PROGRAMOWANIE WSPÓŁBIEŻNE

Dr inż. Wojciech Mikanik
wojciech.mikanik@polsl.pl
pokój 503 (V p.)

Luty 2009

Część I

Wprowadzenie

Wprowadzenie

Program przedmiotu

Wykład

- Wiadomości podstawowe
- Specyfikowanie współbieżności
- Programowanie w modelu z pamięcią wspólną
- Właściwości programów współbieżnych i równoległych
- Programowanie w modelu z pamięcią rozproszoną
- Przykłady

Laboratorium

- Pthreads:
  - Uruchamianie wątków
  - Muteksy
  - Zmienne warunkowe (monitory)
- OpenMP

Literatura

1. M. Ben-Ari: *Podstawy programowania współbieżnego i rozproszonego*, Warszawa, WNT 1996.
2. Z. Czech (red.): *Systemy operacyjne i języki obliczeń równoległych*, wyd. I, Politechnika Śląska, skrypt uczelniany nr 2121, Gliwice 1998.
3. Z. Weiss, T. Gruźlewski: *Programowanie współbieżne i rozproszone*, Warszawa, WNT 1993.

Slajdy wykładowe:
`http://sun.aei.polsl.pl/pub/wmikanik/usmpw/usmpw.html`

Wiadomości pomocnicze

1. Algorytmy sekwencyjne:
   - sortowania
   - numeryczne (np. mnożenie macierzy)
   - algorytmy kombinatoryczne (np. problem komiwojażera)
   - inne
2. Język C oraz C++ (Programowanie Komputerów)

Pseudokod

- Oparty na C/C++
- Dodane deklaracje i instrukcje, które będą stopniowo wprowadzane podczas wykładu
- Krótki opis składni dostępny przez WWW
  `http://sun.aei.polsl.pl/pub/wmikanik/usmpw/usmpw.html`
- Notacja, a nie język programowania
- Środek, nie cel

Proces sekwencyjny

Oznaczenia:

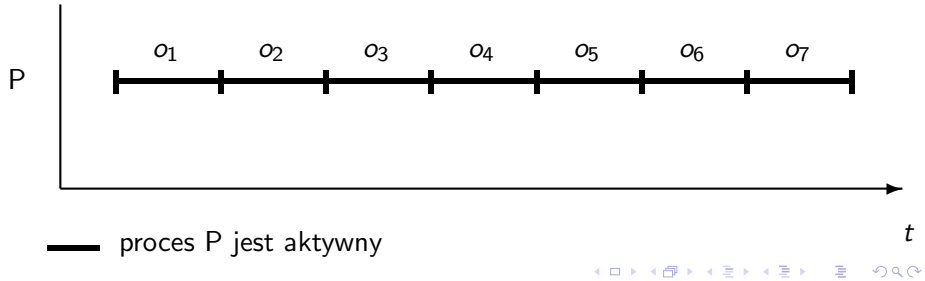
P — proces

o_i — operacja i-ta

$t(\underline{o_i})$ — czas rozpoczęcia operacji o_i

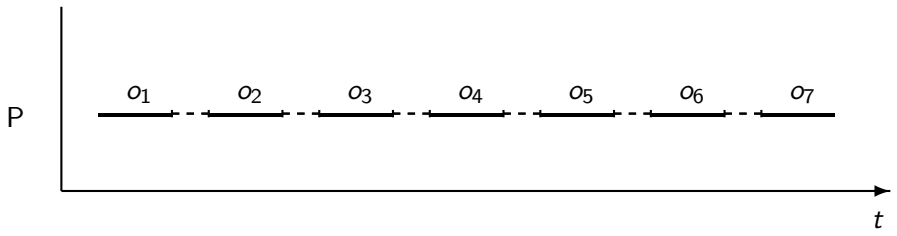
$t(\overline{o_i})$ — czas zakończenia operacji o_i

Dla procesu sekwencyjnego zachodzi $t(\overline{o_i}) \leq t(\underline{o_{i+1}})$
$t(\overline{o_i}) = t(\underline{o_{i+1}})$



Proces sekwencyjny

$t(\overline{o_i}) < t(\underline{o_{i+1}})$



– – – proces P jest nieaktywny

Procesy współbieżne

Oznaczenia:

P_k — k-ty proces

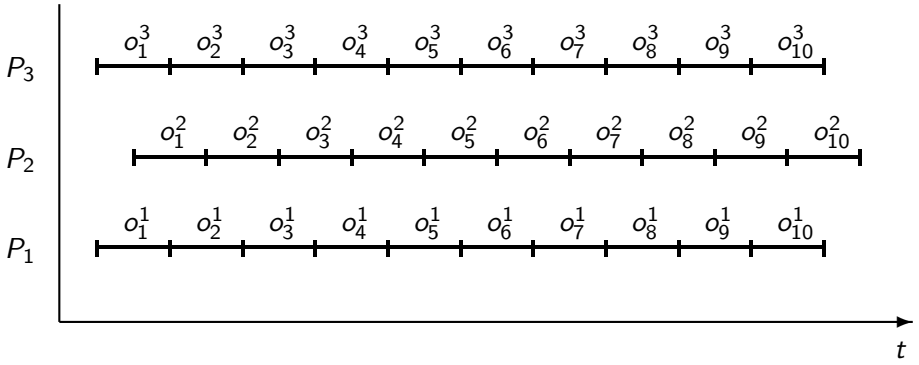
o_i^k — i-ta operacja k-tego procesu

Sekwencja operacji procesu P_1: $o_1^1, o_2^1, o_3^1, o_4^1, o_5^1, o_6^1$
Sekwencja operacji procesu P_2: $o_1^2, o_2^2, o_3^2, o_4^2, o_5^2, o_6^2$
Możliwe sekwencje operacji współbieżnego wykonania P_1 i P_2:

- $o_1^1, o_1^2, o_2^1, o_2^2, o_3^1, o_3^2, o_4^1, o_4^2, o_5^1, o_5^2, o_6^1, o_6^2$
- $o_1^1, o_2^1, o_3^1, o_4^1, o_5^1, o_6^1, o_1^2, o_2^2, o_3^2, o_4^2, o_5^2, o_6^2$
- $o_1^1, o_2^1, o_3^1, o_4^1, o_1^2, o_2^2, o_5^1, o_6^1, o_3^2, o_4^2, o_5^2, o_6^2$
- ...

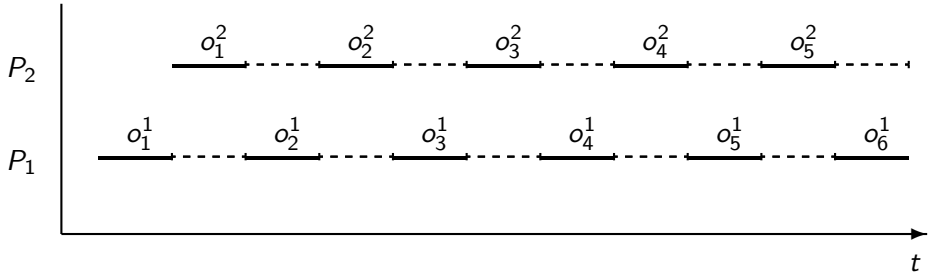
NIC NIE WIEMY O WZGLĘDNEJ SZYBKOŚCI WYKONANIA PROCESÓW WSPÓŁBIEŻNYCH

Procesy współbieżne
Równoległość operacji procesów



$t(\underline{o_1^1}) < t(\underline{o_1^2}) < t(\overline{o_1^1})$

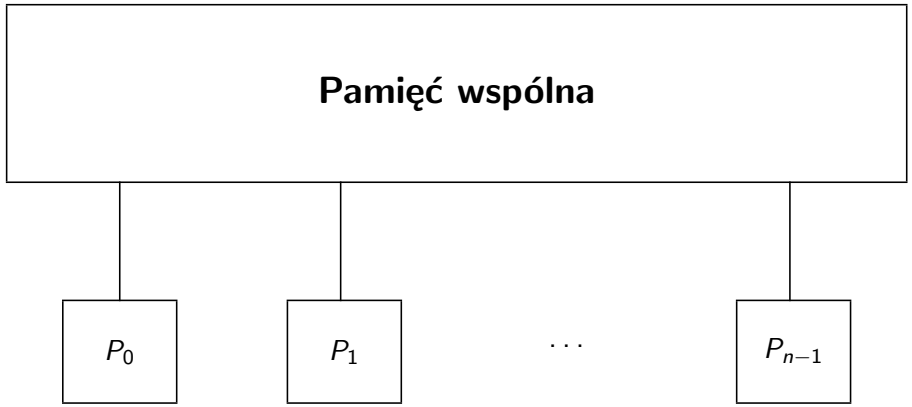
Procesy współbieżne
Przeplatanie operacji procesów



$t(\overline{o_1^1}) = t(\underline{o_1^2})$

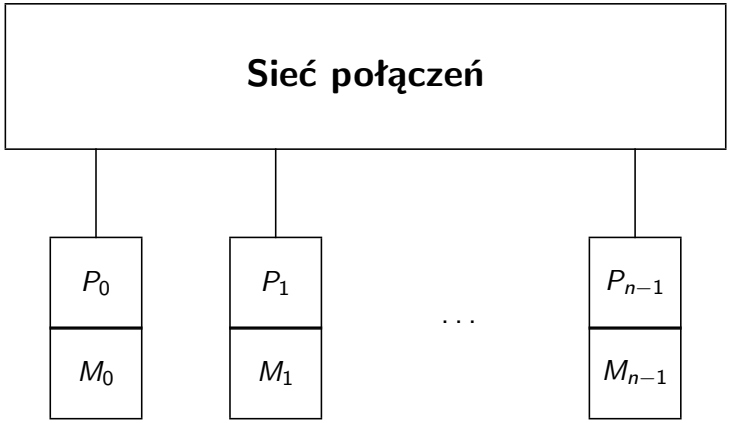
NIC NIE WIEMY O WZGLĘDNEJ SZYBKOŚCI WYKONANIA PROCESÓW WSPÓŁBIEżNYCH

Modele programowania współbieżnego
Pamięć wspólna (jedna przestrzeń adresowa)



Swobodny dostęp do dowolnej komórki pamięci

Modele programowania współbieżnego
Pamięć rozproszona (wiele przestrzeni adresowych)



Każdy proces ma swoje własne dane, do których ma wyłączny dostęp.

