Wykład 5

Synchronizacja (część II)

Wady semaforów

- Jeden z pierwszych mechanizmów synchronizacji
- Generalnie jest to mechanizm bardzo niskiego poziomu - trochę odpowiadający programowaniu w assemblerze.
- Duża podatność na błędy, trudno wykazać poprawność programu
- Przykład: Jeżeli zapomnimy o operacji signal, nastąpi blokada
- Bardziej strukturalne mechanizmy synchronizacji
  - Regiony krytyczne
  - Monitory

Regiony krytyczne

- Współdzielona zmienna v typu T jest deklarowana jako:

  var *v*: **shared** *T*

- Dostęp do zmiennej v wykonywany przy pomocy operacji

  region *v* **when** *B* **do** *S*

- B jest wyrażeniem logicznym
- Tak długo, jak instrukcja *S* się wykonuje, żaden inny proces nie może się odwołać do zmiennej *v*.
- Jeżeli wyrażenie B nie jest spełnione, to proces jest wstrzymywany do momentu jego spełnienia.

Przykład: producent-konsument z ograniczonym buforem

```
var buffer: shared record
                pool: array [0..n–1] of item;
                                count,in,out: integer
            end;

region buffer when count < n
                do begin
                                pool[in] := nextp;
                                in:= in+1 mod n;
                                count := count + 1;
                        end;

region buffer when count > 0
                do begin
                                nextc := pool[out];
                                out := out+1 mod n;
                                count := count – 1;
                        end;
```

- Deklaracja zmiennej współdzielonej.
- Wstawienie elementu nextp do bufora. (Producent).
- Usunięcie elementu nextc z bufora (Konsument).

Idea monitora (a właściwie zmiennej warunkowej)

- Udostępnienie procesom operacji pozwalającej procesowi wejść w stan uśpienia (zablokowania) – *wait* oraz operacji *signal* pozwalającej na uśpienie obudzonego procesu.
- Ale tu natrafiamy na (stary) problem wyścigów, który ilustruje poniższy przykład:

```
if (Jeszcze_nie_bylo_zdarzenia_muszę_wykonać_wait)

    wait() // to zaczekam
```

- Co się stanie, jeżeli zdarzenie na które czeka proces zajdzie po instrukcji if, ale przed uśpieniem procesu ? Proces zgubi zdarzenie (i być może nigdy się nie obudzi)
- W takim razie wykonajmy cały ten kod wewnątrz sekcji krytycznej ?
  - Ale gdy proces wykona wait() - to przejdzie w stan uśpienia nie zwalniając sekcji krytycznej. Przy próbie wejścia do sekcji przez inny proces na pewno dojdzie do blokady.
- Rozwiązanie: Atomowa operacja wait powodująca jednoczesne uśpienie procesu i wyjście z sekcji krytycznej

Monitory

```
monitor mon {
  int foo;
  int bar;
  public void proc1(...) {
  }
  public void proc2(...) {
  }
};
```

- Pseudokod przypominający definicję klasy C++.
- Współdzielone zmienne *foo* oraz *bar* są dostępne wyłącznie z procedur monitora.
- Procesy synchronizują się wywołując procedury monitora (np. proc1 i proc2)
  - Tylko jeden proces (wątek) może w danej chwili przebywać w procedurze monitora. Gwarantuje to automatycznie wzajemne wykluczanie.
- Mówimy że proces "*przebywa wewnątrz monitora*".

Zmienne warunkowe (ang. condition)

- Problem: proces postanawia zaczekać wewnątrz monitora aż zajdzie zdarzenie sygnalizowane przez inny proces.
  - Jeżeli proces po prostu zacznie czekać, nastąpi blokada, bo żaden inny proces nie będzie mógł wejść do monitora i zasygnalizować zdarzenia.
- Zmienne warunkowa (typu condition). Proces, ***będący wewnątrz monitora***, może wykonać na niej dwie operacje.
  - Niech deklaracja ma postać: *Condition* C*;*
  - C.*wait()* Zawiesza wykonanie procesu, i jednocześnie zwalnia monitor pozwalając innym procesom wejść do monitora.
  - C.*signal()* Jeżeli nie ma procesów zawieszonych przez operację *wait* nic się nie dzieje. W przeciwnym wypadku dokładnie jeden proces zawieszony przez operacje *wait* zostanie wznowiony. (od następnej instrukcji po *wait*).
  - Możliwa jest trzecia operacja C.signallAll() wznawiająca wszystkie zawieszone procesy.

