perror (clnt, "call failed");
22     }
23     #ifndef DEBUG
24     clnt_destroy (clnt);
25     #endif /* DEBUG */
26 }
27
28
29 int
30 main (int argc, char *argv[])
31 {
32     char *host;
33
34     if (argc < 3) {
35         printf ("usage: %s server_host\n", argv[0]);
36         exit (1);
37     }
38     host = argv[1];
39     rkill_prg_1 (host, atoi(argv[2]));
40     exit (0);
41 }
```

---

```

program RKILL_PRG {
  version RKILL_VERSION_1 {
    int rkill(int pid, int sig) = 1;
  } = 1;
} = 0x21000001;

```

**Protokół transportowy.** Kod programu klienta z przykładu 1.3 zawiera w linii 18 wywołanie funkcji `clnt_create()` tworzącej uchwyt komunikacyjny dla warstwy RPC. Parametrem tej funkcji jest nazwa protokołu transportowego. Zmiana domyślnego protokołu UDP na TCP nie spowoduje zmiany funkcjonalnej aplikacji, co pokazuje niezależność mechanizmu RPC od warstwy transportowej.

**Plik Makefile.** Nazwę standardowo generowanego przez `rpcgen` pliku `Makefile.rkill` warto zmienić na `Makefile` aby nie używać przełącznika `-f` komendy `make`. Zawartość tego pliku również warto zmienić, ponieważ znajduje się tam wywołanie komendy `rpcgen` powodujące ponowne tworzenie kodu źródłowego po stwierdzeniu modyfikacji specyfikacji w pliku `rkill.x`. Należy w tym celu usunąć następujące 2 linie:

```

$(TARGETS) : $(SOURCES.x)
  rpcgen $(RPCGENFLAGS) $(SOURCES.x)

```

Drugą zalecaną modyfikacją pliku `Makefile` jest modyfikacja kodu dla operacji `clean`. Standardowo powoduje ona usunięcie kodu źródłowego w języku C! Należy usunąć odwołanie do zmiennej `TARGETS`, uzyskując następujące linie:

```

clean:
  $(RM) core $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)

```

Ostatnia zmiana ma na celu poprawę jakości tworzonego kodu. Zmienna `CFLAGS` ustawiana w pliku `Makefile` umożliwia przesłanie dodatkowych opcji dla kompilatora. Warto włączyć w tym miejscu wypisywanie wszystkich ostrzeżeń:

```

CFLAGS += -g -Wall

```

i doprowadzić kod aplikacji do stanu, w którym kompilacja będzie przebiegała bez żadnych ostrzeżeń.

## Zadania

1. Przetestuj działanie klienta usługi `rusers` z przykładu 1.1.
2. Stwórz aplikację `rkill` korzystając z generatora kodu `rpcgen`. Aplikacja klienta powinna być wywoływana z 2 argumentami: nazwą komputera docelowego i identyfikatorem procesu. Serwer powinien wysyłać sygnał `SIGINT` do wskazanego procesu i zwracać wynik z funkcji `kill()`. Wynik wykonania zdalnej procedury powinien zostać przedstawiony po stronie klienta.
3. Stwórz rozszerzoną aplikację `rkill` udostępniającą dwuargumentową funkcję `rkill()` umożliwiającą wysłanie dowolnego sygnału do dowolnego procesu.
4. Przetestuj działanie aplikacji dla protokołów transportowych UDP i TCP.

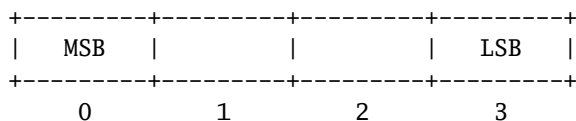
## 1.4 XDR

### 1.4.1 Wprowadzenie

XDR (ang. *eXternal Data Representation*) jest standardem reprezentacji danych niezależnym od architektury sprzętu i języków programowania. Jednocześnie XDR dostarcza narzędzi do konwersji danych z lokalnego formatu do formatu niezależnego. XDR został wprowadzony przez firmę Sun Microsystems równolegle z sieciowym systemem plików NFS (zobacz punkt ??). Szczegółowy opis standardu znajduje się w dokumencie RFC 1014 [?].

Podstawowe zasady kodowania XDR to:

- reprezentacja liczb całkowitych jako 32-bitowe ciągi typu *big-endian*:



- reprezentacja liczb rzeczywistych w formacie IEEE (również 4 bajty)
- reprezentacja wszystkich innych typów zajmuje zawsze wielokrotność 4 bajtów (niektóre bajty mogą zostać wypełnione zerami)
- interpretacja ciągów bajtów pozostawiona jest nadawcy i odbiorcy wiadomości — typy nie są kodowane

Pełna dokumentacja funkcji z biblioteki XDR znajduje się na stronie pomocy systemowej [xdr\(3\)](#).

### 1.4.2 Potoki XDR

Biblioteka XDR składa się ze zbioru funkcji w języku C służących do konwersji podstawowych typów danych do i z standardu XDR. W XDR wyróżnia się następujące dwa pojęcia:

**Potok XDR** jest to strumień danych zakodowanych zgodnie ze standardem XDR. Istnieją trzy typy potoków XDR: potoki na standardowym wejściu/wyjściu, potoki w pamięci i potoki komunikatów.

**Filtr XDR** jest to procedura, której zadaniem jest kodowanie/dekodowanie danych określonego typu. Filtry dla podstawowych typów danych są dostarczone z biblioteką XDR. Nowe filtry można konstruować w oparciu o filtry typów prostych.

**Potok na standardowym wejściu/wyjściu.** Potok taki umożliwia czytanie/pisanie bezpośrednio z pliku. Przykład 1.4 pokazuje kod programu zapisującego do pliku liczbę całkowitą i znak:

W linii 12 tworzony jest potok XDR funkcją `xdrstdio_create()`, której argumentem jest predefiniowana stała `XDR_ENCODE` wskazująca na potok do kodowania (zapisu). W przypadku odczytu należy użyć stałej `XDR_DECODE`. Utworzenie potoku powoduje zainicjowanie struktury XDR, która jest uchwyttem reprezentującym potok. Funkcje `xdr_*` służą do kodowania/dekodowania potoku w zależności od typu potoku. Po wykonaniu programu można przeanalizować jego zawartość w celu weryfikacji zasad kodowania XDR, np. wykonując poniższą komendę:

```
# hexdump -C /tmp/xdr.test
00000000 00 00 01 02 00 00 00 61          |.....a|
```

**Przykład 1.4:** Potok XDR zapisujący do pliku

---

```

1  #include <stdio.h>
2  #include <rpc/xdr.h>
3
4  int main()
5  {
6      FILE* f;
7      XDR   xdrh;
8      int   x = 258;
9      char  c = 'a';
10
11     f = fopen("/tmp/xdr.test", "w");
12     xdrstdio_create(&xdrh, f, XDR_ENCODE);
13     xdr_int(&xdrh, &x);
14     xdr_char(&xdrh, &c);
15     fclose(f);
16 }

```

---

Plik ma więc 8 bajtów. Pierwsze 4 bajty reprezentują liczbę dziesiętną o wartości 258 ( $1 \cdot 256 + 2$ ). Zapis tej samej zmiennej za pomocą funkcji `fwrite()` spowodowałby inne ułożenie bajtów w pliku. Pozostałe 4 bajty pliku reprezentują pojedynczy znak, co powoduje wyzerowanie nieużywanych bajtów.

**Potok w pamięci.** Tworzenie potoku XDR zapisującego dane do bufora w pamięci umożliwia funkcja:

```
void xdrmem_create(XDR* handle, char* addr, int size, xdr_op op);
```

Ten rodzaj potoku ma zastosowanie przy komunikacji procesów poprzez interfejs gniazdek BSD w protokole UDP. Wiadomość przed wysłaniem może być w całości przygotowana w pamięci.

**Potok komunikatów.** Potok ten umożliwia buforowanie danych przekazywanych między procesami. Potoki komunikatów mają zastosowanie przy komunikowaniu procesów poprzez gniazdko TCP. Do tworzenia potoku służy funkcja o poniższym nagłówku:

```
void xdrrec_create(XDR* handle, int sndSize, int rcvSize, char* io,
                  readProc, writeProc);
```

Argument `sndSize` określa rozmiar bufora nadawczego, a `rcvSize` bufora odbiorczego. Parametr `io` identyfikuje mechanizm komunikacyjny (struktura `FILE`, gniazdko BSD). Parametr `readProc` to nazwa procedury, która jest wywoływana gdy w buforze zabraknie danych do odczytu. Analogicznie `writeProc` jest wywoływana gdy brakuje miejsca w buforze nadawczym. Nagłówek funkcji do obsługi bufora wygląda następująco:

```
int fun(char* io, char* buf, int n);
```

gdzie `n` wskazuje ilość bajtów do przesłania. Funkcja powinna zwrócić ilość faktycznie przesłanych danych.

Korzystanie z potoków komunikatów ułatwiają następujące funkcje:

```
xdrrec_endofrecord(XDR* handle, bool_t sendNow)
```

wymuszenie zakończenia komunikatu i jego wysłanie jeżeli parametr `sendNow` ma wartość `TRUE`.

`xdrrec_skiprecord(XDR* handle)`

przejsie do czytania następnego komunikatu z pominięciem pozostałej części komunikatu bieżącego.

`xdrrec_eof(XDR* handle)`

sprawdzenie dostępności danych do odczytu w buforze odbiorczym.

### Inne przydatne funkcje

`int xdr_getpos(XDR* handle)`

zwraca bieżącą pozycję w potoku.

`bool_t xdr_setpos(XDR* handle, int pos)`

ustawienie pozycji w potoku.

### 1.4.3 Filtry XDR

Filtr jest funkcją, której zadaniem jest kodowanie i dekodowanie określonych typów danych. Istnieją filtry proste, złożone i pochodne. Filtry proste umożliwiają kodowanie typów prostych języka C, np. `char`, `int`, np.:

```
bool_t xdr_int(XDR* handle, int* value);
```

Drugim argumentem filtru jest zawsze wskaźnik do danej określonego typu.

**Filtry złożone** służą do konwersji danych złożonych z typów prostych, a więc np. łańcuchów znaków czy tablic. Obsługiwane są następujące typy danych:

<code>string</code>	łańcuch znaków
<code>opaque</code>	tablica bajtów o ustalonej wielkości
<code>bytes</code>	tablica bajtów o zmiennej wielkości
<code>vector</code>	tablica danych dowolnego typu o ustalonej wielkości
<code>array</code>	tablica danych dowolnego typu o zmiennej wielkości
<code>union</code>	unia
<code>reference</code>	wskaźnik
<code>pointer</code>	wskaźnik rozpoznający wskaźnik pusty NULL

**Filtry pochodne** są tworzone w oparciu o inne istniejące filtry. Przykładem może być filtr do kodowania struktury danych.

**Zarządzanie pamięcią.** Odczytując złożoną strukturę danych odbiorca może nie wiedzieć ile ona będzie zajmować miejsca w pamięci. Alokację pamięci może jednak przeprowadzić automatycznie biblioteka XDR. Oto przykład dla łańcuchów tekstowych:

```
XDR xdr;
char *buf;
...
buf = NULL;
xdrstdio_create(&xdr, f, XDR_DECODE);
xdr_string(&xdr, &buf, 1024);      /* odczyt napisu z potoku */
...
xdr_free(xdr_string, &buf);
```

Użyta funkcja `xdr_free()` służy do zwalniania pamięci zaalokowanej przez procedury XDR. Parametr pierwszy tej procedury to wskaźnik na odpowiedni filtr XDR.

#### 1.4.4 Filtry XDR tworzone przez `rpcgen`

Filtry pochodne XDR mogą być tworzone automatycznie przez generator kodu `rpcgen`. Wymaga to przygotowania odpowiedniej specyfikacji zbliżonej notacją do składni języka C. W poniższym przykładzie zostanie utworzony filtr do kodowania struktury DANE:

```
struct DANE
{
    int    x;
    char   c;
    float  f[10];
};
```

Struktura DANE składa się z pola `x` typu `int`, pojedynczego znaku `c` i 10-elementowej tablicy `f` przechowującej liczby zmiennoprzecinkowe. Generowanie odpowiednich filtrów realizuje polecenie `rpcgen`:

```
# rpcgen DANE.x
```

co powoduje powstanie plików `DANE.h` i `DANE_xdr.c` zawierających odpowiednio definicje typów na poziomie języka C i implementację filtru XDR. Pliki te zostały przedstawione w przykładach 1.5 i 1.6:

Język programu `rpcgen` pozwala na definiowanie tablic o zmiennym rozmiarze. Następujące zapisy mogą znaleźć się w specyfikacji struktury danych:

```
x[10]    10-elementowa tablica,
x<10>    tablica o maksymalnym rozmiarze równym 10,
x<>      tablica o nieokreślonym rozmiarze.
```

Wykorzystanie tablic o dynamicznym rozmiarze powoduje wygenerowanie struktur danych języka C dla każdej dynamicznej tablicy. W przypadku wspomnianej struktury DANE jeżeli tablica `f` miałaby określony tylko maksymalny rozmiar to definicja typu w języku C wyglądałaby następująco:

```
struct DANE {
    int x;
    char c;
    struct {
        u_int f_len;
        float *f_val;
    } f;
};
```

W strumieniu XDR w takim przypadku przed wypisaniem zawartości tablicy zostanie najpierw umieszczony jej rozmiar.

Nowe typy danych oraz stałe definiowane są na poziomie specyfikacji dla `rpcgen` podobnie jak w języku C:

```
const MAX = 100;

typedef int Tablica<MAX>;
```

### Przykład 1.5: Plik nagłówkowy dla filtru XDR

---

```
1 #ifndef _DANE_H_RPCGEN
2 #define _DANE_H_RPCGEN
3
4 #include <rpc/rpc.h>
5
6
7 #ifdef __cplusplus
8 extern "C" {
9 #endif
10
11
12 struct DANE {
13     int x;
14     char c;
15     float f[10];
16 };
17 typedef struct DANE DANE;
18
19 /* the xdr functions */
20
21 #if defined(__STDC__) || defined(__cplusplus)
22 extern bool_t xdr_DANE (XDR *, DANE*);
23
24 #else /* K&R C */
25 extern bool_t xdr_DANE ();
26
27 #endif /* K&R C */
28
29 #ifdef __cplusplus
30 }
31 #endif
32
33 #endif /* !_DANE_H_RPCGEN */
```

---

### Przykład 1.6: Implementacja filtru XDR

---

```
1 #include "DANE.h"
2
3 bool_t
4 xdr_DANE (XDR *xdrs, DANE *objp)
5 {
6     register int32_t *buf;
7
8     int i;
9     if (!xdr_int (xdrs, &objp->x))
10         return FALSE;
11     if (!xdr_char (xdrs, &objp->c))
12         return FALSE;
13     if (!xdr_vector (xdrs, (char *)objp->f, 10,
14                     sizeof (float), (xdrproc_t) xdr_float))
15         return FALSE;
16     return TRUE;
17 }
```

---

**Wskaźniki.** Definicja typów dla programu `rpcgen` pozwala na definiowanie wskaźników, które będą następnie poprawnie odtwarzane przy odczycie. Wspomniana struktura `DANE` mogłaby więc być uzupełniona o wskazanie na następny element:

```
struct DANE {
    int x;
    DANE* next;
};
```

Do kodowania wskaźników wykorzystywany jest filtr `xdr_pointer()`.

## Zadania

1. Napisz program dekodujący dane zapisane w pliku z przykładu 1.4.
2. Sprawdź w jaki sposób odbywa się binarne kodowanie łańcuchów znaków.
3. Napisz program zapisujący do pliku i odczytujący strukturę o następujących polach: liczba typu `int`, tablica znaków o długości 5, liczba zmiennoprzecinkowa.
4. Napisz program, który zapisze do pliku listę rekordów przechowujących liczby typu `int`. Zdefiniuj w tym celu strukturę zawierającą wskaźnik na następny rekord. Sprawdź zachowanie biblioteki XDR w przypadku zapisu listy cyklicznej.
5. Napisz program klienta i serwera aplikacji udostępniającej informacje o pliku: rozmiar pliku, identyfikator właściciela, daty modyfikacji i dostępu, itp.
6. Zaproponuj implementację udoskonalonej funkcji `xdr_pointer()` obsługującej dowolnie złożone (a więc również i cykliczne) dynamiczne struktury danych.

## 1.5 Asynchroniczne RPC

### 1.5.1 Przebieg zdalnego wywołania

Biblioteka RPC po wywołaniu zdalnej procedury oczekuje standardowo około 25 sekund na odpowiedź. Po upływie tego czasu pieńek klienta zgłasza wystąpienie błędu przekroczenia czasu oczekiwania (*timeout*) pomimo, że odpowiedź nadchodzi. Można to zweryfikować wstawiając do implementacji zdalnej metody wywołanie funkcji:

```
sleep(30);
```

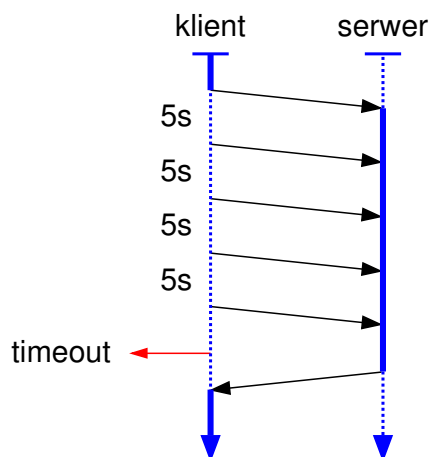
powodującej wydłużenie czasu realizacji zdalnego wywołania. Wykonanie programu klienta kończy się w takiej sytuacji błędem:

```
# rkill_client localhost 2314
call failed: RPC: Timed out
```

Błąd wynika oczywiście z przekroczenia czasu oczekiwania na odpowiedź. Ciekawa jest jednak reakcja serwera:

```
# rkill_server
Zdalna procedura
Zdalna procedura
Zdalna procedura
Zdalna procedura
Zdalna procedura
```

Okazuje się, że zdalna procedura została wywołana aż 5 razy. Jest to wynik strategii stosowanej podczas korzystania z bezpołączeniowego protokołu transportowego UDP. Wysłanie komunikatu UDP do serwera i brak odpowiedzi po upływie czasu określonego parametrem `RETRY_TIMEOUT` powoduje wysłanie kolejnego komunikatu (zobacz rys. 1.3). Domyślna wartość parametry `RETRY_TIMEOUT` wynosi 5 sekund, co powoduje, że w ciągu 25 sekund klient zdąży wysłać 5 komunikatów z żądaniem. Po upływie czasu określonego parametrem `TIMEOUT` następuje zasygnalizowanie błędu warstwie wyższej, czyli aplikacji. Serwer obsługuje poszczególne żądania sekwencyjnie, ale warstwa komunikacyjna systemu operacyjnego odbiera przesyłane komunikaty i buforuje je, co powoduje ich późniejsze wykonanie.

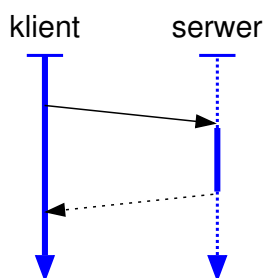


**Rys. 1.3:** Schemat komunikacji z serwerem po protokole UDP dla długotrwałych procedur

Inne zachowanie biblioteki RPC można zaobserwować gdy protokołem transportowym będzie protokół TCP. W takiej sytuacji klient nie retransmituje żądania, ponieważ ma gwarancję, że komunikat dotarł do serwera. Błąd przekroczenia czasu spowoduje przerwanie zdalnego wywołania, ale serwer odbierze jedynie pojedyncze żądanie.

### 1.5.2 Proste wywołanie asynchroniczne

Jeżeli nie można określić maksymalnego czasu wykonania zdalnej procedury, pozostaje wykonanie wywołania asynchronicznego, czyli takiego, w którym serwer ma dowolnie dużo czasu na realizację żądania. Asynchronizm powoduje również, że klient nie jest blokowany realizacją zdalnego wywołania i może kontynuować swoje przetwarzanie. Rysunek 1.4 przedstawia schemat komunikacji w przypadku wywołania asynchronicznego.



**Rys. 1.4:** Wywołanie asynchroniczne

Oczywistą konsekwencją wywołania asynchronicznego jest niemożliwość przekazania wyniku zdalnej procedury do klienta. Oznacza to, że definicja usługi z punktu 1.3.1 musi zostać zmieniona do postaci:

```
program RKILL_PRG {
    version RKILL_VERSION_1 {
        void rkill(int pid) = 1;
    } = 1;
} = 0x21000001;
```

Najprostszą realizacją asynchronicznych wywołań jest modyfikacja domyślnego czasu oczekiwania na odpowiedź. Czas oczekiwania klienta jest zdefiniowany w nagłówku pieńka klienta poprzez statyczną strukturę `TIMEOUT`:

```
static struct timeval TIMEOUT = { 25, 0 };
```

W strukturze `timeval` pole `tv_sec` określa liczbę sekund, a pole `tv_usec` liczbę mikrosekund (zobacz np. stronę pomocy systemowej `utimes(3)`). Czas ten można również zmodyfikować w pliku implementacyjnym klienta korzystając z funkcji `clnt_control()`<sup>2</sup>. W przypadku przykładu 1.3 należy w tym celu zmienić plik `rkill_client.c` w miejscu, gdzie jest tworzony uchwyt komunikacyjny:

```
struct timeval tm = { 60, 0 };
...
clnt = clnt_create (host, RKILL_PRG, RKILL_VERSION_1, "udp");
...
clnt_control(clnt, CLSET_TIMEOUT, &tm);
```

Szczególną zmianą czasu oczekiwania jest ustawienie go na wartość 0, co w praktyce oznacza ignorowanie odpowiedzi i permanentne zgłaszanie błędu `timeout`. Błąd przekroczenia czasu oczekiwania powinien oczywiście być ignorowany:

```
struct rpc_err err;
...
result_1 = rkill_1(rkill_1_pid, clnt);
if (result_1 == NULL) {
    clnt_geterr(clnt, &err);
    if (err.re_status != RPC_TIMEDOUT) {
        clnt_perror (clnt, "call failed");
    }
}
```

Funkcja `clnt_geterr()` umożliwia pobranie szczegółowych informacji o zaistniałym błędzie<sup>3</sup>. Struktura `rpc_err` posiada pole `re_status`, które przechowuje kod zaistniałego błędu. W powyższym przykładzie wyświetlana będzie więc informacja o wszystkich błędach oprócz `RPC_TIMEDOUT`.

Przykład 1.7 zawiera pełen kod implementacji klienta wykorzystującego asynchroniczne wywołanie:

### 1.5.3 Modyfikacja pieńka serwera

Ciekawą, choć niezalecaną oficjalnie implementacją asynchronicznego wywołania procedury jest modyfikacja pieńka serwera tak, aby wysyłał odpowiedź na żądanie klienta *zanim* wykona zdalną procedurę. Zabezpiecza to oczywiście klienta przed błędem typu `timeout`

<sup>2</sup>Funkcja `clnt_control()` umożliwia również zmianę parametru `RETRY_TIMEOUT`.

<sup>3</sup>Jest to analogia do zmiennej globalnej `errno` przechowującej kod błędu ostatnio wywołanej funkcji systemowej. Zobacz również stronę pomocy systemowej `errno(3)`.

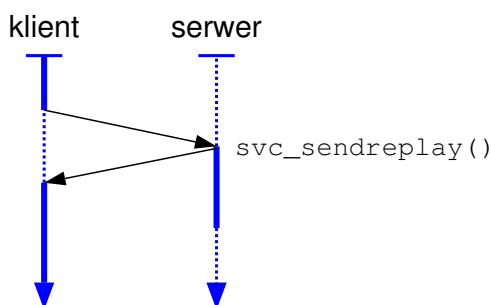
**Przykład 1.7:** Implementacja klienta wykorzystującego asynchroniczne wywołanie procedury

---

