EWSIE
EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA

Programowanie współbieżne i rozproszone

WYKŁAD 6

Jan Kazimirski

Programowanie wielowątkowe

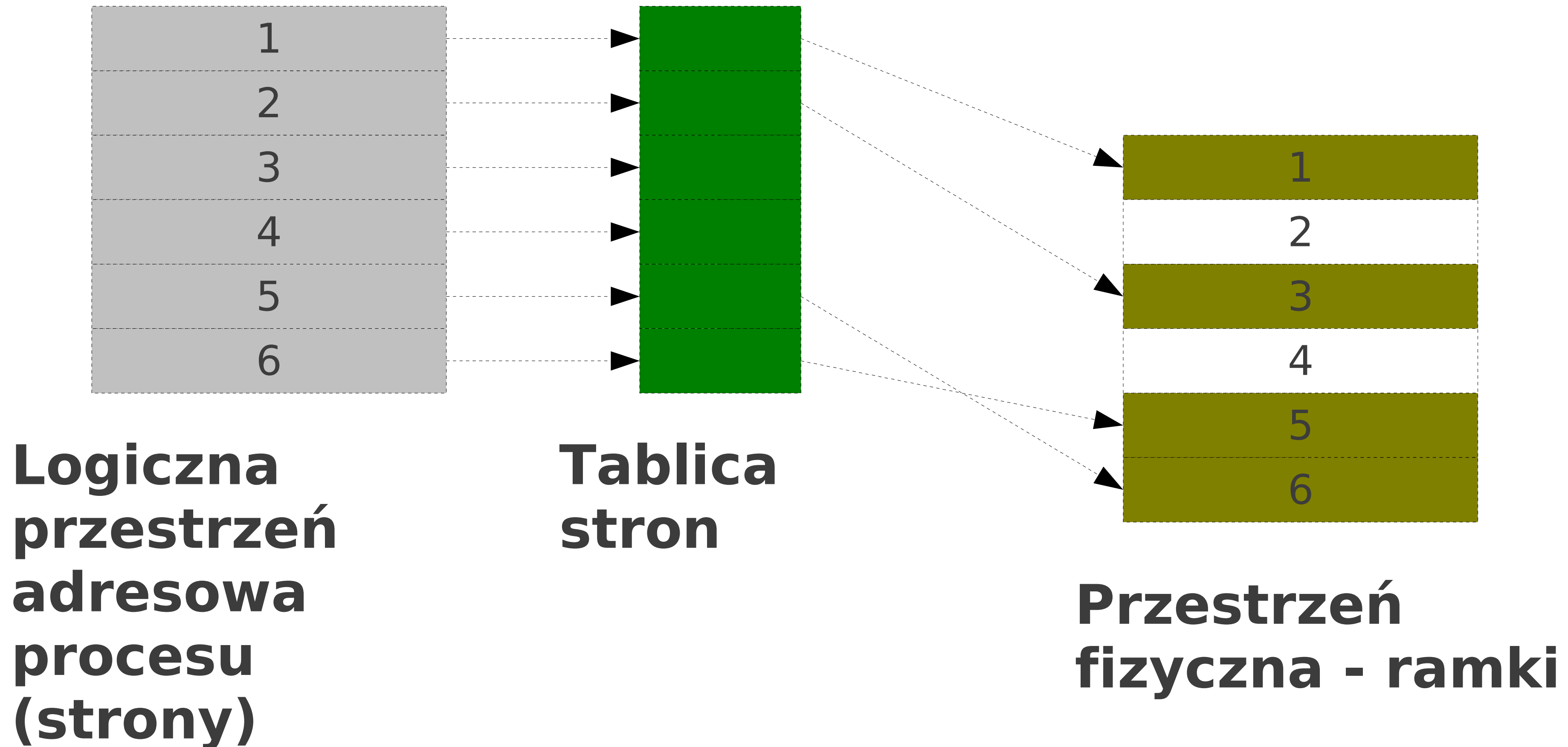
Wielowątkowość

Technika pozwalająca na jednoczesne wykonywanie kilku czynności w ramach jednego programu.

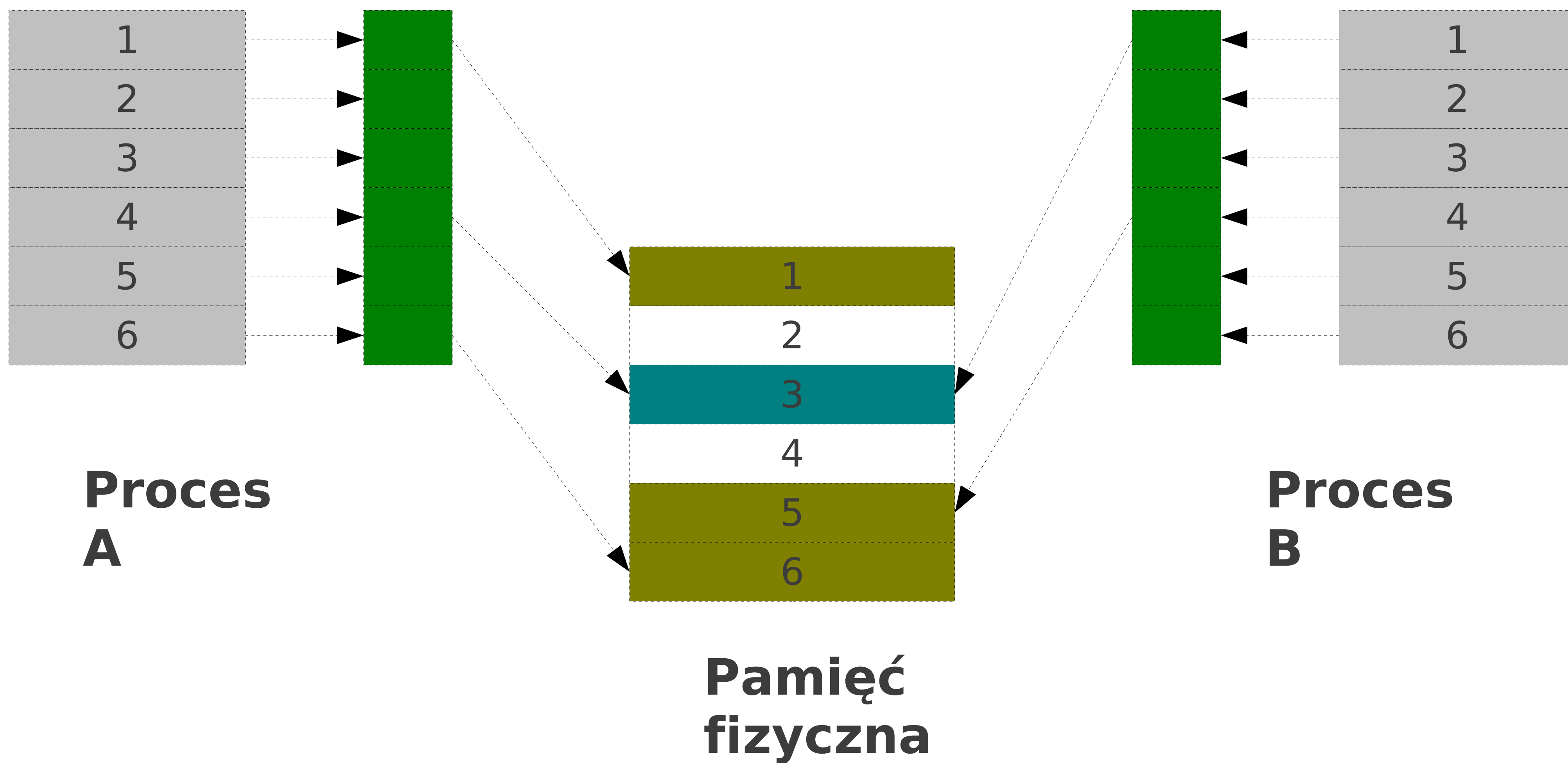
Wielowątkowość a wielozadaniowość

- Wielozadaniowość
  - Osobne procesy pod kontrolą systemu operacyjnego
  - Pełna izolacja procesów
  - Trudna komunikacja między procesami
- Wielowątkowość
  - Ciągi instrukcji w ramach jednego procesu
  - Współdzielenie zasobów
  - Łatwa komunikacja
  - Konieczność synchronizacji

Procesy i pamięć wirtualna



Procesy i pamięć wirtualna c.d.

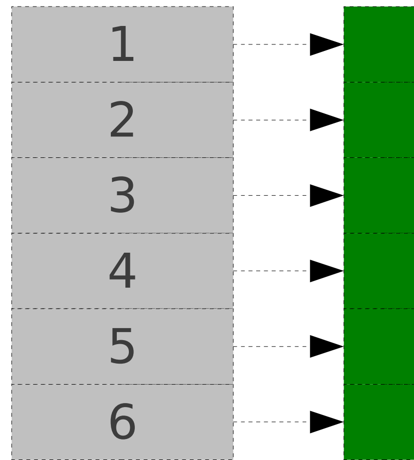


Uruchomienie procesu

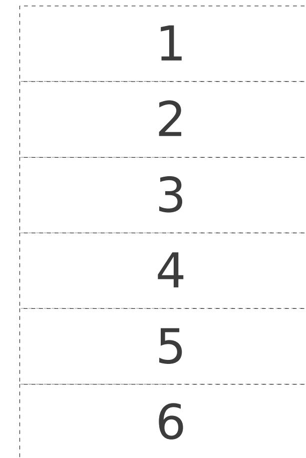
Utworzenie logicznej przestrzeni adresowej dla procesu.