Przykład 1: Implementacja semafora zliczającego przy pomocy monitora

```
monitor Semafor {
  int Licznik;
  Condition NieZero;

  // coś na kształt konstruktora
  Semafor(int i) {
      Licznik=i;
  }

  void wait() {
      if (Licznik==0)
            NieZero.wait();
      Licznik=Licznik-1;
  }

  void signal() {
      Licznik=Licznik+1;
      NieZero.signal();
  }
};
```

- Deklaracja:
  - Semafor S(1);
- Proces potrzebujący wzajemnego wykluczania.

S.wait()

// sekcja krytyczna

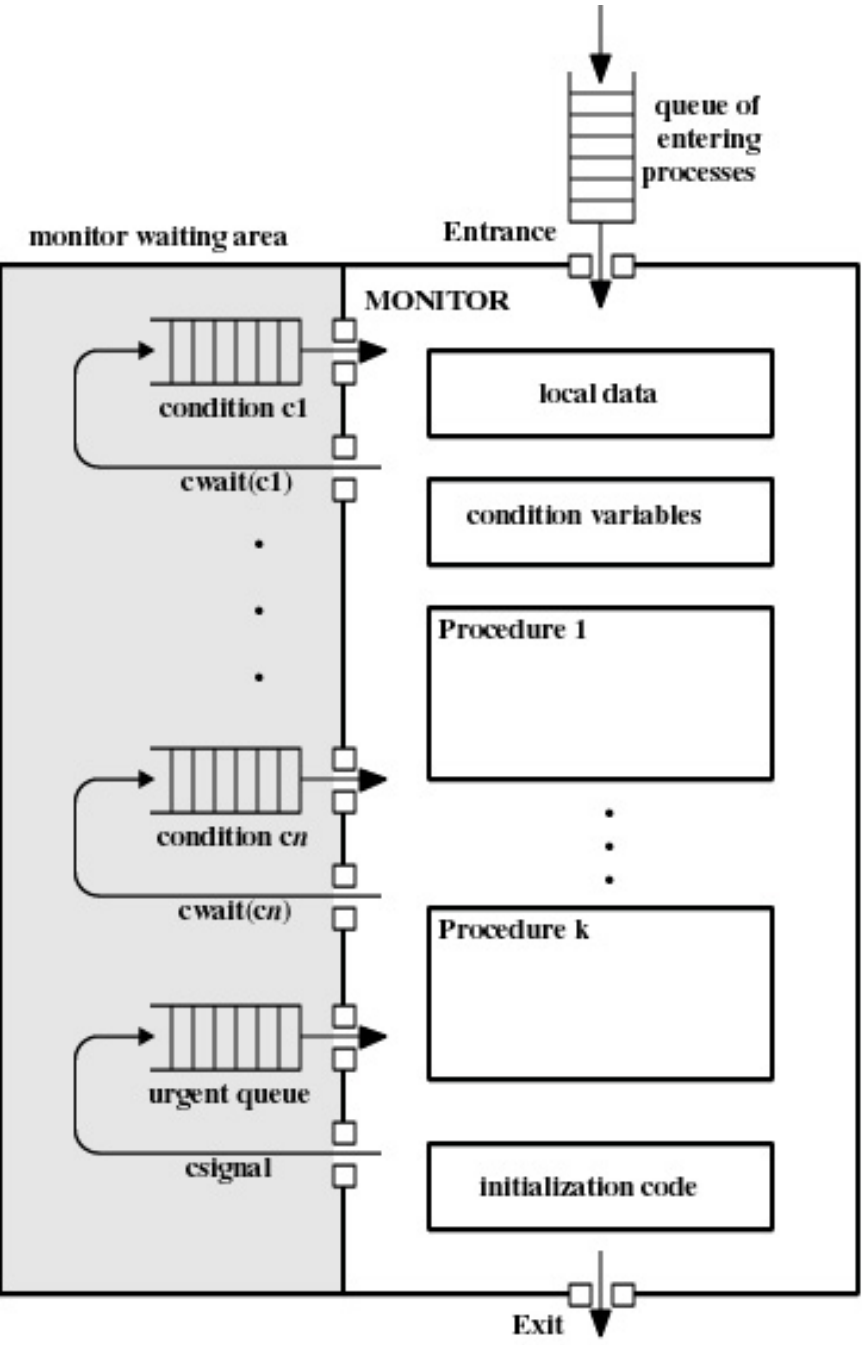
S.signal()

