

A wide-angle photograph of a winter landscape. In the foreground, a snow-covered field with tracks leads towards a dense forest of evergreen trees on a rising slope. The sky is filled with large, textured, white clouds, with some blue visible on the left. The overall tone is cold and serene.

Systemy Operacyjne

Synchronizacja procesów

Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*



Kooperacja procesów

Sposoby kooperacji:

- Bezpośrednie współdzielenie przestrzeni adresacji (zarówno kod, jak i dane)
- Współdzielenie danych przez system plików lub komunikaty

Skutki kooperacji:

- Utrata spójności danych
- Wzajemne blokowanie

Zobacz: wykład pt. "Procesy"

Zobacz: wykład pt. "Komunikacja międzyprocesowa"

Producent - konsument - pamięć dzielona

- **Producent** to proces produkujący informację, którą konsumuje **konsument**
- Przykład: kompilator - asembler, asembler - loader, klient - serwer, etc.
- Dwa typy buforów:
 - Nieskończony - konsument czeka, gdy bufor jest pusty; producent zawsze może umieszczać dane,
 - Skończony - konsument czeka, gdy bufor jest pusty; producent czeka, gdy bufor jest pełny.
- `in` - następna wolna pozycja w buforze
- `out` - pierwsza pełna pozycja w buforze
- `in == out` - bufor jest pusty
- `((in + 1) % BUFFER_SIZE) == out` - bufor jest pełny

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



Producent - konsument

```
while (true) {  
    /* produce an item in nextProduced */  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Producent

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in nextConsumed */  
}
```

Konsument

Producent - konsument

```
while (true) {  
    /* produce an item in nextProduced */  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Producent

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in nextConsumed */  
}
```

Konsument

Warunek wyścigu (ang. race condition) to sytuacja, w której dwa lub więcej procesów wykonuje operację na zasobach dzielonych (odczyt lub zapis), a ostateczny wynik tej operacji jest zależny kolejności tego dostępu.

Warunek wyścigu

Niskopoziomowy `count++`

```
register1 = count  
register1 = register1 + 1  
count = register1
```

Niskopoziomowy `count--`

```
register2 = count  
register2 = register2 - 1  
count = register2
```

Niech `count == 5`

T_0 :	producer	execute	<code>register₁ = count</code>	<code>{register₁ = 5}</code>
T_1 :	producer	execute	<code>register₁ = register₁ + 1</code>	<code>{register₁ = 6}</code>
T_2 :	consumer	execute	<code>register₂ = count</code>	<code>{register₂ = 5}</code>
T_3 :	consumer	execute	<code>register₂ = register₂ - 1</code>	<code>{register₂ = 4}</code>
T_4 :	producer	execute	<code>count = register₁</code>	<code>{count = 6}</code>
T_5 :	consumer	execute	<code>count = register₂</code>	<code>{count = 4}</code>



Sekcja krytyczna

Sekcja krytyczna (ang. *critical section*) to segment kodu, w którym proces może zmieniać wartości zmiennych, aktualizować tabele, pisać do pliku, etc. Podstawową własnością sekcji krytycznej jest to, że w tym samym czasie żaden inny proces nie może realizować swojej sekcji krytycznej (obejmującej te same zasoby).

- **Sekcja wejścia** (ang. *entry section*) - segment kodu, w którym zgłaszane jest żądanie dostępu do zasobu celem realizacji wzajemnego wykluczenia.
- **Sekcja krytyczna** (ang. *critical section*)
- **Sekcja wyjścia** (ang. *exit section*) - segment kodu, w którym zgłaszane jest zwolnienie zasobu.
- **Sekcja pozostałego kodu** (ang. *remainder section*) - nie związana z obsługą współdzielenia zasobów część pozostała część kodu.



