



Politechnika Łódzka

Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

Programowanie sieciowe

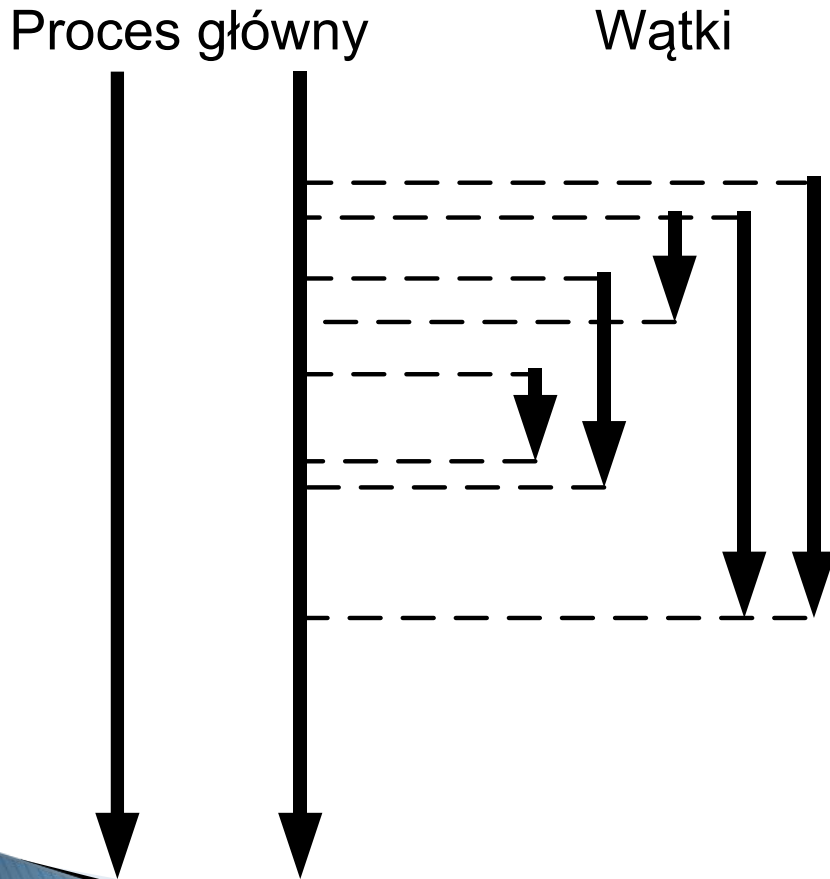
Wykład 4 Techniki programowania współbieżnego



Instytut Informatyki Stosowanej
Politechniki Łódzkiej

Opracował: dr hab. inż. Radosław Wajman

Idea wielozadaniowości



Algorytm szeregowania – ustala kolejność wykonywanych zadań, uwzględnia priorytety

Dzielenie czasu – rozdzielenie czasu procesora dla wielu pracujących na nim wątków

Wywłaszczenie – możliwość „odgórnego” wstrzymania wątku przez planistę (scheduler) i przełączenie zadania (np. Win95 i nowsze, Linux)

Brak wywłaszczenia – tzw. wielozadaniowość kooperacyjna: programy same muszą „zwalniać” procesor. Wadliwa aplikacja może zawiesić cały system (np. Win31)

Wątki – priorytety

Z każdym wątkiem programu może skojarzony zostać priorytet, informujący o tym, jak należy traktować wątek w stosunku do innych wątków programu.

Priorytet jest liczbą całkowitą określającą relatywną ważność wątku w stosunku do innych wątków.

Priorytet wątku nie ma wpływu na szybkość wykonywania wątku, a jedynie na pierwszeństwo w wyborze do wykonania.

Reguły rządzące tym, kiedy zmieniać kontekst wykonania są następujące:

- **Wątek może dobrowolnie odstąpić prawo do wykonania innym wątkom.**
Sytuacja taka ma miejsce, gdy wątek musi czekać na zwolnienie zasobu lub, gdy wykonał metodę `sleep`. Wybierany jest wówczas wątek z oczekujących o najwyższym priorytecie.
- **Wątek może zostać wyparty przez wątek o wyższym priorytecie**
tj., gdy tylko wątek o wyższym priorytecie potrzebuje procesor, to go bierze.

Wątek główny

Wraz z uruchomieniem programu jeden wątek, zwany wątkiem głównym, startuje natychmiast.

Wątek główny jest ważny, ponieważ:

- Jest wątkiem mogącym zapoczątkowywać inne wątki, zwane wątkami potomnymi
- Często musi być ostatnim wątkiem kończącym wykonanie programu.

Wątek główny niczym nie różni się od innych wątków programu.

By uzyskać nad nim kontrolę musimy utworzyć do niego referencję bądź uchwyt.

Tworzenie nowego wątku (C/C++)

```
HANDLE WINAPI CreateThread(  
    SECURITY_ATTRIBUTES lpThreadAttributes,  
    SIZE_T dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId );
```

Pomić C++

Parametry:

- *lpThreadAttributes* – wskaźnik do struktury opisującej parametry bezpieczeństwa nowego wątku. Domyślnie **NULL**.
- *dwStackSize* – określa rozmiar stosu dla nowego wątku. Domyślnie **0**.
- *lpStartAddress* – wskaźnik do funkcji, której instrukcje będą wykonywane w nowo utworzonym wątku programu
- *lpParameter* – lista argumentów przekazywanych do nowego wątku
- *dwCreationFlags* – określa początkowy status procesu, np. **CREATE_SUSPENDED**
- *lpThreadId* – wskaźnik do adresu nowego wątku

Wartość zwracana

Wartością zwracaną przez funkcję **CreateThread** jest uchwyt nowo utworzonego wątku,
lub wartość **-1** w przypadku wystąpienia błędu.

Tworzenie nowego wątku: przykład

Kod wykonywany w oddzielnym wątku
jest opisany na następnym slajdzie

```
HANDLE th1, th2, th3, th4;  
th1 = CreateThread(NULL, 0, my_thread, (void*)1, CREATE_SUSPENDED, NULL);  
th2 = CreateThread(NULL, 0, my_thread, (void*)2, CREATE_SUSPENDED, NULL);  
th3 = CreateThread(NULL, 0, my_thread, (void*)3, CREATE_SUSPENDED, NULL);  
th4 = CreateThread(NULL, 0, my_thread, (void*)4, CREATE_SUSPENDED, NULL);
```

Przykład tworzy 4 wątki domyślnie zawieszone
(`CREATE_SUSPENDED`) wykonujące ten sam kod (`my_thread`)
ale z różnymi parametrami (1..4).

Aby wznowić któryś z wątków, należy skorzystać z funkcji
`ResumeThread`.

Procedura wykonywana w oddzielny wątku

Typ: `LPTHREAD_START_ROUTINE`

Przykład:

```
DWORD WINAPI my_thread(LPVOID arg)
{
    int i = (int)arg;
    for (int j = 1; j < 10; j++)
    {
        Sleep(0);
        printf("%d", i);
    }

    return 0;
}
```

`LPVOID arg` – jest wartością argumentu przekazaną w wywołaniu funkcji **CreateThread**.

W przykładzie jest to liczba typu `int`.

```
VOID WINAPI Sleep(
    DWORD dwMilliseconds);
```

Funkcja **Sleep** usypia aktualnie wykonywany wątek na *dwMilliseconds* milisekund. Gdy wartość parametru jest równa **0**, to następuje wymuszone przełączenie do następnego wątku w kolejce.