// pozostałe czynności

Semantyka Hoare'a i semantyka Mesa

- Przypuśćmy że proces P wykonał operację *wait* i został zawieszony. Po jakimś czasie proces Q wykonuje operacje *signal* odblokowując P.
- Problem: który proces dalej kontynuuje pracę: P czy Q. Zgodnie z zasadą działania monitora tylko jeden proces może kontynuować pracę.
- Semantyka Mesa
  - Proces który wywołał operacje *signal* (Q) kontynuuje pierwszy.
  - P może wznowić działanie, gdy Q opuści monitor.
  - Wydaje się być zgodna z logiką, po co wstrzymywać proces który zgłosił zdarzenie.
- Semantyka Hoare'a
  - Proces odblokowany (P) kontynuuje jako pierwszy.
  - Może ułatwiać pisanie poprawnych programów. W przypadku semantyki Mesa nie mamy gwarancji, że warunek, na jaki czekał P jest nadal spełniony (P powinien raz jeszcze sprawdzić warunek).
- Aby uniknąć problemów z semantyką najlepiej przyjąć że operacja *signal* jest zawsze ostatnią operacją procedury monitora.

Struktura monitora



- Zasada działania: W danej chwili w procedurze monitora może przebywać jeden proces.
- Z każdą zmienną warunkową związana jest kolejka procesów, które wywołały *wait* i oczekują na zasygnalizowanie operacji.
- Po zasygnalizowaniu warunku proces (który wykonał operację signal) przechodzi do kolejki *urgent queue*.
  - Zatem implementacja realizuje semantykę Hoare'a

Przykład 2: Problem producent-konsument z buforem cyklicznym

```
monitor ProducentKonsument {
  int Licznik=0,in=0,out=0;
  Condition Pelny;
  Condition Pusty;
  int Bufor[N];

  void Wstaw(int x) {
      if (Licznik==n)
            Pełny.wait();
      Bufor[in]=x;
      in=(in+1)%N;
      Licznik++;
      Pusty.signal();
  }

  int Pobierz() {
      if (Licznik==0)
            Pusty.wait();
      int x=Bufor[out];
      out=(out+1)%N;
      Licznik=Licznik-1;
      Pelny.signal();
      return x;
  }
};
```

- Rozwiązanie dla wielu konsumentów i wielu producentów.
- Przyjmujemy, że w buforze są przechowywane liczby całkowite (int).
- Producent chcąc wstawić element do bufora wywołuje procedurę monitora *Wstaw*.
- Konsument chcąc pobrać element z bufora wywołuje *Pobierz*.
- Gdy bufor jest pusty, to konsumenci są wstrzymywani na zmiennej warunkowej *Pusty*.
- Gdy bufor jest pełny to producenci są wstrzymywani na zmiennej warunkowej *Pełny*.

Tworzenie wątku w Javie

```
class Worker extends Thread
{
  public void run() {
    System.out.println("Wątek roboczy");
  }
}

public class First
{
  public static void main(String args[]) {
    Worker runner = new Worker();
    runner.start();
    System.out.println("Wątek główny");
        runner.join();
  }
}
```

