Programowanie współbieżne w C++ — Wątki

(wersja wstępna)

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Wydziału Elektroniki
Politechnika Wrocławska

Kurs: Zaawansowane metody programowania

Niniejsza prezentacja została wykonana przy użyciu systemu składu PDFLaTeX *oraz stylu* `beamer`, *którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:

`http://latex-beamer.sourceforge.net`

1. Współbieżność
   - Rodzaje współbieżności

Rodzaje współbieżności

- Programowa współbieżność – (*software concurrency*) określana jest również jako współbieżność wirtualna (*virtual concurrency*)

  Może być realizowany, gdy komputer posiada jeden procesor z pojedynczym rdzeniem.

- Sprzętowa współbieżność – (*hardware concurrency*) określana jest również jako współbieżność prawdziwa (*true concurrency*)

  Możliwa jest tylko w przypadku komputerów wieloprocesorowych lub z pojedynczym procesorem wielordzeniowym. O ilości współbieżnych procesów decyduje liczba niezależnych jednostek obliczeniowych (łączna liczba rdzeni we wszystkich procesorach).

Rodzaje współbieżności

System wieloprocesorowy/rdzeniowy nie wyklucza programowej realizacji współbieżności. Zazwyczaj procesów, które muszą *jednocześnie* działać, jest znacznie więcej niż dostępnych rdzeni.

Współbieżne procesy vs współbieżne wątki

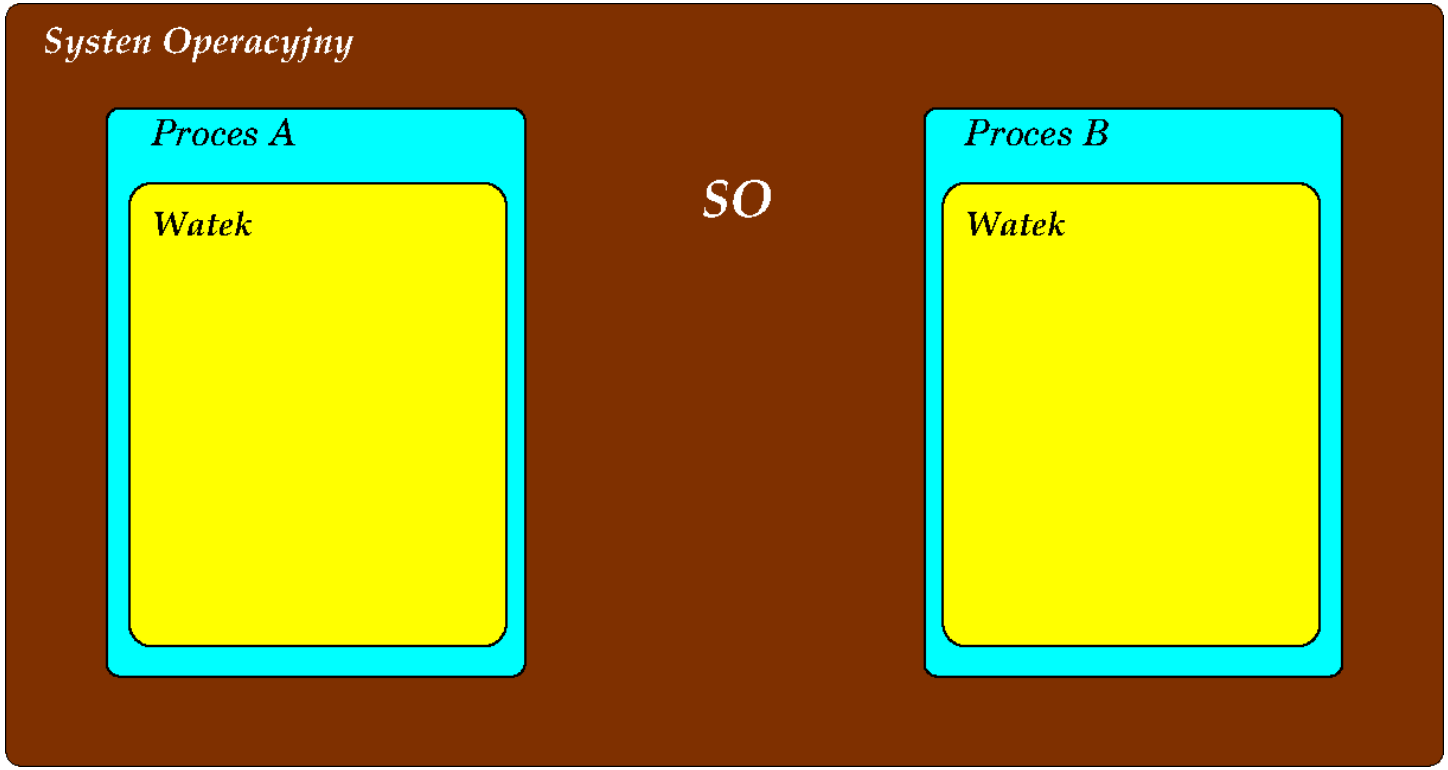
- Przełączanie się między zadaniami, rozumianymi jako osobne procesy, wymaga zmiany całego kontekstu zadania. Dotyczy on zarówno stanu wykonanywania programu jak też danych.
- Przełączanie się między wątkami wymaga zmiany jedynie kontekstu związanego z realizacją programu.