Zwalnianie zasobów istniejącego wątku

Wątek jest zasobem systemu operacyjnego – wymaga zwolnienia.

W przypadku pracującego wątku następuje jego przerwanie.

```
BOOL WINAPI CloseHandle(HANDLE hObject);
```

Parametry:

- *hObject* – uchwyt wątku utworzonego funkcją **CreateThread**.

Uwaga!

W języku C/C++ zaalokowana pamięć nie zostanie zwolniona podczas zwalniania pamięci wątku.

Wznawianie/Zawieszanie wykonywania wątku

```
BOOL WINAPI ResumeThread(HANDLE hThread);
```

```
BOOL WINAPI SuspendThread(HANDLE hThread);
```

Funkcja **ResumeThread** wznowia wykonywanie zawieszonego wątku, szczególnie jeśli wątek został utworzony z parametrem **CREATE_SUSPENDED**.

SuspendThread zawiesza (chwilowo wstrzymuje) wykonywanie wątku.

W przypadku błędu obie funkcje zwracają -1 (kod błędu z *GetLastError*).

Funkcje posługują się *licznikiem zawieszeń*.
Gdy wartość *licznika* == 0, wątek pracuje.

```
th1 = CreateThread(NULL, 0, my_thread, (void*)1, CREATE_SUSPENDED,  
    NULL);
```

```
ResumeThread(th1);
```

```
SuspendThread(th1);
```

Kończenie wątku

Wątek może zakończyć działanie wywołując funkcję `ExitThread` z dowolnym kodem powrotu.

```
VOID WINAPI ExitThread(DWORD dwExitCode);
```

Procedura uruchamia zamykanie w oddzielny wątku i zwraca parametr `DWORD`, Można wykorzystać instrukcję `return` języka C do zwrócenia kodu powrotu.

Do odczytania kodu powrotu służy funkcja `GetCodeThread`:

```
BOOL WINAPI GetExitCodeThread(HANDLE hThread,  
                                LPDWORD lpExitCode);
```

Przykład:

```
DWORD code;  
GetExitCodeThread(th1, &code);  
printf("Kod zakonczenia watku: %d\n", code);
```

Oczekiwanie na zakończenie wątku

Podstawową funkcją do synchronizacji poprzez oczekiwanie na kończenie działającego wątku jest funkcja:

```
DWORD WINAPI WaitForSingleObject(  
    HANDLE hHandle,  
    DWORD dwMilliseconds);
```

Oczekuje ona *dwMilliseconds* na zakończenie wątku.
Wartość **INFINITE** spowoduje oczekiwanie w nieskończoność.

Wynik:

WAIT_OBJECT_0 – wątek zakończył się samoistnie (np. **ExitThread**)

WAIT_TIMEOUT – przekroczony czas oczekiwania (*dwMilliseconds*).

Przykład:

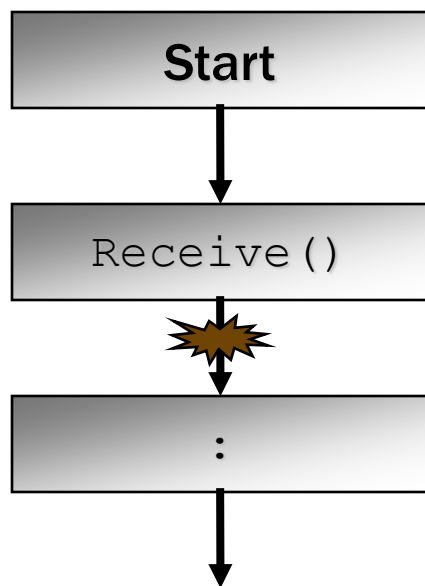
```
th1 = CreateThread(NULL, 0, my_thread, (void*)1, 0/*CREATE_SUSPENDED*/, NULL);  
WaitForSingleObject(th1, INFINITE);
```

Serwer wielowątkowy

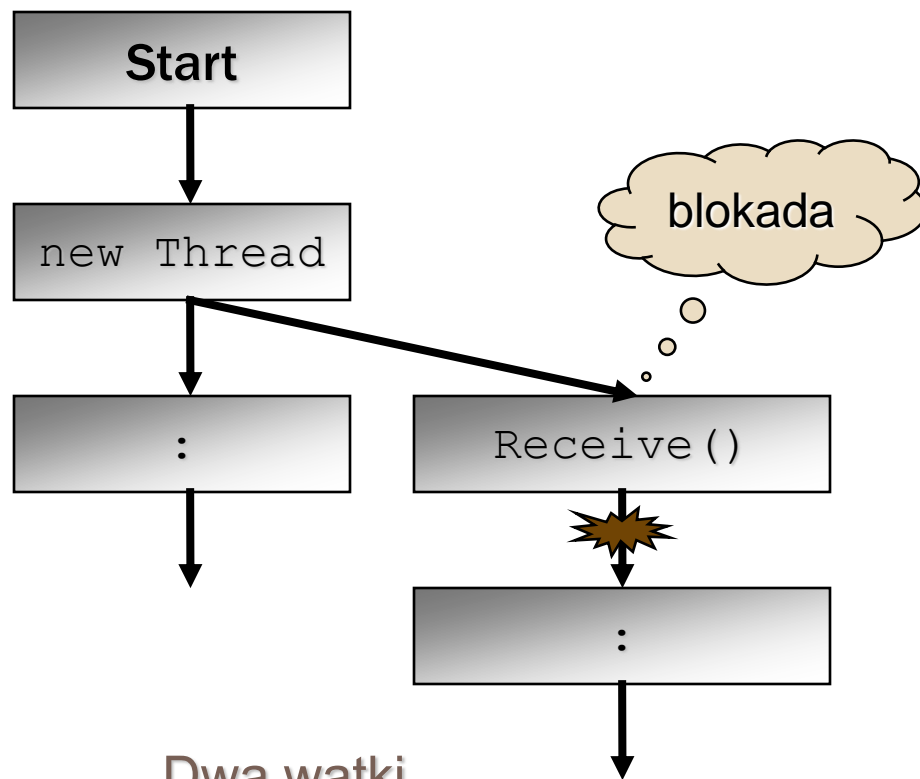
- ▶ Jeżeli istnieje konieczność obsługi wielu połączeń klienckich jednocześnie, serwer musi generować osobne wątki dla każdego z nich.
- ▶ Obiekt / klasa wątku (ang. *Thread*) w C# realizuje to w prosty sposób.
 - Klasa dostarcza nie-statyczne metody, które umożliwiają łatwy dostęp do funkcjonalności obiektu wątku.
 - Np. obiekt *TCPListener* po zaakceptowaniu klienta zwraca gniazdo. Zaraz po jego utworzeniu wystarczy jego obsługę przerzucić do osobnego wątku .