Sekcje kodu

W ramce oznaczone zostały sekcje sterujące przebywaniem w sekcji krytycznej.

do {

entry section

critical section

exit section

remainder section

} while (TRUE);



Wymagania dot. rozwiązania sekcji krytycznej

Rozwiązanie problemu sekcji krytycznej musi spełniać następujące wymagania:

- **Wzajemne wykluczenie** (ang. *mutual exclusion*) oznacza, że jeśli jeden proces wykonuje swoją sekcję krytyczną, to żaden inny proces nie może wykonać swojej sekcji krytycznej.
- **Postęp** (ang. *progress*) oznacza, że jeśli żaden proces nie jest w sekcji krytycznej i jakiś proces chciałby wejść do swojej sekcji krytycznej, to tylko procesy nierealizujące swojej sekcji kodu pozostałego mogą brać udział w decydowaniu, który z nich wejdzie do swojej sekcji krytycznej.
- **Skończony czas oczekiwania** (ang. *bounded waiting*) oznacza, że czas oczekiwania na wejście do sekcji krytycznej dla każdego procesu powinien być ograniczony.

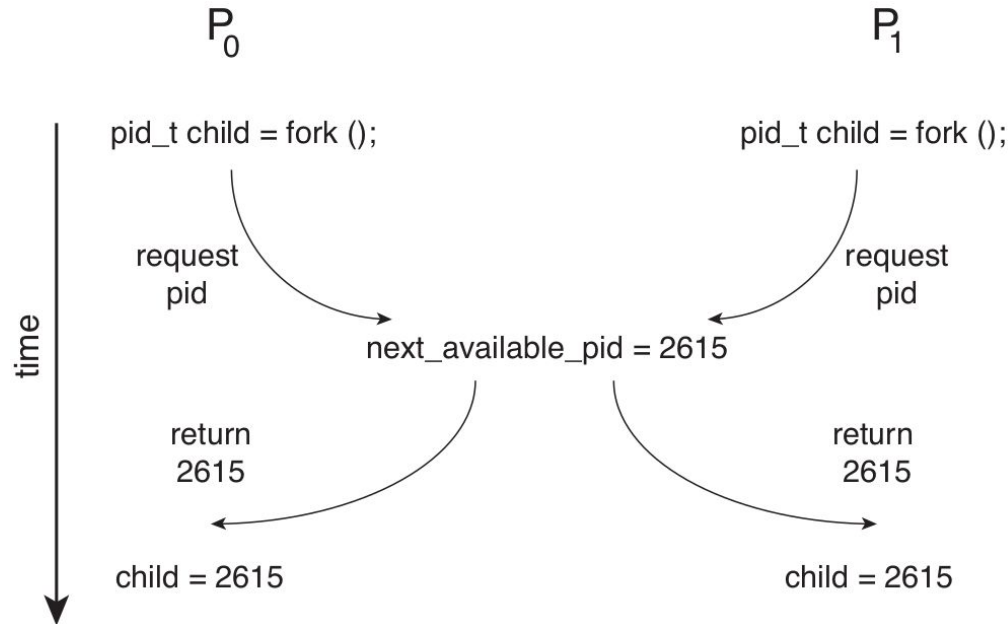


Zakleszczenie, ang. *deadlock*

Zakleszczeniem jest sytuacja, kiedy dwa procesy wzajemnie czekają na zwolnienie zasobów.

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Sekcja krytyczna - przydzielanie PID po `fork()`





Wywłaszczenie jądra (ang. *Kernel preemption*)

Wywłaszczenie - technika, w której planista (algorytm szeregujący zadania, ang. *dispatcher*) może wstrzymać aktualnie wykonywane zadanie, aby umożliwić wykonywanie innemu zadaniu. Zawieszenie (np. zapętlenie) zadania nie powoduje zawieszenia całego systemu.

Wywłaszczenie jądra - jądro pozwala na wywłaszczenie własnego kodu, co oznacza, że w wykonywanie jego kodu może zostać przerwane na czas wykonywania przez procesor innego zadania.

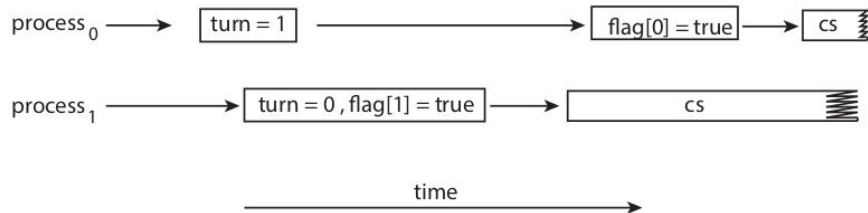
Wywłaszczanie przerw - przerwania są zwykle niewywłaszczalne, dlatego powinny być krótkie.