```
1 #include "rkill.h"
2
3 void
4 rkill_prg_1(char *host, int pid)
5 {
6     CLIENT *clnt;
7     void *result_1;
8     struct timeval tm = { 0, 0 };
9     struct rpc_err err;
10
11 #ifndef DEBUG
12     clnt = clnt_create (host, RKILL_PRG, RKILL_VERSION_1, "udp");
13     if (clnt == NULL) {
14         clnt_pcreateerror (host);
15         exit (1);
16     }
17     clnt_control(clnt, CLSET_TIMEOUT, (char*)&tm);
18 #endif /* DEBUG */
19
20     result_1 = rkill_1(pid, clnt);
21     if (result_1 == (void *) NULL) {
22         clnt_geterr(clnt, &err);
23         if (err.re_status != RPC_TIMEDOUT) {
24             clnt_perror (clnt, "call failed");
25         }
26     }
27 #ifndef DEBUG
28     clnt_destroy (clnt);
29 #endif /* DEBUG */
30 }
31
32
33 int
34 main (int argc, char *argv[])
35 {
36     char *host;
37
38     if (argc < 3) {
39         printf ("usage: %s server_host pid\n", argv[0]);
40         exit (1);
41     }
42     host = argv[1];
43     rkill_prg_1 (host, atoi(argv[2]));
44     exit (0);
45 }
```

---

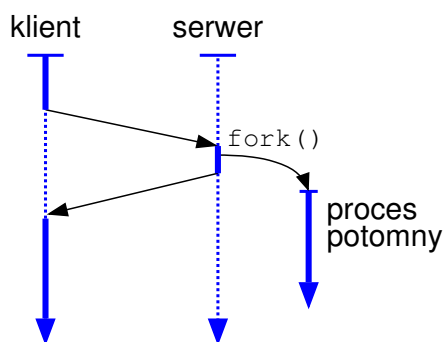
i dodatkowo daje mu potwierdzenie poprawnego wywołania procedury po stronie serwera (zobacz rys. 1.5). Podobnie jednak jak w przykładzie z zerowym czasem oczekiwania na odpowiedź nie będzie możliwe przesłanie wyniku procedury. Przykład 1.8 pokazuje zmodyfikowaną procedurę pieńka serwera. Wewnątrz instrukcji `switch` (od linii 26) następuje zainicjowanie zmiennych reprezentujących filtry kodujące argument i wynik zdalnej procedury oraz zmiennej wskazującej na procedurę, która ma być wywołana. Linia 50 zawiera wywołanie zdalnej procedury. Wcześniej jednak (linia 46) następuje odesłanie odpowiedzi. Jako wynik przetwarzania przesyłany jest pusty wskaźnik `NULL`. W przypadku usług, które udostępniają większą liczbę funkcji obsługę asynchronicznych wywołań należałoby przenieść do wnętrza instrukcji `switch`.



Rys. 1.5: Wywołanie asynchroniczne z wczesnym odesłaniem odpowiedzi

#### 1.5.4 Procesy potomne

Ostatnia propozycja realizacji wywołania asynchronicznego polega na wykorzystaniu procesów potomnych do realizacji samej procedury. Ponieważ proces główny serwera będzie zajmował się jedynie odbiorem żądań i tworzeniem nowych procesów, jego odpowiedź będzie trafiała natychmiast do klientów (zobacz rys. 1.6). Całe przetwarzanie (potencjalnie długie) odbędzie się w nowym procesie potomnym. Umożliwia to współbieżne wykonywanie wielu zdalnych procedur przez pojedynczy serwer. Przykład 1.9 prezentuje implementację zdalnej procedury takiego serwera. Pełen kod serwera musi uwzględnić całościowo problem zarządzania procesami potomnymi, a więc m.in. problem zarządzania procesami *zombie*.



Rys. 1.6: Wywołanie asynchroniczne z procesem potomnym

Procesy potomne lub przetwarzanie wielowątkowe<sup>4</sup> można wykorzystać również w przypadku synchronicznego wywołania procedur w celu zrównoleżenia przetwarzania po stro-

<sup>4</sup>Zobacz punkt 1.8.

**Przykład 1.8:** Implementacja procedury z pieńka serwera dla asynchronicznego wywołania z wczesnym odesłaniem odpowiedzi (bez funkcji main())

---

```
1 #include "rkill.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <rpc/pmap_clnt.h>
5 #include <string.h>
6 #include <memory.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9
10 static void *
11 _rkill_1 (int *argp, struct svc_req *rqstp)
12 {
13     return (rkill_1_svc(*argp, rqstp));
14 }
15
16 static void
17 rkill_prg_1(struct svc_req *rqstp, register SVCXPRT *transp)
18 {
19     union {
20         int rkill_1_arg;
21     } argument;
22     char *result;
23     xdrproc_t _xdr_argument, _xdr_result;
24     char *(*local)(char *, struct svc_req *);
25
26     switch (rqstp->rq_proc) {
27     case NULLPROC:
28         (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
29         return;
30
31     case rkill:
32         _xdr_argument = (xdrproc_t) xdr_int;
33         _xdr_result = (xdrproc_t) xdr_void;
34         local = (char *(*)(char *, struct svc_req *)) _rkill_1;
35         break;
36
37     default:
38         svcerr_noproc (transp);
39         return;
40     }
41     memset ((char *)&argument, 0, sizeof (argument));
42     if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
43         svcerr_decode (transp);
44         return;
45     }
46     if (!svc_sendreply(transp, (xdrproc_t) _xdr_result, NULL)) {
47         svcerr_systemerr (transp);
48     }
49     else {
50         result = (*local)((char *)&argument, rqstp);
51     }
52     if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
53         fprintf (stderr, "%s", "unable to free arguments");
54         exit (1);
55     }
56     return;
57 }
```

---

**Przykład 1.9:** Implementacja zdalnej procedury korzystająca z procesu potomnego

---

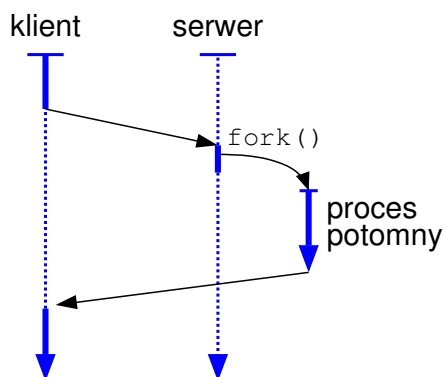
```