Utworzenie tablicy stron

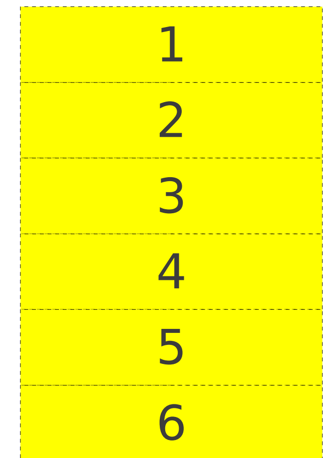
Pomimo „posiadania” swojej przestrzeni adresowej proces nie zajmuje jeszcze ani jednego bajta pamięci fizycznej poza obszarem tablicy stron



Strony logiczne



Ramki fizyczne

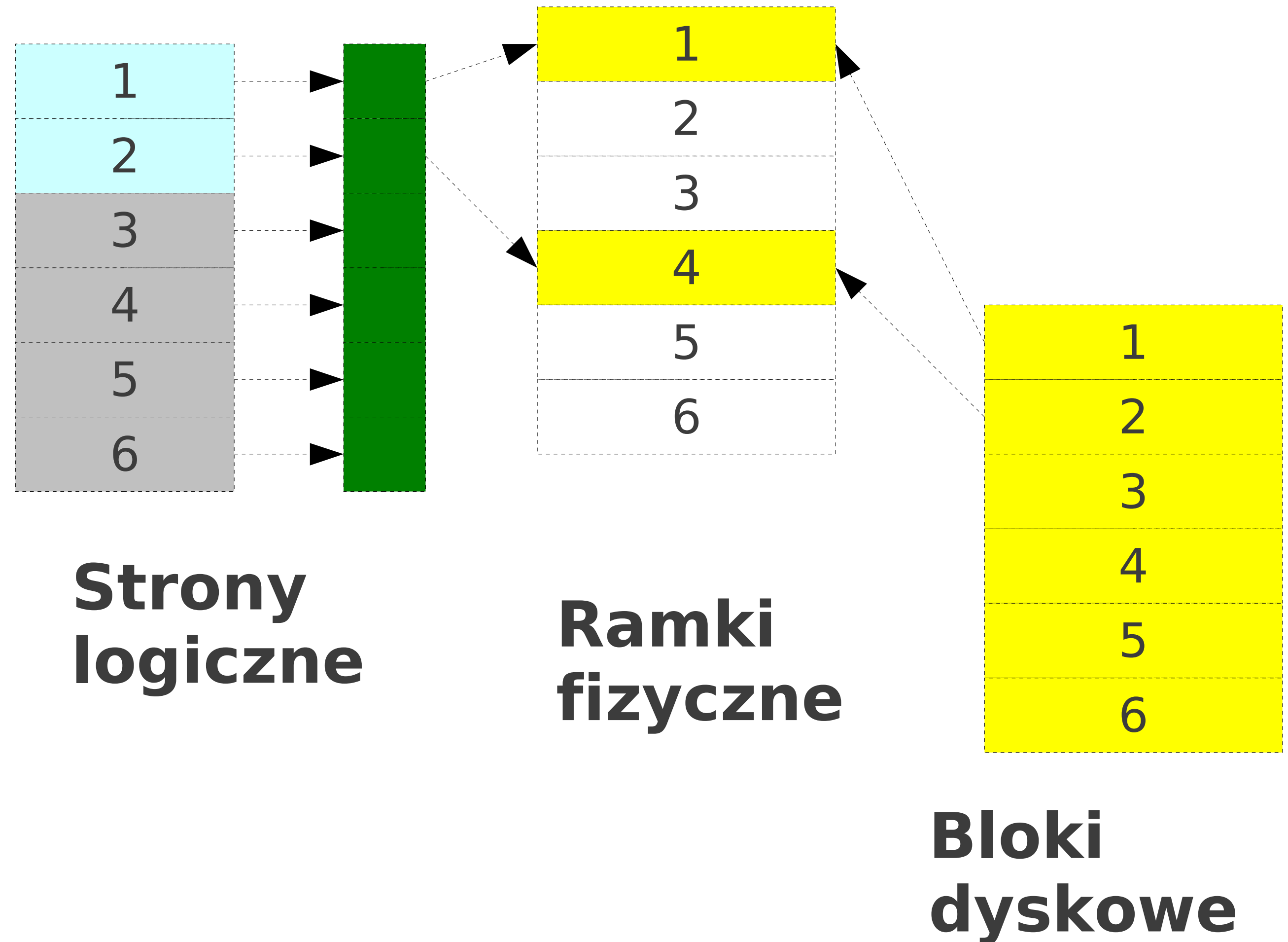


Bloki dyskowe

Uruchomienie procesu c.d.

System przydziela fizyczne ramki do kilku pierwszych stron programu i ładuje fragment programu z dysku.

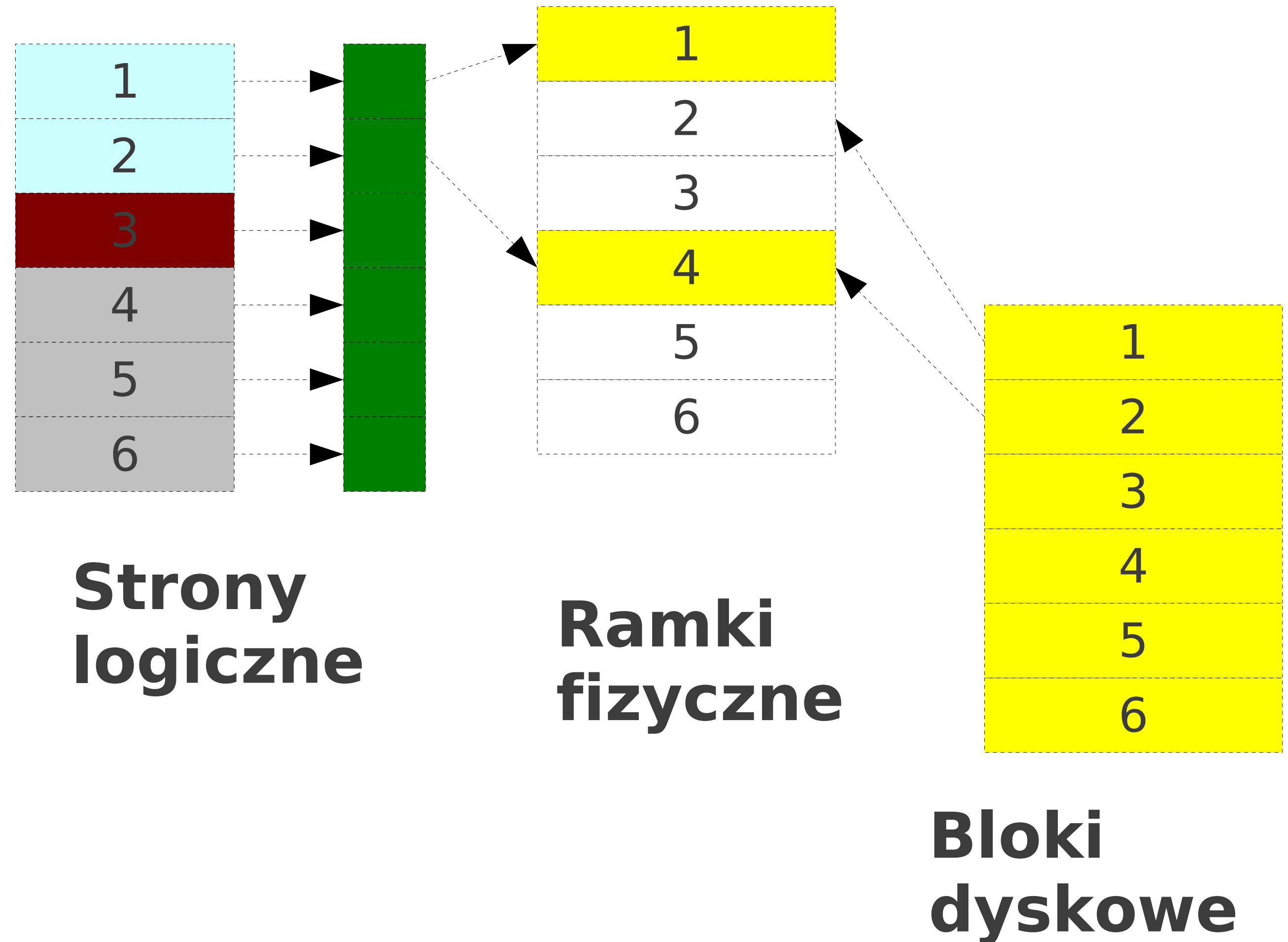
Rozpoczyna się wykonywanie programu



Wykonywanie procesu

W trakcie wykonania proces może sięgnąć do strony, która nie została załadowana do pamięci.