Cecha	Jądro wywłaszczające	Jądro niewywłaszczające
Definicja	Pozwala na usuwanie i podmianę procesu wykonywanego w trybie kernela, a w efekcie wykonywane jest zadanie o najwyższym priorytecie.	Pozwala na wywłaszczenie procesu wykonywanego w trybie kernela, co oznacza konieczność czekania na jego zakończenie.
Warunek wyścigu	Występuje - wiele procesów jest aktywnych	Nie występuje - jeden proces jest aktywny
Responsywność	Większa i deterministyczna responsywność	Mniejsza i niedeterministyczna responsywność
Implementacja	Skomplikowany projekt i implementacja	Mniej skomplikowany projekt i implementacja
Bezpieczeństwo	Większa stabilność pracy i użyteczność	Mniejsza stabilność pracy i użyteczność
Semafor	Nie wymaga użycia semaforów	Dane współdzielone wymagają semaforów
Programowanie RT	Większa użyteczność w programowaniu RT	Mniejsza użyteczność w programowaniu RT
Wywłaszczanie	Jest	Brak
Przykłady	Linux od 2.6, IRIX, Solaris, NetBSD od v5 Mikrokernele: Windows NT, Vista, 7 i 10	Windows XP, Windows 2000, Linux do 2.4

Wywłaszczający - ang. *preemptive*

Algorytm Peterson'a

- Dwa procesy P_0 oraz P_1
- Dwa procesy współdzielą:
`int turn;`
`boolean flag[2];`
- Zmienna `turn` wskazuje, którego procesu jest kolej na wejście do sekcji krytycznej.
- Tablica `flag` wskazuje, czy proces jest gotowy na wejście do sekcji krytycznej.



do {

```
flag[i] = TRUE;  
turn = j;  
while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

} while (TRUE);



Synchronizacja sprzętowa

Mechanizmy:

- Blokowanie przerwań
- Test and set lock - TSL
- Swap
- TLS + czas oczekiwania

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```




Test and set lock - TSL

- Wymagane wsparcie procesora do realizacji instrukcji atomowych
- Realizacja sekwencyjna instrukcji atomowych (także w przypadku SMP)
- Instrukcja atomowa: TestAndSet()

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}  
  
do {  
    while (TestAndSet(&lock))  
        ; // do nothing  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
} while (TRUE);
```



Swap

- Wymagane wsparcie procesora do realizacji instrukcji atomowych
- Realizacja sekwencyjna instrukcji atomowych (także w przypadku SMP)
- Instrukcja atomowa: Swap()
- Inicjalizacja globalnych zmiennych:

```
boolean waiting[n];  
boolean lock;
```

```
void Swap(boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
do {  
    key = TRUE;  
    while (key == TRUE)  
        Swap(&lock, &key);  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
} while (TRUE);
```



TLS + czas oczekiwania

Test and set lock oraz Swap:

- Spełniają wymaganie wzajemnego wykluczenia, ale
- Nie spełniają wymagania dot. skończonego czasu oczekiwania
- Obok: algorytm spełniający wszystkie wymagania dot. sekcji krytycznej

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
} while (TRUE);
```



Semafor

- Semafor S to zmienna całkowita
- Semafor można modyfikować tylko w:
 - `wait()`
 - `signal()`
- Kiedy jeden proces modyfikuje semafor, żaden inny nie może tego robić
- Testowanie warunku $S \leq 0$ oraz inkrementacja $S--$ musi wykonać się bez przerwania

```
wait(S) {  
    while S <= 0  
        ; // no-op  
    S--;  
}  
  
signal(S) {  
    S++;  
}
```



Typy semaforów

- Semafor binarny (ang. *binary semaphore*)
= **mutex lock** od: *mutual exclusion*, czyli wzajemne wykluczenie
Przy zastosowaniu do sekcji krytycznej:
 - procesy współdzielą semafor, mutex=1,
 - każdy proces działa jak na listingu obok.
- Semafor zliczający (ang. *counting semaphore*)
Zastosowanie do sekcji krytycznej, kiedy dany zasób ma wiele instancji.