Wątki

- ▶ `Accept()` i `Receive()` blokują działanie aplikacji
 - Czekają na połączenie lub aż dane zostaną odebrane



Jeden wątek



Dwa wątki

Tworzenie wątku (C#)

► `System.Threading` namespace

```
using System.Threading;
```

```
void ThreadProc()  
{  
    :  
    while (true)  
    {  
        s.ReceiveFrom(...);  
    }  
}  
void MainProc()  
{  
    :  
    Thread t = new Thread(new ThreadStart(ThreadProc));  
    t.Start();  
    :  
}
```

Uruchamianie głównego wątku serwera – C#

```
private TcpListener tcpListener;  
private Thread listenThread;  
  
public int Server()  
{  
    this.tcpListener = new TcpListener(IPAddress.Any, 7);  
    this.listenThread = new Thread(new ThreadStart(ListenForClients));  
    this.listenThread.Start();  
    this.listenThread.Join();  
    return 0;  
}
```

Funkcja realizująca nasłuchiwanie

```
private void ListenForClients()
{
    try
    {
        this.tcpListener.Start();    //run listening
    }
    catch (Exception e)
    {
        this.SetTextOnListBox1("Server could not start. Error of binding");
        return;
    }
    this.SetTextOnListBox1("Server run on port 7");
    while (true)
    {
        //blocks until a client has connected to the server
        TcpClient client = this.tcpListener.AcceptTcpClient();
        this.SetTextOnListBox1("New client connected from: " + client.Client.RemoteEndPoint);

        //create a thread to handle communication with connected client
        Thread clientThread = new Thread(new ParameterizedThreadStart(HandleClientComm));
        clientThread.Start(client);
    }
}
```


Wątek obsługi klienta

```
private void HandleClientComm(object client)
{
    TcpClient tcpClient = (TcpClient)client;
    String clientIP = "" + tcpClient.Client.RemoteEndPoint.AddressFamily;
    NetworkStream clientStream = tcpClient.GetStream();

    int size = 1024;
    byte[] message = new byte[size];
    int bytesRead;
    ASCIIEncoding encoder = new ASCIIEncoding();

    while (true)
    {
        bytesRead = 0;
        try { //blocks until a client sends a message
            bytesRead = clientStream.Read(message, 0, size);
            clientStream.Write(message, 0, bytesRead);
        } catch { break; //a socket error has occurred }

        if (bytesRead == 0) break; //the client has disconnected from the server

        //message has successfully been received
        this.SetTextOnListBox1(" Message from " + tcpClient.Client.RemoteEndPoint + ": " +
                               encoder.GetString(message, 0, bytesRead));
        //System.Diagnostics.Debug.WriteLine(encoder.GetString(message, 0, bytesRead));
    }
    tcpClient.Close();
}
```

Delegacja metody do GUI

```
// Ta delegacja włącza asynchroniczne wywołanie
// w przypadku dostępu do własności kontrolki.
delegate void SetTextCallback(string text);

private void SetTextOnListBox1(string text)
{
    // InvokeRequired porównuje ID wątku wywołania z wątkiem
    // tworzącym np. kontrolkę.
    // Jeśli wątki są różne, zwraca true.
    if (this.listBox1.InvokeRequired)
    {
        SetTextCallback d = new SetTextCallback(SetTextOnListBox1);
        this.Invoke(d, new object[] { text });
        //Wykonuje daną delegację w wątku, który „posiada” kontrolkę
        //i jej podstawowy uchwyt do formularza
    }
    else
    {
        this.listBox1.Items.Insert(0, text);
    }
}
```

Więcej o delegacjach:

<https://arvangel.wordpress.com/2011/08/27/c-var-delegacje-metody-anonimowe-i-wyrazenia-lambda/>

Implementacja wątków w JAVA

By utworzyć nowy wątek trzeba utworzyć obiekt typu **Thread**.

Można to zrobić na dwa sposoby:

- Zaimplementować interfejs **Runnable**

- Rozszerzyć klasę **Thread**

Implementacja Runnable

Interfejs **Runnable** jest abstrakcją wykonywalnego kodu.

Można utworzyć wątek bazując na dowolnym obiekcie implementującym ten interfejs.

By implementować ten interfejs, klasa musi jedynie implementować pojedynczą metodę:

public void run()

Ciało tej metody stanowi kod wątku.

Wątek taki kończy się wraz z zakończeniem działania metody **run()**.

Następnie wywołanie konstruktora klasy Thread:

Thread(Runnable obWątku, String nazwa);

tworzy nowy wątek bazując na obiekcie *obWątku* z nazwą *nazwa*.

Metoda **start()** uruchamia metodę **run()** obiektu wątku.

Implementacja wątków w JAVA

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

Rozszerzenie klasy Thread

Innym sposobem tworzenia własnych wątków jest definiowanie nowych klas dziedziczących po klasie **Thread**, a następnie tworzenie ich instancji.

Klasa rozszerzająca musi nadpisać metodę **run()** klasy **Thread**.

Podobnie należy uruchomić metodę **start()** by rozpocząć wykonanie nowego wątku.

Co lepsze?

Zgodnie z metodologią obiektową dziedziczenie klas ma sens, gdy klasa pochodna zmienia coś istotnego w klasie dziedziczonej.

Jeżeli zmianie ulega tylko metoda **run()**, wydaje się, że w takim przypadku bardziej odpowiednia jest implementacja interfejsu **Runnable**.

Powyższe rozważania traktują raczej o dobrym stylu, niż o poprawności kodowania.

Implementacja wątków w JAVA

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

Funkcja realizująca nasłuchiwanie (JAVA)

```
while(! isStopped()){  
    Socket clientSocket = null;  
    try {  
        clientSocket = this.serverSocket.accept();  
    } catch (IOException e) {  
        System.out.println("Server Stopped.") ;  
        return;  
    }  
  
    new Thread(  
        new WorkerRunnable(  
            clientSocket, "Multithreaded Server")  
        ).start();  
}
```

Wątek obsługi klienta (JAVA)

```
public class WorkerRunnable implements Runnable{

    protected Socket clientSocket = null;
    protected String serverText    = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText    = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write(("message to client" +
                this.serverText + " - " + "").getBytes());
            output.close();
            input.close();
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```


Metody `isAlive()` i `join()`

Skąd jeden wątek może wiedzieć, że inny wątek się zakończył?

Odpowiedzi na to pytanie mogą udzielić dwie metody klasy `Thread`:

- `final boolean isAlive()`

zwracająca `true`, gdy wątek, dla którego ją wywołujemy działa, `false` w przeciwnym przypadku

- `final void join() throws InterruptedException`

ta metoda czeka, aż wątek, dla którego wywołujemy zakończy się; dodatkowe wersje `join` pozwalają podać maksymalny czas oczekiwania na zakończenie wątku

Implementacja wątków w Python

Jednym ze sposobów jest mechanizm niskiego poziomu zrównoleglania

Są dwa moduły umożliwiające realizację tego zadania:

- `_thread`
- `threading`

Moduł wątków `thread` jest „przestarzały” od dłuższego czasu. Użytkownicy są zachęcani do używania `threading`.

Dlatego w Pythonie 3 moduł `thread` nie jest już dostępny.

Jednak jego nazwa została zmieniona na `_thread` ze względu na kompatybilność wsteczną w Pythonie3.

Implementacja wątków w Python

Nowszy moduł zapewnia znacznie zaawansowaną obsługę wątków na wyższym poziomie.

Moduł **threading** udostępnia wszystkie metody modułu **_thread** oraz zapewnia kilka dodatkowych metod –

- **threading.activeCount()** — Zwraca liczbę aktywnych obiektów wątku.
- **threading.currentThread()** — Zwraca liczbę obiektów wątku w kontrolce wątku wywołującego.
- **threading.enumerate()** — Zwraca listę wszystkich obiektów wątków, które są aktualnie aktywne.

Oprócz metod moduł posiada klasę **Thread** wraz z metodami:

- **run()** — Metoda **run()** jest punktem wejścia dla wątku.
- **start()** — Metoda **start()** uruchamia wątek przez wywołanie metody **run**.
- **join([czas])** — Metoda **join()** czeka na zakończenie wątków.
- **isAlive()** — Metoda **isAlive()** sprawdza, czy wątek nadal jest wykonywany.
 - **getName()** — Metoda **getName()** zwraca nazwę wątku.
 - **setName()** — Metoda **setName()** ustawia nazwę wątku.

Serwer TCP w Python

