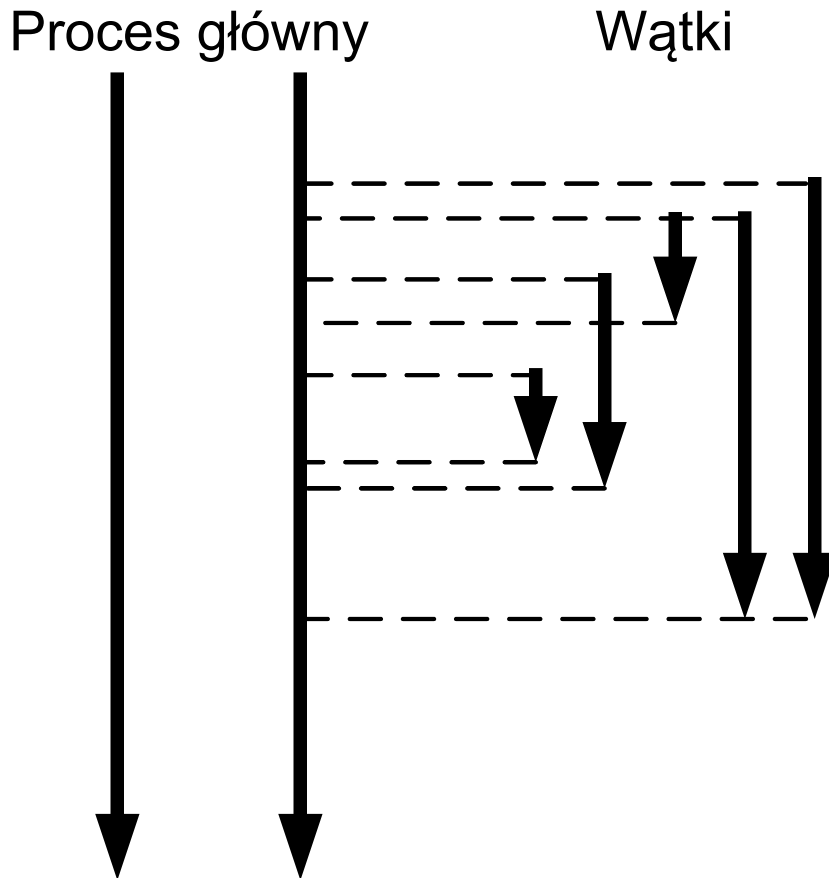
Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

Wielozadaniowość w systemie Microsoft Windows

mgr inż. Tomasz Jaworski
tjaworski@kis.p.lodz.pl
http://tjaworski.kis.p.lodz.pl/

Katedra
Informatyki
Stosowanej

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, Linux)
- **Brak wywłaszczenia** – programy same muszą „zwalniać” procesor. Wadliwa aplikacja może zawiesić cały system (np. Win31)

Katedra
Informatyki
Stosowanej

Tworzenie nowego wątku

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

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* - argument przekazywany do nowego wątku
- *dwCreationFlags* - określa początkowy status procesu, np. **CREATE_SUSPENDED**
- *lpThreadId* - wskaźnik do identyfikatora 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 wykonywane w oddzielny 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 (`th1p`) 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;
}
```

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

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

LPVOID *arg* – jest wartością argumentu przekazaną w wywołaniu funkcji **CreateThread**. W przypadku powyższego przykładu jest to liczba typu int.

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 funkcja **CreateThread**.

Uwaga! Pamięć zaalokowana przy pomocy funkcji `malloc`, `calloc` oraz operatorów `new` i `new[]` nie zostanie zwolniona podczas zwalniania pamięci wątku.

Wznawianie/Zawieszanie wykonywania wątku

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

Funkcja **ResumeThread** wznawia 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);
```

Ponieważ procedura będąca uruchamia w oddzielny wątku 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 oczekiwania za 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);
```

Problem z dostępem do zasobów

Kod do wykonania:

```
x = cnt;
x = x + 1;
cnt = x;
```

Stan początkowy:

```
cnt = 1;
```

Ilość wątków: 2

Problem z dostępem do zasobów

Kod do wykonania:

```
x = cnt;
x = x + 1;
cnt = x;
```

Stan początkowy:

```
cnt = 1;
```

Ilość wątków: 2

Wątek	Operacja	cnt	x_1	x_2
		1	x	x
1	x_1 = cnt	1	1	x
1	$x_1 = x_1 + 1$	1	2	x
2	x_2 = cnt	1	2	1
1	cnt = x_1	2	2	1
2	$x_2 = x_2 + 1$	2	2	2
2	cnt = x_2	**2**	2	2

Katedra
Informatyki
Stosowanej

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
{
    x = cnt;
    x = x + 1;
    cnt = x;
}
```

