# Systemy operacyjne
## Wzajemne wykluczanie i synchronizacja

[2] **Wyścigi w systemie operacyjnym**

**IPC** - komunikacja między procesami (ang. *InterProcess Communication*)

W SO wykonujące się procesy często dzielą obszary wspólnej pamięci, pliki lub inne zasoby. Należy unikać tzw. wyścigów.

*Def. 1*
**Warunkami wyścigu** (ang. *race conditions*) nazywamy sytuację, w której dwa lub więcej procesów wykonuje operację na zasobach dzielonych, a ostateczny wynik tej operacji jest zależny od momentu jej realizacji.

[3] **Przykład wystąpienia wyścigu**

Przykład

```
void echo()
{
        chin = getchar();
        chout = chin;
        putchar( chout );
}
```

**PROCES 1**

```
1
2 chin = getchar();
3
4 chout = chin;
5 putchar( chout );
6
7
```

**PROCES 2**

```
chin = getchar();
chout = chin;

putchar( chout );
```

[4] **Sekcja krytyczna**

Aby zapobiec warunkom wyścigu należy stworzyć mechanizm zabraniający więcej niż jednemu procesowi dostępu do zasobów dzielonych w tym samym czasie. Należy wprowadzić mechanizm wzajemnego wykluczania *WW* (ang. *mutual exclusion*).

*Def. 2*
**Sekcja krytyczna** - fragment programu, w którym występują instrukcje dostępu do zasobów dzielonych. Instrukcje tworzące sekcje krytyczne muszą być poprzedzone i zakończone operacjami realizującymi wzajemne wykluczanie.

Wybór właściwych operacji realizujących WW stanowi istotę każdego systemu operacyjnego.

[5] **Warunki konieczne implementacji SK**

Dla prawidłowej implementacji sekcji krytycznych muszą być spełnione następujące 3 warunki, przy czym nie czynimy żadnych założeń dotyczących szybkości działania procesów, czy też liczby procesorów:

1. wewnątrz SK może przebywać tylko jeden proces,
2. jakikolwiek proces znajdujący się poza SK, nie może zablokować innego procesu pragnącego wejść do SK,
3. każdy proces oczekujący na wejście do SK powinień otrzymać prawo dostępu w rozsądnym czasie.

[6] **Mechanizmy realizujące wzajemne wykluczanie**

Dwa podejścia:

1. Mechanizmy z aktywnym oczekiwaniem na wejście do SK,
   (a) blokowanie przerwań,
   (b) zmienne blokujące (*niepoprawne*),
   (c) ścisłe następstwo (*niepoprawne*),
   (d) algorytm Petersona,
   (e) instrukcja **TSL**.
2. Mechanizmy z zawieszaniem procesu oczekującego na wejście do SK.
   (a) sleep i wakeup (*niepoprawne*),
   (b) semafory,
   (c) monitory,
   (d) komunikaty.

[7] **Mechanizmy z aktywnym oczekiwaniem**

**1. Blokowanie przerwań (ang. *disabling interrupts*)**

- każdy proces wchodząc do SK blokuje przerwania, a wychodząc odblokowuje,
- zaleta: proces znajdujący się w SK może uaktualnić zawartość zasobów dzielonych bez obawy, że inny proces będzie interweniował.
- wada: jeśli proces na wyjściu z SK nie odblokuje przerwań, to nastąpi upadek systemu; ponadto, w przypadku systemów wieloprocerowych technika nieskuteczna,
- technika blokowania przerwań może być stosowana w jądrze SO przy uaktualnianiu niektórych systemowych struktur danych, lecz nie może być wykorzystywana do realizacji wzajemnego wykluczania w przestrzeni użytkownika.

[8] **2. Zmienne blokujące (ang. *lock variables*)**

Rozwiązanie programowe. Niech będzie dana zmienna dzielona o nazwie *lock*. Niech początkowo lock ma wartość 0. Kiedy proces P chce wejść do SK, to sprawdza wartość lock.

- jeżeli lock = 0, to ustawia lock na 1 i wchodzi do SK;
- jeżeli nie, to proces czeka aż lock stanie się równe 0.