Następuje „Błąd braku strony” - **PAGE FAULT**

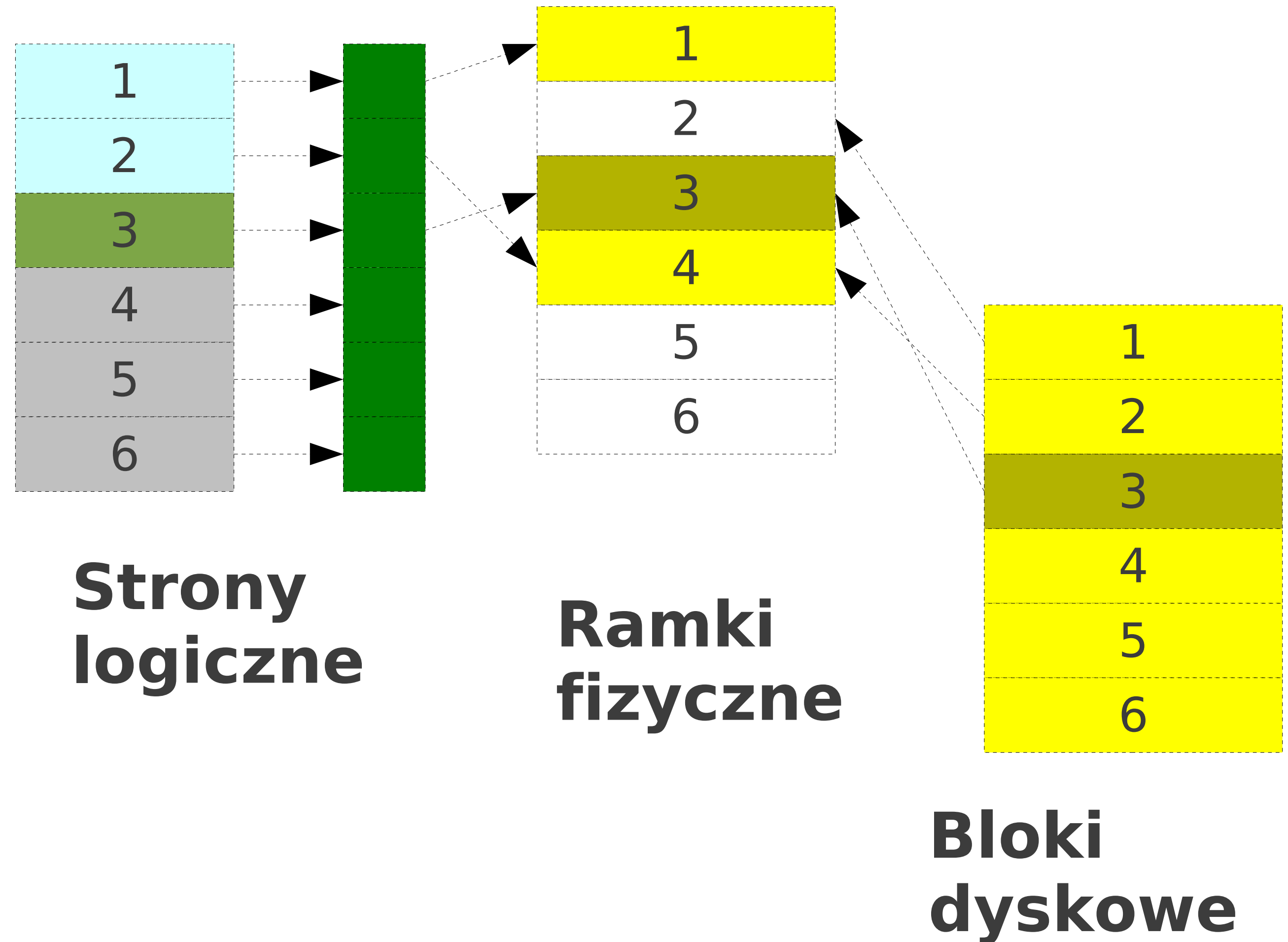


Wykonywanie procesu c.d.

Po wystąpieniu błędu braku strony proces zostaje wstrzymany.

System operacyjny ładuje z dysku odpowiedni fragment i mapuje go na odpowiednią stronę

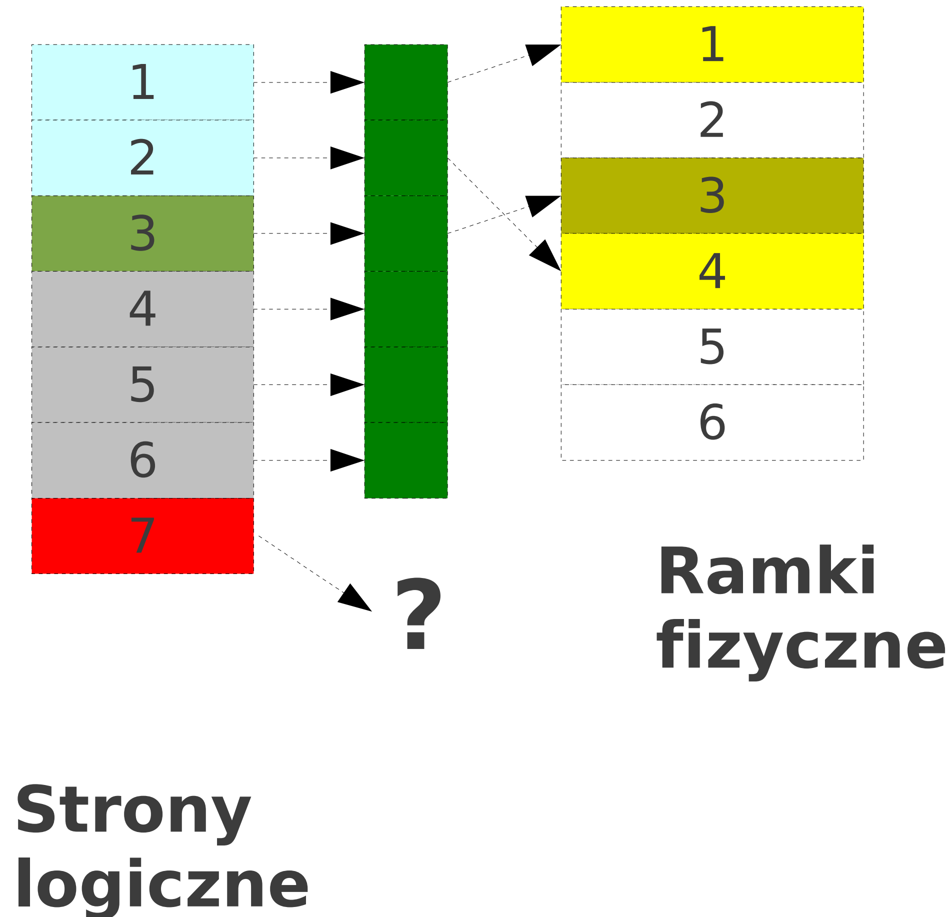
Po zakończeniu operacji dyskowej proces jest budzony i kontynuowany



Wykonywanie procesu c.d.

Mechanizm stronicowania pozwala zapewnić ochronę przestrzeni adresowej procesów

Jeżeli proces próbuje uzyskać dostęp do pamięci poza swoją przestrzenią adresową, to następuje **błąd ochrony pamięci.**