Część II

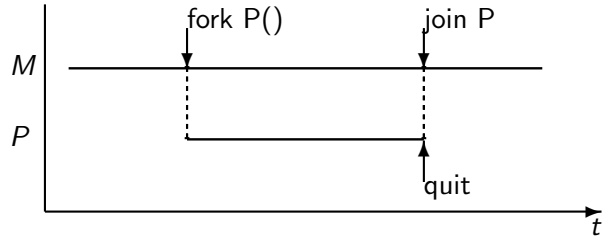
Specyfikowanie współbieżności

Omawiane zagadnienia

1. Instrukcje `fork`, `join`, `quit`
2. Instrukcja `cobegin`
3. Instrukcja `parfor`

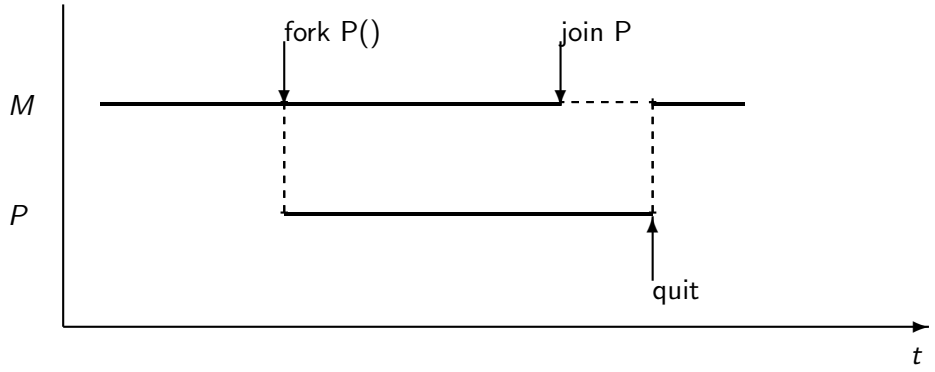
Instrukcje: fork, join, quit

```
void P(void){
  ... //instrukcje
  ...
  ...
  quit;
}//P
void M(void){
  ...
  fork P();
  ...
  join P;
  ...
  ...
}//M
main(void){
  M();
}
```



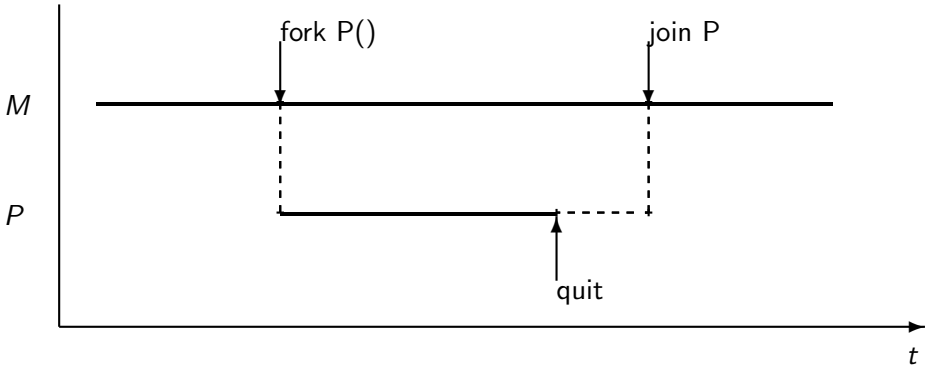
Instrukcje: fork, join, quit

M czeka na *P*



Instrukcje: fork, join, quit

P czeka na *M*



Nazwy procesów

```
void P(void){
  ... //instrukcje
  ...
  ...
  quit;
}//P
void M(void){
  ...
  fork P();
  fork P();
  ...
  join P;  //na zakonczenie ktorego procesu
  ...      //czekamy?
}//M
main (void){
  M();
}
```

Nazwy procesów

```
void P(void){
  ... //instrukcje
  quit;
}//P
void M(void){
  ...
  fork P() alias a;
  fork P() alias b;
  ...
  join a;  //czekamy na zakonczenie procesu a
  ...
  join b;  //czekamy na zakonczenie procesu b
  ...
}//M
main (void){
  M()
}
```


Przykład zastosowania fork, join, quit

```
int a[2*N];
int sumaP;
void sumuj(int p, int k){
  int i;
  int suma = 0;
  for (i = p; i < k; i ++)
    suma += a[i];
  sumaP = suma;
  quit;
}//sumuj
```

```
void M(void){
  int i;
  int suma = 0;

  czytaj_dane (a, 2 * N);
  fork sumuj (0, N) alias P;
  for (i = N; i < 2 * N; i ++)
    suma += a[i];
  join P;
  suma += sumaP;
  printf ("Suma = %d\n", suma);
  return;
}//M
main(void){
  M();
}
```


Przykład zastosowania fork, join, quit

Modyfikacja funkcji sumuj

```
void M(void){
  int suma;

  czytaj_dane (a, 2*N);
  fork sumuj (...) alias P;
  sumuj (...);
  join P;
  ...
  printf ("Suma elementow = %d\n", suma);
  return;
}//M
```


Przykład zastosowania fork, join, quit

Modyfikacja funkcji sumuj

```
int a[2*N];

void sumuj(int p, int k,
           int * s){
  int i;
  *s = 0;
  for (i = p; i < k; i ++)
    *s += a[i];
  quit;
}//sumuj
```

```
void M(void){
  int suma;
  int sumaP;

  czytaj_dane (a, 2*N);
  fork sumuj (0, N, &sumaP) alias P;
  sumuj (N, 2 * N, &suma);
  join P;
  suma += sumaP;
  printf ("Suma = %d\n", suma);
  return;
}
main (void){
  M();
}
```

Instrukcja cobegin

```
cobegin{
  S1;
  S2;
  ...
  Sn;
}
```

```
cobegin{
  a = 2;
  b = 4 * sin(12);
  ...
  z = silnia(102);
}
```


Instrukcja cobegin

Obliczanie sumy elementów tablicy

```
int a[2*N];
void main(void){
  int s1 = 0, s2 = 0;
  czytaj_dane (a, 2*N);
  cobegin{
    { int i;
      for (i = 0; i < N; i++)
        s1 += a[i];
    }
    { int i;
      for (i = N; i < 2*N; i++)
        s2 += a[i];
    }
  } //cobegin
  printf ("Suma elementow = %d\n", s1 + s2);
  return;
}
```


Instrukcja cobegin a fork, join, quit

```
cobegin {
  S1;
  S2;
  ...
  Sn;
}
```

```
...
fork P2();
fork P3();
...
...
fork Pn();
S1;
join P2;
join P3;
...
join Pn;
...
}
```

```
P2(){
  S2;
  quit;
}
P3(){
  S3;
  quit;
}
...
Pn(){
  Sn;
  quit;
```


Instrukcja parfor

```
cobegin {
  S1;
  S2;
  ...
  Sn;
}
```

```
parfor i = 1 to n do {
  Si;
}
```

```
parfor i = wart_początkowa to wart_końcowa do {
  Si;
}
```

```
int a[n], i;
parfor i = 0 to n - 1 do {
  a[i] = i;
} //parfor;
```

Przykład: Pthreads

```
fork int pthread_create(pthread_t *tid, pthread_attr_t *tattr,
                        void*(*start_routine)(void *),
                        void *arg);

     pthread_t tid;
     extern void *start_routine(void *arg);
     void *arg = .....;
     int ret;
     ret = pthread_create(&tid, NULL, start_routine, arg);

quit void pthread_exit(void *status);

join int pthread_join(thread_t tid, void **status);

     pthread_t tid = <id of a thread>;
     int ret, status;
     ret = pthread_join(tid, &status);
```


Część III

Pamięć wspólna: komunikacja i synchronizacja

Omawiane zagadnienia

1. Interferencja procesów
2. Muteksy
3. Semafory
4. Monitory
5. Porównanie zmiennych warunkowych i semaforów

Interferencja procesów

```
main(){
  int x;

  cobegin{
    { //proces P1
      ...  //instrukcje
      x = x + 1;
      ...  //instrukcje
    }
    { //proces P2
      ...  //instrukcje
      x = x + 2;
      ...  //instrukcje
    }
  }//cobegin
}
```

Interferencja procesów

```
main(){
  int x;
  cobegin{
    { //proces P1
      ...  //instrukcje
(1.1) rej1 = x;
(1.2) rej1 = rej1 + 1;
(1.3) x = rej1;
      ...  //instrukcje
    }
    { //proces P2
      ...  //instrukcje
(2.1) rej2 = x;
(2.2) rej2 = rej2 + 2;
(2.3) x = rej2;
      ...  //instrukcje
    }
  }//cobegin
}
```


Interferencja procesów

- zwiększenie wartości x o 3
  - (1.1) rej1 = x;
  - (1.2) rej1 = rej1 + 1;
  - (1.3) x = rej1;
  - (2.1) rej2 = x;
  - (2.2) rej2 = rej2 + 2;
  - (2.3) x = rej2;
- zwiększenie wartości x o 2
  - (1.1) rej1 = x;
  - (1.2) rej1 = rej1 + 1;
  - (2.1) rej2 = x;
  - (1.3) x = rej1;
  - (2.2) rej2 = rej2 + 2;
  - (2.3) x = rej2;
- zwiększenie wartości x o 1
  - (2.1) rej2 = x;
  - (1.1) rej1 = x;
  - (1.2) rej1 = rej1 + 1;
  - (2.2) rej2 = rej2 + 2;
  - (2.3) x = rej2;
  - (1.3) x = rej1;

Zachodzi interferencja procesów $P1$ i $P2$

Interferencja procesów

```
main(){
  int x;           //wspólny obszar danych

  cobegin{
    { //proces P1
      ...  //instrukcje
      x = x + 1;   //sekcja krytyczna
      ...  //instrukcje
    }
    { //proces P2
      ...  //instrukcje
      x = x + 2;   //sekcja krytyczna
      ...  //instrukcje
    }
  }//cobegin
}
```


Mutex

- MUTual EXclusion
- ```
  typedef struct {
      enum {locked, unlocked} status;
      queue_t q;
  } mutex;
  ```
- ```
  void init (mutex &m){
    m.status = unlocked;
    m.q = NULL;
  };
  ```

Mutex
Procedury *lock* i *unlock*

- ```
  void lock (mutex &m) {
    if (m.status == locked)
      zablokuj bieżący proces i umieść go w m.q
    else
      m.status = locked;
  };
  ```
- ```
  void unlock (mutex &m) {
    if (!  empty (m.q))
      pobierz proces z m.q i odblokuj go
    else
      m.status = unlocked;
  };
  ```


Muteks
Wzajemne wykluczanie

```
main(){
  int x;           //wspólny obszar danych
  mutex mx ;

  cobegin{
    { //proces P1
      ...  //instrukcje
      lock(mx);
      x = x + 1;   //sekcja krytyczna
      unlock(mx);
      ...  //instrukcje
    }
    { //proces P2
      ...  //instrukcje
      lock(mx);
      x = x + 2;   //sekcja krytyczna
      unlock(mx);
      ...  //instrukcje
    }
  }//cobegin
```


Przykład: Pthreads
Muteksy

- Inicjalizacja

  ```
  int pthread_mutex_init(pthread_mutex_t *mp,
                         const pthread_mutexattr_t *mattr);
  ```
- Usunięcie

  ```
  int pthread_mutex_destroy(pthread_mutex_t *mp);
  ```
- Stosowanie
  - `int pthread_mutex_lock(pthread_mutex_t *mp);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mp);`
  - `int pthread_mutex_trylock(pthread_mutex_t *mp);`

Semafor
Deklaracja i inicjalizacja

- ```
  typedef struct {
    int wartość;          //zmienna całkowita
    typ_kolejki kolejka; //kolejka (zbiór)
  } semafor;
  ```
- ```
  void inicjalizacja (semafor &s,
                      int wart_pocz){
    s.wartość = wart_pocz;
    s.kolejka = NULL; //kolejka jest pusta
  }
  s = 0;
  ```

Semafor
Operacje *P* i *V*

- ```
  void P (semafor &s){ //operacja P
    if (s.wartość <= 0)
      zablokuj bieżący proces i umieść go w s.kolejka
    else
      s.wartość -= 1;
  }
  ```

- ```
  void V (semafor &s){ //operacja V
    if (jakiś proces jest zablokowany na semaforze s)
      pobierz proces z s.kolejka i odblokuj go
    else
      s.wartość += 1;
  }
  ```


Semafor
Wzajemne wykluczanie

```
main(){
  int x;           //wspólny obszar danych
  semafor semafor_x = 1;

  cobegin{
    { //proces P1
      ...  //instrukcje
      P (semafor_x);
      x = x + 1;   //sekcja krytyczna
      V (semafor_x);
      ...  //instrukcje
    }
    { //proces P2
      ...  //instrukcje
      P (semafor_x);
      x = x + 2;   //sekcja krytyczna
      V (semafor_x);
      ...  //instrukcje
    }
  }//cobegin
```


Przykład: Pthreads
Semafory

- Nazwane
  - ```
    sem_t *sem_open( const char *name, int oflag,
                     /*unsigned long mode,
                     unsigned int value */ ...);
    ```
  - `int sem_close(sem_t *sem);`
  - `int sem_unlink(const char *name);`
- Anonimowe
  - ```
    int sem_init( sem_t *sem, int pshared,
                  unsigned int value);
    ```
  - `int sem_destroy(sem_t *sem);`
- Wspólne funkcje
  - `int sem_post(sem_t *sem);`
  - `int sem_wait(sem_t *sem);`
  - `int sem_trywait(sem_t *sem);`
  - `int sem_getvalue(sem_t *sem, int * sval);`

Producenci i Konsumenci

- Sformułowanie problemu
  Dwa rodzaje procesów:
  - Producenci, wykonując wewnętrzną funkcję *Produkuj*, tworzą elementy danych, które muszą być potem przekazane do Konsumentów,
  - Konsumenci po otrzymaniu elementu danych wykonują pewne obliczenia w wewnętrznej funkcji *Konsumuj*.

