9. Problem wzajemnego wykluczania i sekcji krytycznej

9.1 Przeplot i współużywalność zasobów

Wyróżniamy dwa rodzaje zasobów:
1. Zasoby współużywalne - mogą być wykorzystane przez dowolną liczbę procesów.
2. Zasoby nie współużywalne - w danym odcinku czasu mogą być wykorzystane tylko przez jeden proces.

Przykład zasobu nie współużywalnego
- urządzenia wejścia / wyjścia.
- wspólny obszar pamięci.

Gdy kilka procesów czyta a przynajmniej jeden dokonuje zapisu wynik odczytu zależeć może od sposobu realizacji przeplotu .

Przykład 1 – wątki korzystają ze zmiennej dzielonej

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 8

pthread_t tid[NUM_THREADS]; // Tablica identyfik.
watkow
static int x;

void kod(int num) {
    for(;;) {
        x = 0;
        x = x+1;
        printf(„watek: %d wartość: %d\n”,num,x);
    }
}

int main(int argc, char *argv[]){
   int i;
   // Tworzenie watkow ------------
   for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, kod,(void *)
    (i+1));
   ...
}
```

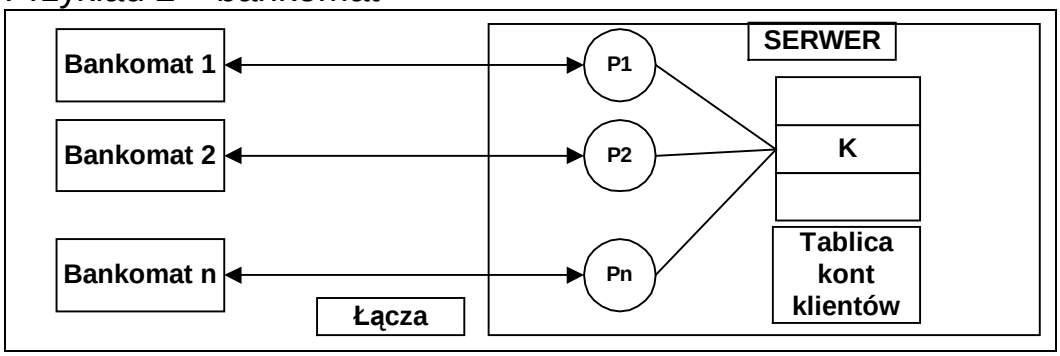
Wątek1	Wątek 2	Wątek 3	x
x=0			0
x = x + 1			1
	x=0		0
		x=0	0
printf x			0
	x = x + 1		1
	printf x		1
		x = x + 1	2
		printf x	2

Możliwa realizacja przykładu 1 – każdy watek daje inny wynik

```
Watek 1 wartosc 0
Watek 2 wartosc 1
Watek 3 wartosc 2
```

Wyniki działania przykładu 1

Przykład 2 – bankomat



Bankomaty dołączone do serwera bankowego K – stan konta klienta

Algorytm wypłaty:
Sprawdź czy żądana kwota x nie przekracza kwoty K rachunku. Gdy nie wydaj zgodę na wypłatę a następnie zmniejsz kwotę K rachunku o kwotę wypłaty x. Gdy żądana kwota x przekracza K odmów wypłaty.

```
int depozyt[N]; // Depozyty-zmienna globalna

int czy_wypłata(int x,int nr_kl)
{
  if(x <= depozyt[nr_kl]) {
    (* przełączenie *)
    depozyt[nr_kl] = depozyt[nr_kl]-x;
    return(1);
  } else {
    return(0);
  }
}
```

Sprawdzanie czy można dokonać wypłaty x z konta klienta

Proces 1	Proces 2	depozyt [nr_kl]
if(x<=depozyt[nr_kl])		100
	if(x<=depozyt[nr_kl])	100
depozyt[nr_kl] = depozyt[nr_kl]-x		20
	depozyt[nr_kl] = depozyt[nr_kl]-x	-60
return(1);		-60
	return(1);	-60

Wypłata kwoty x=80 z dwóch bankomatów przy stanie konta K=100.

Proces 1	Proces 2	K
MOV AX,X		100
MOV BX,K		100
CMP BX,AX		100
Przełączenie kontekstu z P1 na P2		
	MOV AX,X	100
	MOV BX,K	100
	CMP BX,AX	100
Przełączenie kontekstu z P2 na P1		
JB xxx		100
SUB BX,AX		100
MOV X, BX		20
	JB xxx	20
	SUB BX,AX	20
	MOV X, BX	-60

Kod maszynowy dla powyższego przykładu (dla uniprocesora pracującego z przeplotem).

Przykład 3 – Bank wpłaty i wypłaty – problem utraconej aktualizacji

```
void wyplata (int konto, int kwota) {
  int stan;
  stan = czytaj(konto);
  pisz(konto,stan – kwota);
}

void wplata (int konto, int kwota) {
   int stan;
  stan = czytaj(konto);
  pisz(konto,stan + kwota);
}

stan konta 1 – 100
stan konta 2 – 200
stan konta 3 - 300
```

Transakcja 1
Przeniesienie 10 zł z konta 1 na 2