Należy pamiętać

- Przełączanie między wątkami, aczkolwiek jest szybsze niż przełączanie między osobnymi procesami, wciąż jednak generuje dodatkowy narzut.

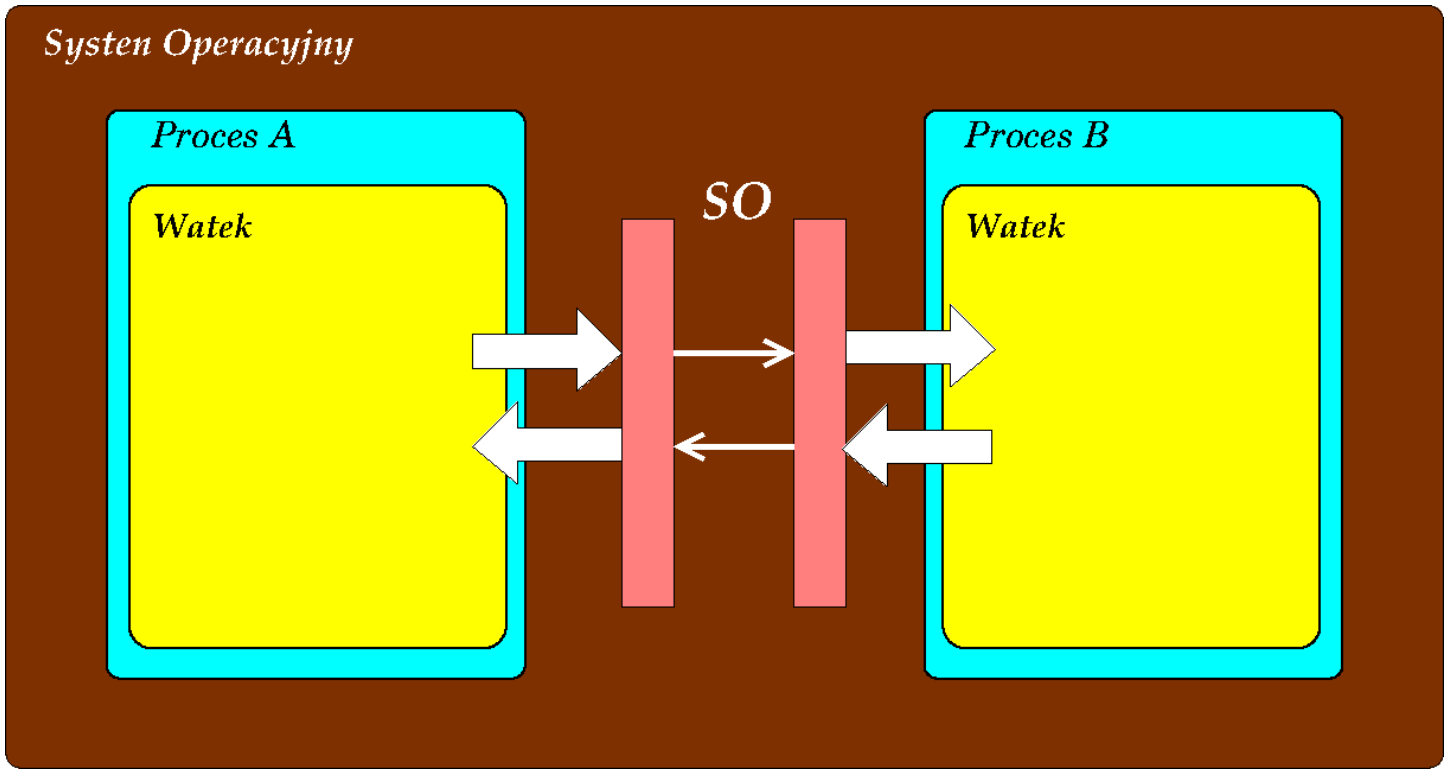
  Należy więc ograniczyć liczbę wątków do niezbędnej ilości.

