Podstawy programowania komputerów

Wykład 14: „Programowanie współbieżne w C”

swalover@ie.tu.koszalin.pl

Definicja programowania współbieżnego

- Programowanie współbieżne jest tworzeniem programów, których wykonanie powoduje uruchomienie pewnej liczby procesów współbieżnych (zazwyczaj procesy te są zależne).
- Dwa procesy nazywamy współbieżnymi, jeżeli każdy z nich rozpoczął się przed zakończeniem drugiego procesu.
- W różnych kontekstach używane są następujące określenia: współbieżny (concurrent), równoległy (parallel), rozproszony (distributed).

Pierwsze dwa używane są zamiennie, przy czym częściej określenie „równoległy” oznacza „wykonywany współbieżnie na tym samym komputerze”. Określenie „rozproszony” oznacza „wykonywany współbieżnie na oddzielnych komputerach połączonych w sieć”.

Dlaczego współbieżność?

- Klasyczne programy współbieżne są wykonywane na maszynach wieloprocesorowych. Celem zastosowania równoległych komputerów i równoległych programów jest zmniejszenie czasu rozwiązywania zadania.
- Na komputerze jednoprocesorowym nie można liczyć na zwiększenie prędkości obliczeń „pseudo-współbieżnych”, bo oprócz kodu zadań procesor musi wykonywać pewien kod związany z ich przełączaniem.
- W rzeczywistości częste są sytuacje, gdy większość czasu pracy zadania nie jest zużywana na pracę procesora. Np. podczas wolnych operacji na pamięci zewnętrznej procesor mógłby równocześnie wykonywać inną pracę lub można wyeliminować częściowo czekanie przez zadanie na dane wprowadzane przez użytkownika z klawiatury.

Pseudo-współbieżne wykonywanie funkcji w C

- Język C nie zawiera mechanizmów umożliwiających programowanie współbieżne (porównaj z Ada).
- Pseudo-współbieżne wykonywanie funkcji realizowane jest nie na poziomie systemu operacyjnego lecz na poziomie programu w C z wykorzystaniem jedynie elementów języka standardowego.
- Kiedy zadania (funkcje, programy) wykonywane są w sposób współbieżny na jednym procesorze, to w rzeczywistości na zmianę wykonywane są pewne małe fragmenty tych zadań - minimum jedna instrukcja (ang. statement) języka C.
- Kod pseudo-współbieżny C jest przenośny zarówno na różne kompilatory jak i różne platformy sprzętowe.

Bufor typu jmp_buf

W pliku nagłówkowym setjmp.h znajduje się deklaracja typu jmp_buf który definiuje strukturę do przechowywania informacji o stanie programu:

```
typedef struct {
        unsigned  j_sp,   j_ss,
        unsigned  j_flag, j_cs;
        unsigned  j_ip,   j_bp;
        unsigned  j_di,   j_es;
        unsigned  j_si,   j_ds;
    } jmp_buf[1];
```

Funkcje setjmp i longjmp

W pliku nagłówkowym setjmp.h znajdują się również deklaracje funkcji które służą do zapamiętania, a następnie odtworzenia stanu programu. :

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

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 goto).

Funkcje setjmp i longjmp, cd.

- 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, program znajdzie się w punkcie powrotu z funkcji setjmp, 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 przy zinterpretowaniu kolejnej instrukcji, czy też nastąpił skok przy pomocy funkcji longjmp.

Funkcje setjmp i longjmp, cd.

Przykładowy kod z wykorzystaniem instrukcji warunkowej:

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

Objaśnienia:

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

Funkcje setjmp i longjmp, cd.

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.

Przykład wykorzystania instrukcji switch:

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

Przełączanie zadań

- Aby dokonać przełączenia procesora pomiędzy dwiema funkcjami potrzebujemy dla każdego procesu 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 używamy wywołania: `longjmp(buf1,1);`

do funkcji f2: `longjmp(buf2,1);`

Przełączanie zadań, cd.

Przed wykonaniem skoku do drugiego procesu, każda funkcja musi wywołać funkcję: `setjmp(buf);`

Zapamiętane przez tę funkcję informacje o stanie programu

będą mogły być wykorzystane do powrotu do miejsca, w którym działanie funkcji zostało zawieszone. Sekwencje przełączające zadania będą wyglądały tak:

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

Objaśnienie: 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.

Przełączanie przy nie znanej ilości procesów

Dla obsługi nie znanej z góry ilości procesów trzeba:

- zdefiniować bufor typu jmp_buf dla każdej funkcji,
- umożliwić określenie następnej funkcji w "łańcuszku„

```
struct el {
   jmp_buf buf;
   struct el *next;
   };//lista jednokierunkowa
```

Objaśnienia: 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.

Przełączanie przy nie znanej ilości procesów, cd.

- Do pełnej manipulacji listą jednokierunkową potrzebne są co najmniej dwie zmienne, wskazujące na dwa kolejne węzły listy:

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

`cur` wskaźnikiem na węzeł odpowiadający aktywnej funkcji

`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);
```

- Po zakończeniu działania procesu trzeba go usunąć z listy:

```
cur=last->next=cur->next;
longjmp(cur->buf,1);//sterowanie do kolejnego procesu
```

Zapis praktyczny

- Stosując definicje preprocesora można zapisać sekwencje przełączające w sposób czytelniejszy:

```
#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 umieszcza się odpowiednio na początku i na końcu funkcji. Ostatnie makro jest właściwą sekwencją przełączającą zadania. Nazwa makro _ (podkreślenie) robi go mniej widocznym w kodzie.

Zapis praktyczny, cd.

Do rozróżniania czy funkcja jest wykonywana współbieżnie czy nie, można użyć lokalnej zmiennej is_a_process:

```
#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);
```