Tak więc:

- lock = 0 oznacza, że nie ma procesu w SK,
- lock = 1 oznacza, że jest w SK.

Rozwiązanie **niepoprawne**, występuje problem wyścigu.

[9] **3. Ścisłe następstwo (ang. *strict alternation*)**

**PROCES 0**

```
1 while( TRUE )
2 {
3     while( turn != 0 )
4         /* wait */;
5     critical_section();
6     turn = 1;
7     noncritical_section();
8 }
```

**PROCES 1**

```
while( TRUE )
{
    while( turn != 1 )
        /* wait */;
    critical_section();
    turn = 0;
    noncritical_section();
}
```

- początkowo turn=0, został naruszony warunek 2. P0 może zostać zablokowany przez P1 znajdujący się poza SK. Stan taki nazywamy **stanem zagłodzenia**.

- rozwiązanie wymaga ścisłego następstwa (przełączania), nie można wydrukować dwóch kolejnych plików przez ten sam proces,
- rozwiązanie **niepoprawne**, wyeliminowany problem wyścigu zastąpiony problemem zagłodzenia.

[10] **4. Algorytm Petersona (I)**

- Łącząc ideę ścisłego następstwa ze zmiennymi blokującymi T. Dekker pierwszy znalazł rozwiązanie (1965) wzajemnego wykluczania. W 1981 r. Peterson znalazł prostsze rozwiązanie tego problemu.
- Każdy proces przed wejściem do SK wywołuje enter_region z własnym numerem jako parametrem, zaś po wyjściu leave_region.

[11] **Algorytm Petersona (II)**

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N]; /* initially 0 */

enter_region(int process)      /* process nr 0 or 1 */
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while( (turn == process) && (interested[other] == TRUE) );
}

leave_region(int process)
{
    interested[process]=FALSE;
}
```

[12] **5. Instrukcja TSL**

Wspomaganie sprzętowe, niektóre komputery są wyposażone w instrukcję **TEST AND SET LOCK** (**TSL**)

- instrukcja TSL wykonuje się niepodzielnie w następujący sposób:
  - czyta zawartość słowa pamięci do rejestru,

  - zapamiętuje wartość rejestru w pamięci,

- operacje czytania i pisania są niepodzielne, tzn. inny proces nie ma dostępu do miejsca pamięci, aż nie nastąpi zakończenie instrukcji TSL,

W celu użycia TSL posłużymy się zmienną dzieloną o nazwie *flag*, przy pomocy której będziemy koordynować dostęp do zasobów dzielonych.

- kiedy flag = 0, to każdy proces może ją ustawić na 1 stosując TSL, a następnie wejść do SK,
- wychodząc z SK ustawia wartość flag na 0 stosując zwykłą instrukcje move.

[13] **Organizacja sekcji krytycznej z wykorzystaniem TSL**

```
1 enter_region:                    leave_region:
2     tsl register, flag               mov flag, #0
3     cmp register, #0                 ret
4     jnz enter_region
5     ret
```

W register inicjalnie 1. Procesy ubiegające się o dostęp do SK muszą wywoływać procedury *enter_region* i *leave_region* we właściwym porządku.

**Wady rozwiązań opartych na koncepcji aktywnego oczekiwania**

- strata czasu procesora,
- możliwość blokady systemu przy wielopriorytetowym sposobie szeregowania procesów, tzw. zjawisko **inwersji priorytetów**.

[14] **Rozwiązania z zawieszaniem procesu oczekującego**

**1. Sleep and Wakeup**

Najprostszym rozwiązaniem jest utworzenie dwóch wywołań systemowych **sleep()** i **wakeup()**.

- wywołanie **sleep()** powoduje, że proces wywołujący zostaje zawieszony do momentu, gdy inny proces nie obudzi danego poprzez wywołanie wakeup(),
- funkcja **wakeup** wywoływana jest z jednym argumentem, numerem procesu, który ma być obudzony.

[15] **Przykład wykorzystania sleep()/ wakeup()**

Problem **producent-konsument** (problem ograniczonego bufora).

Niech dwa procesy dzielą wspólny bufor o skończonym wymiarze. Proces o nazwie producent (Pr) będzie umieszczał informacje (inf) w kolejnych miejscach buforu. Proces o nazwie konsument (Ko) będzie pobierał informacje z tego buforu.

Przyjmujemy założenia:

- jeżeli Pr napotka na bufor pełny, to ma być zawieszony,
- jeżeli Ko napotka na bufor pusty, to ma być zawieszony,

Niech w zmiennej o nazwie *count* będzie zapamiętywana liczba miejsc zajętych w buforze. Ponadto niech maksymalna liczba miejsc będzie równa N.

**Producent**: if( count == N ) { zaśnij } else { dodaj *inf* i count++ }

**Konsument**: if( count == 0 ) { zaśnij } else { pobierz *inf* i count– }

[16] **Producent-Konsument z RC (z wyścigiem)**