Komunikacja między procesami



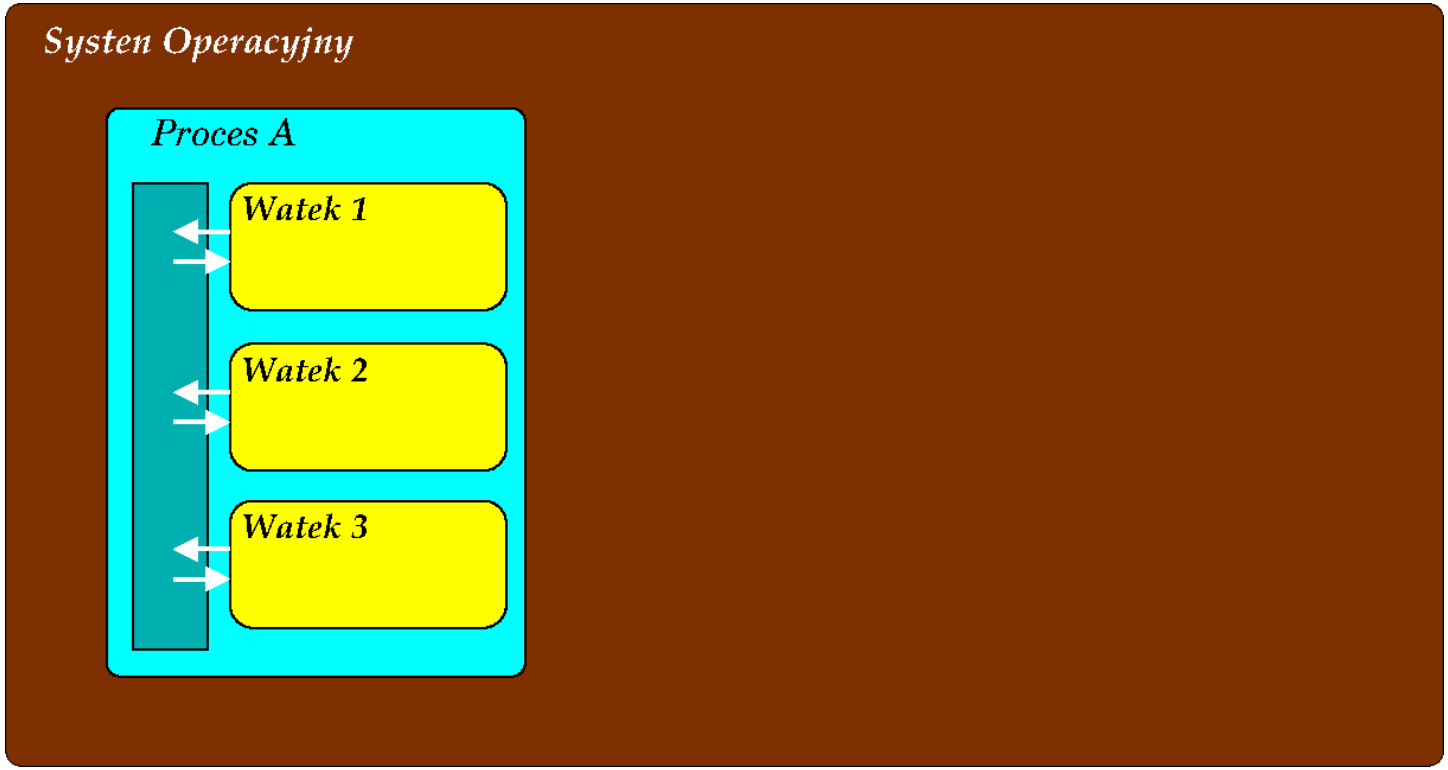
Procesy między sobą mogą komunikować się wykorzystując mechanizmy, które dostarcza system operacyjny.