```
wyplata(1,10);
wplata(2,10);
```

Transakcja 2
Przeniesienie 20 zł z konta 3 na 2

```
wyplata(3,20);
wplata(2,20);
```

Po transakcjach powinno być:

```
stan konta 1 –  90
stan konta 2 – 230
stan konta 3 - 280
```

Proces 1	Proces 2	K1	K2	K3
		100	200	300
Czytaj(1)->100		100	200	300
Pisz(1,100-10)		90	200	300
	Czytaj(3)->300	90	200	300
	Pisz(3,300-20)	90	200	280
Czytaj(2)->200		90	200	280
	Czytaj(2)->200	90	200	280
	Pisz(2,200+20)	90	220	280
Pisz(2,200+10)		90	210	280
		90	210	280

Wynik:
Z konta 2 znikło 10 zł

9.2 Problem wzajemnego wykluczania i warunki jego rozwiązania

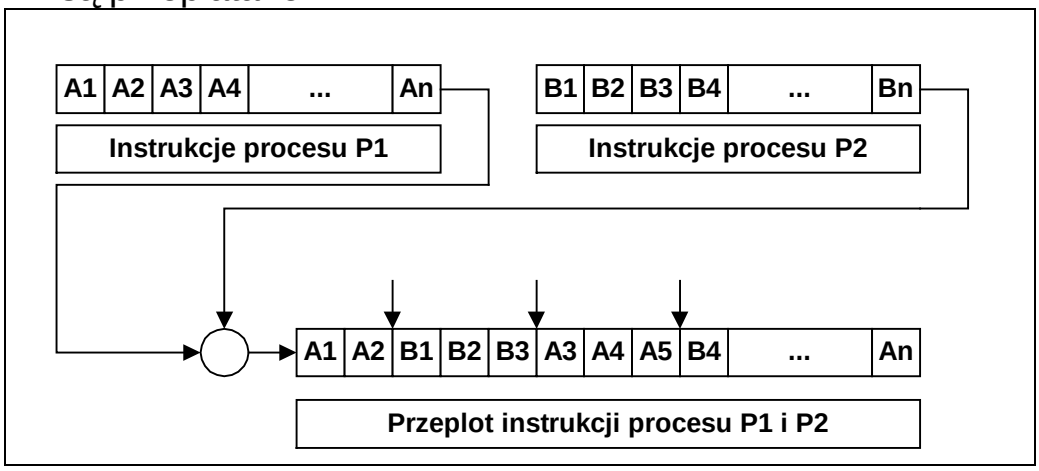
Operacja atomowa
Sekwencja jednego lub wielu działań elementarnych które nie mogą być przerwane. Wykonuje się w całości albo wcale. Działania pośrednie nie mogą być obserwowane przez inny proces.

Operacja atomowa drobnoziarnista (*ang. fine grained*)
Operacja wykonywana przez pojedynczą atomową instrukcję kodu maszynowego.

Operacja atomowa gruboziarnista (*ang. coarse grained*)
Sekwencja operacji drobnoziarnistych której zapewniono niepodzielność innymi metodami.

Zakładamy że:
- Odczyt z pamięci komórki o adresie X jest operacją atomową
- Zapis do pamięci komórki o adresie X jest operacją atomową

W powyższym przykładzie instrukcje atomowe kodu procesów P1 i P2 są przeplatane.



Instrukcje procesów P1 i P2 wykonywane w trybie przeplotu

- Nie możemy poczynić żadnych założeń dotyczących momentów przełączenia procesów P1 i P2
- Nie da się określić wyniku działania powyższych procesów.

Wynik działania aplikacji współbieżnej nie może być uzależniony od sposobu przełączania procesów. Musi być prawidłowy dla wszystkich możliwych przeplotów.

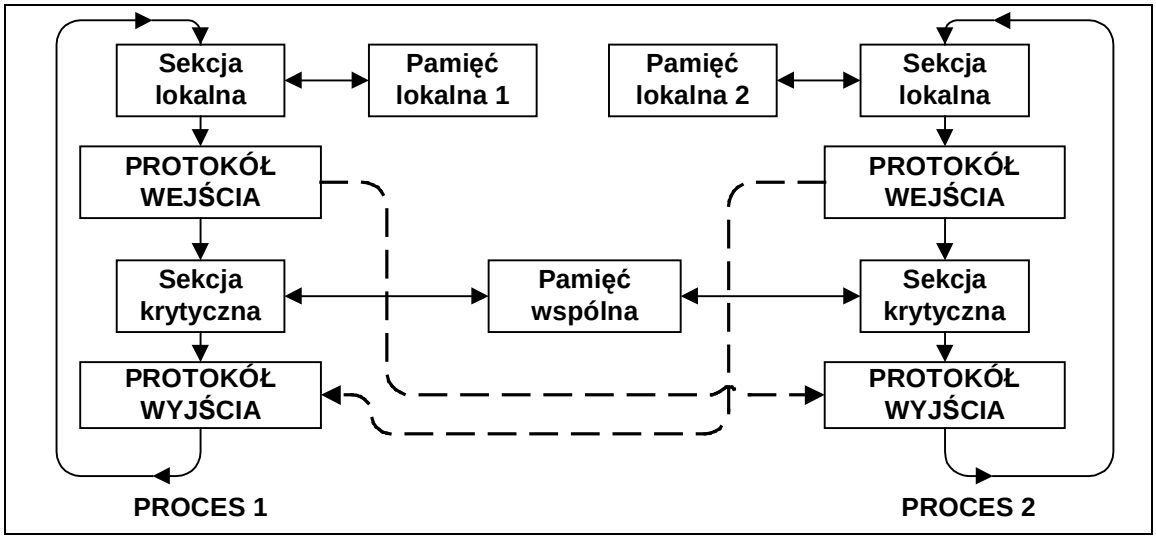
Gdy procesy współbieżne do wzajemnej komunikacji używają wspólnej pamięci, wyniki takiej komunikacji mogą okazać się przypadkowe. Prawidłowa komunikacja współbieżnych procesów przez wspólny obszar pamięci wymaga dotrzymania warunku wzajemnego wykluczania.

Wzajemne wykluczanie - wymaganie aby ciąg operacji na pewnym zasobie (zwykle pamięci) był wykonany w trybie wyłącznym przez tylko jeden z potencjalnie wielu procesów.

Sekcja krytyczna – ciąg operacji na pewnym zasobie (zwykle pamięci) który musi wykonany w trybie wyłącznym przez tylko jeden z potencjalnie wielu procesów.

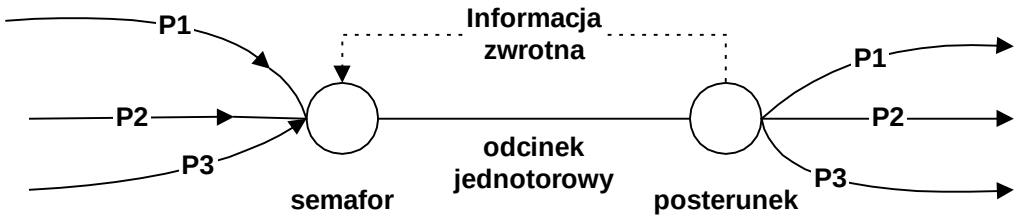
Przy wejściu do sekcji proces wykonuje **protokół wejścia** w którym sprawdza czy może wejść do sekcji krytycznej.

Po wyjściu z sekcji wykonuje **protokół wyjścia** aby poinformować inne procesy że opuścił już sekcję krytyczną i inny proces może ją zająć.



Model programowania z sekcją lokalną i sekcją krytyczną