```
#define N 100
int count=0;

void producer(void)
{
while (TRUE){
        produce_item();
        if (count == N)
            sleep();
        enter_item();
        count = count + 1;
        if (count == 1)
            wakeup(consumer);
        }
}
```

```
void consumer(void)
{
while (TRUE){
        if (count == 0)
            sleep();
        remove_item();
        count = count - 1;
        if (count == N-1)
            wakeup(producer);
        consume_item();
        }
}
```

**Wada:** sygnał *wakeup* może nie zostać przechwycony i utracony, co prowadzi do blokady.

[17] **2. Semafory: definicja**

- 1965 r. - E. W. Dijkstra proponuje zmienną typu całkowitego do zliczania sygnałów *wakeup*,

- definicja semafora: zmienna nazwana **semaforem**, inicjowana nieujemną wartością całkowitą i zdefiniowana poprzez definicje **niepodzielnych** operacji **P(s)** i **V(s)**:

  **P(S)**: while $S <= 0$ do
  ;
  $S := S - 1$;

  **V(S)**: $S := S + 1$;

- holenderskie P i V od *proberen (testować)* i *verhogen (zwiększać)*, teraz dla semaforów wielowartościowych zazwyczaj *down()/up()*, *wait()/signal()*, a dla semaforów binarnych często *lock()/unlock()*.

[18] **Semafory: realizacja**

```
struct semaphore
{
    int      count;
    queue_t  queue;
}

void down( semaphore s )
{
    s.count--;
    if( s.count < 0 )
    {
        wstaw proces do
            kolejki s.queue;
        zablokuj proces;
    }
}
```

```
void up( semapahore s )
{
    s.count++;
    if( s.count <= 0 )
    {
        usuń jeden z procesów
            z s.queue i wstaw
            do kolejki gotowych;
    }
}
```

[19] **Semafory: algorytm producent-konsument**

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full   = 0;
int count = 0;
```

```
void producer(void)
{
    while (TRUE)
    {
        produce_item();
        down( empty );
        down( mutex );
        enter_item();
        up( mutex );
        up( full );
    }
}
```

```
void consumer(void)
{
    while (TRUE)
    {
        down( full);
        down( mutex );
        remove_item();
        up( mutex );
        up( empty );
        consume_item();
    }
}
```

[20] **Mutex - semafor binarny**

- stosowane, gdy nie trzeba zliczać wystąpień sygnałów a jedynie organizować wzajemne wykluczanie (ang. ***mut****ual* ***ex****clusion*).
- szybka i prosta implementacja np. dla pakietu wątków poziomu użytkownika, w register inicjalnie 1.

```
mutex_lock:
        TSL REGISTER, MUTEX
        CMP REGISTER, #0
        JZE ok
        CALL thread_yield
        JMP mutex_lock
ok:   RET
```

```
mutex_unlock:
        MOVE MUTEX, #0
        RET
```

[21] **3. Monitory**

W celu łatwiejszego pisania programów realizujących wzajemne wykluczanie Hoare (1974) i Hansen (1975) zaproponowali mechanizm synchronizacji wysokiego poziomu zwany **monitorem**.

