IV. Programowanie współbieżne

Język C nie zawiera oczywiście żadnych mechanizmów umożliwiających programowanie współbieżne (np. takich, jak w języku Ada). W rozdziale niniejszym przedstawię implementację modułu umożliwiającego pseudo-współbieżne wykonywanie funkcji w języku C. Termin "pseudo-współbieżność" jest tutaj bardzo ważny, gdyż w żadnym wypadku zastosowane rozwiązanie nie umożliwia realizacji rzeczywistej współbieżności na maszynach wieloprocesorowych. Pomimo tego, dla zwiększenia czytelności opisu, będę w dalszej jego części używał terminów "współbieżny" oraz "pseudo-współbieżny" wymiennie.

Implementacja wspomnianego modułu będzie pretekstem do zastosowania wielu technik opisanych w poprzednich rozdziałach tej książki. Dlatego też przed przystąpieniem do czytania tego rozdziału polecam przeczytanie rozdziałów poprzednich. Z drugiej strony implementacja ta jest przykładem niecodziennego stylu programowania w języku C, charakteryzującego się bardzo intensywnym użyciem preprocesora.

Współbieżne wykonywanie funkcji nie będzie realizowane na poziomie systemu operacyjnego lecz na poziomie programu w C, i będzie ono wykonane z wykorzystaniem jedynie elementów języka standardowego. Oznacza to między innymi, że moduł będzie przenośny zarówno na różne kompilatory (należy jednak ostrożnie stosować opcje optymalizacji) jak i różne platformy sprzętowe. Inną konsekwencją realizacji przełączania zadań całkowicie na poziomie języka C jest "gruboziarnistość" zrealizowanej pseudo-współbieżności. Jeżeli dwa (lub więcej) zadania (funkcje, programy) mają być wykonywane w sposób współbieżny na jednym procesorze, to w rzeczywistości na zmianę wykonywane są pewne małe fragmenty tych zadań. Najmniejszą taką cząstką może być instrukcja procesora, która nie może już być podzielona. Takie rozwiązanie byłoby pseudo-współbieżnością "drobnoziarnistą" i gwarantowałoby maxymalne złudzenie rzeczywistej współbieżności. W przedstawionym poniżej module najmniejszą częścią funkcji, która musi być wykonana, zanim sterowanie zostanie przekazane do innej funkcji, jest jedna instrukcja (ang. *statement*) języka C.

1. Dlaczego współbieżność?

Klasyczne programy współbieżne są wykonywane na maszynach wieloprocesorowych. Celem zastosowania równoległych komputerów i równoległych programów jest zmniejszenie złożoności czasowej rozwiązywanego zadania. Jest sprawą oczywistą, że w przypadku programu wykonywanego pseudo-współbieżnie na komputerze jednoprocesorowym nie można liczyć na zwiększenie prędkości obliczeń. Co więcej, wykonanie w takim przypadku kilku zadań musi trwać dłużej niż trwałoby wykonanie tych zadań sekwencyjnie jedno po drugim. Dzieje się tak dlatego, że oprócz kodu zadań procesor musi wykonywać pewien kod związany

z ich przełączaniem. Można by w takim razie powiedzieć, że pseudowspółbieżność jest sztuką dla sztuki. Nie jest to prawdą, a najlepszym na to dowodem jest popularność programów typu DESQview czy Windows, umożliwiających pseudowspółbieżne wykonywanie programów. Twierdzenie, że wielozadaniowość realizowana na jednym procesorze nie może przynieść zysków czasowych jest prawdą dopóty, dopóki zadania cały czas wymagają pracy procesora. W rzeczywistości częste są sytuacje, gdy większość czasu pracy zadania nie jest zużywana na pracę procesora. Na przykład operacje na pamięci zewnętrznej są zwykle na tyle wolne w porównaniu z szybkością procesora, że mógłby on równocześnie wykonywać inną pracę. Jeszcze bardziej skrajnym przypadkiem jest czekanie przez zadanie na dane wprowadzane przez użytkownika z klawiatury. Jeżeli jedno z zadań utknęło w takim wąskim gardle, procesor może poświęcić swój czas na wykonanie innych zadań. Można to zrealizować właśnie poprzez pseudowspółbieżność.

Zyskiwanie czasu w takich sytuacjach nie jest jednak jedynym motywem zastosowania wielozadaniowości. Programy wykonujące kilka zadań na raz mogą być bardzo wygodne dla użytkownika. Przykładem niech będzie edytor tekstów zapisujący co jakiś czas redagowany tekst "w tle".

Jak już wcześniej wspomniałem, realizacja współbieżnego wykonywania funkcji będzie polegała na wykonywaniu na zmianę kolejnych fragmentów każdej z funkcji. Do przekazywania sterowania z jednej funkcji do drugiej posłużą nam funkcje **setjmp** i **longjmp**.

2. Funkcje setjmp i longjmp

Deklaracje tych funkcji wyglądają następująco:

```
int setjmp(jmp_buf);
void longjmp(jmp_buf,int);
```

Są one funkcjami standardowymi. Ich deklaracje, oraz deklaracja typu jmp_buf znajdują się w pliku nagłówkowym "**setjmp.h**". Służą one do zapamiętania, a następnie odtworzenia stanu programu. W praktyce oznacza to, że funkcja **longjmp** umożliwia wykonanie skoku do jakiegoś miejsca, w którym stan programu został wcześniej zapamiętany przy pomocy funkcji **setjmp**. Jak wskazuje sama nazwa funkcji, jest to skok daleki, nie ograniczony do wnętrza funkcji (jak skok przy pomocy instrukcji **goto**). Typ jmp_buf definiuje strukturę, w której przechowywane są informacje o stanie programu.

Wywołanie funkcji **setjmp** powoduje zapamiętanie w zmiennej typu jmp_buf, przekazanej do niej jako argument, informacji o stanie programu. Funkcja zwraca wartość 0.