W danej chwili w sekcji krytycznej może przebywać tylko jeden proces.



Na odcinku jednotorowym może przebywać tylko jeden pociąg

Rozwiązanie problemu wzajemnego wykluczania musi spełniać następujące warunki:

1. W sekcji krytycznej może być tylko jeden proces to znaczy instrukcje z sekcji krytycznej nie mogą być przeplatane.
2. Nie można czynić żadnych założeń co do względnych szybkości wykonywania procesów.
3. Proces może się zatrzymać w sekcji lokalnej nie może natomiast w sekcji krytycznej. Zatrzymanie procesu w sekcji lokalnej nie może blokować innym procesom wejścia do sekcji krytycznej.
4. Każdy z procesów musi w końcu wejść do sekcji krytycznej.

9.3 Niesystemowe metody wzajemnego wykluczania.

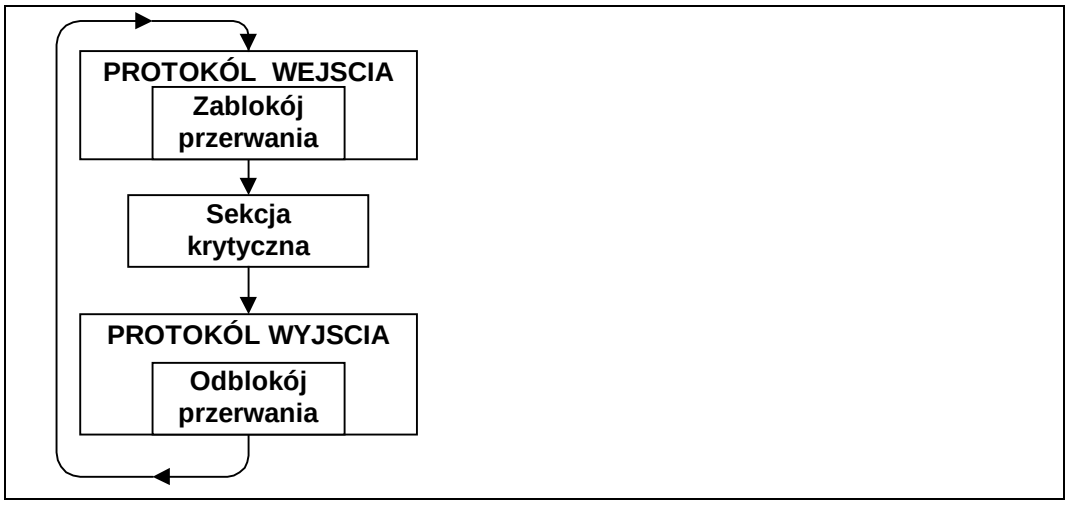
9.3.1 Blokowanie przerwań

Metoda zapewnienia wzajemnego wykluczania poprzez blokowanie przerwań opiera się na fakcie że proces może być przełączony przez:
1. Przerwanie które aktywuje procedurę szeregującą
2. Wywołanie wprost procedury szeregującej lub innego wywołania systemowego powodującego przełączenie procesów.

Gdy żaden z powyższych czynników nie zachodzi procesy nie mogą być przełączane.

Metoda ochrony sekcji krytycznej poprzez blokowanie przerwań opiera się na następujących zasadach:

1 Protokół wejścia do sekcji – następuje zablokowanie przerwań.
2. Protokół wyjścia z sekcji – następuje odblokowanie przerwań.
3. Wewnątrz sekcji krytycznej nie wolno używać wywołań systemowych mogących spowodować przełączenie procesów.



Ochrona sekcji krytycznej przez blokowanie przerwań

Wady metody:
1. Przełączanie wszystkich procesów jest zablokowane.
2. System nie reaguje na zdarzenia zewnętrzne co może spowodować utratę danych.
3. Skuteczne w maszynach jednoprocesorowych

Zastosowanie metody:
Wewnątrz systemu operacyjnego do ochrony wewnętrznych sekcji krytycznych.

9.3.2 Metoda zmiennej blokującej (nieprawidłowa)

Operacje atomowe:
W większości procesorów operacje zapisu i odczytu bajtu, krótkiego słowa (2 bajty), słowa (4 bajty) są operacjami atomowymi.

W procesorach Intel 486 następujące operacje odczytu i zapisu są operacjami atomowymi:
- Bajt
- Krótkie słowo (2 bajty) gdy jest wyrównane do granicy 16 bitów
- Słowo (4 bajty) gdy jest wyrównane do granicy 32 bitów

W procesorach Pentium dodatkowo
- Podwójne słowo (8 bajty) gdy jest wyrównane do granicy 64 bitów

Metoda polega na użyciu zmiennej o nazwie lock.
Gdy zmienna lock = 0 sekcja jest wolna, gdy lock = 1 sekcja jest zajęta.
Proces przy wejściu testuje wartość tej zmiennej. Gdy wynosi ona 1 to czeka, gdy zmieni się na 0 wchodzi do sekcji ustawiając wartość zmiennej lock na 1.