- **monitor** stanowi zbiór procedur, zmiennych i struktur danych, które są zgrupowane w specjalnym module. W każdej chwili tylko jeden proces aktywny może przebywać w danym monitorze.
- monitory są konstrukcją języka wysokiego poziomu. Za właściwą implementację wzajemnego wykluczania jest odpowiedzialny kompilator.

- w przedstawionej koncepcji brak mechanizmu, który powodowałby zawieszenie procesu,

[22] **Monitory: zapis**

Zapis monitora w hipotetycznych językach.

```
monitor Buffer
  var
    byte    b[100];
    integer head, tail;
  procedure insert( int item )
  begin
    ...
  end;
  procedure remove( int item )
  begin
    ...
  end;
end monitor;
```

```
monitor Buffer
{
    char    b[100];
    integer head, tail;
    public void insert( Item i )
    {
      ...
    }
    public Item remove( void )
    {
      ...
    }
}
```

[23] **Monitory: wstrzymywanie procesu**

Zaproponowano wprowadzenie **zmiennych warunkowych** z dwoma operacjami **wait( zmienna )** oraz **signal( zmienna )**.

- kiedy procedura monitora wykrywa sytuację, że nie może kontynuować obliczeń wykonuje operację *wait* na pewnej zmiennej warunkowej. Proces wykonujący procedurę jest zawieszany.
- inny proces może teraz wejść do sekcji krytycznej. Przy wyjściu wykonuje *signal* w celu obudzenia zawieszonego procesu na zmiennej warunkowej.

Po wywołaniu *signal*:

- Hoare: proces obudzony kontynuuje działanie, a wywołujący jest zawieszany,
- Hansen: proces wywołujący musi natychmiast opuścić monitor.

[24] **Monitory: algorytm producent-konsument (I)**

```
monitor Buffer
    condition full, empty;
    integer count;

    procedure enter;
    begin
        if count = N
            then wait(full);
        enter_item;
        count := count + 1;
        if count = 1
            then signal(empty);
    end;

    procedure remove;
    begin
        if count = 0
            then wait(empty);
        remove_item;
        count := count - 1;
        if count = N-1
            then signal(full);
    end;

    count := 0;
end monitor;
```

[25] **Monitory: algorytm producent-konsument (II)**

```
procedure producer;
begin
    while true do
    begin
            produce_item;
            Buffer.enter;
    end
end;
```

```
procedure consumer;
begin
    while true do
    begin
            Buffer.remove;
            consume_item;
    end
end;
```

**Mechanizm monitorów, własności:**

- wait()/signal() zapobiega możliwości gubienia sygnałów występującej przy sleep()/wakeup(),
- niewiele języków wysokiego poziomu jest wyposażone w monitory (Euclid, Concurrent Pascal)
- część języków posiada mechanizmy niepełne (Java i *synchronized*),
- rozwiązanie nierealizowalne w środowisku rozproszonym ze względu na wymóg dostępności wspólnej pamięci.

[26] **4. Komunikaty (ang. *message passing*)**

Oparte na dwóch wywołaniach systemowych:

- **send**( *destination*, &*message* );

- **receive**( *source*, *&message* );

Sposoby adresowania wiadomości:

1. **adresowanie bezpośrednie**, każdy proces posiada unikatowy adres. Można stosować poniższy mechanizm spotkań:
   - jeżeli *send* wywołane przed *receive*, to proces wysyłający zawieszany do momentu przesłania wiadomości po wywołaniu *receive*,
   - jeżeli *receive* wywołane przed *send*, to proces odbierający zawieszany do momentu przesłania wiadomości po wywołaniu *send*.
2. **adresowanie pośrednie** poprzez skrzynkę pośredniczącą pełniącą funkcję bufora pośredniczącego. Argumentem wywołań *send* i *receive* jest adres skrzynki a nie adres konkretnego procesu.