Komunikacja między procesami



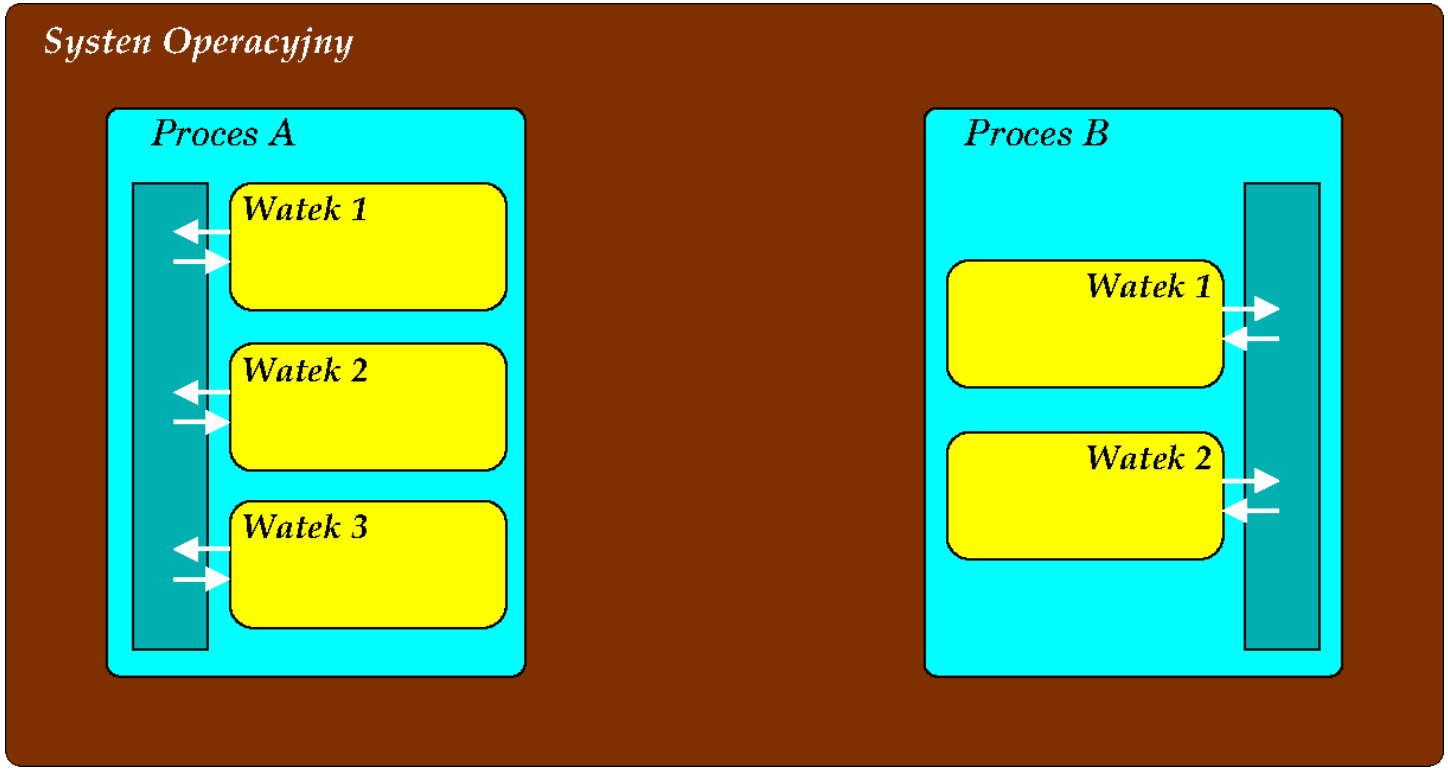
Komunikacja ta może być realizowana na różne sposoby, np. poprzez gniazda (ang. *sockets*), pamięć dzieloną, czy też przekierowanie wyjść i wejść standardowych.

Komunikacja między wątkami



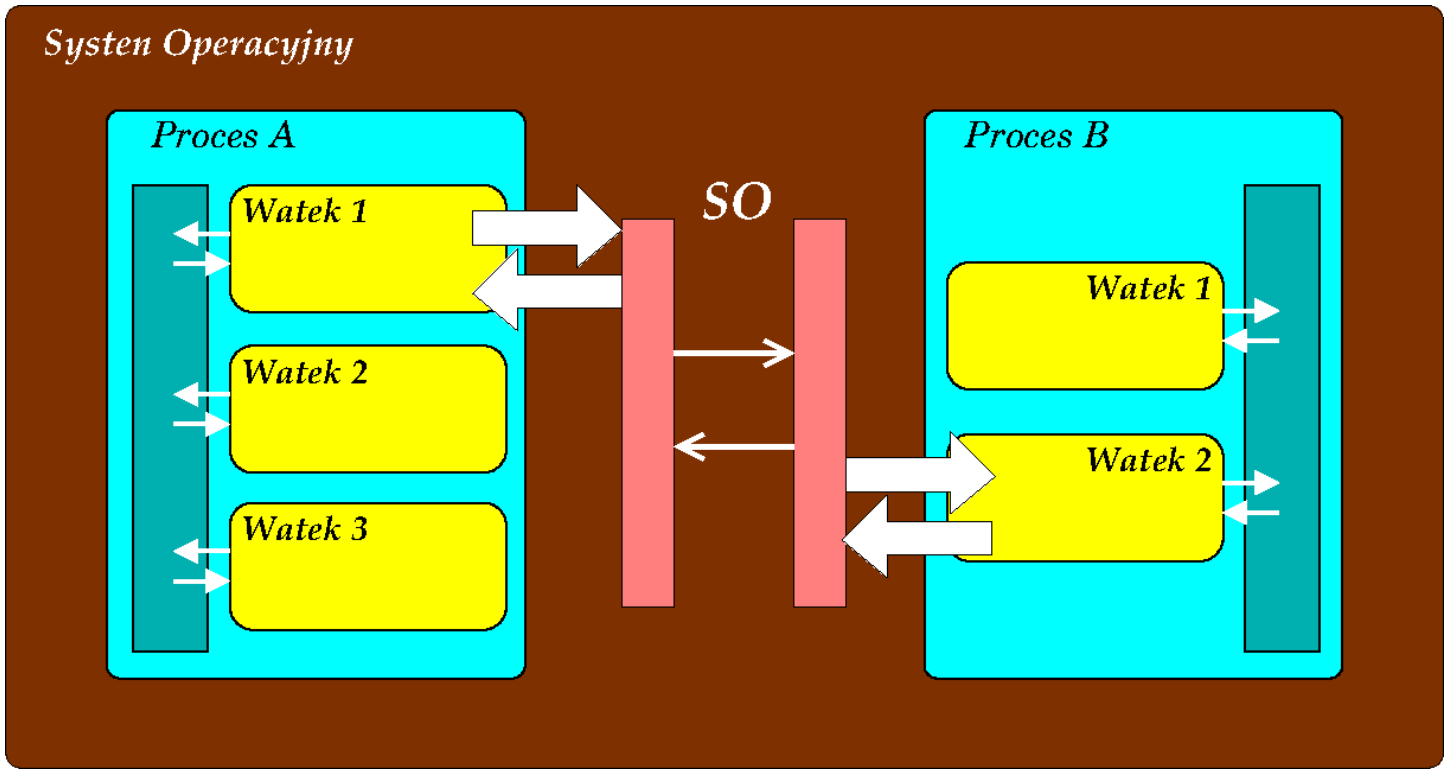
W procesach wielowątkowych komunikacja wewnątrz procesu między poszczególnymi wątkami odbywa się poprzez obszar pamięci procesu.