```
int lock = 0;

do {
  sekcja_lokalna;
  // Protokół wejścia
  while(lock != 0) (* czekanie aktywne *);
  lock = 1;
  sekcja_krytyczna;
  lock = 0;  // Protokół wyjścia
} while(1);
```

Kompilator może przetłumaczyć powyższy kod w następujący sposób:

```
CHECK:   MOV AL, lock
         TEST AL,AL
         JNZ CHECK
         MOV lock, 1
             sekcja_krytyczna
         MOV lock, 0
```

Proces 1	Proces 2	lock
MOV AL, lock		0
TEST AL,AL		
Przełączenie kontekstu z P1 na P2		
	MOV AL, lock	0
	TEST AL,AL	0
	JNZ CHECK	
Przełączenie kontekstu z P2 na P1		
JNZ CHECK		0
MOV lock, 1		1
Przełączenie kontekstu z P1 na P2		
P1 w sekcji krytycznej	MOV lock, 1	1
	P2 w sekcji krytycznej	1

Metoda jest niepoprawna, gdyż operacja testowania wartości zmiennej lock i ustawiania jej na 1 może być przerwana (nie jest niepodzielna).
Dodatkową wadą metody jest angażowanie procesora w procedurze aktywnego czekania.

9.4 Wykorzystania wsparcia sprzętowego do ochrony sekcji krytycznej

Uwaga:
Do zapewnienia wzajemnego wykluczania należy dysponować atomową operacją: ***czytaj – modyfikuj – zapisz***

Wiele mikroprocesorów zawiera instrukcje wspierające sprzętowo wzajemne wykluczanie. Są to instrukcje typu

1. **TAS** - sprawdź i przypisz - (*ang. TAS -Test And Set*)
2. **CAS** - porównaj i zamień – (ang. *Compare And Swap*)

Pozwalają one wykonać kilka operacji w sposób nieprzerywalny.

Operacja **CAS** wymaga trzech argumentów:

- *mem* - lokacja pamięci (zmienna)
- *old* - spodziewana (stara) zawartość zmiennej mem
- *new* - nowa zawartość zmiennej mem

Działanie operacji **CAS *mem*, *new*, *old***

Atomowo wykonaj operację:

- Gdy zawartość zmiennej *mem* jest równa *old* wykonaj podstawienie *mem* = *new* .
- Gdy zawartość zmiennej *mem* nie jest równa *old* nie rób nic.

Operacja może także ustawiać flagi procesora (np. ZF) aby ułatwić testowanie zmiennej *mem*.

W procesorze SPARC Version 9 występują trzy instrukcje typu czytaj – modyfikuj - zapisz:

- ldstub – load store unsigned byte
- swap
- cas

(Na podstawie http://developers.sun.com/solaris/articles/atomic_sparc/)

Instrukcja **ldstub**:
Używana w systemie Solaris do zapewnienia wzajemnego wykluczania. Instrukcja atomowo zapisuje wartość **0xff** w bajcie lock_byte i zwraca jej poprzednią zawartość.