1  #include "rkill.h"
2
3  void *
4  rkill_1_svc(int pid,  struct svc_req *rqstp)
5  {
6      if (fork()==0)
7      {
8          printf("Zatrzymywanie procesu %d.\n", pid);
9          kill(pid, 9);
10         sleep(30);
11         exit(0);
12     }
13
14     return 1;
15 }

```

---

nie serwera i jednocześnie minimalizacji czasu odpowiedzi dla krótkich żądań. Wymaga to modyfikacji pieńka serwera w celu utworzenia nowego procesu (wątku) dla każdego wywołania zdalnej procedury. Schemat przetwarzania zdalnego wywołania procedury w takim podejściu został zaprezentowany na rys. 1.7.



Rys. 1.7: Wywołanie synchroniczne z procesem potomnym

## Zadania

1. Przetestuj obsługę zdalnego wywołania metody `rkill()` po dodaniu do jej implementacji opóźnienia 30-sekundowego.
2. Sprawdź obsługę zdalnego wywołania w przypadku użycia protokołu transportowego TCP.
3. Zaimplementuj wywołanie asynchroniczne poprzez ustawienie zerowego czasu odczekiwania na odpowiedź. Aplikacja klienta nie powinna sygnalizować żadnego błędu, jeżeli przesłanie żądanie udaje się przesłać.
4. Zmodyfikuj pieńek serwera standardowej implementacji zdalnego wywołania metody (z domyślnym czasem oczekiwania na odpowiedź), tak aby odsyłał potwierdzenie wykonania metody zaraz po odebraniu żądania. Sprawdź jak realizowana będzie

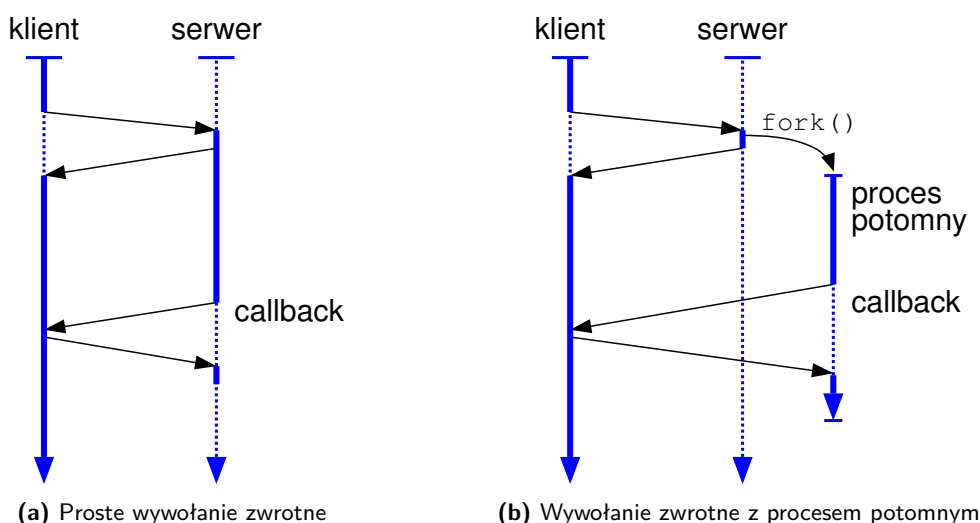
teraz współbieżna obsługa długich, 30-sekundowych żądań pochodzących od dwóch różnych klientów.

5. Zmodyfikuj implementację z poprzedniego punktu przenosząc obsługę żądań klientów do procesów potomnych. Ponownie zweryfikuj obsługę współbieżnych żądań.
6. Stwórz implementację serwera, w której wykorzystywane będą procesy potomne przy realizacji synchronicznych wywołań procedur. Implementacja taka powinna umożliwiać jednocześnie wywoływanie tej samej procedury przez wielu klientów, umożliwiając przy tym odbiór wyników z tych procedur.

## 1.6 Wywołanie zwrotne

### 1.6.1 Wprowadzenie

Omówione dotąd modele wywołań zdalnych nie dają możliwości odbioru wyniku działania procedury zdalnej w przypadku dowolnie długotrwałego przetwarzania. Zwrócenie wyniku może jednak zostać zrealizowane jako wywołanie zwrotne, w którym serwer, po zakończeniu przetwarzania, wywołuje procedurę zdalną klienta. Rysunek 1.8a pokazuje przykład realizacji takiego wywołania. W wywołaniu zwrotnym klient staje się na pewien czas serwerem, aby móc odebrać wynik zdalnej procedury. Istotną zaletą wywołania zwrotnego jest możliwość nieprzerwanej pracy klienta. Z reguły wymaga to jednak uruchomienia dodatkowego wątku, którego zadaniem będzie oczekiwanie na odbiór wyniku z serwera.



Rys. 1.8: Wywołanie zwrotne

Wywołanie zwrotne można oczywiście łączyć z obsługą żądań w procesach potomnych co zobrazowano na rysunku 1.8b. Takie podejście zostanie zaprezentowane w następujących przykładach.

### 1.6.2 Przykład aplikacji

Jako przykład rozważmy znów serwer `rkill` z punktu 1.3.1:

```
program RKILL_PRG {
    version RKILL_VERSION_1 {
```

```

    void rkill(int pid, int sig) = 1;
} = 1;
} = 0x20000000;

```

Zwróćmy uwagę, że funkcja `rkill()` nie zwraca żadnej wartości. Definicję serwera należy uzupełnić o definicję wywołania zwrotnego klienta (plik `rkillcb.x`):

```

program RKILL_CB_PRG {
    version RKILL_CB_VERSION_1 {
        void sendresult(int res) = 1;
    } = 1;
} = 0x40000000;

```

Serwer ten definiuje funkcję `sendresult()`, której zadaniem będzie przesłanie wyniku przetwarzania do klienta. Funkcja ta również nie zwraca żadnego wyniku.

Definicje serwerów należy przetworzyć generatorem kodu `rpcgen`:

```

# rpcgen -N -a rkill.x
# rpcgen -N -a rkillcb.x

```

Oto lista modyfikacji jakie należy wprowadzić do wygenerowanych plików:

1. Usunięcie pliku `Makefile.rkillcb`. Całość sterowania kompilacją znajdzie się w pliku `Makefile.rkill`.
2. Modyfikacja pliku `Makefile.rkill`. Należy uzupełnić listę modułów programowych wchodzących w skład kodu klienta i serwera dopisując odpowiednie wartości do zmiennych `TARGETS_SVC.c` i `TARGETS_CLNT.c`:

```

TARGETS_SVC.c = ... rkillcb_clnt.c rkillcb_client.c
TARGETS_CLNT.c = ... rkillcb_svc.c rkillcb_server.c

```

Powyższa modyfikacja wskazuje na to, że aplikacja klienta będzie się składać z właściwego klienta (m.in. pliki `rkill_client.c` i `rkill_clnt.c`) oraz z serwera wywołania zwrotnego (pliki `rkillcb_server.c` i `rkillcb_svc.c`). Analogicznie w skład aplikacji serwera wejdzie dodatkowo klient wywołania zwrotnego (pliki `rkillcb_client.c` i `rkillcb_clnt.c`).

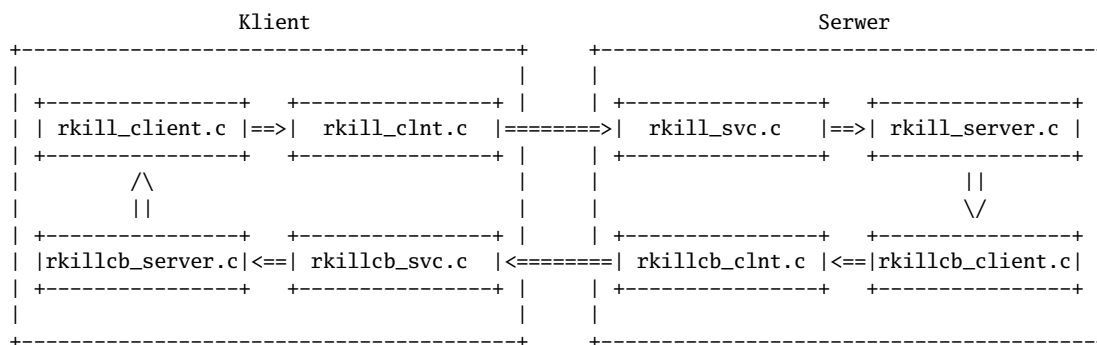
3. Przekazywanie argumentów po stronie klienta (plik `rkill_client.c`). Należy odczytać argumenty z linii poleceń i przekazać je poprzez funkcję `rkill_prg_1()` aż do pieńka klienta czyli do funkcji `rkill_1()`.
4. Modyfikacja klienta wywołania zwrotnego (plik `rkillcb_client.c`): całkowite usunięcie definicji funkcji `main()`, zmiana nazwy funkcji `rkill_cb_prg_1()` na `rkill_cb_1()`<sup>5</sup>, wyprowadzenie argumentu `res` na zewnątrz i dodanie deklaracji tej funkcji do pliku nagłówkowego `rkillcb.h`. Funkcja `main()` dla serwera zdefiniowana jest w pliku `rkill_svc.c`.
5. Implementacja właściwego serwera (przedstawiona w przykładzie 1.10). Uwaga: istotne jest dołączenie odpowiednich plików nagłówkowych.
6. Implementacja serwera wywołania zwrotnego. Zadaniem serwera jest odbiór wyniku zdalnej procedury i wyświetlenie go na ekranie.
7. Zmiana nazwy funkcji `main()` w pieńku serwera wywołania zwrotnego (plik `rkillcb_svc.c`) na `main2()` i dodanie jej deklaracji do pliku nagłówkowego `rkillcb.h`. Definicja funkcji `main()` dla aplikacji klienta znajduje się w pliku `rkill_client.c`.

<sup>5</sup>Dla uniknięcia konfliktu z funkcją o tej samej nazwie z pieńka serwera wywołania zwrotnego.

8. Rozszerzenie implementacji właściwego klienta. Klient po wywołaniu zdalnej procedury (`rkill_prg_1()`) może po prostu wykonać kod serwera wywołania zwrotnego (funkcja `main2()` z pliku `rkillcb_svc.c`).

Przeprowadzenie powyższych zmian może wymagać modyfikacji plików nagłówkowych oraz ich dołączania do implementacji poszczególnych fragmentów kodu.

Przeptyw sterowania w ramach poszczególnych plików składowych aplikacji został przedstawiony na rysunku 1.9.



Rys. 1.9: Przeptyw sterowania w wywołaniu zwrotnym RPC

### 1.6.3 Procedura `svc_run()`

Po wprowadzeniu powyższych modyfikacji klient spowoduje wywołanie zdalnej procedury, serwer wykona ją, wywoła procedurę zwrotną po stronie klienta, ale przetwarzanie po stronie klienta nie zakończy się. Wynika to z charakteru pracy standardowego serwera, którego zadaniem jest praca w pętli nieskończonej. Implementacja klienta korzysta ze standardowego serwera (funkcja `main2()` w pliku `rkillcb_svc.c`). Serwer RPC wywołuje funkcję `svc_run()`, której zadaniem jest nieskończone oczekiwanie i obsługa żądań RPC. Klient korzystający z wywołania zwrotnego powinien oczekiwać tylko na jedno wywołanie RPC od serwera, co wymaga zmodyfikowania funkcji `svc_run()` przedstawionej w przykładzie 1.11.

Procedura `svc_run2()` monitoruje funkcją `select()` deskryptory wskazane zmienną `svc_fdset` i po stwierdzeniu odbioru nowego żądania wywołuje funkcję `svc_getreqset()` przetwarzającą wywołanie. Zewnętrzna zmienna `done` informuje procedurę czy nastąpiło już poprawne wywołanie procedury, na którą oczekuje klient. W linii 23 następuje wyjście z procedury `svc_run2()` po poprawnym obsłużeniu wywołania RPC. Pieniek serwera wywołania zwrotnego (plik `rkillcb_svc.c`) powinien więc wywoływać funkcję `svc_run2()` zamiast standardowej `svc_run()`. Zmienna `done` powinna być inicjowana wewnątrz aplikacji klienta (plik `rkill_client.c`) i modyfikowana w implementacji procedury zwrotnej (plik `rkillcb_server.c`). Wymaga to dodania deklaracji tej zmiennej do pliku nagłówkowego `rkillcb.h` i definicji do kodu klienta. Poprawna kompilacja kodu klienta wymaga również dołączenia definicji funkcji `svc_run2()` do kodu klienta, co oznacza konieczność wskazania w pliku `Makefile` nazwy pliku z implementacją:

```
TARGETS_CLNT.c = ... svc_run2.c
```

Deklaracja funkcji `svc_run2()` powinna również trafić do pliku nagłówkowego `rkillcb.h`.

### Przykład 1.10: Implementacja serwera z wywołaniem zwrotnym

---

```
1 #include <unistd.h>
2 #include <signal.h>
3 #include <arpa/inet.h>
4 #include "rkill.h"
5 #include "rkillcb.h"
6
7 void *
8 rkill_1_svc(int pid, int sig, struct svc_req *rqstp)
9 {
10     static char * result = NULL;
11
12     printf("rkill(%d,%d)\n", pid, sig);
13     if (fork()==0)
14     {
15         int res;
16         sleep(3);
17         res = kill(pid, sig);
18         rkill_cb_1(inet_ntoa(rqstp->rq_xprt->xp_raddr.sin_addr), res);
19         printf("rkill(%d,%d) ==> %d\n", pid, sig, res);
20         exit(0);
21     }
22
23     return (void *) &result;
24 }
```

---

### Przykład 1.11: Zmodyfikowana procedura svc\_run()

---

```
1 #include <rpc/rpc.h>
2 #include <errno.h>
3 #include "rkillcb.h" /* tu znajduje sie deklaracja zmiennej done */
4
5 void svc_run2()
6 {
7     fd_set readfds;
8
9     for (;;)
10    {
11        readfds = svc_fdset;
12        switch (select(_rpc_dtablesize(), &readfds, (fd_set*)NULL, (fd_set*)NULL,
13                    (struct timeval*)NULL))
14        {
15            case -1:
16                if (errno == EINTR) continue;
17                perror("svc_run: - select failed");
18                return;
19            case 0:
20                continue;
21            default:
22                svc_getreqset(&readfds);
23                if (done) return;
24        }
25    }
26 }
```

---

Funkcja `svc_run()` standardowo nie kończy się nigdy, dlatego pieńek serwera wyświetla błąd po jej zakończeniu. Komunikat ten, jak również i występujące po nim zakończenie procesu funkcją `exit()`, należy oczywiście usunąć.

#### 1.6.4 Rejestracja tymczasowej usługi

Klient uruchamiając lokalny serwer musi go zarejestrować w usłudze `portmap`. Rejestracja powinna odbywać się z użyciem tymczasowego numeru usługi RPC, gdyż w przeciwnym wypadku może wystąpić konflikt pomiędzy użytkownikami uruchamiającymi swoje aplikacje na jednym komputerze. Rejestracji usługi dokonuje pieńek serwera zawarty w pliku `rkillcb_svc.c` (funkcja `main2()` z punktu 1.6.2). Przykład 1.12 prezentuje funkcję `register_tmp()` rejestrującą tymczasowy serwer RPC. Jest to zmodyfikowana wersja funkcji `main()` standardowego pieńka serwera. Funkcja `pmap_set()` jest odpowiedzialna za rejestrację usługi RPC w usłudze `portmap`. Pętla `while` w linii 5 dokonuje próbnej rejestracji pod kolejnymi numerami począwszy od `0x40000000`.

Aplikacja klienta przed wywołaniem zdalnej metody powinna zarejestrować procedurę RPC do wywołania zwrotnego, następnie oczekiwać w funkcji `svc_run2()` i na końcu wyrejestrować tymczasowo zarejestrowaną usługę.

Numer tymczasowo zarejestrowanej usługi musi trafić do serwera, aby mógł on przekazać zwrotnie wynik przetwarzania do właściwego klienta. Najprościej można to zrobić podczas samego wywołania zdalnej procedury. Definicja usługi `rkill` musi więc ulec modyfikacji:

```
program RKILL_PRG {
    version RKILL_VERSION_1 {
        void rkill(int prognum, int pid, int sig) = 1;
    } = 1;
} = 0x20000000;
```

Wymagana będzie więc zmiana implementacji klienta wywołania zwrotnego (plik `rkillcb_client.c`) w celu przekazania mu właściwego numeru usługi RPC. Ostateczna implementacja funkcji `main()` w aplikacji klienta wygląda więc następująco:

```
int
main (int argc, char *argv[])
{
    char *host;
    int prognum;
    int pid;
    int sig;

    if (argc < 4) {
        printf ("usage: %s server_host pid sig\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    pid = atoi(argv[2]);
    sig = atoi(argv[3]);
    done = 0;

    SVCXPRT *transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL)
    {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
    }
}
```

**Przykład 1.12:** Dynamiczna rejestracja usługi

---

```

1  int register_tmp(SVCXPRT *transp)
2  {
3      long prognum = 0x40000000;
4
5      while(pmap_set(prognum, RKILL_CB_VERSION_1, IPPROTO_UDP, transp->xp_port)==0)
6      {
7          prognum++;
8      }
9      if (!svc_register(transp, prognum, RKILL_CB_VERSION_1,
10                     rkill_cb_prg_1, IPPROTO_UDP))
11     {
12         fprintf(stderr, "%s", "unable to register transient procedure.");
13         exit(1);
14     }
15     return prognum;
16 }

```

---