Tworzenie procesu

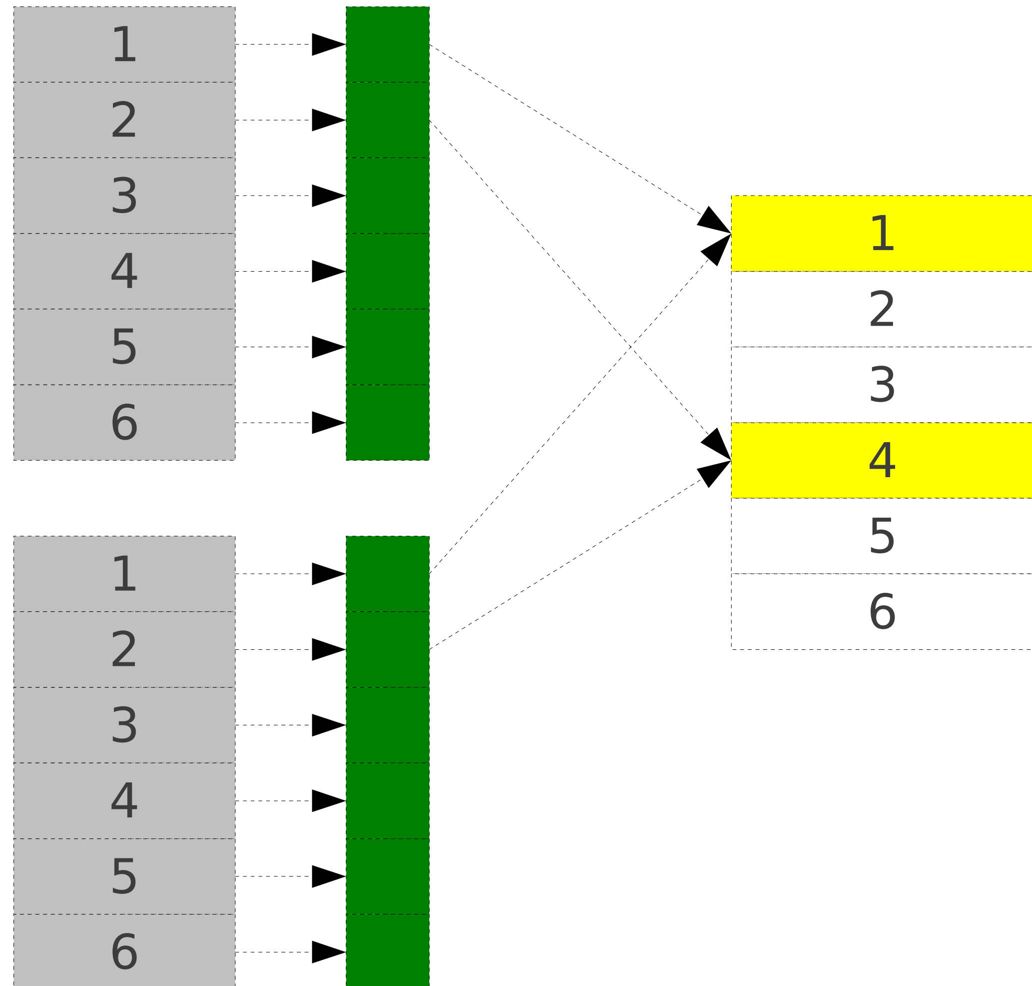
Tworzenie nowego procesu w systemach Unix/Linux to tzw. fork-and-exec.

W pierwszym etapie Wywoływana jest funkcja systemowa fork.

Tworzona jest kopia przestrzeni adresowej procesu rodzica.

Przestrzenie adresowe obu procesów mapowane są na te same ramki pamięci fizycznej

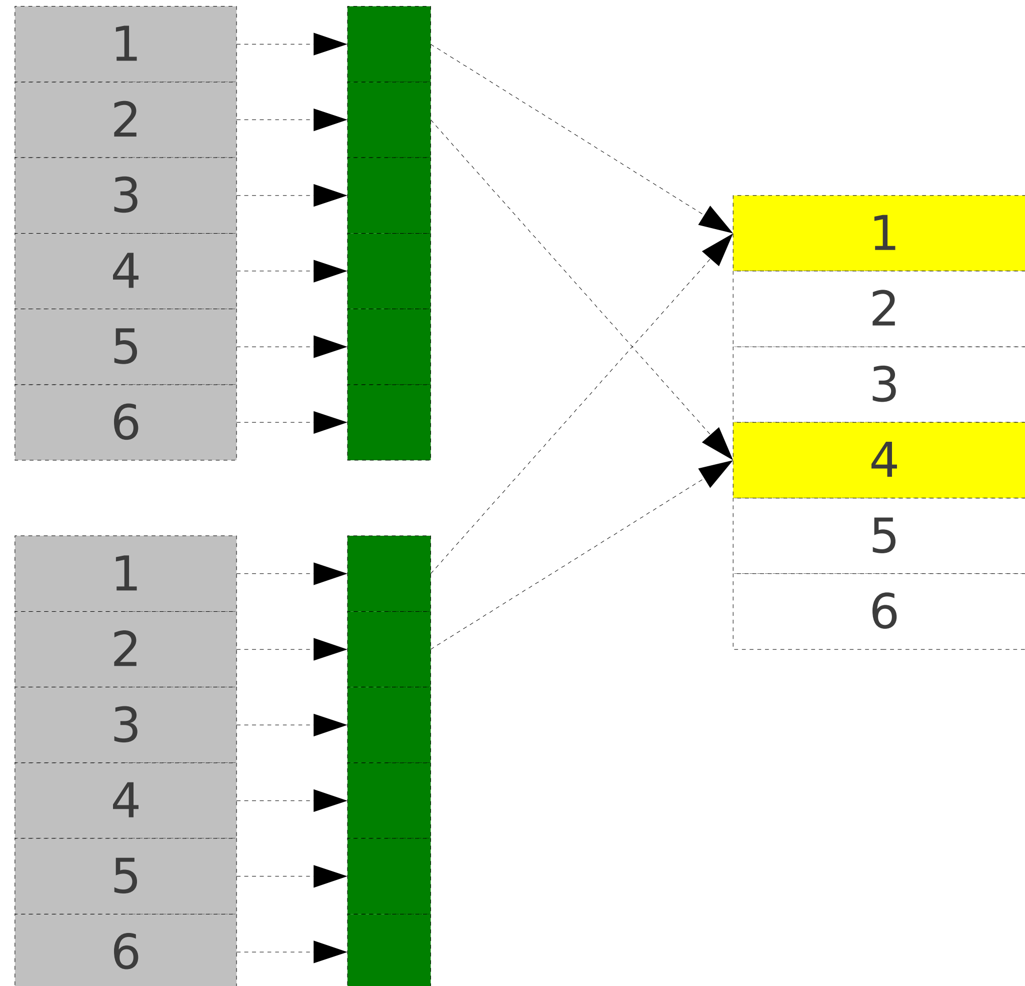
Nowy proces poza tablicą stron nie zajmuje w ogóle pamięci fizycznej



Tworzenie procesu c.d.

W trakcie wykonania oba procesy mogą korzystać z tej samej ramki pamięci jeżeli realizują na niej tylko operacje odczytu

Każdy proces widzi tę ramkę jako stronę swojej własnej przestrzeni adresowej, ale fizycznie zajmują one tylko jedną ramkę w pamięci



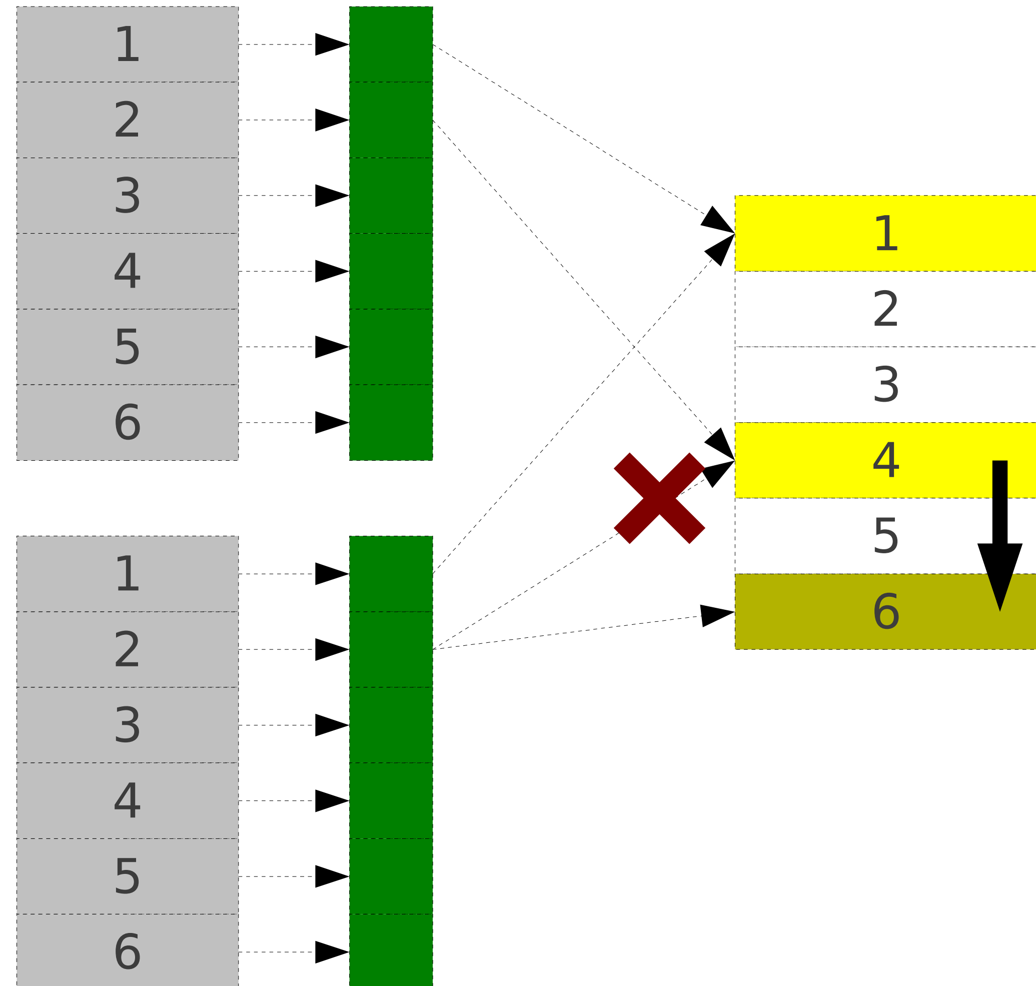
Tworzenie procesu c.d.