Funkcję **longjmp** wywołuje się z dwoma argumentami. Pierwszym jest zmienna, w której wcześniej zapamiętano stan programu przy pomocy funkcji **setjmp**. Drugi argument jest liczbą całkowitą. Wywołanie funkcji **longjmp** powoduje

odtworzenie stanu programu jaki został zapamiętany w zmiennej typu jmp_buf przekazanej jako pierwszy argument. W wyniku takiego wywołania funkcji **longjmp** program znajduje się w punkcie powrotu z funkcji **setjmp** (bo w takim momencie został zapamiętany stan programu), przy czym wartość zwracana przez funkcję **setjmp** jest równa drugiemu argumentowi funkcji **longjmp** (lub 1 jeżeli drugi argument był równy 0). Na podstawie wartości funkcji **setjmp**, program jest w stanie odróżnić czy została ona normalnie wywołana w wyniku zinterpretowania kolejnej instrukcji, czy też nastąpił skok przy pomocy funkcji **longjmp**.

Działanie funkcji **setjmp** i **longjmp** jest czasami trudne do zrozumienia. Poniższy przykład powinien wyjaśnić niejsności.

```
if(setjmp(buf))
 {
   /* ciąg instrukcji */
 }
 /* ... */
 longjmp(buf,3);
```

Po wywołaniu funkcji **setjmp** warunek nie będzie spełniony (setjmp zwróci wartość 0) i *ciąg instrukcji* nie zostanie wykonany. W efekcie wywołania funkcji **longjmp** w innej części programu, sterowanie zostanie przekazane do miejsca powrotu z funkcji **setjmp**, ale tym razem zostanie zwrócona wartość równa 3, a więc warunek będzie spełniony i *ciąg instrukcji* zostanie wykonany.

Ponieważ po "powrocie" funkcja **setjmp** zwraca wartość przekazaną jako argument funkcji **longjmp**, nic nie stoi na przeszkodzie, żeby przy pomocy różnych funkcji **longjmp** przekazywać różne wartości i na ich postawie identyfikować miejsce, z którego nastąpił daleki skok, na przykład przy pomocy instrukcji **switch**:

```
switch(setjmp(buf))
 {
   case 1 : /* z punktu 1 */ break;
   case 2 : /* z punktu 2 */ break;
   case 3 : /* z punktu 3 */ break;
 }
```

3. Przełączanie zadań

Zastanówmy się na początek, w jaki sposób dokonać przełączenia procesora pomiędzy dwiema funkcjami. Jak wcześniej napisałem, użyjemy pary funkcji **setjmp** i **longjmp** do wykonania dalekich skoków pomiędzy procesami (funkcjami). Dla każdego procesu będziemy potrzebować jednej zmiennej typu jmp_buf służącej do zapamiętania stanu programu w momencie przekazania sterowania do drugiego procesu. Obie zmienne muszą być globalne, aby obie funkcje mogły się do nich odwołać.

```
jmp_buf buf1,buf2;
```

W celu wykonania skoku do funkcji **f1** będziemy używać wywołania

```
longjmp(buf1,1);
```

Analogicznie, żeby skoczyć do funkcji **f2**

```
longjmp(buf2,1);
```

Przed wykonaniem skoku do drugiego procesu, każda funkcja musi wywołać funkcję **setjmp**(buf). Zapamiętane przez tę funkcję informacje o stanie programu będą mogły być w przyszłości wykorzystane do powrotu do miejsca, w którym działanie funkcji zostało zawieszone. Tak więc sekwencje przełączające zadania będą wyglądały mniej więcej tak:

```
if(setjmp(buf1)==0)longjmp(buf2,1);   /* w funkcji f1 */
if(setjmp(buf2)==0)longjmp(buf1,1);   /* w funkcji f2 */
```

Funkcja **longjmp** zostanie wywołana tylko wtedy, gdy setjmp zwróci wartość zero. Nastąpi to więc po wywołaniu setjmp w celu zapamiętania kontekstu programu, a nie nastąpi po powrocie w to miejsce przy pomocy dalekiego skoku.

Spróbujmy teraz uogólnić to rozwiązanie na nieznaną z góry ilość procesów. Trzeba zdefiniować jakąś strukturę danych, która zapewniałaby istnienie jednego bufora typu jmp_buf dla każdej funkcji, a także umożliwiałaby określenie jaka jest następna funkcja w "łańcuszku".

```
struct el {
            jmp_buf buf;
            struct el *next;
          };
```

Każdą funkcja będzie posiadała własny element typu struct el. W polu buf tego elementu będzie zapamiętywany kontekst tej funkcji w chwili przełączania sterowania do kolejnego zadania. Pole next struktury będzie wskazywało element typu struct el skojarzony z funkcją, do której ma być przekazane sterowanie. W ten sposób powstanie zapętlona lista o węzłach typu struct el. Lista jest jednokierunkowa, gdyż każdy jej element zawiera tylko pole wskazujące następny element. Do pełnej manipulacji listą jednokierunkową (w tym do usuwania elementów z listy) potrzebne są co najmniej dwie zmienne, wskazujące na dwa kolejne węzły listy:

```
struct el *cur,*last;
```

Zmienna cur będzie zawsze wskazywać na węzeł odpowiadający aktywnej funkcji, zaś zmienna last - na węzeł odpowiadający poprzedniej funkcji. Sekwencja przełączania zadań zapisana przy użyciu tych zmiennych będzie wyglądała następująco:

```
   if(setjmp(cur->buf)==0)
longjmp((last=cur,cur=(cur->next))->buf,1);
```

Argumentem funkcji **longjmp** jest dość skomplikowane wyrażenie:

```
(last=cur,cur=(cur->next))->buf
```

Analiza tego wyrażenia rozpocznie się od przypisania zmiennej last wartości zmiennej cur. Następnie zmiennej cur jest przypisywany wskaźnik do następnego

węzła listy. Pole buf tego węzła zostaje argumentem funkcji **longjmp** (zostanie wykonany skok do następnej funkcji).

Pola buf w liście muszą być zainicjowane przed pierwszym wywołaniem sekwencji przełączającej zadania. Żeby to osiągnąć, umieścimy na początku każdej funkcji następujący warunek:

```
if(setjmp(cur->buf)==0)return;
```

Jeżeli funkcja zostanie wywołana, to jej kontekst zostanie zapamiętany w polu cur->buf i nastąpi od razu powrót do funkcji wywołującej.