```
class serverThread (threading.Thread):  
  
    def __init__(self):  
        threading.Thread.__init__(self)  
        self.ServerSocket = socket.socket()  
  
    def run(self):  
        try:  
            self.ServerSocket.bind((host, port))  
        except socket.error as e:  
            print(str(e))  
        self.ServerSocket.listen()  
        print(f'Server is listing on the port {port}...')  
        while True:  
            Client, address = self.ServerSocket.accept()  
            cliTh = clientThread(Client, address)  
            cliTh.start()
```

Serwer TCP w Python

```
class clientThread (threading.Thread):
    def __init__(self, Client, address):
        threading.Thread.__init__(self)
        self.Client = Client;
        self.address = address;
    def printMsg(self, msg):
        print(f'FROM: {self.address[0]}:{str(self.address[1])}: {msg} ({len(msg)} bytes)')
    def run(self):
        try:
            print(f'Connected to: {self.address[0]}:{str(self.address[1])}')
            while True:
                data = self.Client.recv(1024)
                if len(data) <= 0:
                    break
                self.printMsg(data.decode('utf-8'))
                self.Client.send(data)
        except socket.error as e:
            print(str(e))
        self.Client.close()
        print(f'CLOSED connection: {self.address[0]}:{str(self.address[1])}')
```

Mechanizmy synchronizacji wątków

- Sekcja krytyczna
- Semafor
- **Mutex** (*Mutual Exclusion* – wzajemne wykluczenie) – działają podobnie do sekcji krytycznych, są ponadto mechanizmem IPC (*Interprocess Communication*)

Sekcja krytyczna

Obszar kodu objęty sekcją krytyczną może być wykonywany przez tylko jeden wątek w danej chwili czasowej.

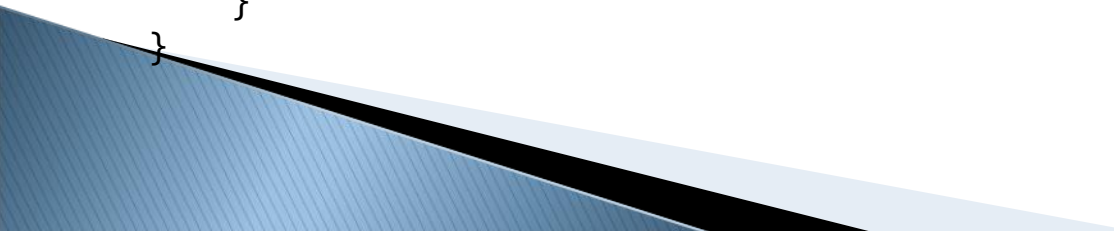
Sekcja krytyczna – C#

a raczej w tym przypadku jej brak

```
class Program
{
    private static int a = 62;
    private static int b = 20;

    static void Main(string[] args)
    {
        for (int i = 0; i < 100; i++)
        {
            new Thread(Divide).Start();
        }
    }

    private static void Divide()
    {
        b = 23;
        if (b != 0)
        {
            Console.WriteLine(a / b);
        }
        b = 0;
    }
}
```



Sekcja krytyczna – C#

```
class Program
{
    private static int a = 62;
    private static int b = 20;
    private static object o = new object();

    static void Main(string[] args) {
        for (int i = 0; i < 100; i++)
        {
            new Thread(Divide).Start();
        }
    }

    private static void Divide() {
        lock (o) {
            b = 23;
            if (b != 0)
            {
                Console.WriteLine(a / b);
            }
            b = 0;
        }
    }
}
```

Sekcja krytyczna – C#

Do klasy zostało dołożone pole o typie `object`

Blokowanie musi odbyć się na typie referencyjnym.

Często zdarza się, że programiści stosują tutaj konstrukcję
`lock(this)` czy też `lock(typeof(MyClass))`.

Należy unikać takich konstrukcji.

Dobre nawyki programowania dla `lock` mówią, że obiekt blokowany powinien być prywatnym polem klasy typu referencyjnego.

Sekcja krytyczna – WINAPI

```
CRITICAL_SECTION cs;
```

```
// .....
```

```
DWORD WINAPI my_thread(LPVOID arg)
```

```
{
```

```
    EnterCriticalSection(&cs);
```

```
    Jakiś KOD;
```

```
    LeaveCriticalSection(&cs);
```

```
}
```

```
int main(void)
```

```
{
```

```
    InitializeCriticalSection(&cs);
```

```
    HANDLE th1, th2, th3;
```

```
    th1 = CreateThread(NULL, 0, my_thread, (void*)1, 0, NULL);
```

```
    th2 = CreateThread(NULL, 0, my_thread, (void*)1, 0, NULL);
```

```
    th3 = CreateThread(NULL, 0, my_thread, (void*)1, 0, NULL);
```

```
    WaitForSingleObject(th1, INFINITE);
```

```
    WaitForSingleObject(th2, INFINITE);
```

```
    WaitForSingleObject(th3, INFINITE);
```

```
    DeleteCriticalSection(&cs);
```

```
}
```



Semafor

Semafor jest *uogólnieniem sekcji krytycznej*. Pozwala na określenie **ilości wątków wykonujących jednocześnie** dany fragment kodu.

Semafor można stosować do wyzwalania zdarzeń, synchronizacji wątków, itd...

Z każdym semaforem skojarzony jest licznik.

Jeśli wartość licznika > 0 , to semafor otwiera możliwość dostępu (sygnalizuje).

Jeśli wartość licznika $= 0$, to semafor zamyka drogę dostępu (nie sygnalizuje)

Semafor

Implementacja semafora wymaga zaprogramowania dwóch metod:

wait i **signal**.