Komunikacja między wątkami



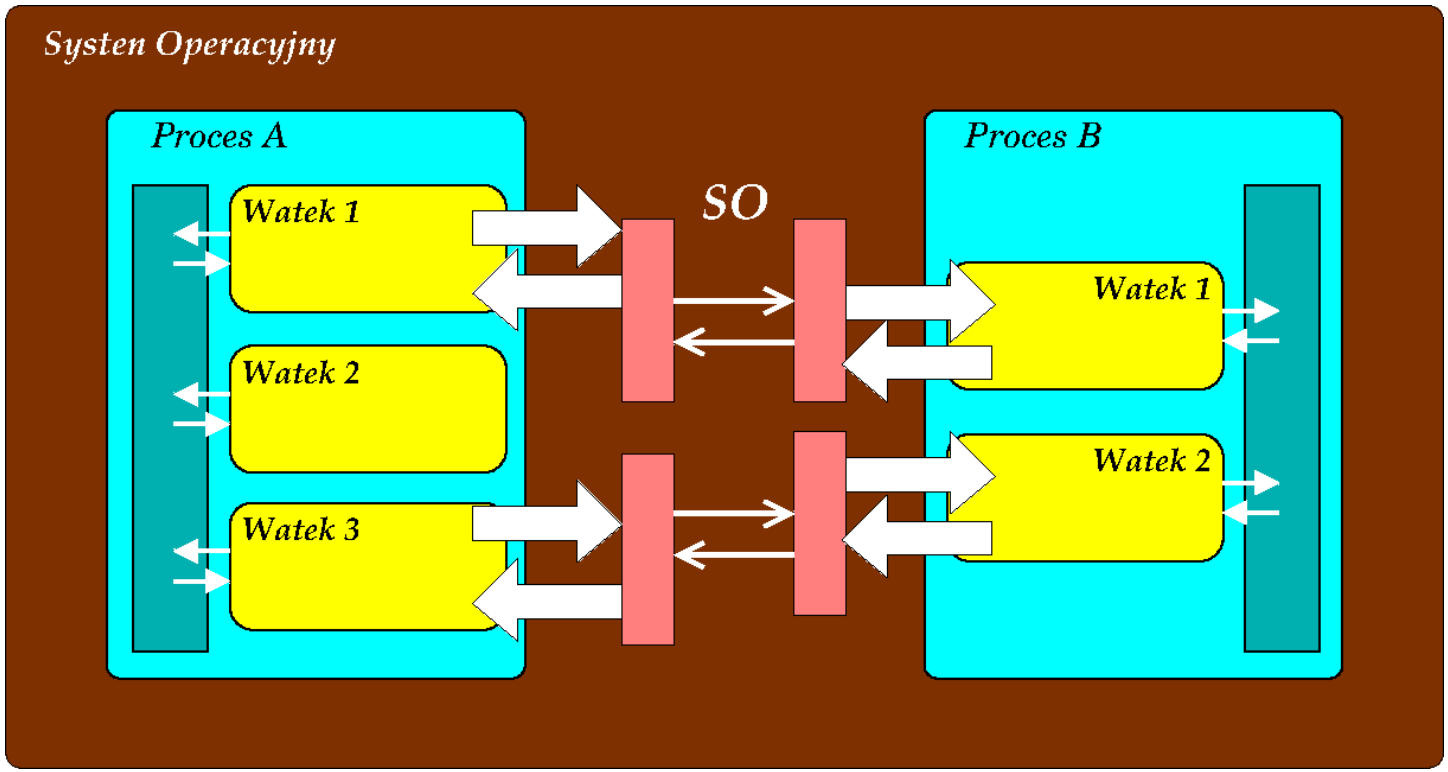
Procesy wielowątkowe mogą pracować niezależnie i zupełnie odizolowane od innych.