Pozostaje jeszcze zastanowić się, co zrobić po zakończeniu działania procesu. Trzeba go oczywiście usunąć z listy, żeby nie był więcej wykonywany. Dokonuje się tego instrukcją

```
cur=last->next=cur->next;
```

W tym miejscu przydaje się zadeklarowana wcześniej na wyrost zmienna last. Następnie trzeba przekazać sterowanie do kolejnego procesu:

```
longjmp(cur->buf,1);
```

4. Zapis praktyczny

Przedstawione powyżej konstrukcje robią dobry użytek z funkcji **setjmp** i **longjmp** umożliwiając przełączanie funkcji - zadań, ale w żadnym wypadku nie nadają się do praktycznego zastosowania w programowaniu.

Stosując definicje preprocesora można zapisać te sekwencje w sposób dużo czytelniejszy. Załóżmy, że zapis ten musi spełniać następujące warunki:

- zamiana napisanej i uruchomionej wcześniej funkcji na postać, w której mogłaby ona być wykonywana współbieżnie, musi być prosta, prawie automatyczna,
- funkcja przystosowana do wykonywania współbieżnego powinna dalej móc być wywoływana w normalny sposób,
- jeżeli funkcja jest ostatnim procesem (wszystkie inne zakończyły już działanie) to nie powinna być wykonywana sekwencja przełączania zadań.

Pierwszym krokiem będzie zastąpienie omówionych w poprzednim podrozdziale sekwencji odpowiednimi definicjami preprocesora:

```
#define BEGIN  if(setjmp(cur->buf)==0)return;
#define END    cur=last->next=cur->next;         \
               longjmp(cur->buf,1);
#define _      if(setjmp(cur->buf)==0)           \
            longjmp((last=cur,cur=(cur->next))->buf,1);
```

Makrodefinicje BEGIN oraz END będą umieszczane odpowiednio na początku i na końcu funkcji. Ostatnie makro jest właściwą sekwencją przełączającą zadania. Ideałem byłaby sytuacja gdyby to makro w ogóle nie było widoczne, dlatego też wybrałem dla niego nazwę jak najmniej rzucającą się w oczy: _ (podkreślenie).

Jak już wcześniej ustaliliśmy, wszystkie makrodefinicje mają być "przeźroczyste" gdy funkcja zostanie wywołana w normalny sposób. Do rozróżniania czy funkcja jest wykonywana współbieżnie czy nie, użyjemy lokalnej zmiennej is_a_process. Zmienna ta będzie miała wartość 1 tylko wtedy, gdy funkcja zostanie wywołana przez funkcję inicjującą procesy.

Definicje rozbudowane o sprawdzanie, czy funkcja jest wykonywana współbieżnie wyglądają następująco:

```
        #define BEGIN     {
\
                           static char is_a_process;
\
                           if((is_a_process=be_a_process)!=0)
\
                            if(setjmp(cur->buf)==0)return;
        #define END    if(is_a_process!=0)                \
                       {                                  \
                        cur=last->next=cur->next;         \
                        longjmp(cur->buf,1);              \
                       }                                  \
                      } /* zamyka nawias otwarty w BEGIN */

        #define _     if(is_a_process!=0)                      \
                      if(setjmp(cur->buf)==0)                  \
longjmp((last=cur,cur=(cur->next))->buf,1);
```

Zmienna be_a_process jest zmienną globalną. Jej wartość wynosi cały czas zero i jest ustawiana na jeden przez funkcję inicjującą procesy na czas inicjującego wywołania procesu. Funkcja inicjująca przydziela pamięć na strukturę struct el dla procesu i umieszcza ją w liście.

```
void  proces(FUNCTION f)
{
 struct el *tmp;
 _number++ ;
 tmp=malloc(sizeof(struct el));
 if(cur)
    {tmp->next=cur->next; cur->next=tmp; }
  else
    {cur=tmp; cur->next=tmp;}
 last=cur;
 cur=tmp;
 be_a_process=1;
 (*f)();
 be_a_process=0;
}
```

FUNCTION jest typem wskazanie do funkcji typu **void**:

```
typedef void (*FUNCTION)(void);
```

W funkcji proces występuje globalna zmienna _number. Jak wskazuje nazwa, jej wartość będzie określała liczbę "żywych" procesów. Proces będzie jej używał do sprawdzenia czy nie jest już przypadkiem ostatnim żywym procesem.

Po zainicjowaniu wszystkich procesów można rozpocząć ich współbieżne wykonywanie. W tym celu zdefiniujemy makro RUN:

```
#define RUN    if(setjmp(_the_end)==0)         \
                 longjmp((cur=cur->next)->buf,1);
```

Globalna zmienna _the_end służy do zapamiętania punktu, z którego zostało wywołane współbieżne wykonywanie procesów, i do którego należy wrócić gdy wszystkie procesy zakończą działanie.

Wyjaśnienia wymaga chyba jeszcze wyrażenie w funkcji **longjmp**:

```
(cur=cur->next)->buf
```

Zapewnia ono rozpoczęcie pseudo-współbieżnego wykonywania funkcji od tej, która została jako pierwsza zadeklarowana za pomocą funkcji proces.

Po wprowadzeniu dwóch dodatkowych zmiennych globalnych, _number i _the_end, wcześniejsze definicje będą wyglądały następująco:

```
#define BEGIN     {                                    \
                    static char is_a_process;
  \
                    if((is_a_process=be_a_process)!=0)
  \
                     if(setjmp(cur->buf)==0)return;

#define END    if(is_a_process!=0)                 \
                 if(-- _number!=0)                 \
                  {                                \
                   cur=last->next=cur->next;       \
                   longjmp(cur->buf,1);            \
                  }                                \
                 else                              \
                  longjmp(_the_end,1);             \
               } /* zamyka nawias otwarty w BEGIN */

#define _    if(is_a_process!=0 && _number>0 )
  \
                if(setjmp(cur->buf)==0)
  \

longjmp((last=cur,cur=(cur->next))->buf,1);
```