Sekcja krytyczna

```
CRITICAL_SECTION cs;
//  .......
DWORD WINAPI my_thread(LPVOID arg)
{
    EnterCriticalSection(&cs);
    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 kod.

Semafory można stosować do wyzwalania zdarzeń, synchronizacji wątków, itd…

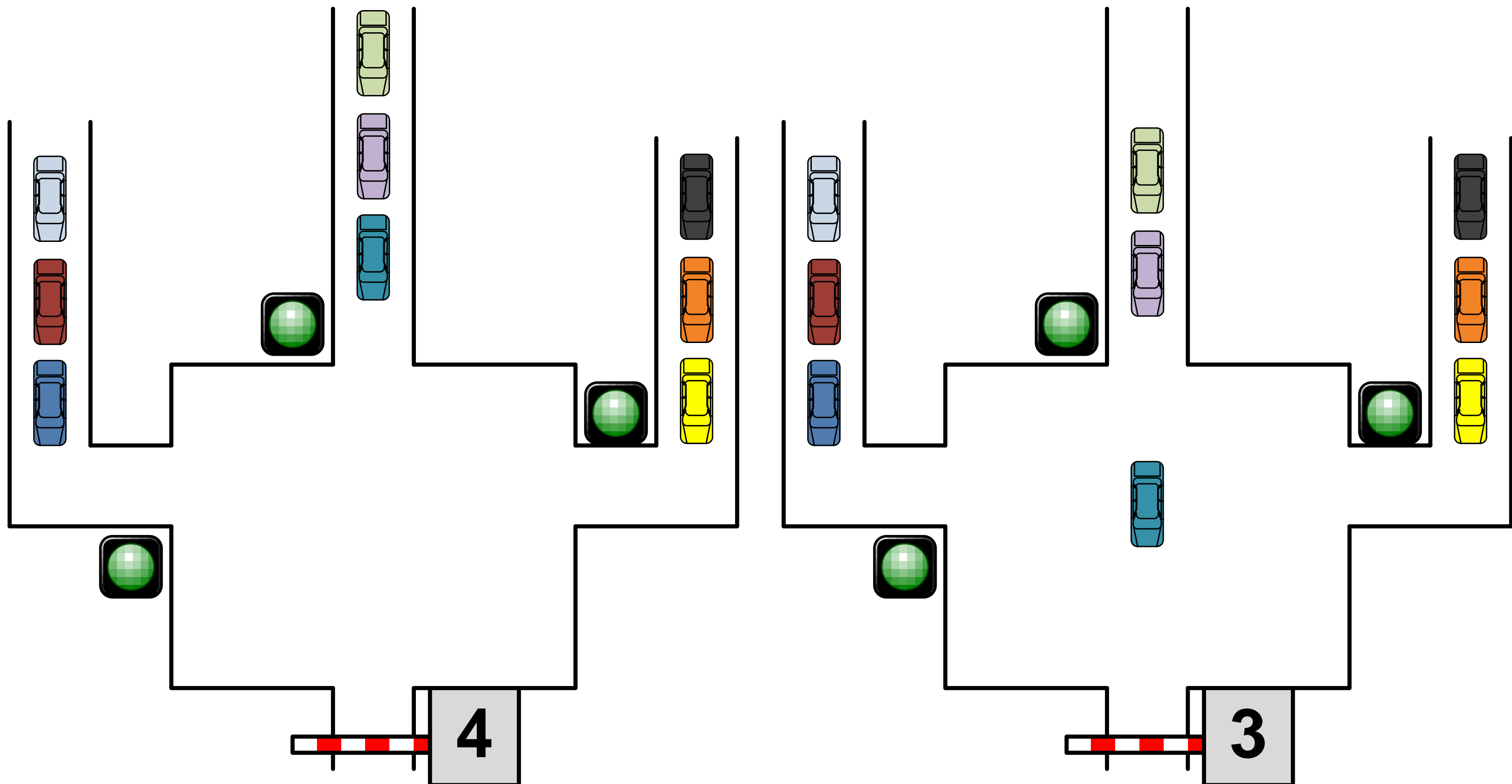
Z każdym semaforem skojarzony jest licznik.