```

    }
    prognum = register_tmp(transp);
    rkill_prg_1(host, prognum, pid, sig);
    svc_run2();
    svc_unregister(prognum, RKILL_CB_VERSION_1);
    svc_destroy(transp);
    exit (0);
}

```

Funkcja `svc_unregister()` wyrejestrowuje tymczasową usługę RPC, a funkcja `svc_destroy()` zwalnia zasoby uchwytu komunikacyjnego.

Dla poprawnej kompilacji projektu deklaracja funkcji `register_tmp()` powinna zostać dołączona do pliku nagłówkowego `rkillcb.h`.

**Zadania**

1. Zweryfikuj poprawność wykonania aplikacji klienta w przypadku współbieżnego uruchamiania wielu kopii na jednym komputerze.
2. Zrealizuj wielowątkową implementację klienta z wywołaniem zwrotnym. Zadaniem dodatkowego wątku ma być oczekiwanie na odpowiedź z serwera.
3. Zrealizuj aplikację rozpraszającą obliczenia na N komputerów z wykorzystaniem RPC. Komputery obliczeniowe powinny udostępniać zdalne wywołanie wykonujące fragment obliczeń i zwracające wyniki poprzez wywołanie zwrotne. Aplikacja klienta prezentuje ostateczny wynik po odebraniu wyników cząstkowych od wszystkich N serwerów.

**1.7 Kontrola praw dostępu**

Zdalne wywołania mogą być uzupełnione o informację identyfikującą użytkownika. Poniższy fragment kodu powinien zostać umieszczony w implementacji klienta:

```
clnt = clnt_create(...);
```

```
...
clnt->cl_auth = authunix_create_default();
```

Funkcja `authunix_create_default()` powoduje dołączenie do wywołania procedury informacji o nazwie komputera klienta, identyfikatorze użytkownika (UID) i identyfikatorze jego grupy (GID). Po stronie serwera informacje te dostępne są poprzez strukturę `svc_req` przekazywaną do zdalnej struktury:

```
struct authunix_parms *aup;
aup = rqstp->rq_clntcred;
printf("Komputer: %s\n", aup->aup_machname);
printf("UID      : %d\n", aup->aup_uid);
printf("GID      : %d\n", aup->aup_gid)
```

## 1.8 Wątki w standardzie POSIX

### 1.8.1 Wprowadzenie

Wiele nowoczesnych systemów operacyjnych umożliwia dekompozycję złożonych zadań nie tylko na rozłączne, współpracujące ze sobą procesy, ale również na wątki. Wątek w systemie operacyjnym to niezależny strumień przetwarzania pracujący w ramach jednego procesu z innymi wątkami. Wątki współdzielą między sobą przestrzeń adresową procesu, mają więc dostęp do tych samych danych statycznych i dynamicznych, ale posiadają swój własny stos umożliwiający im niezależne wykonywanie procedur. Poniższy punkt zawiera opis interfejsu programistycznego w standardzie POSIX. Więcej informacji można znaleźć w pracy [?] oraz dokumentacji [?].

Przykład 1.13 prezentuje aplikację tworzącą nowy wątek realizujący proste przetwarzanie. Kluczowym elementem programu jest wywołanie funkcji `pthread_create()` tworzącej nowy wątek:

```
int pthread_create(pthread_t * thread,
                  pthread_attr_t * attr,
                  void * (*start_routine)(void *),
                  void * arg);
```

Nowy wątek powstaje jako współbieżne wywołanie istniejącej w programie procedury. Procedura taka powinna pobierać wskaźnik `void*` jako argument i zwracać taką wartość jako wynik przetwarzania. Argument funkcji implementującej wątek umożliwia parametryzowanie wątków. Aktualna wartość Parametru przekazywana jest ostatnim argumentem wywołania `pthread_create()`. Poprawne utworzenie wątku inicjuje zmienną typu `pthread_t`, która jest liczbą typu `unsigned int`. Wartość ta staje się unikalnym w ramach procesu identyfikatorem wątku. Wskaźnik do struktury `pthread_attr_t` umożliwia zainicjowanie pewnych atrybutów wątku (m.in. rozmiar stosu, algorytm szeregowania, priorytet).

Wszystkie funkcje implementujące obsługę wątków mają nazwy zaczynające się od `pthread_` i są udokumentowane w sekcji 3p standardowej pomocy systemowej. Każdy program korzystający z wątków powinien dołączyć plik nagłówkowy `pthread.h`. Kompilacja programu korzystającego z wątków wymaga dołączenia biblioteki `pthread`:

```
# gcc -o thread thread.c -lpthread
```

W systemie Linux wątki POSIX są implementowane na poziomie systemu operacyjnego. Ich obecność można zaobserwować korzystając np. z opcji `H` standardowej komendy `ps`:

**Przykład 1.13:** Przykład programu tworzącego nowy wątek

---

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <pthread.h>
4  #include <string.h>
5
6  void* work(void* arg)
7  {
8      printf("Praca wątku\n");
9      sleep(3);
10     return NULL;
11 }
12
13 int main()
14 {
15     pthread_t th;
16
17     printf("Tworzenie wątku...\n");
18     pthread_create(&th, NULL, work, NULL);
19     printf("Wątek uruchomiony...\n");
20     pthread_join(th, NULL);
21     printf("Wątek zakończony.\n");
22     return 0;
23 }

```

---

```

# ./thread &
# ps lxH
F  UID    PID  PPID  PRI  NI     VSZ   RSS  WCHAN  STAT  TTY     TIME  COMMAND
...
0  501    6446  6445   15   0    4360   1936  wait   Ss    pts/2   0:00  /bin/bash
0  501    6457  6446   15   0    5652    460  347332  Sl    pts/2   0:00  ./thread
1  501    6457  6446   16   0    5652    460  -      Sl    pts/2   0:00  ./thread
0  501    6460  6446   15   0    2504    772  -      R+    pts/2   0:00  ps lxH

```

Proces o identyfikatorze 6457 jest procesem wielowątkowym. Składają się na niego wątek główny i dwa wątki poboczne.

## 1.8.2 Zarządzanie wątkami

Wątek może oczekiwać na zakończenie innego wątku funkcją:

```
int pthread_join(pthread_t th, void **thread_return);
```

Jest to funkcja analogiczna do funkcji `wait()` oczekującej na zakończenie procesu potomnego. Wskaźnik `thread_return` zostanie zainicjowany wartością zwróconą przez wątek.

Wykonanie wątku można zakończyć w dowolnym momencie wywołaniem funkcji:

```
void pthread_exit(void *retval);
```

Wartość `retval` może być odczytana przez inny wątek, który zsynchronizuje się z nim wywołaniem `pthread_join()`.

Wątek można zatrzymać z poziomu innego wątku funkcją `pthread_cancel()`:

```
int pthread_cancel(pthread_t thread);
```

Zatrzymywanie wątku można blokować funkcją:

```
int pthread_setcancelstate(int state, int *oldstate);
```

Wątek może „odłączyć” się od procesu i kontynuować pracę niezależnie od wątku głównego. Służy do tego funkcja:

```
int pthread_detach(pthread_t th);
```

Uniezależnienie wątku oznacza, że jego zasoby będą zwolnione po jego zakończeniu. Z drugiej jednak strony nie będzie możliwe zsynchronizowanie z innym wątkiem poprzez wywołanie `pthread_join()`.

### 1.8.3 Obsługa sygnałów

Wszystkie wątki współdzielą między sobą jedną tablicę z adresami procedur obsługi. Każdy wątek może zmieniać przypisaną wcześniej do sygnału procedurę standardową funkcją `signal()`. Istnieje możliwość wysyłania sygnałów do poszczególnych wątków:

```
int pthread_kill(pthread_t thread, int signo);
```

Wątki mogą blokować odbiór określonych sygnałów używając funkcji:

```
int pthread_sigmask(int how, const sigset_t *newmask,
                   sigset_t *oldmask);
```

gdzie `newmask` jest maską sygnałów (4 liczby typu `unsigned long`) tworzoną z użyciem funkcji:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Wywołanie funkcji `pthread_sigmask()` z argumentem drugim ustawionym na `NULL` powoduje odczytanie aktualnej maski sygnałów.

W ramach synchronizacji wątków można wymusić oczekiwanie na określony sygnał:

```
int sigwait(const sigset_t *set, int *sig);
```

### 1.8.4 Synchronizacja wątków

Do synchronizacji wątków wykorzystywane są następujące mechanizmy: zamki (ang. *mutex*), zmienne warunkowe (ang. *conditional variable*) i semafony POSIX.