Poniżej znajduje się pełna zawartość plików **proces.h** i **proces.c** zawierających wszystkie opisane powyżej definicje, a także definicje i deklaracje wszystkich używanych zmiennych. Dodatkowo w pliku **proces.h** zostało zdefiniowane makro ABORT, powodujące zakończenie wszystkich procesów.

```
/* plik proces.c      Adam Sapek  */
#include "proces.h"

struct el *cur=NULL,*last=NULL;            /* definicje
zmiennych globalnych */

jmp_buf _the_end;

unsigned char  _number=0,be_a_process=0;
void proces(FUNCTION f)                          /* funkcja
inicjująca proces */
{
 struct el *tmp;

 _number++ ;
 tmp=malloc(sizeof(struct el));                          /*
przydziel pamięć */
 if(cur!=NULL)
    {tmp->next=cur->next; cur->next=tmp; }             /*
dopisz do kolejki */
  else
    {cur=tmp; cur->next=tmp;}           /* pierwszy proces ->
stwórz kolejkę */
 last=cur;
 cur=tmp;
 be_a_process=1;
 (*f)();                                        /* inicjuj funkcję
jako proces */
 be_a_process=0;
}

/* plik proces.h      Adam Sapek */

#ifndef __proces_h
#define __proces_h

#include <setjmp.h>      /* longjmp i setjmp */

#include <stdlib.h>      /* malloc */

#include <stdio.h>       /* NULL */

/* makro BEGIN jest umieszczane na początku funkcji/procesu  */

#define BEGIN     { static char is_a_process;           \
                    if((is_a_process=be_a_process)!=0)  \
                      if(setjmp(cur->buf)==0)return;


/* makro END jest umieszczane na końcu funkcji/procesu        */

#define END    if(is_a_process!=0)               \
                 if(-- _number!=0)               \
                  {                              \
                   cur=last->next=cur->next;      \
                   longjmp(cur->buf,1);           \
                  }                              \
                 else                            \
                  longjmp(_the_end,1);           \
              } /* zamyka nawias otwarty w BEGIN */

/* makro _ powoduje przekazanie sterowania do następnego procesu
w kolejce */

#define _    if( is_a_process!=0 && _number>0 )          \
              if(setjmp(cur->buf)==0)                    \
```

```
                longjmp((last=cur,cur=(cur->next))->buf,1);

/* makro RUN rozpoczyna współbieżne wykonywanie procesów */
#define RUN
if(setjmp(_the_end)==0)longjmp((cur=cur->next)->buf,1);

/* makro ABORT powoduje natychmiastowe przerwanie WSZYSTKICH
procesów  */

#define ABORT   if(is_a_process!=0)                    \
                   {_number=0; longjmp(_the_end,1);}

/* deklaracje obiektów zewnętrznych */

typedef void (*FUNCTION)(void);

struct el {
            jmp_buf     buf;
            struct el  *next;
          };

extern unsigned char _number,be_a_process;

extern jmp_buf _the_end;

extern struct el *cur,*last;

extern void proces(FUNCTION);
#endif
```

5. Program współbieżny

Na początek wypada podać kilka ogólnych zasad stosowania zdefiniowanych narzędzi. Pisząc program współbieżny najlepiej napisać i uruchomić osobno każdą funkcję, która ma być w programie procesem. Przystosowanie napisanej wcześniej funkcji do pracy współbieżnej jest bardzo łatwe i ogranicza do zmian "kosmetycznych". Po pierwsze na początku funkcji, po deklaracjach zmiennych, należy umieścić makro BEGIN. Drugie makro, END, umieszcza się na końcu funkcji. Teoretycznie ani makro BEGIN nie musi znajdować się na samym początku funkcji, ani makro END na samym końcu, wymagane jest tylko, aby makro BEGIN poprzedzało makro END. Umieszczając wspomniane makra w innym miejscu należy jednak pamiętać, że tylko ta część funkcji, która jest zawarta między nimi, będzie wykonywana w pracy współbieżnej. Kod poprzedzający makro BEGIN wykona się tylko podczas inicjowania procesu funkcją proces, a kod następujący za makrem END nie wykona się w ogóle.

Makro o nazwie _ (podkreślenie) umieszcza się w tych punktach, w których proces ma być przerywany, a sterowanie ma być przekazywane do innych procesów. Żeby uzyskać najlepszą współbieżność, najlepiej dopisać to makro do każdego średnika kończącego instrukcję. Nie zawsze takie rozwiązanie jest najlepsze. Sekwencja przekazania sterowania (makro _) ma swoje koszty, zarówno czasowe jak i pamięciowe. Umieszczając to makro inteligentnie w każdym procesie można w dużym stopniu sterować działaniem programu. Można w ten sposób na przykład uprzywilejować niektóre procesy lub ich fragmenty,

uniemożliwić przerwanie krytycznych sekcji w procesach. Niektóre zasady wyboru lokalizacji makra _ zostaną opisane w dalszej części tego rozdziału, przy opisie przykładowego programu.

Najbardziej zdradzieckim błędem, jaki można popełnić w programie współbieżnym, jest użycie w procesie zmiennej automatycznej. Pamięć na zmienne automatyczne jest przydzielana po wywołaniu funkcji i zwalniania po jej zakończeniu. Z tego powodu zmienne automatyczne różnych procesów mogą znaleźć się w tym samym obszarze pamięci (i zwykle tak się właśnie dzieje). Błędy powstałe "dzięki temu" mogą być bardzo trudne do wykrycia nawet przy pomocy debuggera. Dlatego należy bezwzględnie pamiętać o zamienieniu wszystkich zmiennych automatycznych w funkcjach/procesach na zmienne statyczne. Jest to właściwie jedyna zmiana merytoryczna jakiej należy dokonać w funkcji, żeby mogła stać się procesem współbieżnym.

Programy współbieżne należy konsolidować z modułem **proces.obj** powstałym po skompilowaniu pliku **proces.c**. Można to zrobić na wiele sposobów, np.:

- utworzyć w środowisku Turbo C projekt o zawartości:

```
program.c
proces.c
```

- użyć bezpośrednio kompilatora wsadowego Microsoft C poprzez wywołanie:

```
cl program.c proces.c
```

Przykładowy program

Program demonstracyjny będzie składał się z dwóch procesów. Jeden z nich będzie wczytywał kolejne bajty z podanego pliku i zliczał całkowitą liczbę bitów równych jeden w pliku. Drugi proces na czas pracy pierwszego wygasi ekran i będzie wyświetlał na nim chodzącego "węża". Po przeczytaniu całego pliku proces pierwszy przerwie działanie obydwu procesów i wróci do programu głównego, który wyświetli liczbę jedynek w pliku.