  Zrealizować przekazywanie danych od Producentów do Konsumentów.
- Założenia:
  - jeden proces Producent,
  - jeden proces Konsument,
  - przekazywanymi elementami danych są pojedyncze liczby całkowite.

Producenci i Konsumenci ($1 \rightarrow 1$)

```
main (){
  semafor mozna_czytac = 0;
  semafor mozna_pisac = 1;
  int x; //dana wspolna

  cobegin {
    {  //Producent
      int lx;
      while (1){
        lx = Produkuj();
        P (mozna_pisac);
        x = lx;
        V (mozna_czytac);
      }
    }//Producent
```

```
    { //Konsument
      int lx;
      while (1){
        P (mozna_czytac);
        lx = x;
        V (mozna_pisac);
        Konsumuj (lx);
      }
    }//Konsument
  } // cobegin
}
```


Producenci i Konsumenci ($N \rightarrow 1$)

```
main (){
  semafor mozna_czytac = 0;
  semafor mozna_pisac = 1;
  int x; //dana wspolna

  cobegin {
    parfor i = 1 to N do { //Producenci
      int lx;
      while (1){
        lx = Produkuj();
        P (mozna_pisac);
        x = lx;
        V (mozna_czytac);
      }
    } //parfor - Producenci
```

```
    { //Konsument
      int lx;
      while (1){
        P (mozna_czytac);
        lx = x;
        V (mozna_pisac);
        Konsumuj (lx);
      }
    }//Konsument
  } // cobegin
}
```


Producenci i Konsumenci ($N \rightarrow M$)

```
main (){
  semafor mozna_czytac = 0;
  semafor mozna_pisac = 1;
  int x; //dana wspolna

  cobegin {
    parfor i = 1 to N do { //Producenci
      int lx;
      while (1){
        lx = Produkuj();
        P (mozna_pisac);
        x = lx;
        V (mozna_czytac);
      }
    } //parfor - Producenci
```

```
    //Konsumenci
    parfor i = 1 to M do {
      int lx;
      while (1){
        P (mozna_czytac);
        lx = x;
        V (mozna_pisac);
        Konsumuj (lx);
      }
    } //parfor - Konsumenci
  } // cobegin
}
```


Przypomnienie: Producenci i Konsumenci ($1 \rightarrow 1$)

```
main (){
  semafor mozna_czytac = 0;
  semafor mozna_pisac = 1;
  int x; //dana wspolna

  cobegin {
    {  //Producent
      int lx;
      while (1){
        lx = Produkuj();
        P (mozna_pisac);
        x = lx;
        V (mozna_czytac);
      }
    }//Producent
```

```
    { //Konsument
      int lx;
      while (1){
        P (mozna_czytac);
        lx = x;
        V (mozna_pisac);
        Konsumuj (lx);
      }
    }//Konsument
  } // cobegin
}
```

Bufor cykliczny

```
main(){
  int b[rozmiar], g = 0, o = 0;
  mutex dostep;
  semafor wolne = rozmiar;
  semafor zajete = 0;

  cobegin {
      while (1){ //Producent
        int lx;
        lx = Produkuj();
(1)     P (wolne);
        lock (dostep);
        b[g] = lx;
        g = (g + 1) % rozmiar;
        unlock (dostep);
(2)     V (zajete);
      }//Producent

      while (1){  //Konsument
        int lx;
(3)     P (zajete);
        lock (dostep);
        lx = b[o];
        o = (o + 1) % rozmiar;
        unlock (dostep);
(4)     V (wolne);
        Konsumuj (lx);
    }//Konsument
  } // cobegin
}
```


Czytelnicy i Pisarze

- Sformułowanie problemu
  W programie istnieje zbiór danych, do którego odwołują się procesy współbieżne. Mamy dwa rodzaje procesów:
  - *Czytelnik* jedynie odczytuje dane i może to robić razem z innymi *Czytelnikami*
  - *Pisarz* odczytuje i zapisuje dane; *Pisarz* musi być jedynym procesem, który w danej chwili manipuluje zbiorem danych

  Zrealizować synchronizację wykonania procesów.

Czytelnicy i pisarze

```
main (void){
  semafor pisz = 1;
  mutex sekcja;
  int cc = 0;

  cobegin {
    //P1 .. Pn2
    parfor i := 1 to n2 do {
      P (pisz);
      ...  //zmieniaj dane
      V (pisz);
    }//parfor - Pisarze

    //C1 .. Cn1
    parfor i = 1 to n1 do {
      lock (sekcja);
      cc = cc + 1;
      if (cc == 1)
        P(pisz);
      unlock (sekcja);
      ... //czytaj dane
      lock(sekcja);
      cc = cc - 1;
      if (cc == 0)
        V (pisz);
      unlock (sekcja);
    }// parfor - Czytelnicy
  } // cobegin
}
```


Bariera synchronizacyjna

```
parfor i = 1 to n do { //Procesy P1 ..Pn
  A();
  B();
} //parfor
```

żaden proces $P_1, \ldots, P_n$ nie rozpocznie wykonywania procedury $B()$, dopóki wszystkie procesy $P_1, \ldots, P_n$ nie zakończą wykonywania procedury $A()$.

```
parfor i = 1 to n do { //Procesy P1 ..Pn
  A();
  bariera();
  B();
} //parfor
```

Bariera synchronizacyjna

```
//zmienne globalne i ich inicjalizacja
int ile = 0;
mutex s_ile;
semafor s_czekajacy = 0;

void bariera(void){
  lock(s_ile);
  ile ++;
  if (ile == n)
    for (int i = 0; i < n; i ++)
      V(s_czekajacy);
  unlock(s_ile);
  P(s_czekajacy);
}
```

```
main (){
  //Procesy P1 ..Pn
  parfor i = 1 to n do {
    A();
    bariera();
    B();
  } //parfor
} //main
```


Bariera synchronizacyjna

```
//zmienne globalne
//  i ich inicjalizacja
semafor sb1 = 0;
semafor sb2 = 0;

void bariera(void){
  V (sb1);
  P (sb2);
}//bariera

main (){
  cobegin{
    //Procesy P1 ..Pn
    parfor i = 1 to n do {
      A();
      bariera();
      B();
    } //parfor
```

```
    while (1){  //Koordynator
      int i;
      for (i = 1; i <= n; i++) {
        P (sb1);
      };
      for (i = 1; i <= n; i++) {
        V (sb2);
      };
    }//while - Koordynator
  }//cobegin
} //main
```


Wady semaforów

- *P* i *V* są niestrukturalne:

  ```
  P (semafor);
  sekcja krytyczna
  V (niewłaściwy_semafor);
  ```

- możliwość umieszczenia poza sekcją krytyczną instrukcji, która powinna się w niej znaleüć
- brak związku (składniowego) między zasobem a semaforem
- jeden mechanizm do dwóch typów synchronizacji:
  - wzajemne wykluczanie
  - synchronizacja warunków

Wady semaforów

```
main(){
  int b[rozmiar], g = 0, o = 0;
  mutex dostep;
  semafor wolne = rozmiar;
  semafor zajete = 0;

  cobegin {
      while (1){ //Producent
        int lx;
        lx = Produkuj();
(1)     P (wolne);
        lock (dostep);
        b[g] = lx;
        g = (g + 1) % rozmiar;
        unlock (dostep);
(2)     V (zajete);
      }//Producent
```

```
      while (1){  //Konsument
        int lx;
(3)     P (zajete);
        lock (dostep);
        lx = b[o];
        o = (o + 1) % rozmiar;
        unlock (dostep);
(4)     V (wolne);
        Konsumuj (lx);
    }//Konsument
  } // cobegin
}
```

Monitory

Elementy składowe monitora:

- zmienne permanentne:
  - zmienne współdzielone (zasób)
  - zmienne opisujące warunki (ang. condition variables); zmienne warunkowe
- procedury operujące na zasobach
- instrukcje inicjalizacyjne

Monitor a klasa

Monitor	**Obiekt**
zmienne współdzielone (zasób)	zmienne własne
procedury operujące na zasobach	metody
wszystkie procedury są „publiczne”	metody publiczne, chronione, prywatne
monitor.procedura(p)	obiekt.metoda(p)
instrukcje inicjalizacyjne	konstruktor
	destruktor
zmienne warunkowe	
	dziedziczenie, polimorfizm

Struktura monitora

```
monitor identyfikator {
  private:
    deklaracje zmiennych permanentnych;
  public:
    identyfikator (){ //instrukcje inicjalizacyjne
      inicjalizacja zmiennych permanentnych
    };
    op1 (parametry){ //operacja 1
      deklaracje zmiennych lokalnych w op1;
      kod procedury op1
    };
    :
    opN (parametry){ //operacja N
      deklaracje zmiennych lokalnych w opN;
      kod procedury opN
    };
}
```


Operacje na zmiennych warunkowych

```
void wait (void){
  wstaw proces wykonujący operację do kolejki
    procesów skojarzonej ze zmienną warunkową;
}
```

```
void signal (void) {
  if (jakiś proces jest zawieszony na zmiennej) {
    wstrzymaj proces wykonujący operację;
    wznów jeden z procesów zawieszonych
      na zmiennej warunkowej;
  }
}
```

Zapisywanie operacji: *zmienna_warunkowa.operacja*()
np. `niepusty.signal()`

Bufor cykliczny

```
monitor bufor {
  private:
    const int n = ...
    T elem[n];
    int głowa, ogon;//zakres 0...n-1
    int zajete;     //zakres 0...n
    warunek niepełny, niepusty;
  public:
    //część inicjalizacyjna
    void bufor (void){
      zajete = 0;
      głowa = 0;
      ogon = 0;
    }
```

```
    void wstaw (T p){
      if (zajete == n)
        niepełny.wait();
      elem[ogon] = p;
      zajete += 1;
      ogon = (ogon + 1) % n;
      niepusty.signal();
    }
    void pobierz (T &q){
      if (zajete == 0)
        niepusty.wait();
      q = elem[glowa];
      zajete -= 1;
      glowa = (glowa + 1 ) % n;
      niepełny.signal();
    }
}
```


Producenci i Konsumenci

```
main (){
  bufor b;

  cobegin {
    //Producenci
    parfor i = 1 to N do {
      int lx;
      while (1){
        lx = Produkuj();
        b.wstaw(lx);
      }
    } //parfor - Producenci
```

```
    //Konsumenci
    parfor i = 1 to M do {
      int lx;
      while (1){
        b.pobierz(lx);
        Konsumuj(lx);
      }
    } //parfor - Konsumenci
  } // cobegin
}
```


Czytelnicy i Pisarze

```
const int n1 = ...
const int n2 = ...
main (void){
  monitor_cp arbiter;
  typ_bazy_danych baza_danych;

  cobegin {
    //C1 .. Cn1
    parfor i = 1 to n1 do {
      ...  //instrukcje
      arbiter.poczatek_czytania();
      ... //czytaj dane z bazy
      arbiter.koniec_czytania();
      ... //instrukcje
    }// parfor - Czytelnicy
```

```
    //P1 .. Pn2
    parfor i = 1 to n1 do {
      ... //instrukcje
      arbiter.poczatek_pisania();
      ... //zmieniaj dane w bazie
      arbiter.koniec_pisania();
      ... //instrukcje
    } //parfor - Pisarze
  } //cobegin
}
```


Czytelnicy i Pisarze

```
monitor monitor_cp {
  private:
    int cc = 0;
    bool pisanie = false;
    warunek można_czytać, można_pisać;
  public:
    monitor_cp(){
    }
    void początek_czytania(void){
      if (pisanie || !  empty(można_pisać))
        można_czytać.wait;
      cc = cc + 1;
      można_czytać.signal();
    }; //początek_czytania
    void koniec_czytania(void) {
      cc = cc - 1;
      if (cc == 0)
        można_pisać.signal();
    } //koniec_czytania
}
```