```
int ldstub( int *lock_byte ) {
    int old_value;
    atomic {
        old_value  = *lock_byte;
        *lock_byte = 0xff;
    }
    return( old_value );
}
```

Implementacja blokady:

```
// lock = 0 blokada wolna, lock = FF blokada zajęta

get_lock(int *lock) {
   while(ldstub(*lock) =! 0) { /* Czekanie */ }
}

release_lock(int *lock) {
  *lock = 0;
}
```

Instrukcja CAS - Compare and Swap

```
int cas( int *word,int test_value,int new_value )
{
    int old_value;
    atomic {
        old_value    = *word;
        if ( *word == test_value )
            *word= new_value;
    }
    return( old_value );
}
```

9.5 Sprzętowe wspomaganie wzajemnego wykluczania w procesorach IA-32

W mikroprocesorach IA32 sprzętową ochronę sekcji krytycznej wspomagają instrukcje XCHG (zamień) i CMPXCHG (porównaj i zamień).

Instrukcja Zamień

XCHG *mem*, *reg*

Działanie:

1. Instrukcja powoduje niepodzielną wymianę zawartości komórki ***mem*** i rejestru ***reg***.
2. Na czas wykonania instrukcji dostęp do pamięci operacyjnej jest blokowany, dla kontrolera pamięci wystawiany jest sygnał LOCK. Uniemożliwia to dostęp do pamięci innym procesorom (gdy system jest wieloprocesorowy).

Instrukcja porównaj i zamień

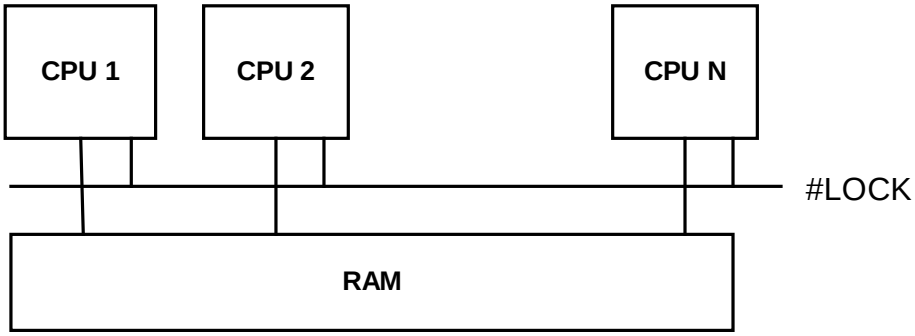
CMPXCHG *mem*, *reg*

1. Porównywana jest wartość akumulatora A z zawartością komórki ***mem***.

- Jeżeli wartości te są równe ustawiana jest flaga ZF i zawartość rejestru ***reg*** jest ładowana do komórki ***mem***.
- Jeżeli zawartość akumulatora A nie jest równa zawartości komórki ***mem*** flaga ZF jest zerowana i do akumulatora A ładowana jest zawartość komórki ***mem***.

2. Instrukcja może być użyta z przedrostkiem LOCK

Blokowanie magistrali:
Procesory Pentium mogą być używane w systemach wieloprocesorowych. Stąd potrzeba blokowania dostępu do pamięci na czas wykonania krytycznych operacji. Architektura IA-32 przewiduje do tego celu sygnał LOCK#.



Rys. 9-1 Sprzętowy sygnał #LOCK blokuje dostęp do pamięci dzielonej.

Gdy sygnał LOCK# jest aktywny dostęp do pamięci przez inne procesory lub moduły aktywne jest zablokowany.

W asemblerze dla procesorów Intel używany jest przedrostek LOCK. Powoduje on zablokowanie dostępu do magistrali na czas wykonania bieżącej instrukcji.

Dla pewnych instrukcji sygnał LOCK# jest aktywowany automatycznie:

- Instrukcja XCHG gdy jeden z operandów odnosi się do pamięci.
- Przy przełączaniu zadań (bit Busy w deskryptorze TSS)
- Przy aktualizacji deskryptorów segmentów.
- Przy aktualizacji katalogu stron i tablicy stron.
- Podczas przerwania, gdy kontroler przerwań przesyła do procesora numer przerwania.

9.6 Wirujące blokady (ang. *Spinlock*)

W oparciu o instrukcję XCHG i zmienną lock można zaimplementować procedurę ochrony sekcji krytycznej - tak zwaną wirującą blokadę.

```
lock:          dd 0                  # 1 – sekcja zajęta
                                     # 0 – sekcja wolna
spin_lock:                           # Zajmij blokadę
CHECK:         MOV EAX,1             # ustaw rejestr EAX na 1
               XCHG EAX ,[lock]      # wymien niepodzielnie
                                     # EAX i zmienna lock
               TEST EAX, EAX         # testuj zawartość EAX
                                     # ustawi to flagę ZF
               JNZ CHECK             # skocz do CHECK gdy
                                     # sekcja była zajeta
               RET                   # zakończ procedurę