W wersji klasycznej funkcja zliczająca liczbę jedynek w pliku mogłaby wyglądać na przykład tak:

```
void bits(void)
 {
  FILE *in;
  unsigned c;
  char name[80];

  printf("\nNazwa pliku : ");
  scanf("%s",name);
  if(NULL!=(in=fopen(name,"rb")))
   while(!feof(in))
    {
     c=getc(in);
```

```
        while(c){ sum+=c&1; c/=2; }
      }
}
```

Funkcja ta pyta o nazwę pliku i próbuje otworzyć do czytania w trybie binarnym plik o podanej nazwie. Jeżeli plik uda się otworzyć, funkcja wczytuje kolejno wszystkie bajty i zlicza kolejne jedynki.

Gdzie umieścić poszczególne makra, żeby przystosować naszą funkcję do pracy współbieżnej ?

Makro END znajdzie się w standardowym miejscu, na końcu funkcji. Makro BEGIN można umieścić albo bezpośrednio po definicjach zmiennych, albo dopiero po wczytaniu nazwy pliku. Wydaje się, że lepiej jest umieścić je w tym drugim miejscu. Pytanie o nazwę pliku pojawi się wtedy w czasie inicjowania procesu funkcją proces. Proszę zauważyć, że nie można makra BEGIN przesunąć jeszcze dalej, za warunek **if.** Gdyby nie udało się otworzyć pliku, makro BEGIN w ogóle nie zostałoby wykonane, a funkcja nie zostałaby poprawnie zainicjowana jako proces.

Normalnie, zakończenie działania jednego procesu współbieżnego nie powoduje przerwania działania pozostałych; proces zakończony jest po prostu wyłączany z kolejki. W naszym programie proces czytający plik jest procesem nadrzędnym. Natychmiast po przeczytaniu całego pliku obydwa procesy powinny być przerwane, a program główny powinien wypisać wynik. W takim celu zostało w pliku **proces.h** zdefiniowane makro ABORT. Powoduje ono natychmiastowe przerwanie wszystkich zadań i skok do miejsca, z którego zostały uruchomione makrem RUN. Makro ABORT należy umieścić bezpośrednio przed makrem END.

Kolejną zmianą jest nadanie zmiennym lokalnym klasy **static.** Wystarczy w tym celu przed definicjami umieścić słowo kluczowe **static.**

Na koniec zostawiłem sprawę najważniejszą - makro _ (podkreślenie). W tej funkcji istnieją dwa potencjalne miejsca, w których można je umieścić: pętla główna i pętla testująca kolejne bity wczytanego bajtu. Zależy nam oczywiście, żeby funkcja czytająca plik nie została zbytnio spowolniona, a więc najlepiej jest umieścić makro _ tylko w pętli głównej. Powstaje pytanie, czy wystarczy to, żeby drugi proces mógł działać płynnie. Jedyną odpowiedź można uzyskać metodą doświadczalną, uruchamiając program. Stwierdziłem, że umieszczenie makra tylko w pętli głównej jest zupełnie wystarczające.

Funkcja **bits** wygląda po zmianach następująco:

```
void bits(void)
{
 static FILE *in;
 static unsigned c;
 static char name[80];
```

```
 printf("\nNazwa pliku : ");
 scanf("%s",name);
 BEGIN
 if(NULL!=(in=fopen(name,"rb")))
 while(!feof(in))
  {
   c=getc(in); _
   while(c){ sum+=c&1; c/=2; }
  }
 ABORT
 END
}
```

Zadaniem drugiego procesu, jak wcześniej napisałem, będzie wygaszenie ekranu na czas działania procesu czytającego plik. Proces ten będzie przykładem adaptacji do pracy współbieżnej funkcji napisanej w zupełnie innym celu. Zostanie tu użyta oryginalna funkcja z rezydentnego programu *Screen Saver*, zaprezentowanego w rozdziale piątym tej książki. W celu zwiększenia czytelności, w funkcji tej dokonałem kilku uproszczeń, a także zamieniłem bezpośrednie odwołania do pamięci video na odpowiednie funkcje biblioteczne. Ponieważ funkcje te nie są zestandaryzowane, ich wersje dla różnych kompilatorów nieco się różnią. Poniżej przedstawię wersję dla kompilatora Turbo C firmy Borland, a na dołączonej do książki dyskietce znajduje się także wersja dla kompilatora Microsoft C 5.1.

Proces wygaszający ekran jest przerywany przez proces pierwszy, gdy ten zakończy swoją pracę, dlatego też jego główna pętla jest nieskończona. Skoro tak, to wydaje się, że umieszczanie makra END na końcu funkcji, w miejscu, w którym nigdy nie zostanie wykonane, mija się z celem. Proszę jednak jeszcze raz spojrzeć na definicje: makra BEGIN i END stanowią nierozłączną parę. W definicji makra END znajduje się nawias zamykający instrukcję blokową otwartą w definicji BEGIN.

W tym miejscu dochodzimy znowu do punktu zasadniczego: gdzie umieścić makro _. Proszę spojrzeć na funkcję **ekr** i spróbować samemu wybrać najodpowiedniejsze miejsce.

```
void  scr(void)
{
 static unsigned direct=3,y,x,a;
 BEGIN;
 clrscr();
 while(1)
    {
     do{  if(rand()%11==0)direct=rand()%8;
          x=snake[0].x+dx[direct];
          y=snake[0].y+dy[direct];
       }
     while(x<1||x>=80||y<1||y>25);
     gotoxy(x,y);                    printf("\xDB");
     gotoxy(snake[0].x,snake[0].y); printf("\xB2");
     gotoxy(snake[1].x,snake[1].y); printf("\xB1");
     gotoxy(snake[2].x,snake[2].y); printf("\xB0");
     gotoxy(snake[3].x,snake[3].y); printf("  ");

     for(a=3;a>=1;a--)
       {
```

```
            snake[a].x=snake[a-1].x;
            snake[a].y=snake[a-1].y;
          }
        snake[0].x=x; snake[0].y=y;
        a=*(char far *)0x46c;
        while(*(char far *)0x46c-a<7);
       }
     END;
    }
```