Czytelnicy i Pisarze

```
    void początek_pisania(void) {
      if (cc > 0 || pisanie)
        można_pisać.wait();
      pisanie = true;
    } //początek_pisania
    void koniec_pisania(void) {
      pisanie = false;
      if (!  empty(można_czytać))
        można_czytać.signal();
      else
        można_pisać.signal();
    } //koniec_pisania;
} //monitor_cp
```


Bariera synchronizacyjna

Monitor

```
const int n = ...;
monitor monitor_b {
  private:
    int czeka = 0;
    warunek bariera;
  public:
    void monitor_b(){
    }
    void synchronizuj(void){
      if (czeka == n - 1) {
        for (i = 1; i <= n - 1; i++)
          bariera.signal();
        czeka = 0;
      } else {
        czeka = czeka + 1;
        bariera.wait();
      }
    }//synchronizuj
} //monitor_b
```

```
main(void){
  monitor_b m;

  parfor i = 1 to n do {
    A();
    //bariera()
    m.synchronizuj();
    B();
  }//parfor
}
```


Zmienne warunkowe a semafory

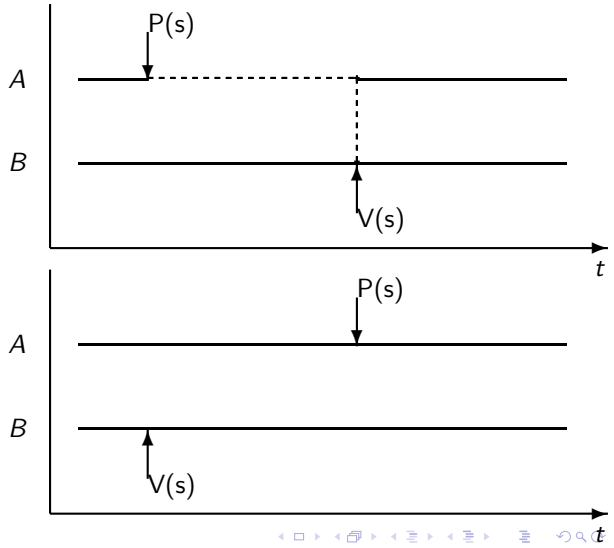
Porównanie operacji

Zmienna warunkowa	**Semafor**
operacja *wait()*	operacja *P()*
zawieszenie procesu wykonującego operację (zawsze)	▶ zmniejszenie wartości licznika ▶ zawieszenie procesu wykonującego operację, jeżeli wartość licznika ujemna
operacja *signal()*	operacja *V()*
nie ma żadnego efektu, jeżeli żaden proces nie jest „zawieszony” na zmiennej	jeżeli żaden proces nie jest „zawieszony” na semaforze, to zwiększa wartość licznika

Zmienne warunkowe a semafory

Semafor

```
semafor s = 0;
cobegin {
  {  //proces A
    ... //instrukcje
    P(s)
    ... //instrukcje
  }
  {  //proces B
    ... //instrukcje
    V(s)
    ... //instrukcje
  }
}//cobegin
```



Zmienne warunkowe a semafory

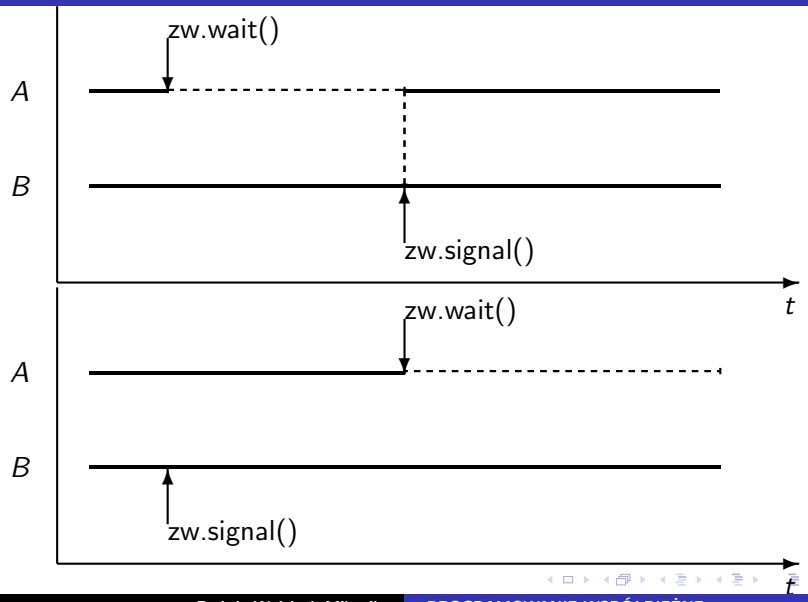
Zmienna warunkowa

```
monitor mn{
  private:
    warunek zw;
  public:
    void m(void) {
    }
    void sprawdz (void){
      zw.wait();
    }
    void ustaw (void){
      zw.signal();
    }
}
```

```
mn m;   //monitor
cobegin {
  {  //proces A
    ... //instrukcje
    m.sprawdz(); // zw.wait()
    ... //instrukcje
  }
  {  //proces B
    ... //instrukcje
    m.ustaw(); // zw.signal()
    ... //instrukcje
  }
}//cobegin
```

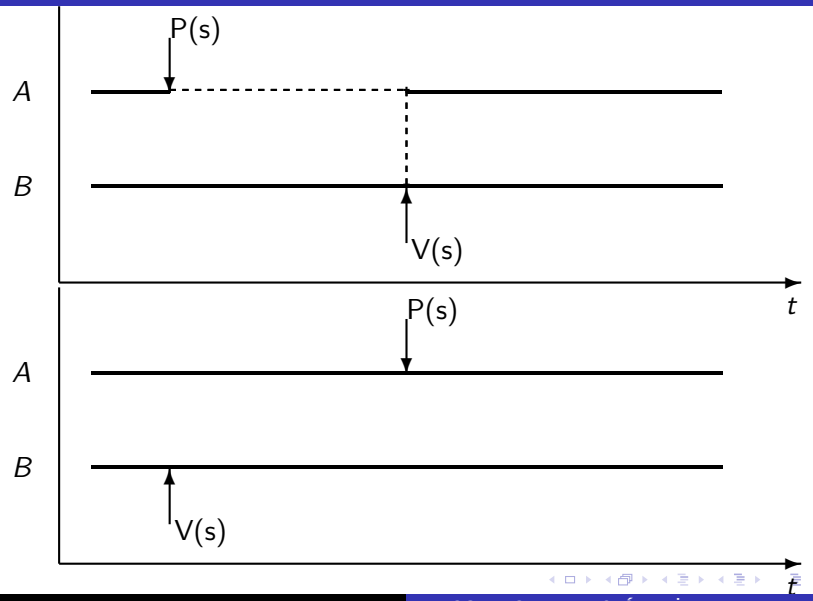

Zmienne warunkowe a semafory

Zmienna warunkowa *zw*



Zmienne warunkowe a semafory

Semafor *s*



Przykład: Pthreads

Zmienne warunnkowe

- Inicjalizacja
  - `PTHREAD_PROCESS_PRIVATE`
  - `PTHREAD_PROCESS_SHARED`
- `pthread_cond_destroy`
- `pthread_cond_wait`
- `pthread_cond_timedwait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`

Inicjalizacja i usuwanie zmiennej warunkowej

- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
-
```
pthread_cond_t cond;
cond = calloc(1, sizeof (pthread_cond_t));
```
- `int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);`

```
pthread_cond_t cv;
ret = pthread_cond_init(&cv, NULL);

cv  = PTHREAD_COND_INITIALIZER;
```

- `int pthread_cond_destroy(pthread_cond_t *cv);`

```
pthread_cond_t cv;
int ret;

ret = pthread_cond_destroy(&cv);
```


pthread_cond_wait(pthread_cond_t *cv);

- `int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);`

```
pthread_cond_t cv;
pthread_mutex_t mutex;
int ret;

...
ret = pthread_cond_wait(&cv, &mutex);
```

- Scenariusz użycia
  1. `pthread_mutex_lock(m);`
  2. `while(condition_is_false) pthread_cond_wait(cv, m);`
  3. `pthread_mutex_unlock(m);`

pthread_cond_signal(pthread_cond_t *cv);

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count() {
   pthread_mutex_lock(&count_lock);

   while (count == 0)
      pthread_cond_wait(&count_nonzero, &count_lock);
   count = count - 1;
   pthread_mutex_unlock(&count_lock);
}

increment_count() {
   pthread_mutex_lock(&count_lock);
   if (count == 0)
      pthread_cond_signal(&count_nonzero);
   count = count + 1;
   pthread_mutex_unlock(&count_lock);
}
```


pthread_cond_broadcast(pthread_cond_t *cv)

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount){
   pthread_mutex_lock(&rsrc_lock);
   while (resources < amount)
      pthread_cond_wait(&rsrc_add,&rsrc_lock);
   resources -= amount;
   pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount) {
   pthread_mutex_lock(&rsrc_lock);
   resources += amount;
   pthread_cond_broadcast(&rsrc_add);
   pthread_mutex_unlock(&rsrc_lock);
}
```

pthread_cond_timedwait

```
int pthread_cond_timedwait( pthread_cond_t *cv,
                            pthread_mutex_t *mp,
                            const struct timespec *abstime);
```

```
pthread_timestruc_t to;
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;
ret = pthread_cond_timedwait(&cv, &mp, &abstime);
pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
  err = pthread_cond_timedwait(&c, &m, &to);
  if (err == ETIMEDOUT)
  break; //timeout
}
pthread_mutex_unlock(&m);
```

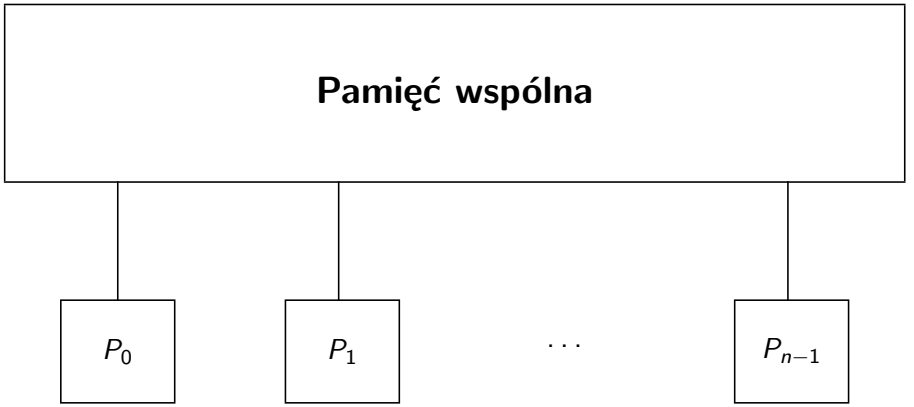

Część IV

Programowanie równoległe

Analiza algorytmów równoległych

- Krok elementarny
  - Krok obliczeniowy
  - Pojedynczy etap przesyłania danych
- Złożoność czasowa algorytmu równoległego $T(n,p)$
- Przyspieszenie
$$S(n,p) = \frac{T_s(n)}{T(n,p)}$$
- Koszt
$$C(n,p) = T(n,p) \cdot p$$
- Efektywność
$$E(n,p) = \frac{S(n,p)}{p}$$
- Często zachodzi $p = p(n)$ i stąd
$$S_p = \frac{T_s}{T_p}, \quad C_p = T_p \cdot p, \quad Ep = \frac{S_p}{p}$$

Parallel Random Access Machine



- Swobodny dostęp do dowolnej komórki pamięci w stałym czasie ($T_{dostpu}(n) = O(1)$)
- Procesory pracują synchronicznie
- Procesory mogą wykonywać różne instrukcje
  Synchroniczny MIMD-SM z darmową komunikacją
- Dowolnie duża przestrzeń adresowa
- Dowolna liczba procesorów