```
// ustawienie wartości początkowej semafora
```

```
counter = 5;
```

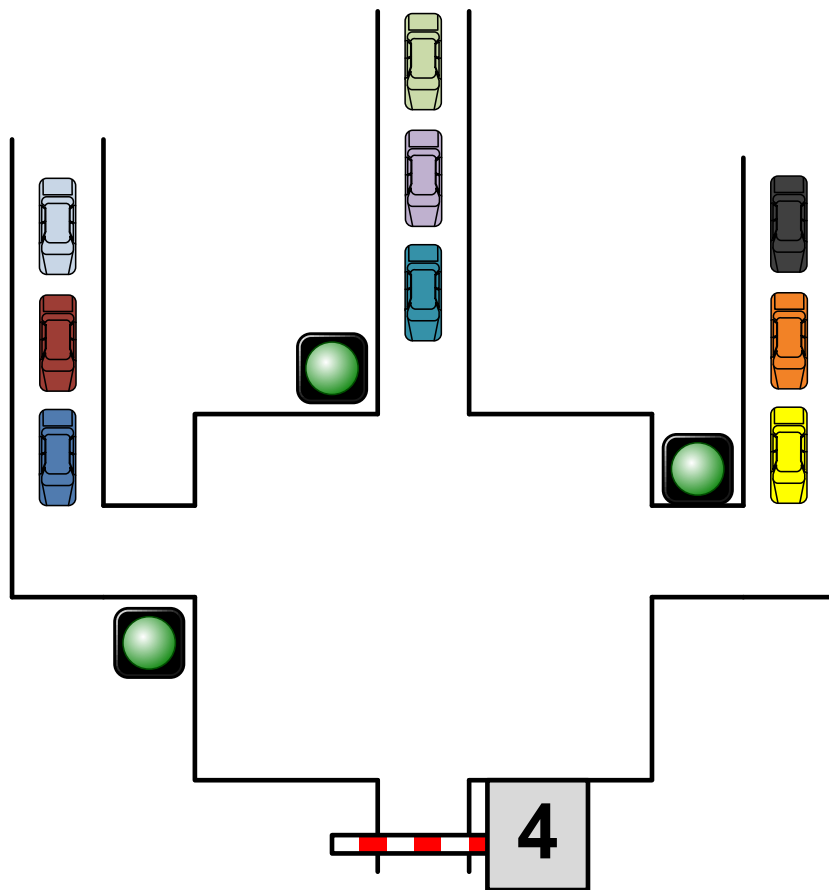
Wait monitoruje semafor i jeśli ten **sygnalizuje**, to **Wait** dopuszcza wykonanie. Jeśli semafor **nie sygnalizuje**, to wątek jest wstrzymywany (aż do ponownej sygnalizacji – **Signal**).

```
void Wait()  
{  
    while(counter<=0) { ... }  
    counter--;  
}
```

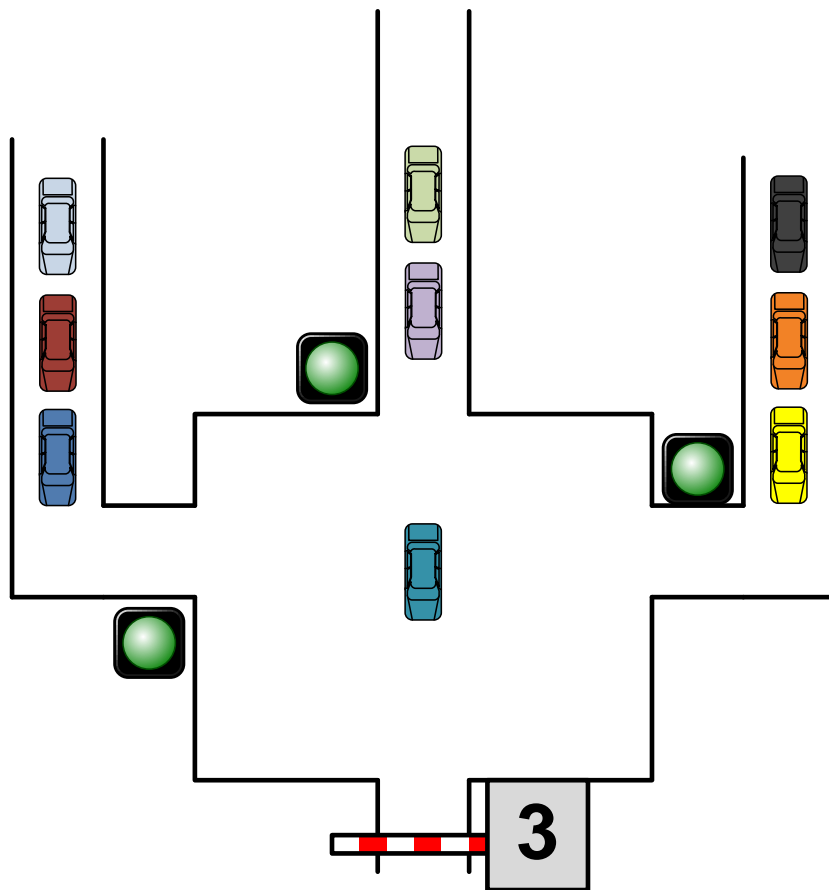
Signal zawsze uruchamia sygnalizację (poprzez zwiększenie licznika).