Jeśli wartość licznika > 0, to semafor **otwiera możliwość dostępu (sygnalizuje)**.
Jeśli wartość licznika == 0, to semafor **zamyka drogę dostępu (nie sygnalizuje)**

Dostępne są dwa typy instrukcji: `wait` i `signal`.

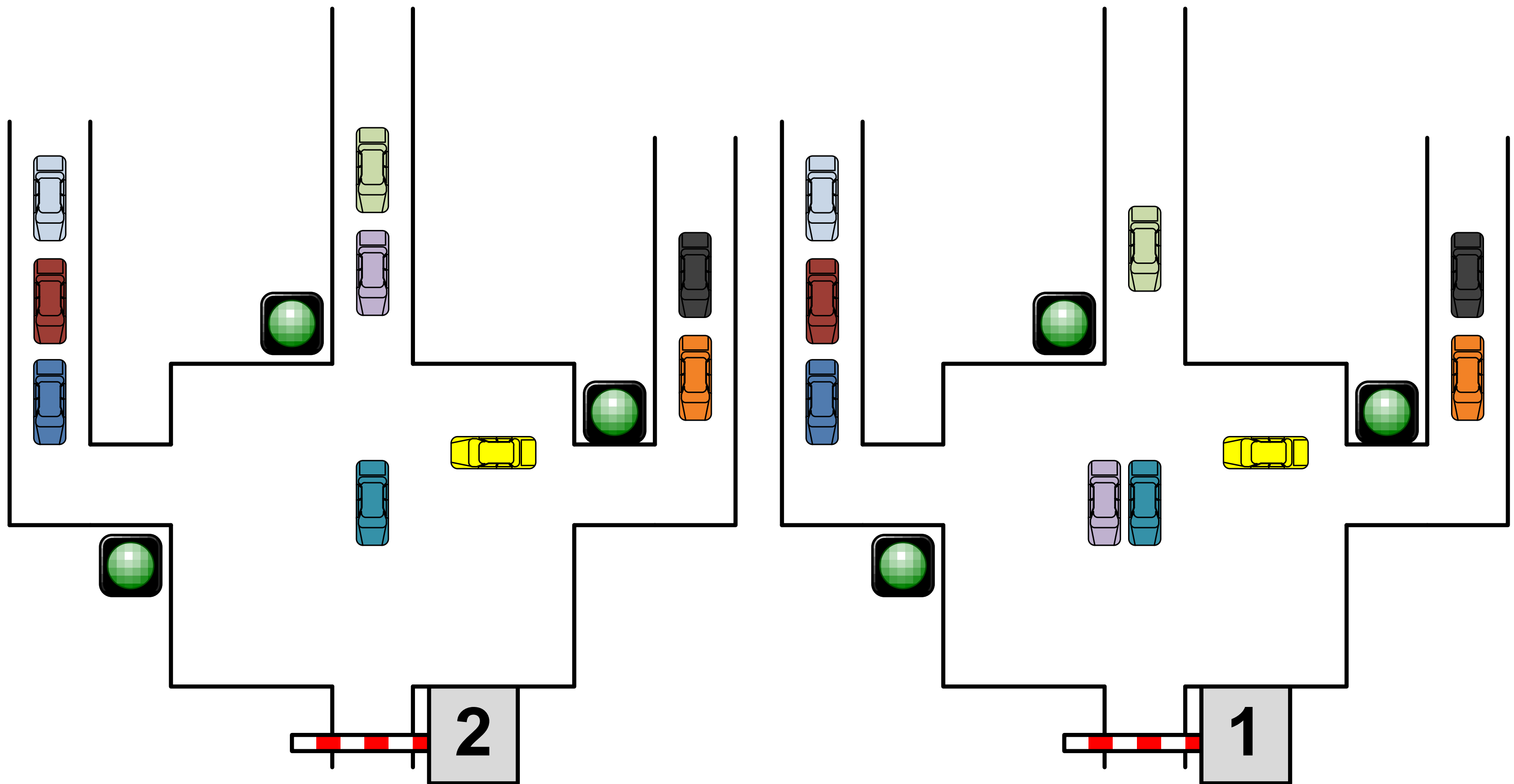
`Wait` monitoruje semafor i jeśli ten **sygnalizuje**, to wait dopuszcza