Parallel Random Access Machine

- EREW PRAM (Exclusive Read, Exclusive Write)
- CREW PRAM (Concurrent Read, Exclusive Write)
- CRCW PRAM (Concurrent Read, Concurrent Write)
  - COMMON — wszystkie wartości zapisywane do komórki pamięci w danym kroku PRAM muszą być równe, w przeciwnym wypadku żadna z nich nie zostanie zapisana
  - ARBITRARY — spośród wartości zapisywanych do komórki pamięci w danym kroku PRAM zostanie zapisana jedna, nie wiadomo która
  - PRIORITY — spośród wartości zapisywanych do komórki pamięci w danym kroku PRAM zostanie zapisana jedna, pochodząca od procesora o najwyższym priorytecie (np. najniższym lub najwyższym numerze)
  - SUM — do komórki pamięci zostanie zapisana suma wszystkich wartości zapisywanych do niej w danym kroku PRAM

Mnożenie macierzy
Algorytm sekwencyjny

```
// (p = 1)
for i := 1 to n do
  for j := 1 to n do
    C[i, j] := 0;
    for s := 1 to n do
      C[i, j] := C[i, j] + A[i, s] · B[s, j];
    end for;
  end for;
end for;
```

$$T(n) = O(n^3)$$
$$C(n) = O(n^3)$$

Mnożenie macierzy
CRCW SUM PRAM

```
//(p = n^3)
parfor i := 1 to n do
  parfor j := 1 to n do
    parfor s := 1 to n do
      C[i, j] := 0;
      C[i, j] := A[i, s] · B[s, j];
    end parfor;
  end parfor;
end parfor;
```

- $T(n, p) = O(1)$
- $S(n, p) = O(n^3) = O(p)$
- $C(n, p) = O(n^3)$
- $E(n, p) = O(1)$

Mnożenie macierzy
CREW PRAM

```
// (p = n^2)
parfor i := 1 to n do
  parfor j := 1 to n do
    C[i, j] := 0;
    for s := 1 to n do
      C[i, j] := C[i, j] + A[i, s] · B[s, j];
    end for;
  end parfor;
end parfor;
```

- $T(n, p) = O(n)$
- $S(n, p) = O(n^2) = O(p)$
- $C(n, p) = O(n^3)$
- $E(n, p) = O(1)$

Mnożenie macierzy
EREW PRAM

// $(p = n^2)$
parfor i := 1 to n do
  parfor j := 1 to n do
    C[i, j] := 0;
    for s := 1 to n do
      C[i, j] := C[i, j] + A[i, (j + s) mod n] · B[(j + s) mod n, j];
    end for;
  end parfor;
end parfor;

- $T(n, p) = O(n)$
- $S(n, p) = O(n^2) = O(p)$
- $C(n, p) = O(n^3)$
- $E(n, p) = O(1)$

Część V

OpenMP

Plan

1. Mnożenie macierzy (przykład)
   - rozwiązanie sekwencyjne
   - rozwiązanie z zastosowaniem pthreads
   - rozwiązanie z zastosowaniem OpenMP
2. Wprowadzenie do OpenMP
3. Wyrażanie wykonania równoległego
4. Rozdzielanie pracy
5. Rodzaje zmiennych
6. Synchronizacja
7. Zagnieżdżanie równoległości
8. Współpraca z zewnętrznymi bibliotekami

Mnożenie macierzy
Wersja sekwencyjna: pseudokod

```
for i := 1 to n do
  for j := 1 to n do
    C[i][j] := 0;
    for s := 1 to n do
      C[i][j] := C[i][j] + A[i][s] * B[s][j];
    end for;
  end for;
end for;
```

Mnożenie macierzy
Wersja sekwencyjna: implementacja w C

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;


  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```


Mnożenie macierzy
PRAM CREW

```
parfor i := 1 to n do
  parfor j := 1 to n do
    C[i][j] := 0;
    for s := 1 to n do
      C[i][j] := C[i][j] + A[i][s] * B[s][j];
    end for;
  end parfor;
end parfor;
```


Mnożenie macierzy
(PONOWNIE) Wersja sekwencyjna: pseudokod

```
for i := 1 to n do
  for j := 1 to n do
    C[i][j] := 0;
    for s := 1 to n do
      C[i][j] := C[i][j] + A[i][s] * B[s][j];
    end for;
  end for;
end for;
```


Mnożenie macierzy
Wersja równoległa: C + biblioteka pthread

Fakty

- Rozsądny rozmiar macierzy ($n \geq 64$)
- Utworzenie wątku kosztuje
- Źródłem mocy obliczeniowej jest procesor, a nie wątek
- Rozpatrujemy system komputerowy z małą lub średnia liczbą procesorów

Decyzja

Wątek i oblicza wartości elementów w wierszach $i, i + NT, i + 2NT, i + 3NT, \ldots$ macierzy wynikowej C
(NT — Liczba wątków)

Mnożenie macierzy

Wersja równoległa: C + biblioteka pthread (cześć I).

```
typedef struct {
  double (*a) [N];
  double (*b) [N];
  double (*c) [N];
  sem_t * s;
  int id;
} thread_args;

void * thread_proc (void * a){
  thread_args * arg = a;
  int i, j, s;
  for (i = arg->id; i < N; i += NT)
    for (j = 0; j < N; j ++){
      arg->c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        arg->c[i][j] += arg->a[i][s] * arg->b[s][j];
    }
  assert (sem_post (arg->s) == 0);
  return NULL;
}
```


Mnożenie macierzy

Wersja równoległa: C + biblioteka pthread (część II)

```
void matrix_mult (double a [N][N], double b[N][N], double c[N][N]){
  int i;
  thread_args ta[NT];
  sem_t s;
  pthread_t p;

  assert (sem_init (&s, 0, 0) == 0);
  for (i = 0; i < NT; i ++){
    thread_args x = { a, b, c, &s, i};
    ta[i] = x;
    assert (0 == pthread_create (&p, NULL, thread_proc, ta + i));
  }
  for (i = 0; i < NT; i ++)
    assert (sem_wait (&s) == 0);
}
```


Mnożenie macierzy

Wersja rownoległa: C + OpenMP

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;

  #pragma omp parallel for private(j, s)
  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```


Mnożenie macierzy

Wersja sekwencyjna: implementacja w C

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;


  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```

OpenMP

- Sposób pisania programów wielowątkowych
- Skoncentrowany na zrównoleglaniu iteracji
  Rozbudowany w wesji 3.0 standardu
- OpenMP definiuje
  - zbiór dyrektyw kompilatora
  - (mały) zbiór podprogramów
  - powiązania (bindings) dla takich języków programowania jak: Fortran, C, C++
- OpenMP **rozszerza**, ale **nie modyfikuje** bazowego języka programowania

OpenMP

- Cele
  - Skalowalne programy równoległe dla komputerów równoległych z pamięcią wspólną
  - Przenaszalna wydajność
  - Szybki i stopniowy rozwój równoległej wersji istniejącego kodu sekwencyjnego
  - Łatwa jednoczesna pielęgnacja obe wersji kodu (sekwencyjnej i równoległej)
  - Tworzenie równoległych programów od podstaw
- Historia
  - Listopad 1996 — spec. ANSI X3H5
  - Październik 1997 — spec. 1.0 OpenMP (FORTRAN)
  - 1998 — spec. 1.0 OpenMP (C/C++)
  - 1999 — spec. 1.1 OpenMP (FORTRAN)
  - 2006 — dostępne kompilatory zgodne ze spec. 2.5 (FORTRAN, C, C++)
  - Feb 2008 — spec. 3.0
- Strona domowa OpenMP: `www.openmp.org`
- OpenMP user group: `www.cOMPunity.org`

Kompilatory wspierające OpenMP

- Intel (Linux, MSWindows)
  - kompilator C++ wersja 9.x ↑
  - kompilator FORTRAN
- Microsoft Visual Studio 2005 ↑ (MSWindows)
- GNU C++ ver. 4.2.0 ↑
- Sun Studio wersja 11 ↑ (Solaris)
- kompilatory firmy The Portland Group, Inc. (C/C++/Fortran) (Intel Linux, Intel Solaris, Intel Windows/NT, MacOSX/Intel)
- IBM XL Fortran and C (IBM AIX)
- HP, SGI, Fujitsu

Wykonanie programu OpenMP

```
void matrix_mult (double a[N][N],
                  double b[N][N],
                  double c[N][N]){
  int i, j, s;

  #pragma omp parallel for private(j, s)
  for (i = 0; i < N; i ++){
    for (j = 0; j < N; j ++){
      c[i][j] = 0.0;
      for (s = 0; s < N; s ++)
        c[i][j] += a[i][s] * b[s][j];
    }
  }
}
```

Programowanie z zastosowaniem OpenMP

- Dyrektywy kompilatora
  - Format
    `#pragma omp` *directive* [*clause*]
  - Region równoległy
    `#pragma omp parallel` [*clause*]
  - Podział pracy
  - Synchronizacja
- Biblioteka
  - Plik nagłówkowy `#include <omp.h>`
  - Funkcje informacyjne
  - Funkcje synchronizacyjne
  - Dynamiczne dostosowanie liczby wątków
  - Zagnieżdżona równoległość
- Zmienne środowiska
  - `OMP_DYNAMIC`
  - `OMP_NUM_THREADS`
  - `OMP_NESTED`
  - `OMP_SCHEDULE`

Wyrażanie równoległości

- Dyrektywa `parallel`

```
#pragma omp parallel
{
  printf("hello world\n");
}
```

- Przydatne funkcje informacyjne
  - Odczytywanie numeru bieżącego wątku: `omp_get_thread_num()`
  - Odczytywanie liczby wątków: `omp_get_num_threads()`
  - Wykrywanie regionu równoległego: `omp_in_parallel()`
- Ustawianie liczby wątków
  - dla całego programu
    - funkcja `omp_set_num_threads()`
    - zmienna środowiska `OMP_NUM_THREADS`
  - dla regionu równoległego
    klauzula `num_threads`
- Dynamiczne dostosowanie liczby wątków
  - funkcja `omp_set_dynamic ()`
  - funkcja `omp_get_dynamic ()`
  - zmienna środowiska `OMP_DYNAMIC`

Dyrektywy podziału pracy

- dyrektywa `sections`
- połączone dyrektywy `parallel sections`
- dyrektywa `for`
- połączone dyrektywy `parallel for`
- dyrektywa `single`

```
#pragma omp single
{
  doSth (defaultDelay);
  single2 = omp_get_thread_num();
}
```

- dyrektywa `master`

```
#pragma omp master
master = omp_get_thread_num();
```


Dyrektywa `sections`

Składnia

```
#pragma omp sections [klauzula]
  {
    [ #pragma omp section ]
      blok1
    [ #pragma omp section
      blok2 ]
    [ #pragma omp section
      blok3 ]
    ...
  }
```

```
cobegin
    blok1
    blok2
    blok3
    ...
coend;
```

Dyrektywa sections

Przykład

```
#pragma omp parallel
{
  #pragma omp sections
  {
    printf ("Operacja wykonana przez wątek %d spośród %d\n",
            omp_get_thread_num (), omp_get_num_threads());

    #pragma omp section
    printf ("Operacja wykonana przez wątek %d spośród %d\n",
            omp_get_thread_num (), omp_get_num_threads());

    #pragma omp section
    {
      printf ("Operacja wykonana przez wątek %d spośród %d\n",
              omp_get_thread_num (), omp_get_num_threads());
    }
    #pragma omp section
    {
      printf ("Operacja wykonana przez wątek %d spośród %d\n",
              omp_get_thread_num (), omp_get_num_threads());
    }
  }
}
```


Dyrektywa sections

sections vs parallel

```
#pragma omp parallel
{
    printf ("Operacja wykonana przez wątek %d spośród %d\n",
            omp_get_thread_num (), omp_get_num_threads());
}
```

`sections`
- Dokładnie 4 wiadomości
- Wypisane numery wątków tworzą kombinację z powtórzeniami 4 liczb całkowitych z zakresu $[0 \ldots \textit{rozmiarZespołu} - 1]$

`parallel`
- Dokładnie *rozmiarZespołu* wiadomości
- Wypisane numery wątków tworzą permutację 4 liczb całkowitych z zakresu $[0 \ldots \textit{rozmiarZespołu} - 1]$