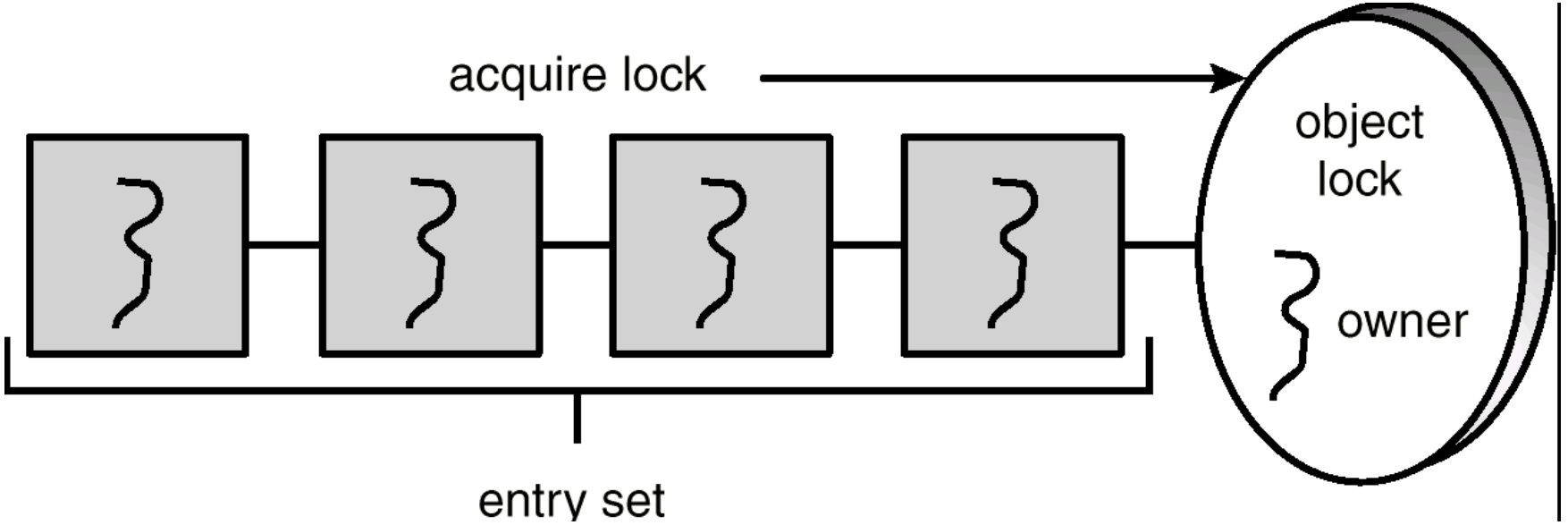
- Dwie metody:
  - Rozszerzenie klasy Thread
  - Implementacja interfejsu Runnable

- Rozszerzenie klasy Thread
  - Metoda run jest wykonywana w odrębnym wątku
  - Deklarujemy obiekt klasy
  - Metoda start() uruchamia wątek.
    - Reprezentowany przez obiekt klasy Worker.
  - Metoda join() zawiesza aktualny wątek do momentu zakończenia wątku reprezentowanego przez obiekt klasy Thread.

Metody synchronizowane w Javie

- Z ***każdym*** obiektem w Javie związany jest *zamek* (ang. lock).
- Metoda jest zsynchronizowana, jeżeli przed jej deklaracją stoi słowo kluczowe *synchronized*.
- Zamek gwarantuje wzajemne wykluczanie metod synchronizowanych obiektu
  - Aby wykonać metodę synchronizowaną wątek musi wejść w posiadanie zamka.
  - Wątek kończąc metodę synchronizowaną zwalnia zamek
  - Jeżeli wątek próbuje wywołać metodę synchronizowaną, a zamek jest już posiadany przez inny wątek (wykonujący właśnie metodę synchronizowaną), to jest zostaje on zablokowany i dodany kolejki wątków oczekujących na zwolnienie zamka.

Blokada (ang. lock) obiektu w Javie



Producent-konsument w Javie z semi-aktywnym oczekiwaniem

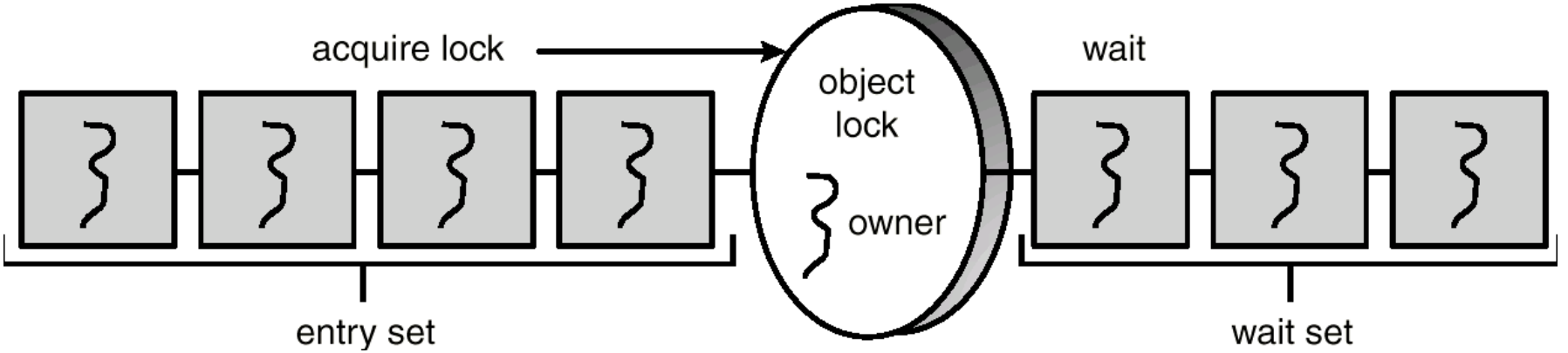
```
class ProducentKonsument {
  int Licznik=0,in=0,out=0;
  static final int N=100;
  int Bufor[N];

  public void Wstaw(int x) {
      while (Licznik==N)
            Thread.yield();
      synchronized(this) {
            Bufor[in]=x;
            in=(in+1)%N;
            Licznik++;
      }
  }

  public int Pobierz() {
      while(Licznik==N)
            Thread.yield();
      synchronized(this) {
            int x=Bufor[out];
            out=(out+1)%N;
            Licznik=Licznik-1;
      }
      return x;
  }
```