wykonanie. Jeśli semafor **nie sygnalizuje**, to wątek jest wstrzymywany (aż do ponownej **sygnalizacji**).

`Signal` zawsze **uruchamia sygnalizację** (poprzez zwiększenie licznika).

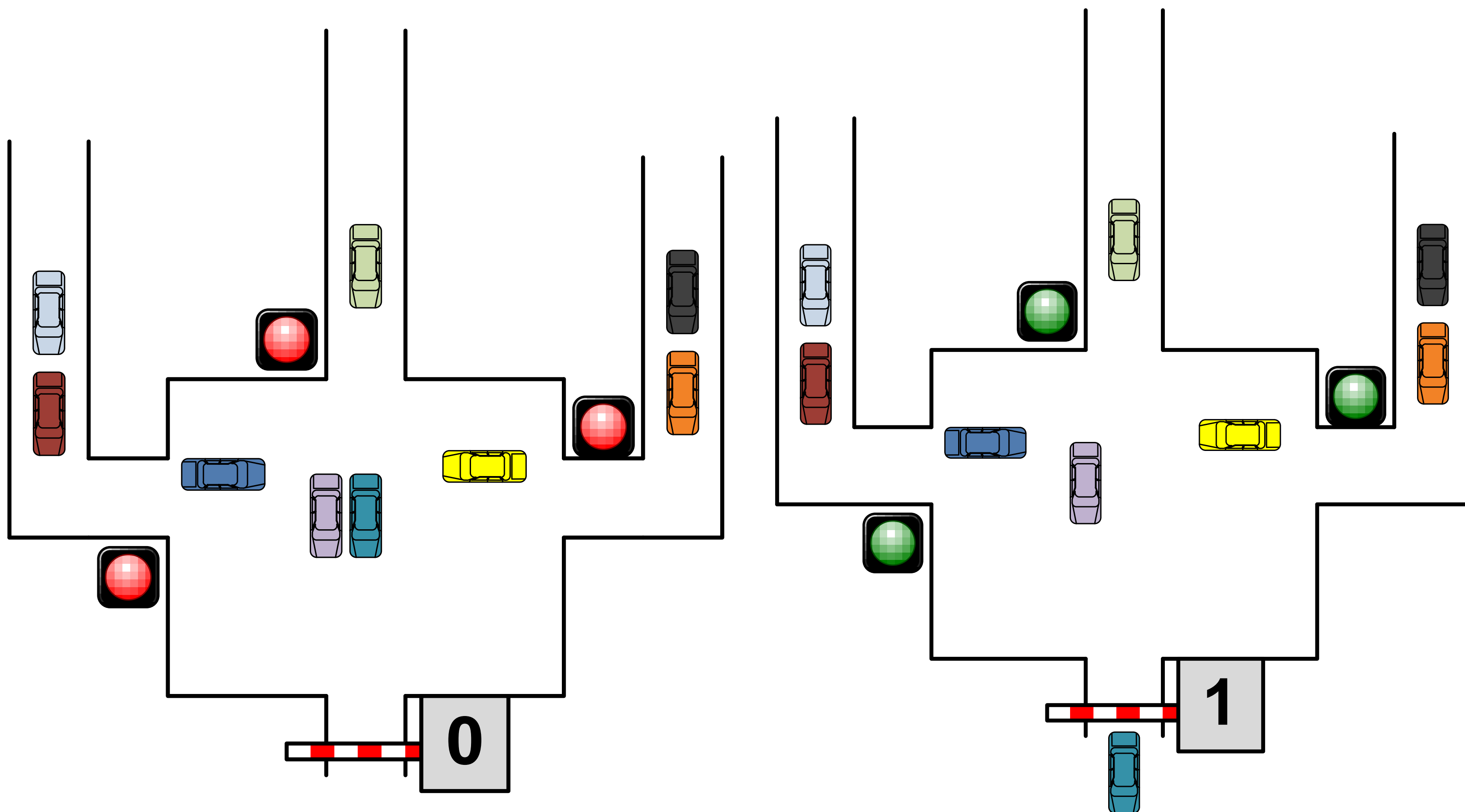
Semafory – bardziej życiowy przykład



Semafory – bardziej życiowy przykład

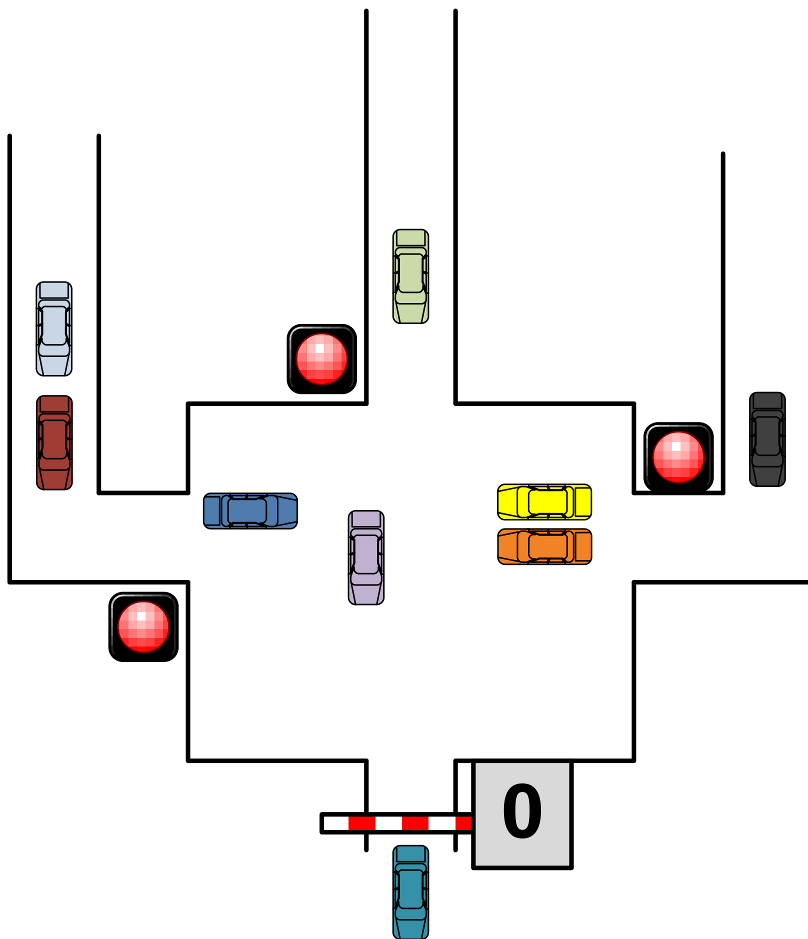