Dyrektywa parallel sections

```
#pragma omp parallel sections
{
  printf ("Operacja wykonana przez wątek %d spośród %d\n",
          omp_get_thread_num (), omp_get_num_threads());

  #pragma omp section
  printf ("Operacja wykonana przez wątek %d spośród %d\n",
          omp_get_thread_num (), omp_get_num_threads());

  #pragma omp section
  {
    printf ("Operacja wykonana przez wątek %d spośród %d\n",
            omp_get_thread_num (), omp_get_num_threads());
  }
  #pragma omp section
  {
    printf ("Operacja wykonana przez wątek %d spośród %d\n",
            omp_get_thread_num (), omp_get_num_threads());
  }
}
```


Dyrektywa for

- Składnia

  `#pragma omp for` [*clause*]
  for-loop

- Przykład

```
int a[ARR_SIZE];
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i < ARR_SIZE; i ++)
      a[i] = 1;
}
```

Dyrektywa `for`

Klauzula `schedule`

- Szeregowanie domyślne: zależne od implementacji
- `schedule(static, rozmiar_porcji)`
  - porcje pracy o rozmiarze `rozmiar_porcji` lub równe rozmiarowi całości podzielonemu przez liczbę wątków (jeżeli `rozmiar_porcji` nie jest podany)
  - cyklicznie, w kolejności numerów wątków (round-robin)
- `schedule(dynamic, rozmiar_porcji)`
  - porcje pracy o rozmiarze `rozmiar_porcji` lub 1 (jeżeli `rozmiar_porcji` nie jest podany)
  - kolejka porcji
- `schedule(guided, rozmiar_porcji)`
  - porcje pracy o rozmiarze malejącym do `rozmiar_porcji` lub do 1 (jeżeli `rozmiar_porcji` nie jest podany)
  - kolejka porcji
- `schedule(runtime)`
  szeregowanie zgodne z wartością zmiennej środowiska `OMP_SCHEDULE`

Dyrektywa `parallel for`

- Składnia
  `#pragma omp parallel for [`*clause*`]`
  for-loop
- Przykład

```
double a [N][N], b [N][N], c [N][N];
int i;
...
#pragma omp parallel for
for (i = 0; i < N; i ++){
  int j, s;
  for (j = 0; j < N; j ++){
    c[i][j] = 0.0;
    for (s = 0; s < N; s ++)
      c[i][j] += a[i][s] * b[s][j];
  }
}
```


Model danych

- Zmienne mogą być
  - Wspólne (`shared`)
  - Prywatne (`private`)
  - Poddane operacji redukcji (`reduction`)
- Domyślnie
  - Zazwyczaj: `shared`
  - Prywatne
    - Zmienne lokalne (ale **nie** `static`) podprogramów wykonywanych przez wątek
    - Zmienne automatyczne bloków kodu wykonywanych przez wątek
    - Zmienne kontrolne zrównoleglanej iteracji `for`

Model danych

Deklaracje

- Klauzule
  - `shared (` *list* `)`
  - `private (` *list* `)`
  - `default (` *list* `)`
  - `firstprivate`
  - `lastprivate`
  - `reduction (`*operator* `:` *list* `)`
- Zmienne własne wątku
  - dyrektywa `threadprivate`
  - klauzula `copyin`

Model danych
Przykład

```
void matrix_mult2(){
  double a [N][N], b [N][N], c [N][N];
  int i;
  ...
  #pragma omp parallel
  {
    #pragma omp for
    for (i = 0; i < N; i ++){
      int j, s;
      for (j = 0; j < N; j ++){
        c[i][j] = 0.0;
        for (s = 0; s < N; s ++)
          c[i][j] += a[i][s] * b[s][j];
      }
    }
  }
}
```


Model danych
Przykład

```
void matrix_mult3(){
  double a [N][N], b [N][N], c [N][N];
  int i, j, s;
  ...
  #pragma omp parallel
  {
    #pragma omp for private(j, s)
    for (i = 0; i < N; i ++){
      for (j = 0; j < N; j ++){
        c[i][j] = 0.0;
        for (s = 0; s < N; s ++)
          c[i][j] += a[i][s] * b[s][j];
      }
    }
  }
}
```


Model danych
Spójność danych

- Spójność sekwencyjna
  - zalety
  - wady
- Przyjety model spójności
  - tymczasowy widok danych
  - punkty synchronizacyjne
    - bariera (zarówno jawna jak i implicite)
    - wejście i wyjście do/z regionu krytycznego (`critical`)
    - wejście i wyjście do/z funkcji operujących na lock-ach
  - dyrektywa `flush` (*list*)

Klauzula `reduction`

- Składnia: `reduction` (*operator* : *list*)
- Lista operatorów: `+, *, -, &, ^, |, &&, ||`
- Przykład:

```
#define ARR_SIZE 100000
int a [ARR_SIZE], i, sum;
...
sum = 0;
#pragma omp parallel
{
  #pragma omp for reduction (+: sum)
  for (i = 0; i < ARR_SIZE; i ++)
    sum += a[i];
}
```

Synchronizacja

- Synchronizacja implikowana
  - dyrektywa `for`
  - dyrektywa `sections`
  - dyrektywa `single`

  klauzula `nowait`
- Synchronizacja jawna (dyrektywy)
  - `master`
  - `critical`
  - `atomic`
  - `barrier`
  - dyrektywa `ordered` i klauzula `ordered`
- Synchronizacja jawna (`lock_t`)

Przykład `nowait`

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < numThreads; i++)
    doSth(i * delay);

  #pragma omp sections
  {
    printf("Wątek #%d \n", omp_get_thread_num());

    #pragma omp section
    printf("Wątek #%d\n", omp_get_thread_num());

    #pragma omp section
    printf("Wątek #%d\n", omp_get_thread_num());
  }
}
```


Dyrektywa `critical`

- Składnia

  `#pragma omp critical [ (`*name*`)]`
  block
- Przykład

```
#define ARR_SIZE 100000
int a [ARR_SIZE], i, sum;
...
sum = 0;
#pragma omp parallel for
for (i = 0; i < ARR_SIZE; i ++)
  #pragma omp critical
  sum += a[i];
printf ("Sum == %d\n", sum);
```


Nazwana dyrektywa `critical`

```
int cs_even = 0, cs_odd = 0;
...
#pragma omp parallel for num_threads (numThreads)
for (i = 0; i < numThreads; i ++)
  if (i % 2)
    #pragma omp critical (odd)
    cs_odd ++;
  else
    #pragma omp critical (even)
    cs_even ++;
```

Przykład dyrektywy ordered

```
#pragma omp parallel for ordered
for (i = 0; i < numThreads; i ++)
{
  do_sth_in_any_order();
  #pragma omp ordered
  doSth(); //to zostanie wykonane w kolejnosci
}
```


Pozostałe zagadnienia

- Zagnieżdżanie równoległości
  - funkcja `omp_set_nested ()`
  - funkcja `omp_get_nested ()`
  - zmienna środowiska `OMP_NESTED`
- Stosowanie bibliotek

Część VI

Pamięć rozproszona: komunikacja i synchronizacja

Omawiane zagadnienia

1. Przesyłanie wiadomości
2. Podstawowe instrukcje: `send`, `receive`
3. Nazwy procesów
4. Synchronizacja nadawcy i odbiorcy
5. Komunikacja selektywna (rozkazy dozorowane)
6. Kanały

Przesyłanie wiadomości

- Podstawowe operacje:
  - `send` *lista wyrażeń* `to` *określenie odbiorcy*
  - `receive` *lista zmiennych* `from` *określenie nadawcy*
- Założenie: komunikat dotrze do odbiorcy
- Zagadnienia
  - Określenie nadawcy i odbiorcy
    - Nazwy procesów (grup procesów)
    - „Urządzenia" komunikacyjne: kanały
  - Synchronizacja nadawcy i odbiorcy

Nazwy procesów

Określeniem odbiorcy (nadawcy) jest unikalna nazwa procesu lub grupy procesów

- identyfikator procesu w języku programowania
- numer procesu (MPI)
- PID + IP (UNIX + sieć)
- możliwość definiowania grup procesów

Nazwy procesów

- `fork`, `join`, `quit`

  `fork` *P*() `alias a;`

- `cobegin`

```
cobegin {
  as nazwa procesu {
    instrukcja 1.1;
    ...
    instrukcja 1.N1;
  };
  ...
  as nazwa procesu {
    instrukcja K.1;
    ...
    instrukcja K.NK;
  };
}
```

```
parfor zmienna sterująca = wp to wk do{
  as nazwa procesu<zmienna sterująca> {
    instrukcja 1;
    instrukcja 2;
    ...
    instrukcja N;
  };
};
```


Producenci i Konsumenci

```
main (){ //1->1

  cobegin {
    as Producent {  //Producent
      int lx;
      while (1){
        lx = Produkuj();
        send lx to Konsument;
      }
    }//Producent

    as Konsument { //Konsument
      int lx;
      while (1){
        receive lx from Producent;
        Konsumuj (lx);
      }
    }//Konsument
  } // cobegin
}
```

```
main (){ //N->1

  cobegin {
    parfor i = 1 to N do {//Producenci
      int lx;
      while (1){
        lx = Produkuj();
        send lx to Konsument;
      }
    } //parfor

    as Konsument { //Konsument
      int lx;
      while (1){
        receive lx;
        Konsumuj (lx);
      }
    }//Konsument
  } // cobegin
}
```

Obliczanie minimum

Zadanie: W komputerze z pamięcią rozproszoną wykonuje się $N+1$ procesów współbieżnych $P_0, P_1, \ldots, P_N$. Każdy z tych procesów wykonuje następujący kod:

```
int x;
x = generuj();
x = znajdzMin (x, i);
konsumujMin (x);
```

Funkcja int *znajdzMin* (int x, int i):

- Parametry wejściowe:
  - x wartość wygenerowana w bieżącym procesie,
  - i numer bieżącego procesu.
- Zwraca najmniejszą wartość spośród wszystkich wartości wygenerowanych w procesach P_0, P_1, ..., P_N.

Podać kod funkcji *znajdzMin*.

Obliczanie minimum

```
int znajdzMin (int x, int i){
  if (i == 0) {
    int j, xTmp;
    for (j = 1; j <= N; j ++){
      receive xTmp;
      if (xTmp < x)
        x = xTmp;
    }//for
    for (j = 1; j <= N; j ++)
      send x to P<j>;
  } else {
    send x to P<0>;
    receive x from P<0>;
  } // if (i == 0)
  return x;
}
```

```
main(){
  parfor i = 0 to N do {
    as P<i> {
      int x;
      x = generuj ();
      x = znajdzMin (x, i);
      konsumujMin (x);
    } //P<i>
  } //parfor
} //main
```


Obliczanie minimum

Wprowadzenie dodatkowego nazwanego procesu.

```
int znajdzMin (int x){
  send x to Nowy;
  receive x from Nowy;
  return x;
}
main(){
  cobegin {
    parfor i = 0 to N do {
      as P<i> {
        int x;
        x = generuj ();
        x = znajdzMin (x, i);
        konsumujMin (x);
      } //P<i>
    } //parfor
```

```
    as Nowy {
      int j, x, xTmp;
      for (j = 0; j <= N; j ++){
        receive xTmp;
        if (xTmp < x)
          x = xTmp;
      }//for
      for (j = 0; j <= N; j ++)
        send x to P<j>;
    }//Nowy
  }//cobegin
} //main
```

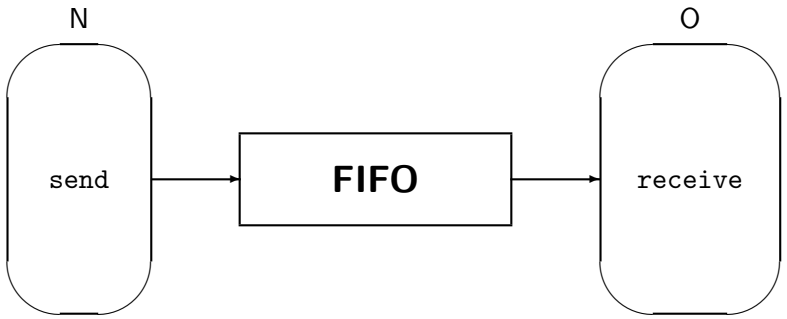

Obliczanie minimum

Bez wprowadzania dodatkowego nazwanego procesu.