Komunikacja między wątkami



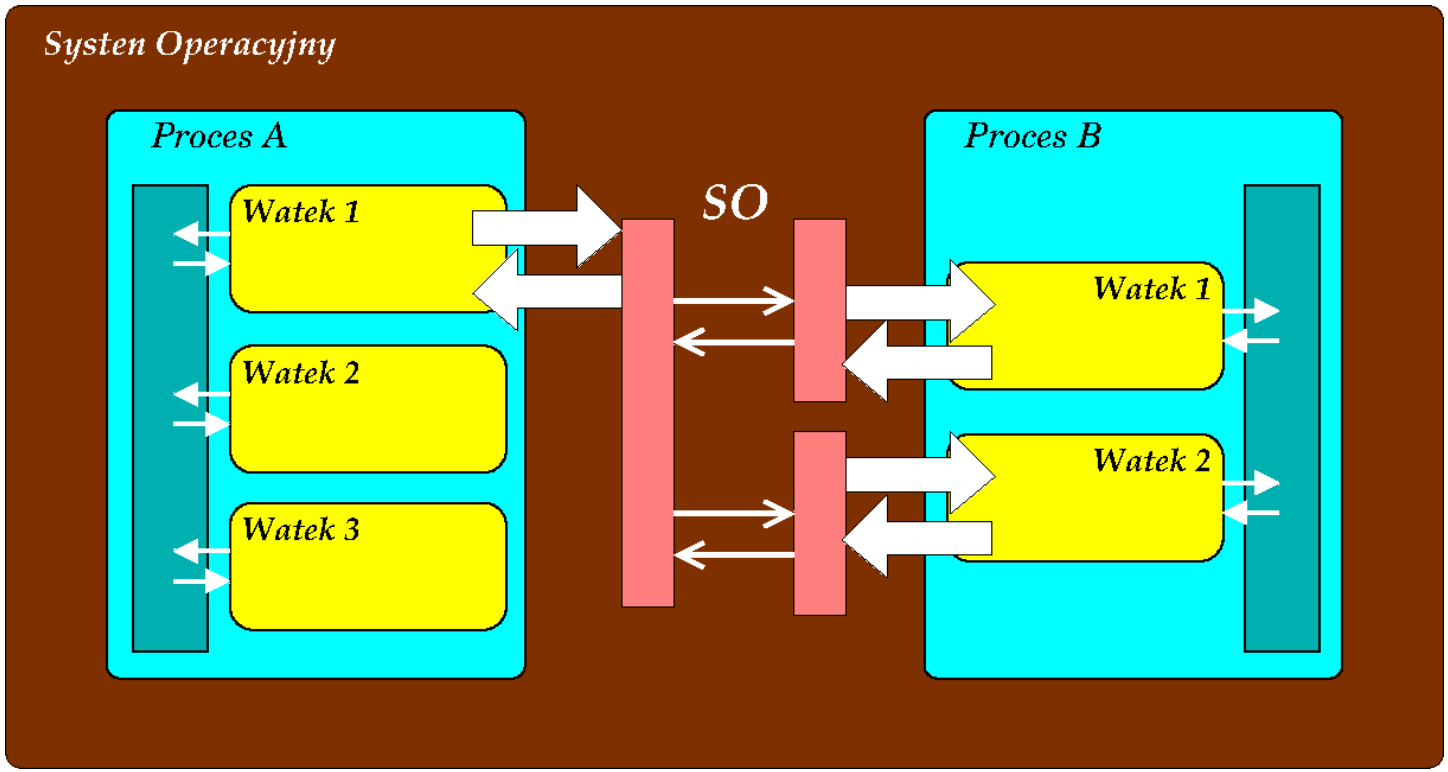
Mogą jednak też komunikować się ze sobą wykorzystując standardowe mechanizmy dostarczane przez system operacyjny. Należy jednak zwrócić uwagę, że w takim przypadku komunikacja zachodzi między wybranymi wątkami.

Komunikacja między wątkami



Komunikacja może zachodzić jednocześnie między różnymi wątkami dwóch procesów i w każdym z kanałów komunikacyjnych mogą być wykorzystywane różne mechanizmy.

Komunikacja między wątkami



Możliwa jest też komunikacja kilku wątków jednego procesu z pojedynczym wątkiem innego procesu.

Ilość wątków

Maksymalna ilość sprzętowych wątków

`std::thread::hardware_concurrency()`

Ta metoda statyczna może zwrócić zero, jeśli na danym sprzęcie nie jest w stanie poprawnie wykryć ilości procesorów lub rdzeni.

```
#include <iostream>
#include <thread>

int main()
{
  std::cout << std::thread::hardware_concurrency() << std::endl;
}
```

Najprostszy wątek w C

```
#include <stdio.h>
#include <pthread.h>


void *simpleThread(void *pArg)
{
  printf("... Komunikat watku\n");
}


int main()
{
  pthread_t  Thr;
  void       *pRes;

  if (pthread_create(&Thr,NULL,simpleThread,NULL) < 0) {
    return 1;
  }
  pthread_join(Thr,&pRes);
}
```

Za działanie wątku odpowiada funkcja `simpleThread`, której wskaźnik jest przekazywany do funkcji tworzącej nowy wątek. Funkcja `simpleThread` staje się rodzajem funkcji `main` dla nowego wątku.