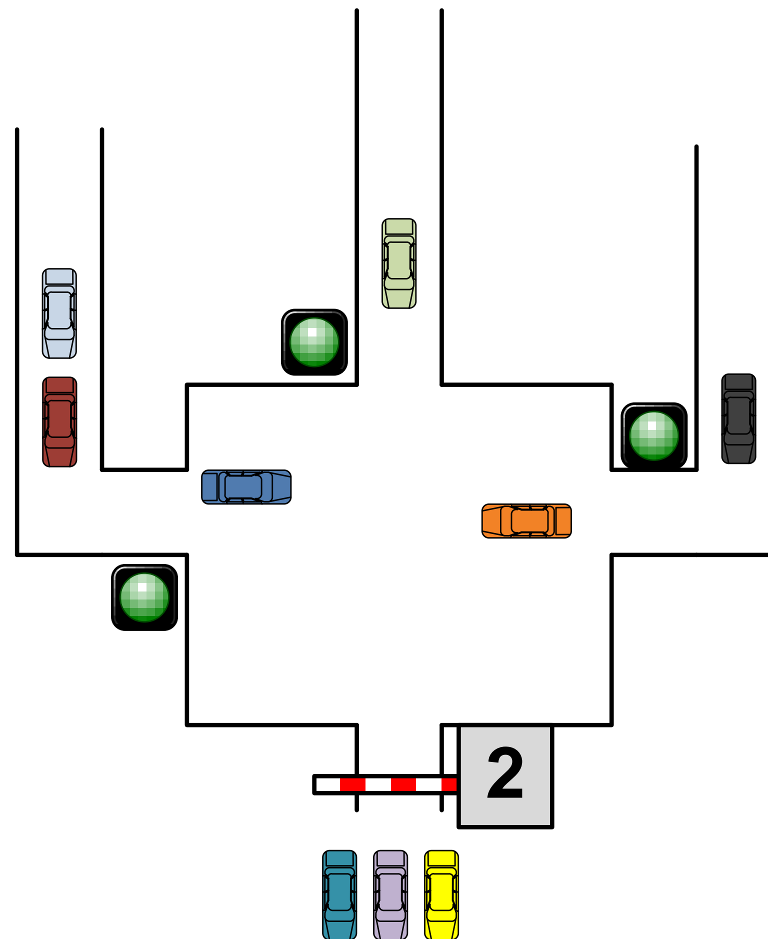
Semafory – bardziej życiowy przykład



Katedra
Informatyki
Stosowanej

Semafory – bardziej życiowy przykład





Semafory – 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);
```

Katedra Informatyki Stosowanej

Semafory – 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.

Dziękuję za uwagę!