```
//pamięć wspólna
int znajdzMin (int x){
  int xp[N+1];
  cobegin {
    parfor i = 0 to N do {
      send x to P<i>;
    }
    parfor j = 0 to N do {
      receive xp[j] from P<j>;
    }
  }//cobegin
  int j;
  x = xp[0];
  for (j = 1; j <= N; j++)
    if (xp[j] < x)
      x = xp[j];
  return x;
}//znajdzMin
```

```
main(){
  parfor i = 0 to N do {
    as P<i> {
      int x;
      x = generuj ();
      x = znajdzMin (x, i);
      konsumujMin (x);
    } //P<i>
  } //parfor
} //main
```

Synchronizacja nadawcy i odbiorcy



n	Rodzaj komunikacji	send	receive
∞	Asynchroniczna	Nieblokująca	Blokująca
0	Synchroniczna	Blokująca	Blokująca
$0 < n < \infty$	Buforowana	Blokująca	Blokująca

Komunikacja selektywna

```
while(1){
  receive lista_1 from nadawca_1;
  receive lista_2 from nadawca_2;
    ⋮
  receive lista_N from nadawca_N;
};
```

```
while(1){
  if (receive od nadawca_1 nie blokuje)
    receive lista_1 from nadawca_1;
  if (receive od nadawca_2 nie blokuje)
    receive lista_2 from nadawca_2;
      ⋮
  if (receive od nadawca_N nie blokuje)
    receive lista_N from nadawca_N;
};
```


Rozkazy dozorowane

dozór $\longrightarrow$ *lista rozkazów*

Dozór =

- wyrażenie logiczne
- wyrażenie logiczne oraz instrukcja przesyłu komunikatu (`send` lub `receive`).

Możliwe stany dozoru:

- dozór fałszywy,
- dozór prawdziwy,
- dozór (chwilowo) ani prawdziwy ani fałszywy.

Rozkazy dozorowane
Przykłady

```
x < 0 -> x = -x;
```

```
for (i = 0, ujemne = 0; i < N; i++)
  a[i] < 0 ->
    ujemne ++;
```

```
wolne, receive x from producent ->
  wolne = false;
```

```
true, send 1 to konsument ->
  ;
```

Instrukcja alternatywy

```
guarded if {
  dozór_1 → instrukcja_1
  dozór_2 → instrukcja_2
    ⋮
  dozór_N → instrukcja_N
}
```

```
int x;
bool wolne = true;
guarded if {
  wolne, receive x from P ->
    wolne = false;
  !wolne, send x to K ->
    wolne = true;
}
```


Instrukcja alternatywy

Wersja z replikatorem

```
guarded if zs = wart_początkowa to wart_końcowa {
  dozór → lista
}
```

```
int x;
guarded if i = 1 to 10 {
  receive x from P<i> ->
    ;
}
```


Instrukcja powtarzania

```
guarded do{
  dozór_1 → instrukcja_1
  dozór_2 → instrukcja_2
    ⋮
  dozór_N → instrukcja_N
}
```

```
int x;
bool wolne = true;
guarded do {
  wolne, receive x from P ->
    wolne = false;
  !wolne, send x to K ->
    wolne = true;
}
```


Instrukcja powtarzania

Wersja z replikatorem

```
guarded do zs = wart_początkowa to wart_końcowa {
  dozór → lista
}
```

```
bool odebrac[N];
int x, suma = 0;
for (i = 0; i < N, odebrac[i++] = true);
guarded do i = 0 to N - 1 {
  odebrac[i], receive x from P<i> -> {
    odebrac[i] = false;
    suma += x;
  }
}
```

Bufor cykliczny

```
void bufor() {
  T elem[N];
  int głowa = 0,
      ogon = 0;      // zakres 0 ..  N - 1
  int zajete = 0;    // zakres 0 ..  N
  guarded do {
    zajete < N,
      receive elem[ogon] from P → {
        zajete = zajete + 1;
        ogon = (ogon + 1) % N;
    }
    zajete > 0,
      send elem[głowa] to K → {
        zajete = zajete - 1;
        głowa = (głowa + 1) % N;
    }
  } // guarded do
}//bufor
```


Producenci i Konsumenci

```
void producent(){
  T komunikat;
  while (1) {
    generuj komunikat;
    send komunikat to B;
  } //while
} //producent

void konsument(){
  T komunikat;
  while (1) {
    receive komunikat from B;
    zużyj komunikat;
  } //while
} //konsument
```

```
void main (){
  cobegin{
    as B { bufor();}
    as P { producent();}
    as K { konsument();}
  } //cobegin
}
```


Producenci i Konsumenci

Wersja bez send w dozorze

```
void bufor() {
  T elem[N];
  int głowa = 0,
      ogon = 0;    // zakres 0..N-1
  int zajete = 0; // zakres 0..N
  guarded do {
    zajete < N,
      receive elem[ogon] from P → {
        zajete = zajete + 1;
        ogon = (ogon + 1) % N;
    }
    zajete > 0,
      receive from K → {
        send elem[głowa] to K;
        zajete = zajete - 1;
        głowa = (głowa + 1) % N;
    }
  } //guarded do
}//bufor
```

```
void konsument(){
  T komunikat;
  while (1) {
    send to B;
    receive komunikat from B;
    zużyj komunikat;
  } //while
} //konsument
```


Producenci i Konsumenci ($N \rightarrow 1$)

```
main (){

cobegin {
 //Producenci
 parfor i = 1 to N do {
  as P<i> {
   int lx;
   while (1){
    lx = Produkuj();
    send lx to Konsument;
   }
  } //P<i>
 } //parfor
```

```
  as Konsument { //Konsument
   int lx;
   while (1){
    guarded if i = 1 to N {
     true, receive lx from P<i> ->
      ; //pusta lista rozkazow
    } //guarded if
    Konsumuj (lx);
   }
  }//Konsument
 } // cobegin
}
```

Kanały

- 1 — 1
- jedno- lub dwukierunkowe
- `channel` *message structure* `end;`
- `send` *lista wyrażeń* `to` *identyfikator*;
  `receive` *lista zmiennych* `from` *identyfikator*;

Niezgodność typu danych

```
cobegin
  as nadawca
    send 7UL to odbiorca;
  as odbiorca
    var float y;
    receive y from nadawca;
coend
```

Rozwiązanie z zastosowaniem kanałów

```
var
  channel
    unsigned long;
  end c;
cobegin
  send 7UL to c;

  var float y;
  receive y from c;
coend;
```


Producenci i Konsumenci (1 → 1)

```
main (){
  channel {
     int;
  } cpm;

  cobegin {
    //Producent
    as Producent {
      int lx;
      while (1){
        lx = Produkuj();
        send lx to cpm;
      }
    }//Producent

    //Konsument
    as Konsument {
      int lx;
      while (1){
        receive lx from cpm;
        Konsumuj (lx);
      }
    }//Konsument
  } // cobegin
}
```


Producenci i Konsumenci (N → 1)

```
var
  channel int; end cpm[1 .. N];
cobegin

   parfor i := 1 to N do
     loop
       send Create() to cpm[i];
     end loop;
   end parfor;

   var int lx;
   loop
     guarded if i:= 1 to N do
       true, receive lx from cpm[i]->
         Use (lx);
     end guarded if;
   end loop;

coend
```


Część VII

Przykłady

Suma

W pewnym programie współbieżnym wykonywanym w komputerze z pamięcią wspólną stosowane są dwie zmienne wspólne o następujących deklaracjach:

```
int a[3n];
int total = 0;
```

Po zainicjalizowaniu elementów tablicy *a* rozpoczyna się współbieżne wykonanie trzech procesów:

```
{ //proces P1
  int i, s;
  for (i = 0, s = 0; i < n; i ++)
    s += a[i];
  total += s;
}
{ //proces P2
  int i, s;
  for (i = n, s = 0; i < 2 * n; i ++)
    s += a[i];
  total += s;
}
{ //proces P3
  int i, s;
  for (i = 2 * n, s = 0; i < 3 * n; i ++)
    s += a[i];
  total += s;
}
```

Po zakończeniu wykonania procesów P1, P2 i P3 ma być spełniona następująca zależność: $total = \sum_{i=0}^{3*n-1} a[i]$

Co należy zrobić, aby powyższa zależność była spełniona po zakończeniu wykonywania procesów P1, P2 i P3?

Odpowiedź uzasadnić.

fork, join, quit

Podać wartość zmiennych całkowitych x, y, a, b po zakończeniu wykonania procesów P_1 i P_2.

```
void P2(void){
  a = x + 2;
  b = a * y;
  quit;
}
void P1(void){
  x = 2;
  fork P2();
  y = 3;
  join P2;
  y = b * x;
}
```


Wyświetlacz (I)

W komputerze równoległym z pamięcią wspólną wykonywany jest program, w którego skład wchodzą m.in. trzy procesy współbieżne:

```
enum rodzaj {wysokosc, predkosc};
{//proces predkosciomerz
  int p;
  for (;;){
    p = odczytaj_predkosc();
    przekaz_do_wyswietlacza (predkosc, p);
  }
}
{//proces wysokosciomierz
  int w;
  for (;;){
    w = odczytaj_wysokosc();
    przekaz_do_wyswietlacza (wysokosc, w);
  }
}
{//proces wyswietlacz
  int w;
  rodzaj r;

  for (;;){
    odbierz_pomiar (r, w);
    switch (r){
      case wysokosc: wyswietl_wysokosc (w);
        break;
      case predkosc: wyswietl_predkosc (w);
        break;
    }
  }
}
```


Wyświetlacz (II)

Podać kod źródłowy funkcji *przekaz_do_wyswietlacza* oraz *odbierz_pomiar* jeśli wiadomo, że:

1. Funkcje te mają następujace nagłówki:

   ```
   void przekaz_do_wyswietlacza (rodzaj r, int wartosc);
   void odbierz_pomiar ( rodzaj &r, int &wartosc);
   ```

2. Wynik każdego pomiaru ma zostać wyświetlony.
3. Czas potrzebny na wykonanie funkcji *odczytaj_predkosc* oraz *odczytaj_wysokosc* jest zmienny; funkcji tych nie trzeba deklarować.
4. Czas potrzebny na wykonanie funkcji *przekaz_do_wyswietlacza* oraz *odbierz_pomiar* powinien być jak najkrótszy.

Skrzynka pocztowa

Zaproponuj strukturę danych oraz podaj kod źródłowy funkcji *inicjalizuj*, *wyślij* oraz *odbierz* dla implementacji skrzynki pocztowej w komputerze równoległym z pamięcią wspólną.
Przyjmij, że funkcja *inicjalizuj* będzie wykonana jako pierwsza.

Graf

Graf wykonania jest skierowanym, acyklicznym grafem. Wierzchołki reprezentują procesy, a krawędzie wskazują kolejność wykonania tych procesów. Każdy proces może rozpocząć wykonanie wtedy i tylko wtedy, gdy wszystkie poprzedzające je procesy zostaną zakończone. Kod każdego procesu można zapisać następująco:

```
void P(void){;
  oczekuj na zakonczenie procesow poprzedzajacych;
  rob swoje;
  zawiadom procesy nastepujace, ze skonczyles;
}
```

Stosując wyłącznie synchroniczne przesyłanie komunikatów, zapewnij aby kolejność wykonania procesów $P_1, \ldots, P_4$ była dokładnie taka, jak określa to graf wykonania. Nie narzucaj żadnych dodatkowych ograniczeń na kolejność wykonania procesów, poza tymi, które określa graf wykonania. Procesy $P_1, \ldots, P_4$ rozpoczynają swoje wykonanie w tym samym czasie:

```
cobegin { P1;  P2;  P3;  P4; }
```


Przyjęcie

Program równoległy zawiera instrukcję **cobegin** podaną niżej.