spin_unlock:   # Zwolnij blokadę      # zwolnij blokadę
               MOV EAX,0             # ustaw rejestr EAX na 0
               XCHG EAX ,[lock]      # wymien niepodzielnie
                                     # EAX i zmienna lock
               RET                   # zakończ procedurę
```

Przykład 9-1 Wykorzystanie instrukcji XCHG do implementacji procedur ochrony sekcji krytycznej

Wadą metod jest użycie aktywnego czekania co powoduje niepotrzebną stratę mocy procesora.

Własności:
W systemach jednoprocesorowych i tak musi dojść do przełączenia wątku, bo kto miałby zmienić wartość testowanej zmiennej? Tak więc, nie opłaca się stosować wirujących blokad w systemie jednoprocesorowym.

Są one stosowane gdy:

- W systemach wieloprocesorowych SMP. Czas oczekiwanie jest mniejszy niż czas przełączenia wątku i nie opłaca się wywłaszczać wątku.
- W systemach wieloprocesorowych SMP – blokada przerwań tu nie wystarczy.
- Gdy nie ma innych metod – np. brak systemu operacyjnego.

Wady:

- Zajmują czas procesora – czekanie aktywne
- Brak mechanizmów zapewniających uczciwość (może wystąpić zagłodzenie)
- Brak mechanizmów zapewniających bezpieczeństwo (można czekać w nieskończoność)

9.7 Wirujące blokady w standardzie POSIX

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

Inicjacja wirującej blokady:

```
int pthread spin init( pthread spinlock t *
spinner, int pshared )
```

Gdzie:

spinner	blokada
pshared	Flaga: PTHREAD_PROCESS_SHARED PTHREAD PROCESS PRIVATE

Gdzie:

PTHREAD_PROCESS_SHARED	blokada może być używana przez wątki różnych procesó**w**
PTHREAD PROCESS PRIVATE	blokada może być używana przez wątki jednego procesu

Funkcja inicjalizuje zasoby potrzebne do działania wirującej blokady i pozostawia ją w stanie otwartym.

Zajęcie blokady:

```
int pthread spin lock( pthread spinlock t *
spinner )
```

Funkcja próbuje zająć blokadę. Gdy jest ona wolna to zostaje zajęta. Gdy jest zajęta to wątek jest blokowany do czasu aż blokada nie zostanie zwolniona.

Zwolnienie blokady:

```
int pthread spin unlock( pthread spinlock t * sp)
```

Funkcja zdejmuje blokadę sp. Gdy jakieś wątki oczekują na zdjęcie blokady jeden z nich (nie jest specyfikowane który) będzie odblokowany.

Warunkowe zajęcie blokady:

`int pthread_spin_trylock(pthread_spinlock_t *sp )`

Funkcja próbuje zająć blokadę sp. Gdy jest ona wolna to zostaje zajęta. Gdy jest zajęta to funkcja zwraca kod błędu EBUSY i wątek nie jest blokowany.

Funkcja zwraca:
EOK - gdy udało się zająć blokadę
EBUSY - gdy blokada jest zajęta

Skasowanie blokady:

`pthread_spin_destroy( pthread spinlock t * sp)`

Funkcja kasuje blokadę sp. Gdy jakieś wątki oczekują na zdjęcie blokady, będą one odblokowane.

Programowe metody zapewnienia wzajemnego wykluczania

We wczesnych procesorach nie było wsparcia sprzętowego dla wzajemnego wykluczania. Stąd wzajemne wykluczanie realizowano w sposób wyłącznie programowy. Obecnie metody te mają znaczenie tylko teoretyczne i historyczne. Wymienić można tu algorytm Dekkera, algorytm Petersona,
Algorytm piekarniczy.

9.7.1 Algorytm Petersona

Rozwiązanie problemu wzajemnego wykluczania dla dwu procesów bez użycia mechanizmów sprzętowych podał T. Dekker.

Prostsze rozwiązanie problemu skonstruowane przez Petersona.

```
#define N 2         // Liczba procesów
#define TRUE 1
#define FALSE 0
int num = 0;
int zainteresowany[N];

// Protokół wejścia
enter_region(int proces) {
  // proces – numer procesu 0 lub 1
  int inny;
  inny = 1 – proces;
  zainteresowany[proces] = TRUE;
  num = proces;
  while((num == proces) &&
 (zainteresowany[inny] == TRUE));
 /* Czekanie aktywne */
}