Najprostszy wątek w C

```c
#include <stdio.h>
#include <pthread.h>


void *simpleThread(void *pArg)
{
  printf("...␣Komunikat␣watku\n");
}


int main()
{
  pthread_t  Thr;
  void       *pRes;

  if (pthread_create(&Thr,NULL,simpleThread,NULL) < 0) {
    return 1;
  }
  pthread_join(Thr,&pRes);
}
```

Za działanie wątku odpowiada funkcja `simpleThread`, której wskaźnik jest przekazywany do funkcji tworzącej nowy wątek. Funkcja `simpleThread` staje się rodzajem funkcji `main` dla nowego wątku.

Wywołanie funkcji `pthread_create` powoduje utworzenie wątku i uruchomienie w nim funkcji `simpleThread`.

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

Prototyp funkcji `pthread_create`

Najprostszy wątek w C

```c
#include <stdio.h>
#include <pthread.h>


void *simpleThread(void *pArg)
{
  printf("...␣Komunikat␣watku\n");
}


int main()
{
  pthread_t  Thr;
  void       *pRes;

  if (pthread_create(&Thr,NULL,simpleThread,NULL) < 0) {
    return 1;
  }
  pthread_join(Thr,&pRes);
}
```

Za działanie wątku odpowiada funkcja `simpleThread`, której wskaźnik jest przekazywany do funkcji tworzącej nowy wątek. Funkcja `simpleThread` staje się rodzajem funkcji `main` dla nowego wątku.

Wywołanie funkcji `pthread_join` jest konieczny, aby główny wątek procesu *zaczekał* na zakończenie wątku wykonywanego przez funkcję `simpleThread`.

```c
int pthread_join(pthread_t thread, void **retval);
```

Prototyp funkcji `pthread_join`

Najprostszy wątek w C

```c
#include <stdio.h>
#include <pthread.h>


void *simpleThread(void *pArg)
{
  printf("...␣Komunikat␣watku\n");
}


int main()
{
  pthread_t  Thr;
  void       *pRes;

  if (pthread_create(&Thr,NULL,simpleThread,NULL) < 0) {
    return 1;
  }
  pthread_join(Thr,&pRes);
}
```

Za działanie wątku odpowiada funkcja `simpleThread`, której wskaźnik jest przekazywany do funkcji tworzącej nowy wątek. Funkcja `simpleThread` staje się rodzajem funkcji `main` dla nowego wątku.

Wywołanie funkcji `pthread_join` jest konieczny, aby główny wątek procesu *zaczekał* na zakończenie wątku wykonywanego przez funkcję `simpleThread`.

Jeżeli główny wątek nie zaczeka na zakończenie wątków pobocznych, to prowadzi do *niekontrolowanego* przerwania ich działania.

Najprostszy wątek w C++

```
#include <iostream>
#include <thread>


void simpleThread()
{
    std::cout << "... Komunikat watku\n";
}


int main()
{
    std::thread Thr(simpleThread);
    Thr.join();
}
```

W najprostszym przypadku tworzenie nowych wątków w C++ jest analogiczne do tego, jakie jest znane z C. Tworzenie wątku można zrealizować w konstruktorze obiektu klasty `std::thread`, który będzie przechowywał dane o tym wątku. Jako parametr konstruktora przekazujemy wskaźnik na funkcję wykonującą dany wątek.

Metoda `join` realizuje tę samą operację jak funkcja `pthread_create`. Nie potrzebuje ona dodatkowych parametrów, gdyż wszystkie niezbędne dane znajdzie w obiekcie `Thr`.

Kompilacja i konsolidacja

Dla wersji g++ 5.0.0

```
g++ -std=c++11 watek.cpp -lpthread
```

Co najmniej od wersji g++ 6.3.0

```
g++ watek.cpp -lpthread
```

Chcą skorzystać z możliwości tworzenia wątków i ich obsługi niezbędne jest konsolidowanie programu z biblioteką `pthread`. Zawiera ona implementację wątków opartą na standardzie POSIX, której API jest dostosowane dla języków C/C++.