### [27] Komunikaty: algorytm producent/konsument (I)

Założenia wstępne:

- wiadomości są tego samego rozmiaru,
- wiadomości wysłane lecz nie odebrane są buforowane automatycznie przez system operacyjny,
- użyto N wiadomości, analogicznie do N miejsc dzielonego bufora,
- komunikaty są traktowane jako medium transportowe dla informacji, tzn. mogą być wypełnione bądź bez informacji,
- algorytm rozpoczyna wysłanie przez konsumenta N pustych komunikatów do producenta,
- aby producent mógł wysłać nową informację do konsumenta musi mieć dostępny pusty komunikat odebrany od konsumenta - liczba komunikatów w obiegu między producentem a konsumentem jest stała i niezależna od szybkości produkcji czy konsumpcji informacji.

### [28] Komunikaty: algorytm producent/konsument (II)

```
#define N 100
```

```
void producer( void )
{
    int item;
    message m;



    while( TRUE )
    {
        produce_item( &item );
        receive( consumer, &m );
        build_message( &m, &item );
        send( consumer, &m );
    }
}
```

```
void consumer( void )
{
    int item, i;
    message m;
    for( i = 0;  i < N;  i++ )
        send( producer, &m );

    while( TRUE )
    {
        receive( producer, &m );
        extract_item( &m, &item );
        send( producer, &m );
    }
}
```

[29] **Problem ucztujących filozofów (I)**

- pięciu filozofów przy stole z rozstawionymi pięcioma talerzami i pięcioma widelcami pomiędzy talerzami,
- każdy filozof tylko je i myśli, do jedzenia potrzebuje talerza (z zawartością) i dwóch widelców,
- widelec zasobem dzielonym przez sąsiadujących filozofów,
- jak zorganizować synchronizację?

[30] **Problem ucztujących filozofów (II)**

```
#define N 5

void philosopher( int i )
{
        while( TRUE )
        {
                think()
                take_fork( i );
                take_fork( ( i + 1 ) % N );
                eat();
                put_fork( ( i + 1 ) % N );
```

```
                put_fork( i );
        }
}
```

Rozwiązanie **niepoprawne** - możliwość blokady.

[31] **Problem ucztujących filozofów (III)**

```
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT ( i + N - 1) % N
#define RIGHT ( i + 1 ) % N
int state[N];
semaphore mutex = 1;
semaphore s[N];
```

```
void philosopher(int i) {
        while (TRUE) {
                think( );
                take_forks(i);
                eat( );
                put_forks(i);
        }
}

void take_forks(int i) {
        down(&mutex);
        state[i] = HUNGRY;
        test(i);
        up(&mutex);
        down(&s[i]);
}
```

```
void put_forks(i) {
        down(&mutex);
        state[i] = THINKING;
        test(LEFT);
        test(RIGHT);
        up(&mutex);
}

void test(i) {
        if (state[i] == HUNGRY && \\
                state[LEFT] != EATING && \
                state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

[32] **Problem czytelników i pisarzy**

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void
writer( void )
{
    while( TRUE )
    {
        think_up_data( );
        down( &db );
        write_data_base( );
        up( &db );
    }
}
```

```
void
reader(void)
{
    while( TRUE )
    {
        down( &mutex );
        rc = rc + 1;
        if( rc == 1 )
            down( &db );
        up( &mutex );
        read_data_base( );
        down( &mutex );
        rc = rc - 1;
        if( rc == 0 )
            up(&db);
        up(&mutex);
        use_data_read( );
    }
}
```

[33] **Problem śpiącego fryzjera**

```
#define CHAIRS 5
semaphore customers = 0;        /* ilu siedzi na krzesłach */
semaphore barbers = 0;          /* ilu wolnych, 0 lub 1 */
semaphore mutex = 1;
int waiting = 0;                void customer( void )
                                {
void barber( void )                 down( &mutex );
{                                   if( waiting < CHAIRS )
    while( TRUE )                   {
    {                                   waiting = waiting + 1;
        down( &customers );             up( &customers );
        down( &mutex );                 up( &mutex );
        waiting = waiting - 1;          down( &barbers );
        up( &barbers );                 get_haircut();
        up( &mutex );               } else {
        cut_hair();                     up( &mutex );
    }                               }
}                               }
```