Właściwie umieszczenie tego makra obok każdego średnika nie szkodzi niczym oprócz niepotrzebnego zwiększenia programu i zmniejszenia czytelności zapisu. Kluczem do poprawnego umieszczenia sekwencji przełączającej zadania jest spostrzeżenie, że funkcja **scr** praktycznie cały czas spędza w pętli opóźniającej (proszę spróbować uruchomić tę funkcję bez tej pętli !):

```
while(*(char far *)0x46c-a<7);
```

Teraz jest już chyba jasne, że wystarczy umieścić makro _ (podkreślenie) w tej pętli, i to nie obok średnika, tylko zamiast niego:

```
while(*(char far *)0x46c-a<7)_
```

Poniżej przedstawiony jest kompletny program.

```
/* plik bity.c */
#include "proces.h"
#include <conio.h>

unsigned long sum=0;

void bits(void)             /* funkcja zlicza ilość jedynek w
bajtach pliku */
{
 static FILE *in;
 static unsigned c;
 static char name[80];
 printf("\nNazwa pliku : "); scanf("%s",name);        /* wczytaj
nazwę pliku */
 BEGIN                                      /* początek części
współbieżnej */
 if(NULL!=(in=fopen(name,"rb")))
 while(!feof(in))
  {
   c=getc(in); _                            /* wczytaj kolejny
bajt z pliku */
   while(c){ sum+=c&1; c/=2; }              /* policz jedynki w
tym bajcie  */
  }
 ABORT                                      /* przerwij procesy
współbieżne */
 END
}

struct  {                                   /* struktura
reprezentuje jednen*/
```

```
        unsigned char x,y;
/* człon węża */
    }snake[4]={{4,4},{3,3},{2,2},{1,1}};  /* wąż ma 4 człony */

void scr(void)          /* funkcja wygasza ekran i wyświetla
chodzącego węża */
{
 static unsigned direct=3,y,x,a;
 static char dx[8]={0,1,2,1,0,-1,-2,-1},
             dy[8]={-1,-1,0,1,1,1,0,-1};

 BEGIN;
 clrscr();
 while(1)
    {
     do{  if(rand()%11==0)      /* średnio co 11 kroków       */
            direct=rand()%8;                 /* losuj nowy
kierunek węża   */
          x=snake[0].x+dx[direct];
          y=snake[0].y+dy[direct];
       }while(x<1||x>=80||y<1||y>25);         /* czy można w tym
kierunku ? */

     gotoxy(x,y);                   printf("██");   /* rysuj
węża           */
     gotoxy(snake[0].x,snake[0].y); printf("▓▓");
     gotoxy(snake[1].x,snake[1].y); printf("▒▒");
     gotoxy(snake[2].x,snake[2].y); printf("░░");
     gotoxy(snake[3].x,snake[3].y); printf("  ");
     for(a=3;a>=1;a--)                            /* przesuń

węża           */
       {
       snake[a].x=snake[a-1].x;
       snake[a].y=snake[a-1].y;
       }
     snake[0].x=x; snake[0].y=y;
     a=*(char far *)0x46c;
     while(*(char far *)0x46c-a<7) _            /* odczekaj 7
taktów zegara  */
    }
 END;
}

void main()
{
clrscr();                                       /* wyczyść ekran
*/
proces(scr);                                    /* inicjuj
procesy           */
proces(bits);
RUN;                                            /* uruchom
procesy           */
clrscr();
printf("W pliku znaleziono %lu jedynek",sum);   /* wypisz wynik
*/
}
```

6. Komunikacja między procesami

Najprostszym sposobem komunikacji między procesami współbieżnymi jest zastosowanie zmiennych globalnych. Metoda ta jest tyleż prosta, co prymitywna. Nie

zapewnia ona synchronizacji: proces, który ma czytać jakąś daną ze zmiennej globalnej, nie wie, czy została ona tam już zapisana. Podobnie, proces zapisujący daną nie wie, czy poprzednia wartość została już pobrana. Dopisanie odpowiednich wskaźników synchronizujących może na tyle zaciemnić zapis, że warto posłużyć się preprocesorem w celu implementacji w miarę ogólnej metody komunikacji między procesami. Dzięki ukryciu szczegółów w definicjach procesora i zadeklarowaniu funkcji, zmiennych i struktur w osobnym pliku można uzyskać bardzo łatwe w użyciu i elegancko wyglądające narzędzia.

Do komunikacji między procesami użyjemy skrytki typu FIFO (ang. *First In First Out*). Procesy "producenci" będą deponować w skrytce dane określonego typu, a procesy "konsumenci" będą je z niej pobierać. Jeżeli skrytka będzie pusta, konsumenci będą czekać na dane, jeżeli zaś będzie pełna, producenci będą czekać na wolne miejsce. Niezbędne jest, aby oczekiwanie któregoś procesu (czy to konsumenta, czy producenta) nie powodowało zawieszenia pozostałych procesów. W przeciwnym przypadku czekający proces nigdy nie doczekałby się zmiany stanu skrytki.

Typ danych przechowywanych w skrytce (w C++ może to być nie tylko typ standardowy, ale także klasa) oraz rozmiar skrytki będzie mógł być definiowany. W przypadku niezdefiniowania typu i/lub rozmiaru będą przyjmowane wartości domyślne: typ **int** i rozmiar równy 100.

Skrytka będzie strukturą zdefiniowaną jak poniżej:

```
struct {
        unsigned long  pocz,koniec;
        typ            wnetrze[rozmiar];
      }_skrytka;
```

Pole pocz będzie wskazywało następny element możliwy do pobrania, a pole koniec, pierwsze wolne miejsce w skrytce. Jeżeli skrytka będzie pusta, wartość pola pocz będzie równa wartości pola koniec. Jeżeli skrytka będzie pełna, zajdzie warunek:

```
_skrytka.koniec-_skrytka.pocz==rozmiar
```

Warunki sprawdzające, czy skrytka jest pełna czy pusta, mogą przydać się w programowaniu, dlatego zdefiniujemy je w postaci identyfikatorów preprocesora:

```
#define PELNA (_skrytka.pocz-_skrytka.koniec==rozmiar)
#define PUSTA (_skrytka.pocz==_skrytka.koniec)
```