Specjalnej uwagi wymagają jedynie operacje zapisu na danej stronie pamięci.

Próba modyfikacji danych ze „wspólnej” strony przez jeden z procesów powoduje wstrzymanie procesu.

System operacyjny przydziela zatrzymanemu procesowi dodatkową ramkę i kopiuje zawartość z ramki „wspólnej” aktualizując tablicę stron.

Proces zostaje obudzony i kontynuuje zapis w swoim „prywatnym” obszarze pamięci.



Przełączanie procesów i wątków

- **Przełączanie procesów**
  - Konieczność zachowania stanu procesu – rejestry, wskaźniki do tablic stron, list deskryptorów plików itd..
- **Przełączanie wątków**
  - Konieczność zachowania stanu rejestrów
  - Niezmienione struktury kontrolne pamięci wirtualnej, deskryptorów plików, uprawnienia, itd..

Wątki a procesy lekkie

- **Procesy lekkie (LWP)**
  - Implementowane w przestrzeni jądra
  - Wykorzystują mechanizmy jądra
- **Wątki**
  - Implementowane w przestrzeni użytkownika
  - Przenośne – wykorzystują standardowe funkcje jądra

Zalety stosowania wątków

- **Wykorzystanie CPU**
  - Komputery wieloprocesorowe (SMP)
  - Procesory wielordzeniowe
  - Zastosowanie potoków, techniki out-of-order execution i superskalarności
- **Zwiększenie przepustowości**
  - Zablokowany wątek (np.. operacja I/O) nie blokuje całej aplikacji

Zalety stosowania wątków c.d.

- **Responsywność**
  - Osobny wątek obsługujący UI nie jest blokowany np., operacjami I/O lub obliczeniami
- **Eliminacja komunikacji IPC**
  - Komunikacja między wątkami procesu jest łatwiejsza niż między procesami

Zalety stosowania wątków c.d.

- **Oszczędność zasobów**
  - Tworzenie procesów jest kosztowne zarówno ze strony CPU, jak i pamięci.
- **Uproszczona obsługa zdarzeń**
  - Obsługa zdarzeń asynchronicznej wymaga zwykle złożonej logiki
  - Stosowanie osobnych wątków obsługujących takie zdarzenia upraszcza program

EWSIE
EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA

Zalety stosowania wątków c.d.

- **Uproszczone programy**
  - Wiele programów „z definicji” jest współbieżnych (np. serwer http, serwer bazy danych, technologie obiektów rozproszonych CORBA, DCOM)
  - Implementacja wielowątkowa jest łatwiejsza, bardziej przejrzysta, odporniejsza na błędy.

EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA EWSIE

Pułapki w stosowaniu wątków

- Brak ochrony zasobów
- Brak ochrony struktur sterujących wątków
- Niespodzianki w przypadku jednoczesnego dostępu do plików
- Niespodzianki w przypadku jednoczesnego wywoływania tych samych funkcji systemowych z tymi samymi danymi

Tworzenie procesów w C

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
        printf("START!\n");
        fork();
        printf("STOP!\n");
}
```

Tworzenie procesów w C c.d.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
        printf("START!\n");
        int pid = fork();
        if(pid==0) {
                printf("Jestem dzieckiem.\n");
                printf("Moj PID to: %d\n",getpid());
                printf("PID rodzica to: %d\n",getppid());
        } else {
                sleep(2);
                printf("Jestem rodzicem.\n");
                printf("Moj PID to: %d\n",getpid());
                printf("PID rodzica to: %d\n",getppid());
        };
}
```

Biblioteka PThreads

- Standard POSIX określający implementację wielowątkowości
- Definiuje interfejs programistyczny C do podstawowych operacji:
  - Tworzenie i kończenie wątku
  - Lokalne dane wątku
  - Mechanizmy synchronizacji między wątkami
- Dostępny dla większości systemów UNIX

PThreads – podstawowe operacje

- Identyfikator wątku – zmienna typu ***pthread_t***
- Wątek rozpoczyna się wywołaniem funkcji dostarczonej przez użytkownika.
- Funkcja wykonywana w ramach wątku ma jeden argument typu void* i zwraca wartość typu void*
- Do utworzenia wątku służy funkcja ***pthread_create***
- Funkcja ***pthread_self*** wywołana w ramach wątku zwraca jego identyfikator

PThreads – podstawowe operacje

- Funkcja ***pthread_equal*** porównuje dwa identyfikatory wątków i zwraca wartość różną od zera jeżeli dotyczą tego samego wątku, lub 0 jeśli nie.
- Funkcja **pthread_exit** kończy wykonywanie wątku.

Przykład tworzenia wątków

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <pthread.h>
04:
05: void* fun(void* msg) {
06:   printf("%s\n",(char*) msg);
07:   return NULL;
08: };
09:
10: int main() {
11:   pthread_t T1,T2;
12:   const char* msg1 = "Wątek 1";
13:   const char* msg2 = "Wątek 2";
14:   pthread_create(&T1,NULL,fun,(void*) msg1);
15:   pthread_create(&T2,NULL,fun,(void*) msg2);
16:
17:   pthread_join(T1,NULL);
18:   pthread_join(T2,NULL);
19:
20:   return 0;
21: }
```

Przykład tworzenia wątków

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <pthread.h>
04:
05: void* fun(void* msg) {
06:   printf("%s\n",(char*) msg);
07:   return NULL;
08: };
09:
10: int main() {
11:   pthread_t T1,T2;
12:   const char* msg1 = "Wątek 1";
13:   const char* msg2 = "Wątek 2";
14:   pthread_create(&T1,NULL,fun,(void*) msg1);
15:   pthread_create(&T2,NULL,fun,(void*) msg2);
16:
17:   pthread_join(T1,NULL);
18:   pthread_join(T2,NULL);
19:
20:   return 0;
21: }
```

Przykład tworzenia wątków

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <pthread.h>
04:
05: void* fun(void* msg) {
06:   printf("%s\n",(char*) msg);
07:   return NULL;
08: };
09:
10: int main() {
11:   pthread_t T1,T2;
12:   const char* msg1 = "Wątek 1";
13:   const char* msg2 = "Wątek 2";
14:   pthread_create(&T1,NULL,fun,(void*) msg1);
15:   pthread_create(&T2,NULL,fun,(void*) msg2);
16:
17:   pthread_join(T1,NULL);
18:   pthread_join(T2,NULL);
19:
20:   return 0;
21: }
```

Przykład tworzenia wątków

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <pthread.h>
04:
05: void* fun(void* msg) {
06:   printf("%s\n",(char*) msg);
07:   return NULL;
08: };
09:
10: int main() {
11:   pthread_t T1,T2;
12:   const char* msg1 = "Wątek 1";
13:   const char* msg2 = "Wątek 2";
14:   pthread_create(&T1,NULL,fun,(void*) msg1);
15:   pthread_create(&T2,NULL,fun,(void*) msg2);
16:
17:   pthread_join(T1,NULL);
18:   pthread_join(T2,NULL);
19:
20:   return 0;
21: }
```

EWSIE EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA

Przykład tworzenia wątków

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <pthread.h>
04:
05: void* fun(void* msg) {
06:   printf("%s\n",(char*) msg);
07:   return NULL;
08: };
09:
10: int main() {
11:   pthread_t T1,T2;
12:   const char* msg1 = "Wątek 1";
13:   const char* msg2 = "Wątek 2";
14:   pthread_create(&T1,NULL,fun,(void*) msg1);
15:   pthread_create(&T2,NULL,fun,(void*) msg2);
16:
17:   pthread_join(T1,NULL);
18:   pthread_join(T2,NULL);
19:
20:   return 0;
21: }
```

Funkcja pthread_create

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

Funkcja tworzy nowy wątek.

Argumenty

thread – wskaźnik do identyfikatora wątku

attr – atrybuty wątku (wartość NULL oznacza domyślne atrybuty)

start_routine – wskaźnik do funkcji startowej wątku

arg – wskaźnik argumentu przekazywany do funkcji startowej

Funkcja zwraca 0 w przypadku powodzenia lub kod błędu w przeciwnym wypadku

Funkcja pthread_join

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

Funkcja czeka na zakończenie podanego wątku

Argumenty

thread – wskaźnik do identyfikatora wątku

retval – jeżeli różna od NULL to będzie ustawiony na status powrotu z zakończonego wątku

Funkcja zwraca 0 w przypadku powodzenia lub kod błędu w przeciwnym wypadku

Przerwanie działania wątku

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <unistd.h>
04: #include <pthread.h>
05:
06: void* fun(void* msg) {
07:   sleep(5);
08:   printf("%s\n",(char*) msg);
09:   return NULL;
10: };
11:
12: int main() {
13:   pthread_t T1;
14:   const char* msg1 = "Wątek 1";
15:   pthread_create(&T1,NULL,fun,(void*) msg1);
16:
17:   return 0;
18: }
```