```
void Signal()  
{  
    counter++;  
}
```

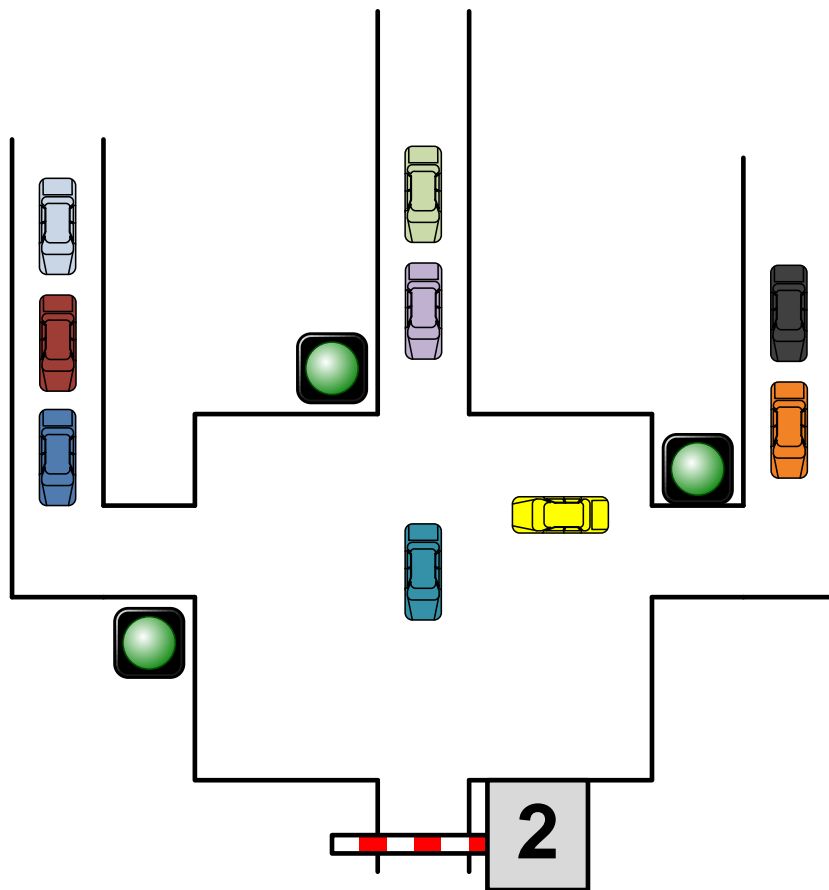
Semafony – przykład



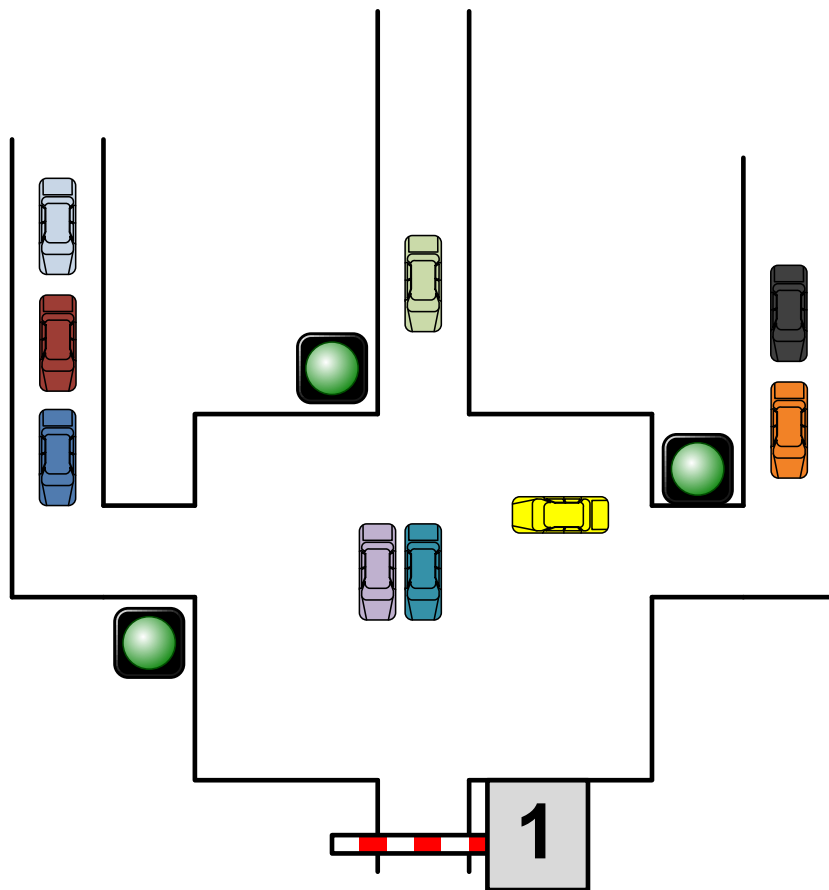
Semafony – przykład



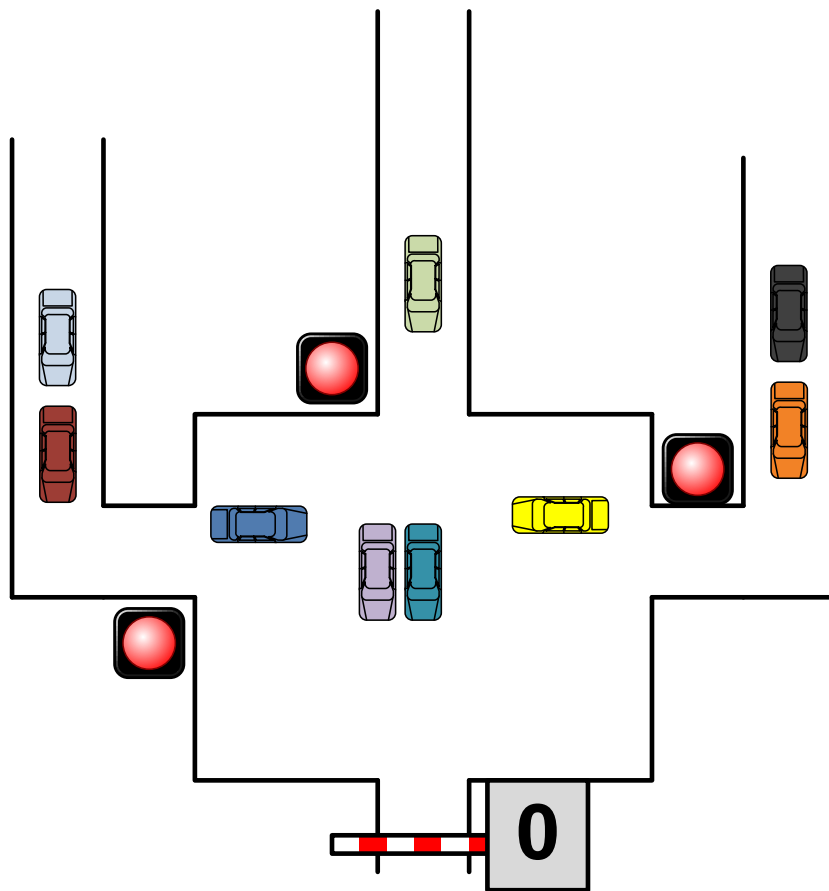
Semafor – przykład



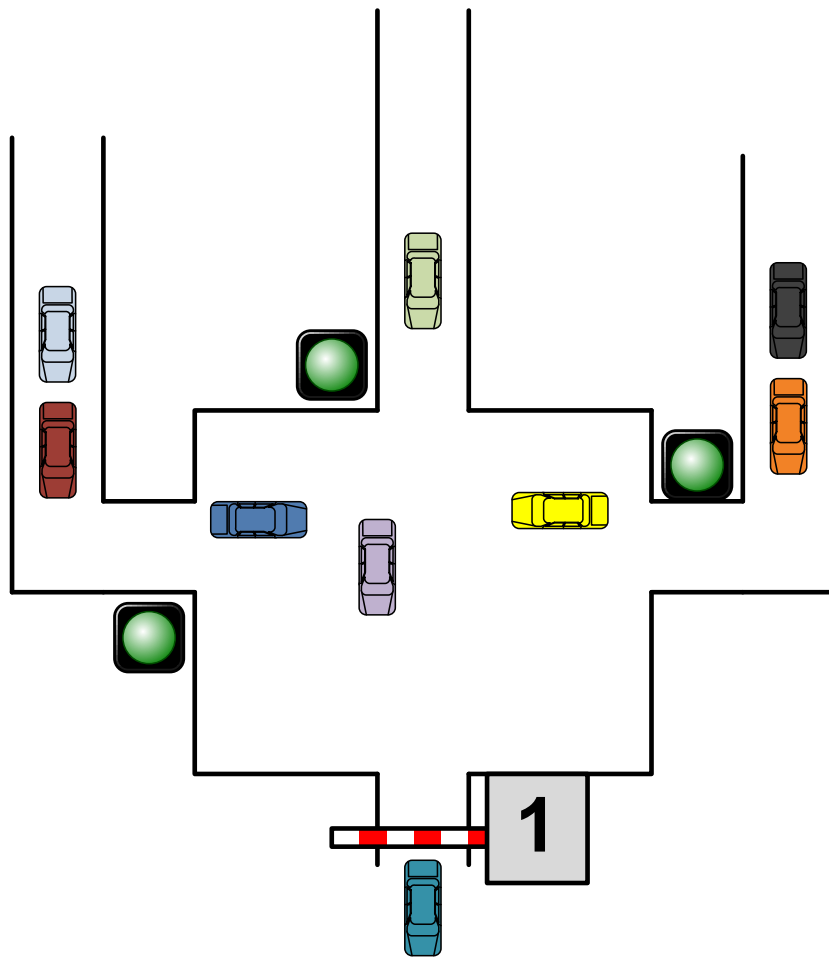
Semafony – przykład



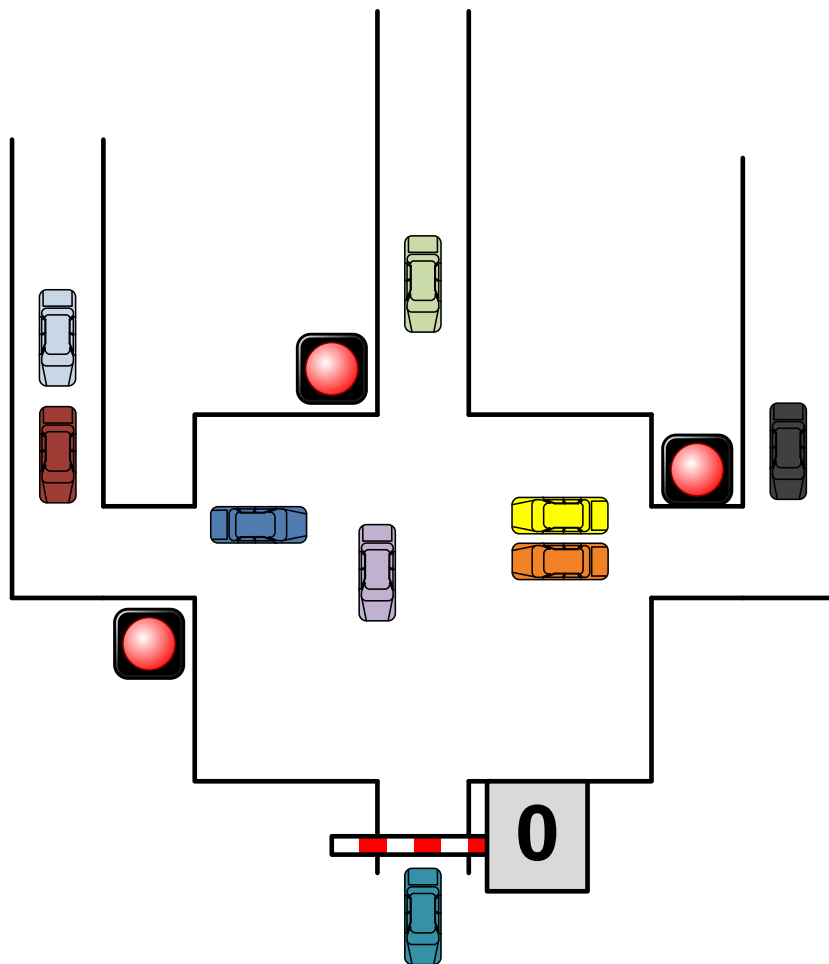
Semafony – przykład



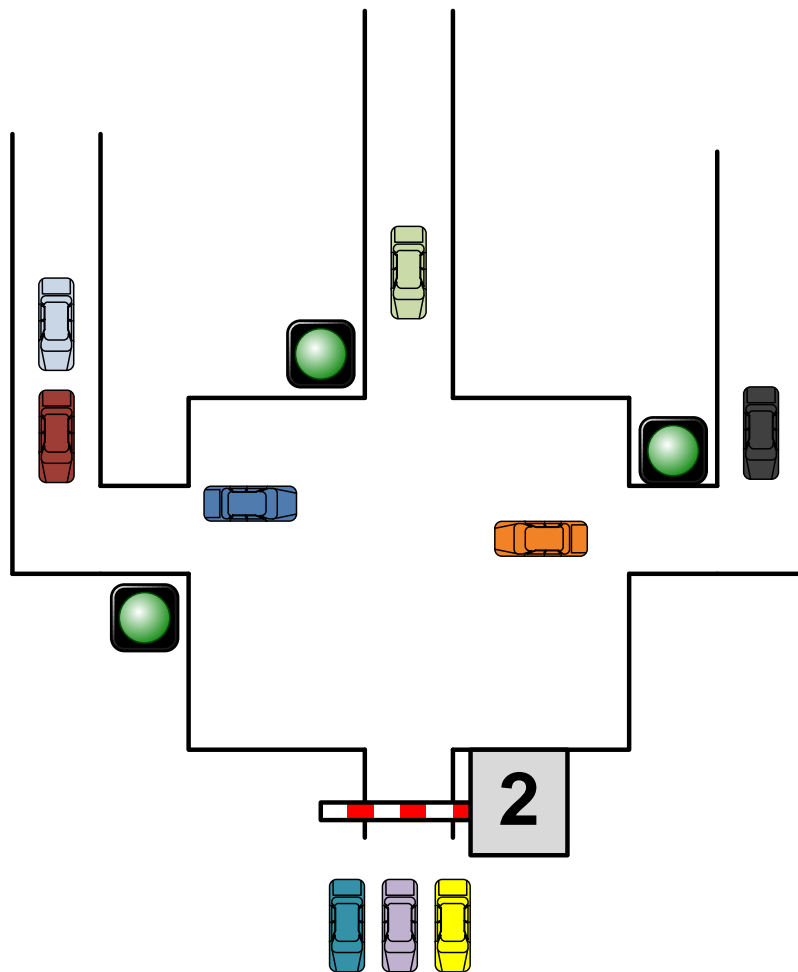
Semafor – przykład



Semafony – przykład



Semafony – przykład



Semafor – Windows API

Semafor tworzy się za pomocą `CreateSemaphore` a zwalnia za pomocą `CloseHandle`.