Funkcje umieszczająca i usuwająca daną ze skrytki są bardzo proste i nie wymagają chyba komentarza:

```
_umiesc(typ x)
{
  if(!PELNA)
   {
```

```
      _skrytka.wnetrze[(_skrytka.pocz++)%rozmiar]=x;
      return 0;
    }
  return 1;
}

_usun(typ *x)
{
  if(!PUSTA)
   {
     *x=_skrytka.wnetrze[(_skrytka.koniec++)%rozmiar];
     return 0;
   }
  return 1;
}
```

Są to funkcje wewnętrzne dla modułu komunikacji między procesami; dla programisty przygotujemy wygodniejsze narzędzie. Jak widać, funkcje **_umiesc** i **_usun** zwracają wartość 0 w przypadku sukcesu i wartość 1 w przypadku niepowodzenia. Wykorzystując tę własność zdefiniujemy makrodefinicje czekające w razie potrzeby na dane lub wolne miejsce w skrytce, ale nie zawieszające przy tym pozostałych procesów.

```
#define  put(x)    while(_umiesc(x))_
#define  get(x)    while(_usun(&x))_
```

Makrodefinicje get i put czekają w pętli **while** aż odpowiednia funkcja zwróci wartość zero, co świadczy o pomyślnym zapisie do skrzynki (put) lub odczycie ze skrytki (get). Jednocześnie w każdym obiegu pętli, wywoływane jest makro _ (podkreślenie), dzięki czemu pozostałe procesy mogą normalnie pracować.

Na koniec dygresja dotycząca użycia plików nagłówkowych, włączanych do programu dyrektywą preprocesora **#include**. Z zasady należy unikać umieszczania w tych plikach kodu, a więc definicji zmiennych i funkcji, które należy definiować w osobnym, niezależnie kompilowanym pliku. Natomiast w pliku nagłówkowym powinny znaleźć tylko ich deklaracje.

Wystarczy rzut oka na przytoczony poniżej fragment, żeby przekonać się, że w pliku nagłówkowym **bufor.h** umieściłem zarówno funkcje jak i zmienne. W tym przypadku sytuacja jest jednak wyjątkowa: typ danych deponowanych w skrytce, a więc typ argumentu funkcji _umiesc i _usun jest znany dopiero w momencie kompilacji programu głównego. Podobnie jest z typem i rozmiarem samej skrytki. Oczywiście, można zaimplementować skrytkę tak, żeby mogła być skompilowana wcześniej i akceptowała dane dowolnego typu. Kosztem tego będzie większe skomplikowanie i brak kontroli zgodności typów danych umieszczanych i pobieranych ze skrytki. Nie będę przesądzał, które rozwiązanie jest lepsze, zachęcam natomiast gorąco do zaimplementowania tego drugiego samodzielnie.

Poniżej przytoczono pełną zawartość pliku **bufor.h**. Zawiera on jedną nie opisaną, ale chyba oczywistą funkcję **_inicjuj**.

```
/* plik bufor.h   Adam Sapek
*/
#ifndef __bufor_h
```

```
#define __bufor_h

#ifndef __proces_h
#include "proces.h"
#endif

#ifndef rozmiar
#define rozmiar 100                              /* domyślny rozmiar
skrytki */
#endif

#ifndef typ
#define typ int                              /* domyślny typ danych w
skrytce */
#endif

#define  put(x)    while(_umiesc(x))_              /* włóż daną do
skrytki */

#define  get(x)    while(_usun(x))_             /* pobierz daną ze
skrytki */

struct {                                                  /* skrytka
FIFO */
         unsigned long  pocz,koniec;
         typ            wnetrze[rozmiar];                /* dane
typu typ */
       }_skrytka;
#define PELNA (_skrytka.pocz-_skrytka.koniec==rozmiar)      /* czy
pełna ? */
#define PUSTA (_skrytka.pocz==_skrytka.koniec)              /* czy
pusta ? */

void _inicjuj(void)                                    /* inicjacja
skrytki */
{
 _skrytka.pocz=_skrytka.koniec=0;
}
_umiesc(typ x)                          /* funkcja umieszcza daną w
skrytce */
{
  if(!PELNA)
   { _skrytka.wnetrze[(_skrytka.pocz++)%rozmiar]=x;
     return 0;
   }
  return 1;
}

_usun(typ *x)                              /* funkcja usuwa daną ze
skrytki */
{
  if(!PUSTA)
   { *x=_skrytka.wnetrze[(_skrytka.koniec++)%rozmiar];
     return 0;
   }
  return 1;
}

#endif
```

Zastosowanie skrytki jest bardzo proste. Program, który jej używa powinien zawierać dyrektywę

```
#include "bufor.h"
```

poprzedzoną ewentualnymi definicjami rozmiaru skrzynki i/lub typu danych, np.:

```
#define rozmiar 10
#define typ   char
```

Do umieszczania danych w skrytce należy stosować makro put, a do ich pobierania makro get. Należy pamiętać, że argumentem makra get jest wskaźnik do zmiennej, w której ma być umieszczona pobrana dana. Poniższy prosty programik ilustruje zastosowanie skrytki.

```
/* plik prod.c */

#define  typ char
#include "bufor.h"
void producent1(void)      /* "produkuje" duże litery i
umieszcza w skrytce */
{
 static char c;
 BEGIN
 for(c='A';c<='Z';c++)
  { put(c); _ }
 END
}
void producent2(void)      /* "produkuje" małe litery i
umieszcza w skrytce */
{
 static char c;
 BEGIN
 for(c='z';c>='a';c--)
  { put(c); _ }
 END
}
void konsument(void)       /* pobiera ze skrytki znak i wypisuje
na ekranie */
{
 static char c,x;
 BEGIN
 for(x=0;x<52;x++)
  {
    get(&c); _
    putchar(c);
  }
 END
}

void main()
{
proces(producent1);            /* inicjuj procesy */
proces(producent2);
proces(konsument);
RUN;                           /* uruchom procesy współbieżnie
*/
}
```

Program jest trywialny, niemniej jednak prześledzenie jego działania może być pouczające. Radzę zwłaszcza poeksperymentować zmieniając liczbę makrodefinicji _ umieszczonych w jednym z procesów producentów, a także rozmiar skrytki.