Zakończenie wątku

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #include <unistd.h>
04: #include <pthread.h>
05:
06: int ret=0;
07:
08: void* fun(void* msg) {
09:   sleep(5);
10:   printf("%s\n",(char*) msg);
11:   return NULL;
12: };
13:
14: int main() {
15:   pthread_t T1;
16:   const char* msg1 = "Wątek 1";
17:   pthread_create(&T1,NULL,fun,(void*) msg1);
18:
19:   pthread_exit(&ret);
20: }
```

Funkcja pthread_exit

void pthread_exit(void *retval);

Funkcja kończy wątek

Argument

retval – status powrotu z wątku

Funkcja nigdy nie wraca

Powrót z funkcji startowej danego wątku jest jednoznaczny z wywołaniem funkcji pthread_exit.

Jeżeli funkcja main kończy się wywołaniem funkcji pthread_exit to program nie zakończy się przed zakończeniem wykonywania wszystkich wątków.

Zmienna ze statusem powrotu nie może być zmienną lokalną.

Czas życia wątku

- W momencie uruchomienia procesu tworzony jest „wątek początkowy”.
- Kolejne wątki tworzone są funkcją pthread_create.
- Nowy wątek znajduje się w stanie „**gotowy do wykonania**”
- Gotowy do wykonania wątek może przejść do stanu „**wykonywany**”.
- Wątek może przejść do stanu „**zablokowany**” jeżeli oczekuje na zasoby.
- Wątek może zostać zakończony lub anulowany przechodząc do stanu „**zakończony**”

Tworzenie wątku

- Funkcja **pthread_create** nie zapewnia synchronizacji z tworzonym wątkiem.
- W momencie powrotu z tej funkcji tworzony wątek może być ciągle w stanie gotowy, ale również może być już zakończony.
- Zachowanie to może powodować trudne do wykrycia (niedeterministyczne) błędy

Wątki i funkcja main

- Wykonywanie wątku zaczyna się od wywołania jego funkcji startowej, która może później wywoływać inne funkcje.
- Uruchomienie wątku jest analogiczne do uruchomienia programu wywołaniem funkcji main.
- Różnice:
  - Różne argumenty funkcji main (argc,argv) i funkcji startowej wątku (void*).
  - Zakończenie funkcji main kończy program.
  - Wątki mogą mieć ograniczony rozmiar stosu.

Wykonywanie wątku

- Stan **gotowy** wątku oznacza że oczekuje on na CPU.
- Wątek przechodzi do stanu **wykonywany** gdy zostanie do niego przydzielony CPU.
- W systemie z 1 CPU tylko jeden wątek może być w stanie **wykonywany**.
- W systemie z N CPU w stanie **wykonywany** może być N wątków jednocześnie.
- Wątek może przejść ponownie do stanu **gotowy** jeżeli wykorzysta już swój kwant czasu CPU.

Blokowanie wątku

- Wątek może przejść do stanu zablokowany:
  - Gdy próbuje ustawić blokadę na zablokowanym semaforze (mutex)
  - Gdy czeka na zmienną warunku
  - Gdy oczekuje na sygnał (sigwait)
  - Gdy oczekuje na operację I/O lub wygenerował błąd braku strony
  - Gdy oczekuje na zakończenie innego wątku

Zakończenie wątku

- Wątek kończy się gdy:
  - Nastąpi powrót z jego funkcji startowej
  - Wywoła funkcję **pthread_exit**
  - Zostanie anulowany funkcją **pthread_cancel**
  - Zakończył się proces
- Zakończony wątek zależnie od jego atrybutów może zostać usunięty (zwolnienie zasobów), albo pozostaje w stanie **zakończony** (wątek „zombie”).

Wątek odłączony

- Zakończony wątek nie może być usuwany gdyż inny wątek może chcieć uzyskać od niego status powrotu.
- Zakończone wątki ciągle zajmują zasoby. Aby uniknąć tego problemu trzeba zmienić stan wątku na odłączony.
- Wątek można utworzyć od razu jako odłączony, lub odłączyć go później funkcją **pthread_detach**.
- Odłączony wątek zaraz po zakończeniu jest usuwany i zasoby są zwalniane.

Funkcja pthread_detach

int pthread_detach(pthread_t thread)

Funkcja odłącza podany wątek. Jego zasoby będą zwolnione od razu po zakończeniu. Może być wywołana tylko raz dla danego wątku.

Argument

thread – identyfikator odłączanego wątku

Wartość zwracana

Zwraca 0 jeżeli operacja zakończyła się poprawnie, lub kod błędu.

Atrybuty wątku

- W momencie tworzenia wątku możemy przekazać mu atrybuty. Wartość NULL powoduje użycie domyślnych wartości.
- Atrybuty wątku przechowywane są w strukturze typu **pthread_attr_t**.
- Funkcja **pthread_attr_ini** inicjuje wartości struktury atrybutów. Po zainicjowaniu struktura atrybutów może być użyta do tworzenia wątków.
- Funkcja **pthread_attr_destroy** usuwa dane atrybutów wątku (struktura może być wtedy ponownie zainicjowana)

Funkcja pthread_attr_init

int pthread_attr_init(pthread_attr_t *attr)

Funkcja inicjuje strukturę atrybutów wątku wartościami domyślnymi. Struktura ta może być później zmodyfikowana i użyta do tworzenia wątków.

Argumenty

attr – wskaźnik do struktury z atrybutami wątku

Wartość zwracana

Teoretycznie może zwrócić kod błędu, implementacja linuxowa zawsze zwraca 0 (poprawne zakończenie).

Funkcja pthread_attr_destroy

int pthread_attr_destroy(pthread_attr_t *attr)

Funkcja usuwa dane atrybutów wątku.

Argumenty

attr - wskaźnik do struktury z atrybutami wątku

Wartość zwracana

Zwraca 0 jeżeli operacja zakończyła się poprawnie, lub kod błędu.

Modyfikacja atrybutów wątku

- Dokładna definicja struktury z atrybutami wątku zależy od implementacji, dane nie powinny być więc modyfikowane ręcznie.
- Biblioteka udostępnia szereg funkcji do modyfikacji poszczególnych atrybutów m.in.:
  - Odłączania wątku
  - Sposobu dziedziczenia parametrów szeregowania
  - Określania polityki szeregowania wątku
  - Ustawiania parametrów stosu wątku

Synchronizacja wątków

- Wątki operują na współdzielonych danych. Jednoczesny dostęp do tych danych wymaga mechanizmów synchronizacji.
- PThreads wykorzystuje dwa podstawowe metody synchronizacji:
  - Semafory typu mutex
  - Zmienne warunku

EWSIE
EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA

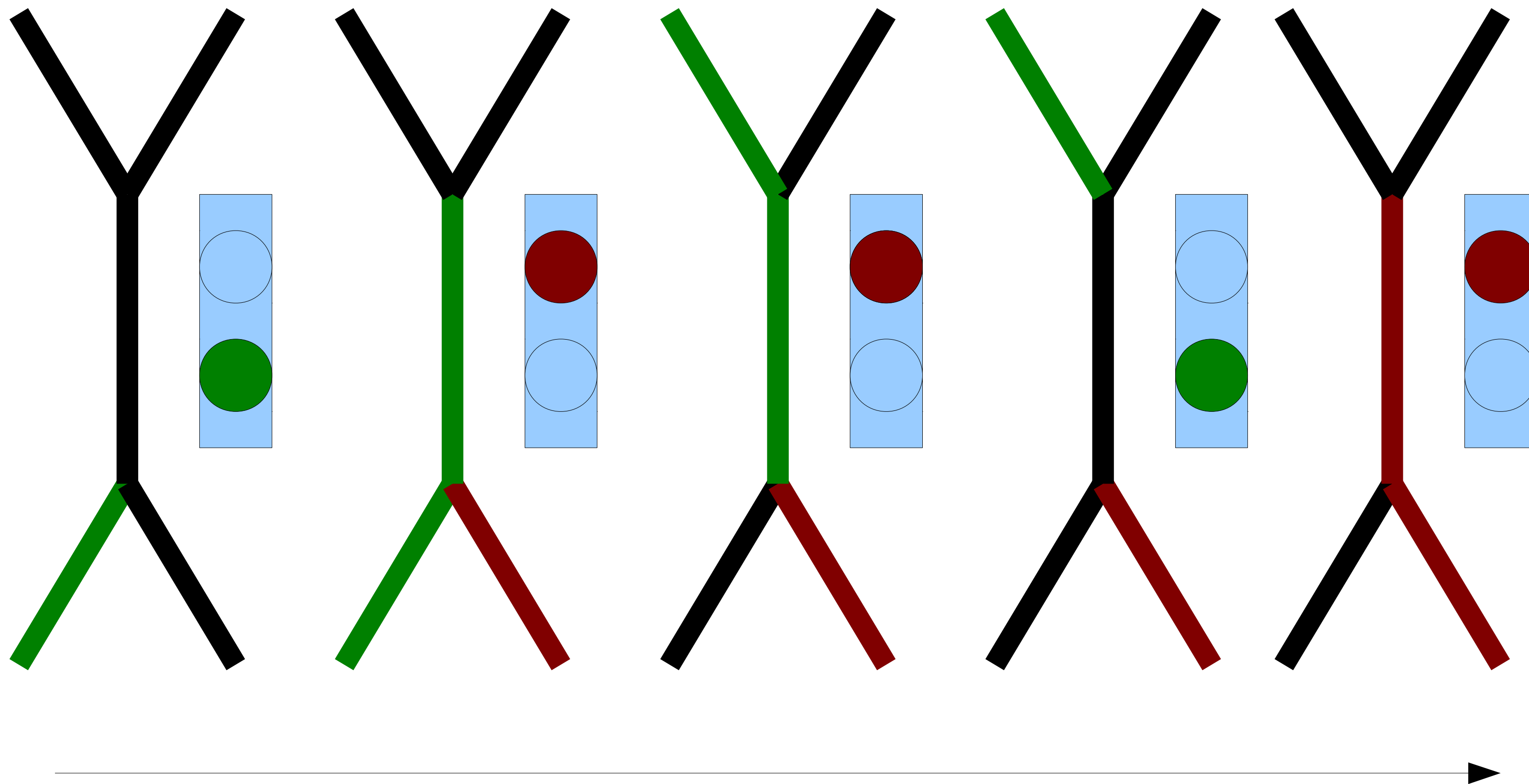
Semafor

- Zmienna całkowita ze zdefiniowanymi operacjami:
  - inicjacja liczbą dodatnią
  - **semWait** - operacja zmniejszenia wartości, wartość ujemna blokuje operację
  - **semSignal** - operacja zwiększenia wartości, jeżeli jest mniejsza lub równa 0, czekający proces jest odblokowany

Mutex

- Semafor binarny (mutex)
  - można go inicjalizować tylko na 0 lub 1
  - **semWait** – testuje semafor, wartość 0 blokuje proces, wartość jeden zmieniana jest na zero
  - **semSignal** – odblokowuje zablokowany proces lub ustawia wartość na 1

Mutex c.d.



Tworzenie i usuwanie mutexu

- Mutex reprezentuje zmienna typu **pthread_mutex_t**
- Mutex z domyślnymi atrybutami można utworzyć statycznie używając makra **PTHREAD_MUTEX_INITIALIZER**
- Mutex można też utworzyć dynamicznie za pomocą funkcji **pthread_mutex_init**
- Po wykorzystaniu mutex można usunąć funkcją **pthread_mutex_destroy**

Funkcja pthread_mutex_init

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

Funkcja inicjalizuje semafor typu mutex

Atrybuty

mutex – wskaźnik do zmiennej reprezentującej mutex

mutexattr – atrybuty mutexu

Wartość zwracana

Zawsze zwraca 0.

Funkcja pthread_mutex_destroy

int pthread_mutex_destroy(pthread_mutex_t *mutex);