```
HANDLE WINAPI CreateSemaphore(  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
    LONG lInitialCount,  
    LONG lMaximumCount,  
    LPCTSTR lpName);
```

Parametry:

- *lpSemaphoreAttributes* – wskaźnik na strukturę przechowującą atrybuty semafora do utworzenia; domyślnie **NULL**,
- *lInitialCount* – wartość początkowa semafora. Dla omawianego przykładu z samochodami będzie to 4.
- *lMaximumCount* – maksymalna wartość licznika semafora. Dla omawianego przykładu będzie to 4. **Jaka będzie interpretacja tej wartości na podstawie omawianego przykładu?**
- *lpName* – nazwa semafora. Jeśli semafor o tej nazwie już istnieje, zwracany jest jego uchwyt. Domyślnie **NULL**.

Zwalnianie zasobów semafora:

```
BOOL WINAPI CloseHandle(HANDLE hObject);
```

Semafor – Windows API

Instrukcja **wait**:

```
DWORD WINAPI WaitForSingleObject(  
    HANDLE hSemaphore,  
    DWORD dwMilliseconds);
```

Instrukcja **signal**:

```
BOOL WINAPI ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG lReleaseCount,  
    LPLONG lpPreviousCount);
```

Parametry:

- *hSemaphore* – uchwyt semafora,
- *dwMilliseconds* – ilość milisekund oczekiwania; domyślnie **INFINITE**,
- *lReleaseCount* – wartość o jaką zostanie zwiększony licznik; domyślnie 1,
- *lpPreviousCount* – wskaźnik na poprzednią wartość; domyślnie **NULL**.

Semafory – C#