Jak było w C++

```
class BezKopiowania {
  public:
   BezKopiowania() {}


};


int main()
{
  BezKopiowania  B1, B2;
  BezKopiowania  B3(B1); // Co zrobic, aby zabronic tego
}
```

Jak było w C++ – stare rozwiązanie

```
class BezKopiowania {
  public:
   BezKopiowania() {}
  private:
   BezKopiowania(const BezKopiowania&) {}
};


int main()
{
  BezKopiowania  B1, B2;
  BezKopiowania  B3(B1); // Teraz juz nie mozna
}
```

Jak jest w C++11 i wyżej

```
class BezKopiowania {
  public:
   BezKopiowania() {}
   BezKopiowania(const BezKopiowania&) = delete;

};


int main()
{
  BezKopiowania  B1, B2;
  BezKopiowania  B3(B1); // Teraz juz nie mozna
}
```

Parametry i struktury danych wątka

```
#include <iostream>
#include <thread>


class BackgroundTask {
 public:

   void operator() () const
   {
        std::cout << "...␣Komunikat␣watku\n";
   }
};


int main()
{
   BackgroundTask   oSimpleTask;
   std::thread      Thr(oSimpleTask);

   Thr.join();
}
```

Operator funkcyjny '() const' przeciążamy wtedy, gdy nie chcemy modyfikować obiektu reprezentującego wątek.

Parametry i struktury danych wątka

```
#include <iostream>
#include <thread>


class BackgroundTask {
 public:

   void operator() ()
   {
        std::cout << "...␣Komunikat␣watku\n";
   }
};


int main()
{
   BackgroundTask   oSimpleTask;
   std::thread      Thr(oSimpleTask);

   Thr.join();
}
```

Przeważnie jednak chcemy modyfikować obiekt. Przeciążamy wówczas operator funkcyjny ().

Parametry i struktury danych wątka

```
#include <iostream>
#include <thread>


class BackgroundTask {
 public:

  void operator() () const {  std::cout << "...␣Komunikat␣watku␣(con)\n";  }
  void operator() ()       {  std::cout << "...␣Komunikat␣watku␣(mod)\n";  }
};


int main()
{
  BackgroundTask  oSimpleTask;
  std::thread     Thr(oSimpleTask);

  Thr.join();
}
```

Jeśli oba operatory są przeciążone, to wywołany zostanie operator dla obiektu modyfikowalnego.

Inicjalizacja wątku bez dodatkowego obiektu

```
#include <iostream>
#include <thread>


class BackgroundTask {
 public:

   void operator() ()
   {
        std::cout << "...␣Komunikat␣watku\n";
   }
};


int main()
{
   std::thread      Thr{ BackgroundTask() };

   Thr.join();
}
```

Do obiektu inicalizującego nowy wątek można przekazać obiekt reprezentujący wątek poprzez listę w stylu języka C.

Inicjalizacja wątku bez dodatkowego obiektu

```
#include <iostream>
#include <thread>


class BackgroundTask {
 public:

  void operator() () const
  {
      std::cout << "...␣Komunikat␣watku\n";
  }
};


int main()
{
  std::thread      Thr( BackgroundTask() );   // <-- Tak NIE MOZNA!

  Thr.join();
}
```

Nie można jednak tworzyć obiektu tymczasowego bezpośrednio w liście parametrów tego konstruktora.

Inicjalizacja wątku bez dodatkowego obiektu

```
#include <iostream>
#include <thread>


class BackgroundTask {
 public:

   void operator() () const
   {
        std::cout << "...␣Komunikat␣watku\n";
   }
};


int main()
{
   std::thread      Thr( (BackgroundTask()) );

   Thr.join();
}
```

Należy zastosować dodatkowe nawiasy. Tak już jest OK.

Inicjalizacja wątku bez dodatkowego obiektu

```
#include <iostream>
#include <thread>









int main()
{
  std::thread      Thr([] {
                        std::cout << "...␣Komunikat␣watku\n";
                      }
                   );

  Thr.join();
}
```

Funkcję wykonywaną w wątku można zdefiniować jako funkcję lambda.

Dostęp do danych

```
#include <iostream>
#include <mutex>
#include <thread>


std::mutex  oMutex;


class BackgroundTask {
 public:

  void operator() ()
  {
    std::lock_guard<std::mutex>  oGuard(oMutex);
    std::cout << "\t...\tKomunikat\twatku\n";
  }
};


int main()
{
  std::thread  oThr1{BackgroundTask()}, oThr2{BackgroundTask()};

  oThr1.join();  oThr2.join();
}
```

Tworzenie obiektu `std::lock_guard` umożliwia automatyczne zwolnienie po uruchomieniu destruktora.

Dostęp do danych

```
#include <iostream>
#include <mutex>
#include <thread>


struct SharedData {
  std::mutex  _Mutex;
  double      _Num;
};

SharedData    Data4Threads;


class BackgroundTask {
 public:

  void operator() ()
  {
    std::lock_guard<std::mutex>  oGuard(Data4Threads._Mutex);
    std::cout << "....Komunikat watku\n";
  }
};


int main()
{
  std::thread  oThr1{BackgroundTask()}, oThr2{BackgroundTask()};

  oThr1.join();  oThr2.join();
}
```

Obiektu klasy `std::mutex` dobrze jest skojarzyć z konkretnymi danymi.

Koniec prezentacji
Dziękuję za uwagę