Funkcja usuwa podany mutex

Argumenty

mutex – zmienna reprezentująca mutex

Wartość zwracana

Funkcja zwraca 0 (sukces) lub kod błędu.

EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA EWSIE

Korzystanie z mutexu

- W podstawowym zakresie korzystanie z mutexu sprowadza się do użycia dwóch funkcji:
- Wątek wchodzący do sekcji krytycznej wywołuje funkcję **pthread_mutex_lock**.
- Mutex jest zablokowany. Każdy kolejny wątek próbujący zablokować mutex, zostanie zablokowany.
- Wątek wychodzący z sekcji krytycznej zwalnia mutex funkcją **pthread_mutex_unlock**.
- Jeżeli inny wątek czeka na mutex to zostaje on odblokowany.

Funkcja pthread_mutex_lock

int pthread_mutex_lock(pthread_mutex_t *mutex);

Funkcja próbuje zablokować mutex. Jeżeli mutex był wolny, to zostaje zablokowany i wątek kontynuuje pracę. Jeżeli mutex był już zablokowany to wątek jest blokowany do czasu zwolnienia mutexu.

Argumenty

mutex – zmienna reprezentująca mutex

Wartość zwracana

Zwraca 0 (sukces) lub kod błedu

Funkcja pthread_mutex_unlock

int pthread_mutex_unlock(pthread_mutex_t *mutex);

Funkcja odblokowuje mutex

Argumenty

mutex – zmienna reprezentująca mutex

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu.

Nieblokujące użycie mutexu

- Próba zablokowania zajętego mutexu powoduje zablokowanie wątku.
- Wątek może do przejęcia mutexu użyć funkcji **pthread_mutex_trylock**
  - Jeżeli mutex jest wolny, zostaje zablokowany
  - Jeżeli mutex jest zablokowany przez inny wątek to funkcja zwraca kod błędu a wołający wątek nie jest blokowany

Funkcja pthread_mutex_trylock

int pthread_mutex_trylock(pthread_mutex_t *mutex);

Funkcja działa podobnie jak int pthread_mutex_lock, ale w przypadku zajętego mutexu nie blokuje wątku tylko wraca z kodem błedu.

Argumenty

mutex – zmienna reprezentująca mutex

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu.

„Rozmiar” mutexa

- Załóżmy, że mamy dwie zmienne wymagające ochrony mutexem. Możliwe rozwiązania:
  - „Duzy mutex” - Jeden mutex chroniący obie zmienne
  - „Mały mutex” - Jeden mutex dla każdej zmiennej
- „Duży” mutex oznacza dużą sekcję krytyczną – duży fragment kodu wykonywany szeregowo
- „Mały” mutex oznacza częste blokowanie i zwalnianie – narzut czasowy

Impas

T1	T2
...	...
R1.lock();	...
...	**R2.lock();**
...	...
R2.lock();	...
	R1.lock();

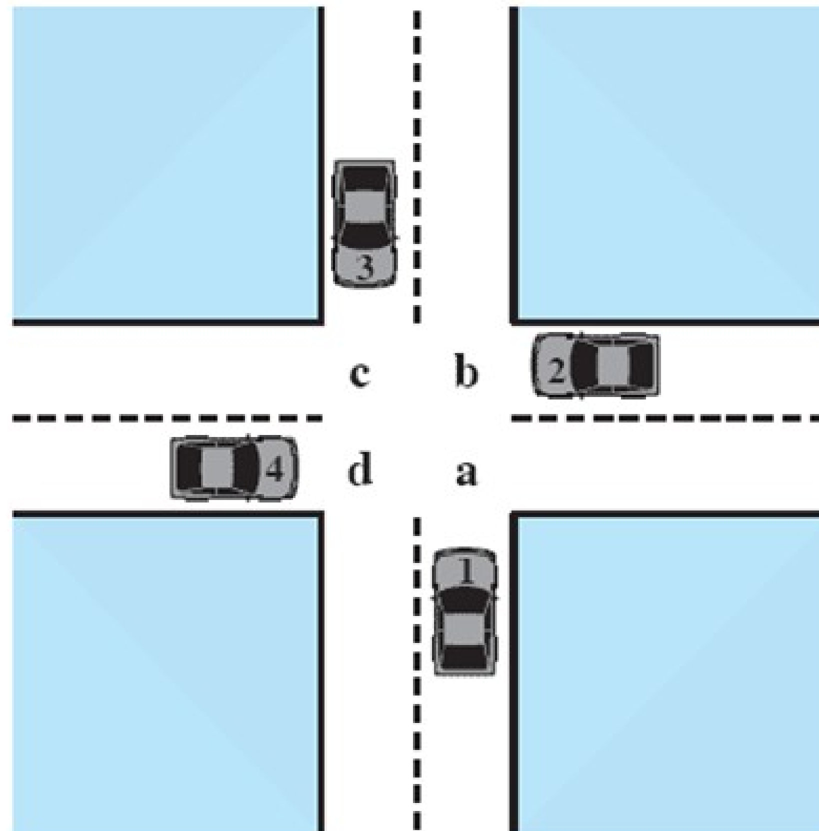
– Dwa wątki T1 i T2, oraz dwa zasoby R1 i R2.

– Każdy wątek wymaga dostępu do obu zasobów

– T1 rezerwuje R1, a T2 rezerwuje R2

– Każdy wątek czeka na zasób zarezerwowany przez drugi wątek

Impas c.d.



Stallings, „Systemy operacyjne …”

Zapobieganie impasom

- **Hierarchia mutexów**
  - Wątki zawsze blokują mutexy w określonym porządku.
- **Blokowanie z wycofywaniem**
  - Wątek próbuje zablokować mutex i jeżeli nie jest on wolny, to wątek odblokowuje wszystkie poprzednie mutexy

Zmienne warunku

- Mutexy zapewniają ochronę dostępu do danych (sekcja krytyczna).
- Stosowanie mutexa obciąża zasoby - „aktywne oczekiwanie”
- Zmienna warunku synchronizuje wątki w oparciu o dane.
- Zmienne warunku zawsze stosuje się razem z mutexem.

Tworzenie i usuwanie zmiennej warunku

- Zmienną warunku reprezentuje zmienne typu **pthread_cond_t**
- Zmienną warunku można utworzyć za pomocą makra PTHREAD_COND_INITIALIZER lub dynamicznie funkcją int **pthread_cond_init**
- Zmienną warunku usuwa się za pomocą funkcji **pthread_cond_destroy**

Funkcja pthread_cond_init

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

Tworzy zmienną warunku o podanych atybutach

Argumenty

cond – zmienna reprezentująca zmienną warunku

cond_attr – atrybuty zmiennej warunku

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu

Funkcja pthread_cond_destroy

int pthread_cond_destroy(pthread_cond_t *cond);

Usuwa zmienną warunku

Argumenty

cond – zmienna reprezentująca zmienną warunku

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu

Zmienna warunku - wykorzystanie

- Wątek oczekujący na spełnienie warunku używa funkcji **pthread_cond_wait**
- Wątek jest blokowany do czasu spełnienia warunku
- W momencie spełnienia warunku należy wywołać funkcję **pthread_cond_signal** (budzenie jednego wątku) lub **pthread_cond_broadcast** (budzenie wszystkich wątków)

Funkcja pthread_cond_signal

int pthread_cond_signal(pthread_cond_t *cond);

Sygnalizuje spełnienie warunku, budzi jeden wątek.

Argumenty

cond – zmienna reprezentująca zmienną warunku

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu

Funkcja pthread_cond_broadcast

int pthread_cond_broadcast(pthread_cond_t *cond);

Sygnalizuje spełnienie warunku, budzi wszystkie czekające wątki.

Argumenty

cond – zmienna reprezentująca zmienną warunku

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu

Funkcja pthread_cond_wait

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Blokuje wątek do czasu spełnienia warunku.

Argumenty

cond – zmienna reprezentująca zmienną warunku

mutex – mutex związany ze zmienną warunku

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu

Funkcja pthread_cond_timedwait

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Blokuje wątek do czasu spełnienia warunku lub na określony czas

Argumenty

cond – zmienna reprezentująca zmienną warunku

mutex – mutex związany ze zmienną warunku

abstime – czas obudzenia wątku niezależnie od warunku

Wartość zwracana

Zwraca 0 (sukces) lub kod błędu

Używanie zmiennej warunku

- Przed wejściem do stanu oczekiwania wątek powinien sprawdzić stan zmiennej warunku
- Testowanie zmiennej warunku i wejście w stan oczekiwania powinno być operacją atomową – wymaga zablokowania mutexa.
- W momencie wejścia w stan oczekiwania mutex jest automatycznie zwalniany.

EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA EWSIE

Używanie zmiennej warunku c.d.

- W momencie budzenia wątku mutex jest automatycznie zablokowany. Należy go odblokować.
- Po obudzeniu wątek powinien sprawdzić stan zmiennej warunku.
- Wątek sygnalizujący spełnienie warunku powinien zrobić to po zablokowaniu mutexu. A następnie zwolnić blokadę.

Używanie zmiennej warunku c.d.

- Dlaczego po obudzeniu wątek powinien sprawdzać zmienną warunku ?
  - Inny wątek mógł już obsłużyć warunek, tzn. warunek może już nie być spełniony (problem producent/konsument).
  - Wątek mógł zostać obudzony bez sygnału (zdarza się czasami w systemach SMP)
  - Można używać w sytuacjach typu „może być coś do zrobienia” zamiast „jest coś do zrobienia”

Pojedyncza inicjalizacja

- Często zachodzi potrzeba aby w programie pewne czynności wykonały się dokładnie raz (inicjalizacja).
- W przypadku wielu wątków nie jest to proste.
- Zamiast tworzyć własną implementację (zmienna z mutexem) można skorzystać z funkcji **pthread_once**.
- Funkcja pthread_once korzysta ze zmiennej typu **pthread_once_t**, którą należy zainicjować za pomocą makra **PTHREAD_ONCE_INIT**.

Funkcja pthread_once