 // Protokół wyjścia
 leave_region(int proces) {
  // proces – numer procesu 0 lub 1
  zainteresowany[proces] = FALSE;
}
```

Algorytm Petersona zapewniający wzajemne wykluczanie dla dwu procesów

9.7.2 Algorytm piekarniczy

Tak zwany algorytm piekarniczy pozwala na rozwiązanie problemu wzajemnego wykluczania dla N procesów. Algorytm podany przez Lamporta.

Klient przy wejściu pobiera numerowany bilet. Wartość numeru na bilecie jest najwyższa ze wszystkich dotychczas wydanych a nie obsłużonych biletów. Gdy stanowisko obsługi się zwolni, ten z czekających klientów jest obsługiwany który posiada bilet o najniższym numerze.

Proces wykonujący protokół wejścia otrzymuje numer – największy z dotychczas przyznanych. Gdy jakiś proces opuszcza sekcję krytyczną, ten z czekających procesów wchodzi do sekcji, który posiada najniższy numer.

9.8 Systemowe metody zapewnienia wzajemnego wykluczania

Niesystemowe metody stosowane są rzadko i ich znaczenie jest raczej teoretyczne.
Powody:

1. Prawie zawsze tworzymy aplikacje działające w środowisku systemu operacyjnego który z reguły dostarcza mechanizmów zapewnienia wzajemnego wykluczania.
2. Realizacja metod wzajemnego wykluczania polega na zawieszeniu pewnych procesów a wznowieniu innych. System operacyjny w naturalny sposób zapewnia takie mechanizmy. Proces zawieszony nie wykonuje czekania aktywnego a zatem nie zużywa czasu procesora.
3. Metody systemowe są znacznie prostsze i powiązane z innymi mechanizmami i zabezpieczeniami. Przykładowo awaryjne zakończenie się procesu w sekcji krytycznej odblokowuje tę sekcję. Można też narzucić maksymalny limit czasowy oczekiwania na wejście do sekcji krytycznej (*ang. Timeout*).

Z niesystemowych metod wzajemnego wykluczania praktycznie stosowane są metody:

1. Wirujące blokady (*ang. Spin Locks*) wykorzystujące sprzętowe wsparcie w postaci instrukcji sprawdź i przypisz oraz zamień. Stosuje się je do synchronizacji wątków ze względu na mały narzut operacji systemowych.
2. Blokowanie przerwań – do ochrony wewnętrznych sekcji krytycznych systemu operacyjnego.

9.8.1 Wzajemne wykluczanie poprzez obiekty typu mutex

Mechanizm zapewniających wzajemne wykluczanie zaimplementowny jest w wielu systemach operacyjnych. W systemach standard Posix mechanizm ten nosi nazwę mutex. Jest to skrót od angielskiego terminu *Mutual Exclusion*. Poniżej opisano implementację dotyczącą wątków POSIX z biblioteki Pthreads.

Tworzenie obiektu typu mutex

```
int mutex_init(mutex_t*mutex, mutexattr_t* attr )
```

mutex Obiekt typu **mutex**
attr Atrybuty określające zachowanie obiektu. Gdy NULL atrybuty zostaną przyjęte domyślne

Wykonanie funkcji pozostawia zmienną mutex w stanie nie zablokowanym.

Zablokowanie sekcji krytycznej

```
int mutex_lock( mutex_t* mutex )
```

mutex Obiekt typu mutex zainicjowany poprzednio przez funkcję mutex_init

Gdy przynajmniej jeden proces wykonał wcześniej funkcję **mutex_lock** zmienna **mutex** oznaczona będzie jako zajęta. Proces bieżący wykonujący tę funkcję zostanie wstrzymany

Zwolnienie sekcji krytycznej
Proces opuszczający sekcję krytyczną powinien poinformować o tym system (wykonać protokół wyjścia).

```
int mutex_unlock( pthread_mutex_t* mutex )
```

mutex Obiekt typu **mutex**

Gdy jakieś procesy czekają na wejście do sekcji to jeden z nich będzie odblokowany i wejdzie do sekcji. Gdy brak takich procesów to sekcja zostanie oznaczona jako wolna.

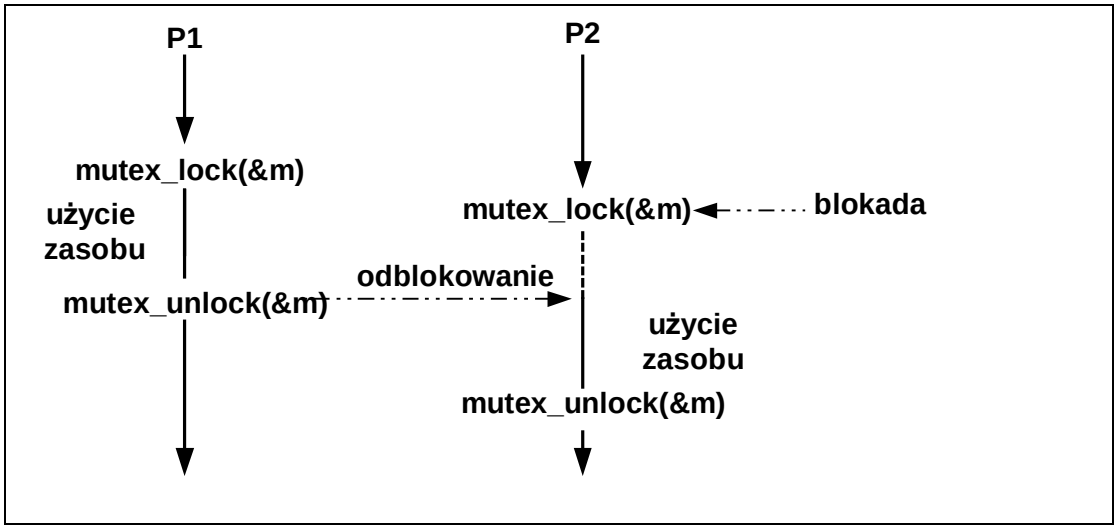
Skasowanie obiektu typu mutex

```
int mutex_destroy( pthread_mutex_t* mutex );
```

Podstawowy schemat ochrony sekcji krytycznej przy użyciu zmiennej **mutex**:

```
mutex_t mutex ;  // Deklaracja zmiennej mutex

mutex_init(&mutex,NULL);     // Inicjalizacja
zmiennej
do {
  ............
  // Zablokowanie sekcji krytycznej
  mutex_lock( &mutex );

    Sekcja krytyczna

  // Zwolnienie sekcji krytycznej
  unlock( &mutex );

} while(1),
```

Ochrona sekcji krytycznej przez obiekt typu mutex



9.9 Pamięci transakcyjne

Wady podejścia blokującego dostęp do sekcji krytycznej

- Gruboziarniste blokady ograniczają stopień równoległości.
- Drobnoziarniste blokady maja duży narzut czasowy i są kłopotliwe dla programisty.
- Podatność na błędy: zakleszczenie, zagłodzenie, pominięte zabezpieczenie sekcji krytycznej, inwersja priorytetów
- Może prowadzić do inwersji priorytetów – wątek o wyższym priorytecie musi czekać aż wątek o niższym priorytecie zwolni dostęp do zasobu.
- Blokady naruszają strukturalność - programów z blokadami nie można swobodnie składać.

Rozwiązania:

- Model procesów i komunikatów – agent zasobu. Wykorzystanie w Occam, Erlang
- Pamięć transakcyjna

Operacje na pamięci dzielonej odbywają się w postaci transakcji tak jakby wykonywał się tylko jeden wątek.

Transakcja – ciąg operacji przeprowadzający zbiór danych z jednego stanu spójnego do drugiego.

- Transakcja definiuje sekwencję operacji na wspólnych danych.
- Jest abstrakcją wyższego rzędu niż muteksy czy semafory.
- Stosowane głównie w systemach baz danych

Własności ACID transakcji:

1. **Niepodzielność** (*ang. Atomicity*) – transakcja ma być wykonana albo w całości albo wcale.

2. **Spójność** (*ang. Consistency*) – transakcja przeprowadza system z jednego spójnego stanu w drugi.

3. **Izolacja** (*ang. Isolation*) – jeżeli współbieżnie przeprowadzane są inne transakcje na wspólnych danych to nie wpływają one na transakcję bieżącą. Pośrednie skutki transakcji nie są widoczne dla innych transakcji.

4. **Trwałość** (*ang. Durabilty*) – zmiany muszą być zapisane w pamięci trwałej.

Ostatnia własność 4 nie jest wymagana w pamięci transakcyjnej.

Występują dwa rodzaje pamięci transakcyjnej:
- Sprzętowa
- Programowa STM *Software transactional memory*

Do realizacji pamięci STM konieczne jest wsparcie sprzętowe – np. instrukcja CAS (Compare and Swap, Compare and Exchange).

Zasada działania pamięci transakcyjnej:
- Program wykonuje się tak jak gdyby transakcje były wykonywane niezależnie od innych wątków.

- Na koniec transakcji wykonuje się sprawdzanie czy transakcja może być zatwierdzona. Gdy tak (dostęp był wyłączny) jest zatwierdzana. Gdy nie jest cofana, dziennik pozwala na cofniecie nieudanej transakcji, i następnie powtarzana.

Jest to realizacja realizacja optymistyczna i nieblokujaca.

Przykład zapisu:
```
// Wstawienie węzła do listy
atomic {
     newNode->prev = node;
```

```
      newNode->next = node->next;
      node->next->prev = newNode;
      node->next = newNode;
 }
```

Przykład z C#
Dwa wątki modyfikują dwa łańcuchy s1 i s2 wstawiając tam:

```
SetStrings("Hello", "World"); // Wątek 1
SetStrings("World", "Hello"); // Wątek 2


public void SetStrings(string s1, string s2){
    m_string1 = s1;
    Thread.Sleep(1); // Symulacja zajętości
    m_string2 = s2;
}
```

Procedura niezabezpieczona

```
public void SetStrings(string s1, string s2){
  Atomic.Do(()=> {
    m_string1 = s1;
    Thread.Sleep(1);
    m_string2 = s2;
  }
 );
}
```

Procedura zabezpieczona dyrektywą Atomic

Implementacje: Python, C#, Concurrent Haskell

Zalety:

- Wysoki stopień równoległości - transakcje działające na różnych danych nie przeszkadzają sobie wzajemnie.
- Liczna klasa błędów, w tym zakleszczenie , nie istnieje
- Odzyskujemy strukturalność - złożenie operacji poprawnych jest nadal operacja poprawna.

Wady:
Nie można wykonywać żadnej operacji której skutki nie mogą być odwrócone (np. operacji wejścia – wyjścia). Przezwycięża się to poprzez buforowanie danych, które nie mogą być odwrócone.

Wnioski:

- Programowanie współbieżne z blokadami nie jest zalecane i nie powinno być stosowane poza programami niskopoziomowymi.
- Zamiast programowania z blokadami stosować można programowanie z przekazywaniem komunikatów lub pamięć transakcyjną. Prowadzi to do zwiększenia bezpieczeństwa i efektywności programów.
- Zmiana paradygmatu tworzenia programów równoległych jest konieczna, aby móc w pełni wykorzystać możliwości procesorów wielordzeniowych.