```
cobegin {
  parfor i = 1 to n {//grupa procesow M
    M(i)
  }
  parfor i = 1 to n { //grupa procesow K
    K(i)
  }
}
M(int i){ //kod funkcji K(int i) jest identyczny
  int j, x;
  x = wyznacz_x();
  j = szukaj_rownego(x, i);  //<<<--
  przetworz(j);
}
```

Napisać funkcję *szukaj_rownego*, która zwraca indeks procesu, należącego do przeciwnej grupy, który zgłosił taką samą wartość, jak bieżący proces lub -1, jeżeli żaden z procesów grupy przeciwnej nie zgłosił takiej wartości. Przyjąć nastepujące założenia:

1. Wszystkie wartości zgłaszane przez procesy są różne,
2. Dostępna jest funkcja *jestM()*, która zwraca 1, jeżeli bieżący proces należy do grupy M oraz 0 w przeciwnym razie.

Zasoby (I)

W komputerze równoległym z pamięcią wspólną wykonywany jest program, którego procesy korzystają z zasobów przechowywanych w globalnej tablicy o następującej deklaracji:

```
zasob a[n];
```

gdzie n jest pewną stała. Na początku wykonania programu wszystkie zasoby są wolne. Aby proces mógł skorzystać z zasobu musi go najpierw „zająć", a po użyciu „zwrócić". W danej chwili z zasobu może korzystać tylko jeden proces (lub żaden). W skład programu wchodzą procesy współbieżne dwóch rodzajów:

```
//proces pierwszego rodzaju
{
  int p;
  int z;
  for (;;){
    p = probuj_pobrac_zasob(&z);
    if (p) {
      korzystaj_z_zasobu(z);
      zwroc_zasob(z);
    } else
      rob_cos();
  }
}
```

```
//proces drugiego rodzaju
{
  int z;
  for (;;){
    pobierz_zasob(&z);
    korzystaj_z_zasobu(z);
    zwroc_zasob(z);
  }
}
```

Zasoby (II)

Podaj kod źródłowy funkcji *probuj_pobrac_zasob*, *pobierz_zasob* oraz *zwroc_zasob* jeśli wiadomo, że:

1. W programie wykonywane jest $\alpha \geq 0$ procesów pierwszego rodzaju i $\beta \geq 0$ procesów drugiego rodzaju.
2. Funkcje te mają następujace nagłówki:

```
int probuj_pobrac_zasob (int *nrzasobu);
void pobierz_zasob (int *nrzasobu);
void zwroc_zasob (int *nrzasobu);
```

3. Funkcja *probuj_pobrac_zasob* zwraca 1, jeżeli udało się pobrać zasób, a 0 w przeciwnym wypadku.
4. Funkcja *pobierz_zasob* zawsze kończy się powodzeniem, tj. zajęciem jakiegoś zasobu na potrzeby procesu, który ją wywołał.
5. Proces wykonujący funkcję *probuj_pobrac_zasob* nie może czekać, aż jakiś zasób będzie wolny.
6. Zasób nie „wie" czy jest wolny czy zajęty.
7. Funkcje *korzystaj_z_zasobu* oraz *rob_cos* są zadeklarowane.
8. Czas potrzebny na wykonanie każdej funkcji powinien być jak najkrótszy.

Do synchronizacji procesów użyj semaforów. Podaj deklaracje wszystkich niezbędnych zmiennych i stałych globalnych.

Studenci

W programie współbieżnym wykonywane jest m.in. *n* procesów *Student*:

```
void Student(){
  bool  zdane;

  ... //to, co studenci zwykle robia w trakcie semestru
  zdane = false
  do {
    przegladaj_notatki();
    zdawaj_egzamin();
    zdane = sprawdz_wyniki();
  } while (! zdane);
  ... //to, co studenci zwykle robia po zdaniu egzaminu
}
```

Uzupełnij podany wyżej kod, tak aby spełnione były następujące warunki:

1. żaden *Student* nie rozpocznie wykonywania procedury *zdawaj_egzamin*, jeżeli choć jeden proces *Student* wykonuje procedurę *przegladaj_notatki*;
2. żaden *Student* nie rozpocznie wykonywania funkcji *sprawdz_wyniki*, jeżeli choć jeden proces *Student* wykonuje procedurę *zdawaj_egzamin*.

Funkcja *sprawdz_wyniki* zwraca wartość **true** z pewnym prawdopodobieństwem $P > 0$.

Przyjmij model z pamięcią wspólną; do synchronizacji procesów użyj wyłącznie semaforów.

Interferencja

Dany jest fragment kodu programu równoległego:

```
... {obliczenia}
parfor i = 1 to n { //procesy P1 .. Pn
  ... //obliczenia
  x := x + ai;
  ... //obliczenia
}
... {obliczenia}
```

Zapewnij, że w trakcie wykonania tego programu nie wystąpi interferencja procesów $P_1, \ldots, P_n$. Przyjmij, że operacja

$$x = x + ai$$

jest jedyną operacją w instrukcji parfor, która odwołuje się do zmiennej x oraz że zmienna x jest jedyną zmienną wspólną w programie. Do realizacji zadania użyj monitora.
Uwaga: Zmienna x nie może być zmienną własną monitora.

Rozesłanie

Dany jest następujący program współbieżny:

```
const int N = ...;
int i;

parfor i = 1 to N {
    P(i);
}
```

Kod źródłowy funkcji *P* jest następujący:

```
void P(int k){
  int x;
  ...
  x = wykonaj (k, x);
  ...
}
```

Funkcja *wykonaj* ma prototyp:

```
int wykonaj(int i, int x);
```

Funkcja ta zwraca wartość zmiennej lokalnej x funkcji *P* wywołanej dla $i = 1$. Napisać funkcję *wykonaj*. Przyjąć, że funkcja *wykonaj* będzie wykonana dokładnie jeden raz w trakcie jednego wykonania funkcji *P*.

Sumowanie

n równouprawnionych procesów $A_i, i = 1, 2, \ldots, n$ generuje ciągi 200 liczb całkowitych i przekazuje je procesowi B, który dokonuje sumowania, a wyliczone wartości zwraca procesom A_i (każdemu procesowi A_i zwracana jest suma wygenerowanych przez ten proces liczb). Liczby przekazywane są porcjami po 10, a proces B odsyła sumę

1. *wszystkich* porcji,
2. *każdej* porcji.

Poczmistrz

Procesy $P_i, i = 1, 2, \ldots, n$ współpracują ze sobą przesyłając do siebie komunikaty za pomocą procesu *Pośrednik*. U *Pośrednika* znajduje się początkowo m komunikatów. Proces P_i pobiera komunikat k od *Pośrednika*, przetwarza go, wyznacza odbiorcę wyniku, a następnie wysyła komunikat z wynikiem i numerem odbiorcy do *Pośrednika*. Potem wykonuje pewne obliczenia lokalne i znów czeka na odebranie komunikatu od *Pośrednika*. Komunikaty nie muszą być odbierane od *Pośrednika* w takiej kolejności, w jakiej zostały wysłane. Zaimplementować procesy P oraz *Pośrednik*. Napisać program współbieżny złożony z procesów P oraz *Pośrednik*. Wartości m i n są parametrami programu.

Zasoby raz jeszcze

W systemie znajdują się dwa typy zasobów: A i B, które są wymienne, ale pierwszy z nich jest wygodniejszy niż drugi. Jest m zasobów typu A i n ($n > m$) zasobów typu B. W systemie działają trzy grupy procesów, które różnią się sposobem zgłaszania zapotrzebowania na zasób:

- procesy pierwszej grupy żądają wyłącznie wygodnego zasobu (typu A) i czekają, aż będzie dostępny,
- procesy drugiej grupy żądają wygodnego zasobu (typu A), lecz jeśli jest niedostępny, to czekają na zasób dowolnego typu,
- procesy trzeciej grupy żądają dowolnego zasobu, ale jeśli żaden nie jest dostępny, to nie czekają.

Zaimplementować opisany system. Liczba procesów każdej grupy oraz wartości m i n są parametrami programu.

Synchronizacja zegarów

Współbieżne procesy $P_i, i = 1, 2, \ldots, n$ wykonują w nieskończonej pętli własne operacje i dodatkowo co m-ty obrót pętli (począwszy od i-tego) synchronizują z innymi procesami swój logiczny lokalny zegar. Logiczny czas każdego procesu jest mierzony liczbą wykonanych obrotów pętli. Synchronizacja jest konieczna, gdyż pętle różnych procesów wykonują się z różną szybkością. Synchronizacja polega na korygowaniu lokalnego zegara na podstawie sygnowanych czasem komunikatów docierających od innych procesów. Odbieraniem komunikatów w imieniu procesu P_i zajmuje się proces-kontroler $K_i, i = 1, 2, \ldots, n$. W celu skorygowania lokalnego czasu proces P_i komunikuje się z procesem K_i i otrzymuje od niego wartość największego czasu t, jaki sygnował dotychczas otrzymane komunikaty. Jeśli t jest większe od lokalnego czasu procesu, to czas ten jest ustawiany na $t + 1$ (skoro odebrano komunikat nadany w chwili t, to lokalny czas odbiorcy musi być późniejszy). Po skorygowaniu swojego czasu proces P_i wysyła do jednego losowo wybranego procesu $K_j, j = 1, \ldots, n$ komunikat sygnowany własnym lokalnym czasem. Napisać program współbieżny złożony z procesów P i K. Wartości m i n są parametrami programu.

Dystrybucja wiadomości

W systemie znajduje się jeden proces producent P i n procesów — konsumentów $K_i, i = 1, 2, \ldots, n$. Producent produkuje wiadomości i wstawia je do bufora o rozmiarze b, z którego są one pobierane przez konsumentów. Każda wiadomość wysłana przez producenta musi zostać odczytana przez wszystkich konsumentów. Każdy konsument musi odebrać wiadomości w takiej kolejności, w jakiej zostały wysłane przez producenta. Procesy K_i odczytują wiadomości z różną częstotliwością. Napisać program współbieżny złożony z procesów K i procesu P. Wartości b i n są parametrami programu.

Farming

W systemie są dwa rodzaje procesów: zarządca Z oraz wykonawcy $W_i, i = 1, 2, \ldots, n$. Zarządca wczytuje zbiór liczb, powoduje posortowanie elementów tego zbioru przy pomocy wykonawców (elementy sortowane są rosnąco), a następnie wypisuje elementy zbioru. Proces wykonawca potrafi wykonać trzy operacje:

- posortować przekazany mu zbiór liczb,
- scalić dwa posortowane zbiory liczb w jeden, także posortowany,
- zakończyć się.

Napisać program współbieżny złożony z procesów W i procesu Z. Wartość n jest parametrem programu.

Akademik

W pewnym akademiku jest jedna łazienka, z której mogą korzystać zarówno kobiety jak i mężczyźni, ale nie jednocześnie. W danej chwili w łazience może znajdować się dowolnie wielu mężczyzn lub dowolnie wiele kobiet. Napisać program współbieżny symulujący korzystanie z takiej łazienki. Każdy z procesów wykonuje cztery czynności:

```
loop
  1. rob cos
  2. wejdz do lazienki
  3. korzystaj z lazienki
  4. wyjdz z lazienki
end loop
```

Czas trwania czynności 1 i 3 jest losowy, być może inny za każdym razem. Liczba mężczyzn n_m i liczba kobiet n_k w akademiku są parametrami programu.

Wspólne konto

Konto w banku jest wspólną własnością grupy n procesów. Każdy proces może wpłacić lub wypłacić pieniądze z konta. Bieżący stan konta jest sumą wszystkich dotychczasowych wpłat minus suma wszystkich dotychczasowych wypłat. Na koncie nigdy nie może powstać debet. Jeżeli kwota, którą proces próbuje wypłacić przewyższa stan konta, jest on wstrzymywany do czasu, gdy wykonanie tej operacji będzie możliwe. Napisać program współbieżny, który realizuje opisane zachowanie.