- Thread.yield pozwala na przekazanie sterowania innemu wątkowi lub procesowi.
  - ***Ciągle jest to aktywne czekanie - nie zalecane***.
- Synchronizowany blok kodu
  - Często nie ma konieczności synchronizowania całej metody
  - W danej jeden wątek może wykonywać synchronizowany blok kodu jednego obiektu.
  - Wątek ten posiada blokadę obiektu
- W książce "Applied Operating Systems Concepts" użyto synchronizowanych metod. Czy jest to poprawne ?
- Powyższe rozwiązanie jest poprawne wyłącznie dla jednego procesu konsumenta i jednego procesu producenta. (***Dlaczego ?***).

Metody wait oraz notify

- Wątek posiadający blokadę obiektu może wykonać metodę *wait*. (tego obiektu)
  - Wątek natychmiast zwalnia blokadę (inne wątki mogą wejść w posiadanie blokady)
  - Zostaje uśpiony..
  - Umieszczany jest w kolejce wątków zawieszonych (ang. wait set) obiektu.
  - Należy obsłużyć wyjątek InterruptedException.
- Wątek posiadający blokadę obiektu może wykonać metodę *notify*.
  - Metoda ta sprawia, że jeden wątek z kolejki wątków zawieszonych zostanie przesunięty do kolejki wątków oczekujących na zwolnienie blokady.
- Metoda *notifyAll* powoduje przeniesienie wszystkich wątków zawieszonych.
- W Javie istnieją również metody *suspend* i *resume*.
  - Są niebezpieczne i nie należy ich stosować !!!

Blokada obiektu w Javie (wersja ostateczna)



- Jeden wątek jest właścicielem – wykonuje kod synchronizowany
- Entry Set - wątki oczekujące na wejście w posiadanie zamka.
- Wait Set – wątki zawieszone poprzez metodę *wait*
- Wywołanie metody *notify* przenosi wątek z wait set do entry set
- Obiekt w Javie odpowiada monitorowi z ***maksymalnie jedną zmienną*** warunku. To rozwiązanie obniża wydajność synchronizacji - każdy wątek po obudzeniu, musi sprawdzić czy obudziło go zdarzenie na które czekał.
  - W przypadku producenta - konsumenta z ograniczonym buforem mamy dwa typy zdarzeń (bufor niepusty oraz bufor niepełny)

Producent konsument z wykorzystaniem metod wait/notify

```
class ProducentKonsument {
  int Licznik=0,in=0,out=0;
  int Bufor[N];

  public synchronized void Wstaw(int x) {
    while (Licznik==N)
        try { wait();}
            catch(InterruptedException e) {;}
    Bufor[in]=x;
    in=(in+1)%N;
    Licznik++;
    notifyAll();
  }

  public synchronized int Pobierz() {
    while (Licznik==0)
        try { wait();}
            catch(InterruptedException e) {;}
    int x=Bufor[out];
    out=(out+1)%N;
    Licznik=Licznik-1;
    notifyAll();
    return x;
  }
};
```

- Słabość metod wait/notify - trzeba „do skutku” sprawdzać warunek.

Synchronizacja w Javie, a monitory.

- W monitorze możemy zadeklarować wiele zmiennych warunkowych.
- Klasa w Javie w przybliżeniu odpowiada monitorowi z jedną zmienną warunkową.
- Różnice są widoczne w przypadku rozwiązania problemu producent-konsument.
  - Wersja z monitorami wykorzystuje dwie zmienne warunkowe
  - W wersji w Javie konsument oczekujący na pojawienie się elementu w buforze może zostać powiadomiony przez innego konsumenta.
  - Z tego powodu po obudzeniu należy raz jeszcze sprawdzić warunek (pętla while).
- Brak zmiennych warunkowych prowadzi do niskiej wydajności: np. budzeni są wszyscy czekający konsumenci ale tylko jeden z nich może kontynuować.
- Specyfikacja Javy mówi, że wątek wywołujący metodę notify kontynuuje pierwszy.
  - Odpowiada to semantyce Mesa.