```
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine) (void));
```

Gwarantuje wykonanie kodu tylko raz

Argumenty

control – zmienna kontrolna

init_routine – wykonywana funkcja z kodem inicjującym

Wartość zwracana

Zawsze zwraca 0.

EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA EWSIE

Atrybuty

- Implementacja PThreads dostarcza łatwy sposób inicjowania obiektów (wątki, mutexy, zmienne warunku itp.) z domyślnymi wartościami
- Modyfikacja zachowania obiektów wymaga dostarczenia struktur z wartościami atrybutów
- Wewnętrzna reprezentacja struktur atrybutów jest zależna od implementacji

Atrybuty mutexu

- Zestaw atrybutów mutexu tworzy się funkcją **pthread_mutexattr_init**, a usuwa funkcją **pthread_mutexattr_destroy**
- Zestaw atrybutów mutexu zależy od implementacji
- Poszczególne atrybuty mogą być testowane i ustawiane za pomoca zestawów funkcji get/set, np. **pthread_mutexattr_settype**, **pthread_mutexattr_gettype**.

Atrybuty zmiennej warunku

- Zestaw atrybutów zmiennej warunku tworzy się funkcją **pthread_condattr_init**, a usuwa funkcją **pthread_condattr_destroy**.
- Zestaw atrybutów zmienne warunku zależy od implementacji.
- Linux nie implementuje żadnych atrybutów. Inne systemy mogą implementować atrybut pshared (zmienna warunku może być używana przez rózne procesy)

Atrybuty wątku

- Standard definiuje następujące atrybuty wątku:
  - detachstate
  - stacksize, stackaddress
  - scope
  - inheritsched
  - schedpolicy, schedparam
- Nie wszystkie atrybuty muszą być implementowane w danej implementacji
- Do testowania i ustawiania atrybutów należy stosować odpowiednie funkcje

Atrybut detachstate

- Atrybut określa w jakim stanie stanie jest tworzony wątek.
- Dopuszczalne wartości:
  - **PTHREAD_CREATE_DETACHED** – wątek tworzony jest w stanie odłączony
  - **PTHREAD_CREATE_JOINABLE** – na wątku można użyć funkcji pthread_join

Atrybuty stacksize, stackaddress

- Pozwalają ustawić rozmiar i położenie stosu wątku
- **UWAGA!** Funkcje te powinny być używane ostrożnie.
- Ustawianie położenia stosu jest szczególnie niebezpieczne gdyż zależy od realizacji sprzętowej i jest bardzo podatne na błędy

Atrybuty szeregowania wątku

- Atrybut inheritsched określa czy tworzony wątek dziedziczy atrybuty szeregowania po wątku macierzystym.
- Polityka szeregowania:
  - SCHED_FIFO – szeregowanie FIFO
  - SCHED_RR – szeregowanie Round-Robin
- Inne parametry szeregowania

Atrybut scope

- Atrybut określa w jaki sposób wątek rywalizuje o zasoby.
- Wartości:
  - **PTHREAD_SCOPE_SYSTEM** – rywalizuje o zasoby z wszystkimi innymi wątkami w systemie
  - **PTHREAD_SCOPE_PROCESS** – rywalizuje o zasoby z innymi wątkami procesu

Anulowanie wątku

- Do anulowania wątku służy funkcja **pthread_cancel**
- Zachowanie przerywanego wątku zależy od jego ustawień, możliwe stany wątku:
  - **Asynchroniczny** – wątek może być przerwany w dowolnym momencie.
  - **Synchroniczny** – może być przerwany tylko w określonych momentach
  - **Nieprzerywalny**

Anulowanie wątku c.d.

- Funkcja **pthread_setcancelstate** ustawia stan wątku z punktu widzenia jego anulowania. Możliwe stany:
  - **PTHREAD_CANCEL_ENABLE** – anulowanie wątku jest możliwe
  - **PTHREAD_CANCEL_DISABLE** – anulowanie wątku nie jest możliwe, polecenie anulowania jest blokowane do czasu zmiany stanu wątku

Anulowanie wątku c.d.

- Funkcja **pthread_setcanceltype** określa sposób wykonania operacji anulowania wątku. Możliwe wartości:
  - **PTHREAD_CANCEL_DEFERRED** – operacja anulowania czeka aż wątek osiągnie punkt anulowania
  - **PTHREAD_CANCEL_ASYNCHRONOUS** – anulowanie wątku jest realizowane natychmiast.

Anulowanie wątku c.d.

- Wątek działający w trybie synchronicznym powinien okresowo sprawdzać czy nie został anulowany wywołując funkcję **pthread_testcancel**
- Standard POSIX określa które funkcje systemowe są z definicji punktami anulowania, oraz które mogą być jako takie zaimplementowane

Dane prywatne wątku

- Każdy wątek posiada własny obszar pamięci na dane specyficzne dla wątku (TSD)
- Obszar TSD indeksowany jest za pomocą kluczy
- Nowy klucz można utworzyć funkcją **pthread_key_create** i usunać go funkcją **pthread_key_delete**.
- Wartość dla danego klucza wątek może ustawić funkcją **pthread_setspecific** i pobrać funkcją **pthread_getspecific**

EUROPEJSKA WYŻSZA SZKOŁA INFORMATYCZNO-EKONOMICZNA EWSIE

Modele programu wielowątkowego

- **Pipeline** - „Linia produkcyjna” - Każdy wątek wykonuje operację na sekwencji danych i przekazuje wynik do kolejnego wątku
- **Work crew** – „Brygada robocza” - Wątki wykonują operacje na niezależnych danych
- **Client/serwer** – Wątek klienta otrzymuje „kontrakt” na wykonanie zadania.