```
class Program {  
    private static Semaphore s = new Semaphore(2, 2); //wart. pocz. 2, max count 2  
    static void Main(string[] args)  
    {  
        for (int i = 0; i < 10; i++)  
        {  
            Thread t = new Thread(Run);  
            t.Name = i.ToString();  
            t.Start();  
        }  
    }  
    public static void Run()  
    {  
        s.WaitOne();  
        Console.WriteLine(Thread.CurrentThread.Name);  
        Thread.Sleep(1000);  
        s.Release();  
    }  
}
```

Tworzymy tutaj obiekt klasy Semaphore o pojemności 2 wątków i z wolnymi dwoma miejscami na wątki.

Następnie tworzymy 10 wątków i przekazujemy im do wykonania metodę Run.

Metoda ta może dzięki semaforze być obsługiwana tylko przez dwa wątki na raz

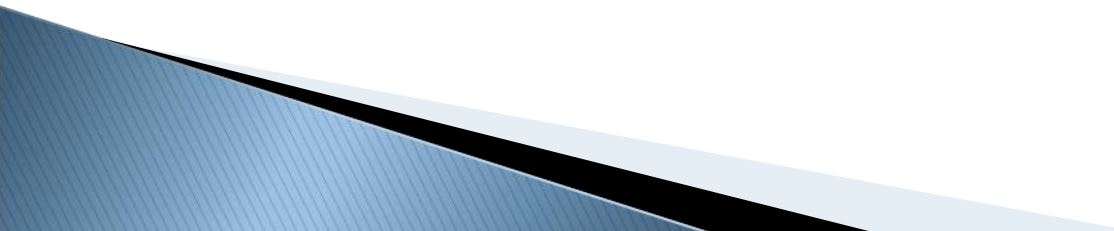
Mutex

Mutex posiada pewną zaletę niedostępną dla dwóch powyższych rozwiązań,

Potrafi synchronizować/blokować dostęp do określonego zasobu pomiędzy różnymi procesami działającymi w systemie.

Mutex może mieć zatem swego rodzaju zasięg globalny w obszarze całego systemu – nie tylko w pojedynczej instancji aplikacji.

Np. w sytuacji, gdy jedna aplikacja może mieć w systemie wiele instancji i każda z tych instancji może zapisywać dane do jednego pliku.



Mutex – C#

```
class Logger
{
    private readonly string MUTEX_GUID = "e1ffff8f-c91d-4188-9e82-c92ca5b1d057";
    private Mutex m_oLoggerMutex = null;

    public Logger()
    {
        m_oLoggerMutex = new Mutex(false, MUTEX_GUID);
    }

    public void Log()
    {
        m_oLoggerMutex.WaitOne();
        {
            StreamWriter oFile = null;
            try
            {
                oFile = File.AppendText("logger.log");
                oFile.WriteLine("Przykładowa linia...");
                oFile.Flush();
            }
            finally
            {
                if (null != oFile)
                {
                    oFile.Close();
                    oFile.Dispose();
                }
            }
        }
        m_oLoggerMutex.ReleaseMutex();
    }
}
```

ID może być dowolne
(pod warunkiem, że wszystkie
aplikacje/wątki
będą się do niej stosować).

Mutex tworzy jeden z przeciążonych konstruktorów,
w którym np. wyłącza się automatyczne ustawienie
i wskazuje jego nazwę.

Zakleszczenie

Specjalny typ błędu związanego z wielowątkowością.

Z zakleszczeniem (ang. *deadlock*) mamy do czynienia np. wtedy,

- gdy jeden wątek otwiera monitor obiektu T1,
- drugi otwiera monitor obiektu T2,
- i pierwszy próbuje wywołać synchronizowaną metodę obiektu T2,
- natomiast drugi próbuje wywołać synchronizowaną metodę obiektu T1.



Oba wątki będą czekały na zwolnienie zasobów blokując je sobie wzajemnie.

Taką sytuację nazywamy zakleszczeniem.

Zakleszczenie jest trudne do wykrycia ponieważ:

- W programie, w którym zakleszczenie jest możliwe zwykle rzadko do niego dochodzi; oba wątki muszą w tym samym czasie być w odpowiednim miejscu kodu,
- W wielu programach występuje więcej niż dwa wątki i dwie synchronizowane metody; może dojść do zakleszczeń zespołowych.

```

public class DeadLockTest {
    public static void main(String args[]) {
        final DeadLockTest t1 = new DeadLockTest();
        final DeadLockTest t2 = new DeadLockTest();

        Runnable r1 = new Runnable() {
            public void run() {
                try {
                    synchronized (t1) {
                        System.out
                            .println("r1 has locked t1, now going
                                Thread.sleep(100);
                        System.out
                            .println("r1 has awake , now going to
                                synchronized (t2) {
                                    Thread.sleep(100);
                                }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };

        Runnable r2 = new Runnable() {
            public void run() {
                try {
                    synchronized (t2) {
                        System.out
                            .println("r2 has aquire the lock of t2 now going to sleep");
                        Thread.sleep(100);
                        System.out
                            .println("r2 is awake , now going to aquire the lock from t1");
                        synchronized (t1) {
                            Thread.sleep(100);
                        }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };

        new Thread(r1).start();
        new Thread(r2).start();
    }
}

```

deadlocktest.DeadLockTest > main > r2 > Runnable > run > try > synchronized (t2) >

Threads ×	
Name	State
system	
main	
Thread-0	On Monitor
Thread-1	On Monitor
DestroyJavaVM	Running
Reference Handler	Waiting
Finalizer	Waiting
Signal Dispatcher	Running
Attach Listener	Running

Output ×	Search Results	Variables	Breakpoints	Watches
<div>DeadLockTest (debug) × Debugger Console ×</div> <pre> debug: r1 has locked t1, now going to sleep r2 has aquire the lock of t2 now going to sleep r1 has awake , now going to aquire lock for t2 r2 is awake , now going to aquire the lock from t1 </pre>				

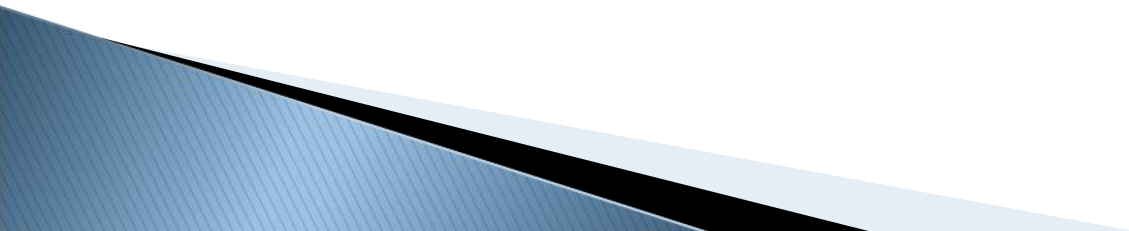
Zagłodzenie

Gdy jeden proces nigdy nie będzie miał szansy wykonania się,

wtedy mówimy o zagłodzeniu procesu (ang. *starvation*)

Projektowana architektura systemu, powinna dopuścić do wykonania również wątków o najniższym priorytecie.

Jeśli algorytm szeregowania procesów jest zły, wtedy istnieje szansa, że wątek zostanie zagłodzony, ponieważ zawsze będą wykonywane inne procesy np. o wyższym priorytecie.



Livelock –specjalny przypadek zagłodzenia

Występuje, gdy obydwa procesy, aby uniknąć deadlock'a zatrzymują wykonywanie kodu, aby dać szansę innym wątkom na wykonanie się.

Dla przykładu, gdy dwie osoby na wąskim korytarzu wybierają ciągle tą samą trasę, aby siebie minąć, ale niestety kończą stale na wzajemnej blokadzie.

Livelock wydaje się podobny do deadlock, bo efekt jest taki sam.

W Livelock stan procesu się jednak zmienia, a w deadlock, pozostaje ciągle taki sam.

W osoby na korytarzu ciągle zmieniają swoją pozycję (np. lewo, prawo).