#### 1.8.4.1 Zamki

Zamki są binarnymi semaforami, a więc mogą znajdować się w jednym z dwóch stanów: podniesiony lub opuszczony. Do inicjowania zamka wykorzystywana jest funkcja:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Poniższy fragment kodu tworzy zamek z domyślnymi atrybutami:

```
pthread_mutex_t m;
...
pthread_mutex_init(&m, NULL);
```

Do wykonywania operacji na zamkach służą następujące funkcje:

`pthread_mutex_lock()`

Zajęcie zamka. Funkcja jest blokująca do czasu aż operacja może zostać wykonana.

`pthread_mutex_trylock()`

Zajęcie wolnego zamka. Próba zajęcia już zajętego zamka kończy się zasygnalizowaniem błędu (EBUSY).

`pthread_mutex_unlock()`

Zwolnienie zamka. Zwolnienia powinien dokonać wątek, który zajął zamek.

`pthread_mutex_destroy()`

Usunięcie zamka.

Wszystkie wymienione funkcje pobierają wskaźnik do struktury `pthread_mutex_t` jako jedyny argument i zwracają wartość typu `int`. Przykład 1.14 pokazuje wykorzystanie zamków do implementacji prostego wzajemnego wykluczania.

#### 1.8.4.2 Zmienne warunkowe

Zmienne warunkowe pozwalają na kontrolowane zasypianie i budzenie procesów w momencie zajścia określonego zdarzenia.

Poniższy fragment kodu tworzy zmienną warunkową z domyślnymi atrybutami:

```
pthread_cond_t c;
...
pthread_cond_init(&c, NULL);
```

Budzenie wątków realizowane jest z wykorzystaniem jednej z dwóch poniższych funkcji:

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

Funkcja `pthread_cond_signal()` budzi co najwyżej jeden wątek oczekujący na wskazanej zmiennej warunkowej. Funkcja `pthread_cond_broadcast()` budzi wszystkie wątki uśpione na wskazanej zmiennej warunkowej.

Kolejne dwie funkcje służą do usypiania wątku na zmiennej warunkowej:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Funkcja `pthread_cond_wait()` oczekuje bezwarunkowo do czasu odebrania sygnału budzącego, podczas gdy funkcja `pthread_cond_timedwait()` ogranicza maksymalny czas oczekiwania. Zaśnięcie wątku wymaga użycia jednocześnie zmiennej warunkowej i zamka. Przed zaśnięciem zamek musi być już zajęty. Zaśnięcie oznacza atomowe zwolnienie zamka i rozpoczęcie oczekiwania na sygnał budzący. Obudzenie wątku powoduje ponowne zajęcie zamka. Poniższy przykład pokazuje zastosowanie zmiennej warunkowej do synchronizacji wątków:

- wątek oczekujący

```
pthread_cond_t c;
pthread_mutex_t m;
...
pthread_mutex_lock(&m);      /* zajęcie zamka */
```

**Przykład 1.14:** Proste zastosowanie zamków

---

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <pthread.h>
4
5 pthread_mutex_t mtx;
6
7 void* work(void* arg)
8 {
9     int id = pthread_self();
10    int i;
11    for(i=0; i<10; i++)
12    {
13        printf("[%d] Czekam...\n", id);
14        pthread_mutex_lock(&mtx);
15        printf("[%d] Sekcja krytyczna...\n", id);
16        sleep(1);
17        printf("[%d] Wyjście...\n", id);
18        pthread_mutex_unlock(&mtx);
19        usleep(100000);
20    }
21    return NULL;
22 }
23
24 int main()
25 {
26    pthread_t th1, th2;
27
28    pthread_mutex_init(&mtx, NULL);
29    pthread_create(&th1, NULL, work, NULL);
30    pthread_create(&th2, NULL, work, NULL);
31    pthread_join(th1, NULL);
32    pthread_join(th2, NULL);
33
34    return 0;
35 }
```

---

```
pthread_cond_wait(&c, &m);    /* oczekiwanie na zmiennej warunkowej */
pthread_mutex_unlock(&m);    /* zwolnienie zamka */
```

- wątek budzący

```
pthread_cond_signal(&c);    /* sygnalizacja zmiennej warunkowej */
```

Pomimo, że nie jest to wymagane, często do poprawnej synchronizacji wywołuje się funkcję budzącą podczas przetrzymywania zamka:

```
pthread_mutex_lock(&m);
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

### 1.8.4.3 Semafor

Do synchronizacji wątków można wykorzystać semafor standardu POSIX. Jest to rozwiązanie całkowicie niezależne od semaforów Systemu V będących częścią zestawu mechanizmów komunikacji międzyprocesowej IPC. Semafor POSIX jest licznikiem przyjmującym wartości od zera wzwyż. Do obsługi semaforów POSIX przewidziano następujące funkcje:

#### `sem_init()`

Funkcja tworząca i inicjująca nowy semafor. Semafor może być strukturą wewnętrzną procesu lub może służyć do synchronizacji niezależnych procesów, podobnie jak semafor IPC<sup>6</sup>. Argumentem funkcji `sem_init()` jest początkowa wartość semafora.

#### `sem_wait()`

Oczekiwanie na wartość semafora większą od zera i obniżenie jej o 1.

#### `sem_trywait()`

Nieblokująca próba zmniejszenia wartości semafora o 1.

#### `sem_post()`

Zwiększenie wartości semafora o 1. Operacja zawsze wykonuje się poprawnie i nie jest blokująca.

#### `sem_getvalue()`

Pobranie aktualnej wartości semafora.

#### `sem_destroy()`

Usunięcie semafora.

Przykład 1.15 prezentuje proste zastosowanie semaforów standardu POSIX. Jest to oczywiście zmodyfikowana wersja przykładu 1.14.

## Zadania

1. Przeciwicz przekazywanie argumentów do wątku i odbiór wartości zwrotnych. Uruchom w tym celu 3 nowe wątki, które będą zwracały podwojoną wartość typu `int`.
2. Utwórz 2 wątki i wstrzymaj ich wykonanie zmienną warunkową. Wątek główny powinien wznowić ich pracę sygnalizując to odpowiedniej zmiennej warunkowej. Sprawdź różnicę pomiędzy funkcją `pthread_cond_signal()` a `pthread_cond_broadcast()`.
3. Zaproponuj implementację operacji `bariera()`, której działanie polegałoby na zsynchronizowaniu wskazanej liczby wątków. Operacja kończy się w momencie jej wywołania przez wszystkie wątki.

<sup>6</sup>Aktualna implementacja LinuxThreads nie wspiera semaforów synchronizujących niezależne procesy.

### Przykład 1.15: Proste zastosowanie semaforów POSIX

---

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5
6 sem_t s;
7
8 void* work(void* arg)
9 {
10     int id = pthread_self();
11     int i;
12     for(i=0; i<10; i++)
13     {
14         printf("[%d] Czekam...\n", id);
15         sem_wait(&s);
16         printf("[%d] Sekcja krytyczna...\n", id);
17         sleep(1);
18         printf("[%d] Wyjście...\n", id);
19         sem_post(&s);
20         usleep(100000);
21     }
22     return NULL;
23 }
24
25 int main()
26 {
27     pthread_t th1, th2;
28
29     sem_init(&s, 0, 1);
30     pthread_create(&th1, NULL, work, NULL);
31     pthread_create(&th2, NULL, work, NULL);
32     pthread_join(th1, NULL);
33     pthread_join(th2, NULL);
34
35     return 0;
36 }
```

---

4. Zaproponuj synchronizację 2 wątków w problemie producenta-konsumenta. Przeanalizuj czy rozwiązanie to działa poprawnie również w przypadku większej liczby producentów i konsumentów.
5. Zastosuj przetwarzanie wielowątkowe po stronie klienta w celu uruchomienia serwera wywołania zwrotnego. Główny wątek klienta powinien kontynuować przetwarzanie, okresowo sprawdzając dostępność wyniku wywołania zdalnej procedury.
6. Zastosuj przetwarzanie wielowątkowe po stronie serwera do obsługi żądań klientów obsługiwanych z wykorzystaniem wywołań zwrotnych. Każde żądanie powinno powodować uruchomienie nowego wątku, który jest odpowiedzialny za obsługę tego żądania. Na końcu wykonywania zdalnej procedury następuje wywołanie zwrotne do klienta i wątek powinien się zakończyć.