7. Współbieżne wejście z klawiatury

Wywołanie w procesie współbieżnym funkcji czytającej dane z klawiatuy powoduje zawieszenie działania pozostałych procesów. Ponieważ wprowadzenie informacji z klawiatury zajmuje relatywnie dużo czasu, jest to dosyć poważna uciążliwość. Poza tym wczytywanie z klawiatury w minimalnym stopniu wykorzystuje procesor i jest najlepszym momentem do wykonania jakiejś pracy przez inne procesy.

Zacznijmy od funkcji najprostszych. Funkcje **getch** i **getche** służą do wczytania znaku z klawiatury. Różnica między nimi polega na tym, że getche powoduje wypisanie wczytanego znaku na ekran. Obydwie funkcje czekają na naciśnięcie klawisza i dlatego użyte w procesie współbieżnym powodują zawieszenie działania wszystkich procesów. Zamiast tych funkcji możemy zdefiniować makrodefinicje preprocesora zwracające kod znaku albo wartość NIC jeżeli w buforze klawiatury nie ma żadnego znaku.

```
#define NIC -1
#define getk()          (kbhit()?getch():NIC)
#define getke()         (kbhit()?getche():NIC)
```

Funkcja **kbhit** zwraca wartość niezerową jeżeli w buforze klawiatury znajduje się znak gotowy do pobrania. Wszystkie trzy wykorzystane funkcje: **kbhit**, **getch** i getche nie są wprawdzie funkcjami standardu ANSI, ale są implementowane zarówno w kompilatorach firmy Borland jak i Microsoft.

Bardziej skomplikowane operacje wejścia realizuje się w języku C przy pomocy rodziny funkcji pochodnych od scanf. Rozszerzymy tę rodzinę o makrodefinicję pscanf umożliwiającą realizację formatowanego wejścia z klawiatury w jednym procesie współbieżnym, podczas gdy pozostałe procesy wykonują inna pracę. Ponieważ musimy ograniczyć się do użycia preprocesora, makro pscanf będzie umożliwiało wczytanie tylko jednej wartości. Będzie to jednak jedyne ograniczenie w stosunku do "oryginału"; wszystkie sekwencje formatujące wejście akceptowane w rodzinie scanf będą realizowane przez makro pscanf.

Idea rozwiązania jest następująca. Kolejne znaki, aż do naciśnięcia klawisza ENTER, są wczytywane do lokalnego bufora tekstowego. W pętli wczytującej znaki znajduje się wywołanie makra _ , co zapewnia wykonywanie się współbieżnie pozostałych procesów. Po wczytaniu całego wiersza, jest on interpretowany przy pomocy standardowej funkcji **sscanf** realizującej formatowane wejście z ciągu znaków. Dzięki zastosowaniu funkcji **sscanf**, możemy być pewni, że wszystkie sekwencje sterujące zostaną poprawnie zinterpretowane.

Definicja makra pscanf wyglądałaby mniej więcej tak:

```
#define pscanf(form,arg)   { static char _buf[100],_x;
\
```

```
                                        static int _c;
\
                                        _x=0;
\
                                        do{
\
                                           _c=getke();
\
                                           if(_c!=NIC)
\
                                             _buf[_x++]=_c;
\
                                           _
\
                                        }while(_c!='\r');
\
                                        sscanf(_buf,form,arg);
\
                                      }
```

Wydaje się konieczne uzupełnienie jej przynajmniej o interpretację klawisza BACKSPACE. Odczytanie naciśnięcia tego klawisza za pomocą funkcji **getche** powoduje przesunięcie kursora na ekranie o jeden znak w lewo. Wystarczy więc wyświetlić w tym miejscu spację i jeszcze raz cofnąć kursor. Ponadto trzeba usunąć ostatni znak z bufora (a więc po prostu dekrementować licznik x).

Rozbudowana w opisany powyżej sposób makrodefinicja pscanf znajduje się w pliku **procesi.h**, którego pełny tekst zamieszczono poniżej.

```
/* plik procesi.h     Adam Sapek */

#ifndef __procesi_h
#define __procesi_h

#ifndef __proces_h
#include "proces.h"
#endif

#include <conio.h>

#define NIC        -1

#define getk()     (kbhit()?getch():NIC)         /* czytaj znak z
klawiatury */
#define getke()    (kbhit()?getche():NIC)        /* j.w. + pisz
czytany znak */

/* Makro pscanf umożliwia formatowane wejście z klawiatury bez
zawieszania   */
/* procesów współbieżnych. Makro akceptuje sekwencje steruj¹ce
rodziny scanf */

#define pscanf(form,arg) {static char _buf[100],_x;    \
                          static int _c;               \
                          _x=0;                        \
                          do{                          \
                          _c=getke();                  \
                          if(_c=='\b')                 \
```

```
                                {                          \
                                  putch(' '); putch('\b'); \
                                  if(_x>0)_x--;            \
                                }                          \
                               else                        \
                                if(_c!=NIC)                \
                                  _buf[_x++]=_c;           \
                               _                           \
                               }while(_c!='\r');           \
                               sscanf(_buf,form,arg);      \
                              }

#endif
```

Makrodefinicji pscanf używa się dokładnie tak, jak funkcji z rodziny **scanf**. Jedyną różnicą, o której trzeba pamiętać, jest to, że ma ona ustaloną liczbę argumentów: ciąg format i jedna zmienna do wczytania.

Poniżej przedstawiam programik ilustrujący ile pracy mogą wykonać inne procesy w czasie, gdy jeden czyta z klawiatury.

```
/* plik mult.c */
#include "procesi.h"

unsigned long i;

void czytaj(void)            /* funkcja pyta o imię i wypisuje
pozdrowienie */
{
 static char name[40];
 BEGIN
 printf("Jak sie nazywasz ?\n");
 pscanf("%[^\r]",name);                    /* wczytaj wiersz do
tablicy name */
 printf("Ahoj %s !\n",name);
 ABORT                                     /* przerwij wszystkie
procesy     */
 END
}
void licz(void)                         /* funkcja mnoży w petli
dwie liczby */
{
 static float x=3.14, y=1.73, z;
 BEGIN
 while(1){ z=y*x; i++; _ }              /* pomnóż x*y i zwieksz
licznik      */
 END
}
void main()
{
 proces(licz);                              /* inicjuj procesy
*/
 proces(czytaj);
 RUN                                        /* uruchom procesy
współbieżnie */
 printf("W miedzyczasie wykonalem %ld mnozen",i);
}
```