Biblioteka POSIX Threads - implementacja monitora

- Dostarcza typy i operacje dla semaforów (sem_t) oraz mutexów (pthread_mutex_t) realizujących wzajemne wykluczanie.
- Dostarcza typ (pthread_cond_t) dla zmiennych warunkowych i niepodzielną operację pthread_cond_wait(condition,mutex) usypiającą wątek na zmiennej warunkowej i jednocześnie zwalniającą blokadę mutex. Po obudzeniu wątku oczekującego na zmiennej warunku (przez pthread_cond_signal albo pthread_cond_broadcast) nastąpi ***ponowna automatyczna re-akwizycja muteksa, przed powrotem z funkcji pthread_cond_wait.***
- Ponadto mamy operację na zmiennych warunkowych pthread_cond_signal (obudza jeden zawieszony wątek) i pthread_cond_broadcast (obudza wszystkie zawieszone wątki).

```
pthread_mutex_t mutex; // Realizuje wzajemne wykluczanie wew. monitora

void Funkcja_Monitora() {
     pthread_mutex_lock(&mutex); // Wejście do monitora
     ....................
    if (            ) {
       pthread_mutex_unlock(&mutex);
       return; // Teraz też opuszczamy monitor - pamiętać o return !!!
    }
    ....................
   pthread_mutex_unlock(&mutex); // Opuszczenie monitora
}
```

POSIX Threads - zmienne warunkowe monitora

- Każda zmienna warunkowa deklarowana jest jako zmienna typu pthread_condition_t i inicjowana przy pomocy pthread_cond_init:

```
pthread_cond_t condition;
pthread_cond_init(&condition,NULL);
```

- Jeżeli wątek przebywający wewnątrz funkcji monitora (a zatem posiadający mutex), zechce wykonać operację wait na zmiennej warunku, może wykonać następujący kod:

```
// Zwolnienie muteksa i oczekiwanie na zmiennej warunku.
pthread_cond_wait(&condition,&mutex);
// Obudzenie nastąpi po wykonaniu operacji signal lub broadcast
// i re-akwizycja muteksa.
```

- Wątek chcący wykonać operację signal monitora (i przebywający wewnątrz monitora tzn. posiadający muteks) wykonuje następujący kod:

```
// Czy to jest semantyka Hoare'a czy też Mesa ?
pthread_cond_signal(&condition);
```

Przekazywanie komunikatów (ang. message pasing)

- Dostarcza dwie operacje.
  - *send*(odbiorca,dane)
  - *receive*(nadawca,dane)
- Send i receive mogą wymagać podania kanału.
- Idealne dla problemu producent konsument.
- Na ogół wymaga kopiowania danych => możliwy spadek wydajności.
- Dobra metoda synchronizacji dla systemów rozproszonych.
  - Wydajna realizacja pamięci współdzielonej w systemie rozproszonym jest bardzo trudna.

Typy operacji send oraz receive

- Blokujące send i blokujące receive.
  - Obydwa procesy są zablokowane do momentu przekazania komunikatu.
  - Nazywane ***spotkaniem*** (Ada, CSP, Occam, Parallel C)
- Nieblokujące send i blokujące receive.
  - Proces wywołujący send nie musi czekać na przekazanie komunikatu.
  - Komunikat umieszczany jest w buforze
- Nieblokujące send i nieblokujące receive.
  - Żaden z pary procesów nie musi czekać na przekazanie komunikatu.
  - Operacja receive sygnalizuje brak komunikatu.
  - Dodatkowa operacje test_completion i wait_completion.

Implementacje przekazywania komunikatów

- Spotkania w Adzie
- Gniazda
  - Wykorzystujące protokół TCP/IP
  - Gniazda domeny Uniksa.
- Biblioteki PVM oraz MPI.
  - Zaprojektowane z myślą o obliczeniach równoległych.
- Zdalne wywołanie procedury (ang. remote procedure call, RPC)
- Zdalne wywołanie metody (ang. remote method invocation, RMI)
- Kolejki komunikatów w Uniksie
- Nazwane (i nienazwane) potoki w Uniksie