```
wait(S) {  
    while S <= 0  
        ; // no-op  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

```
do {  
    wait(mutex);  
  
    // critical section  
  
    signal(mutex);  
  
    // remainder section  
} while (TRUE);
```



Problemy synchronizacyjne

- Problem ograniczonego bufora
- Problem czytelników i pisarzy
- Problem uczujących filozofów



Problem ograniczonego bufora

Założenia:

- Pula buforów, rozmiar puli wynosi n
- Każdy bufor może zawierać jeden obiekt
- Zmienna `mutex` jest semaforem b. do puli
- Na początku `mutex = 1`
- Semaforey `empty` i `full` to liczność pustych/pełnych buforów
- Na początku `empty = n, full = 0`

```
do {  
    . . .  
    // produce an item in nextp  
    . . .  
    wait(empty);  
    wait(mutex);  
    . . .  
    // add nextp to buffer  
    . . .  
    signal(mutex);  
    signal(full);  
} while (TRUE);
```

Producent

```
do {  
    wait(full);  
    wait(mutex);  
    . . .  
    // remove an item from buffer to nextc  
    . . .  
    signal(mutex);  
    signal(empty);  
    . . .  
    // consume the item in nextc  
    . . .  
} while (TRUE);
```

Konsument



Problem czytelników i pisarzy - definicja

Założenia:

- Istnieje współdzielona baza danych
- Dwa typy procesów: piszące i czytające
- Procesy czytające mogą w dowolnej liczbie osiągać dostęp do bazy
- Jeśli jeden proces piszący ma dostęp, w tym czasie żaden inny proces (ani piszący, ani czytający) nie może mieć dostępu

Warianty:

- I. Żaden proces czytający nie czeka na dostęp, chyba, że proces piszący go uzyskał. Ryzyko zagłodzenia pisarzy.
- II. Jeśli pisarz oczekuje na dostęp, żaden czytelnik nie może rozpocząć czytania. Może dojść do zagłodzenia czytelników.



Problem czytelników i pisarzy - rozwiązanie I

- Czytelnicy współdzielą:

```
semaphore mutex, wrt;    // init: 1
int readcount;           // init: 0
```

- wrt jest wspólny także dla pisarzy
- wrt jest muteksem obsługującym pisarzy
- wrt jest także dla pierwszego i ostatniego czytelnika w sekcji krytycznej,
- mutex obsługuje zmienną readcount
- readcount - liczba aktualnie czytających

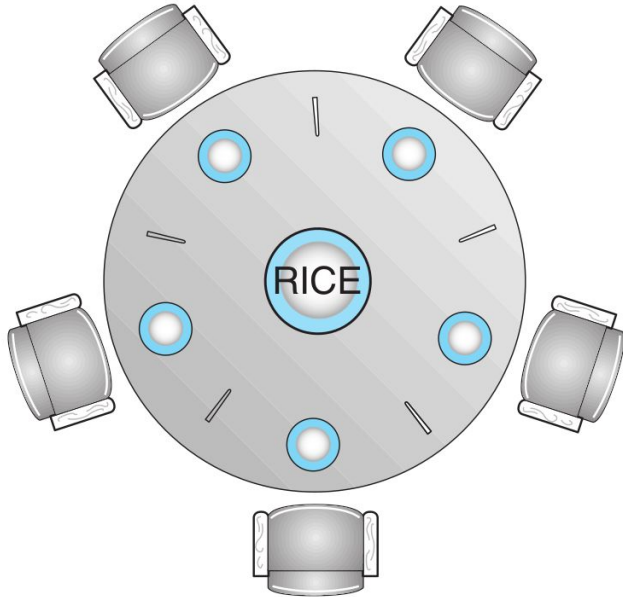
```
do {
    wait(wrt);
    . . .
    // writing is performed
    . . .
    signal(wrt);
} while (TRUE);
```

Pisarz

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
    . . .
    // reading is performed
    . . .
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
} while (TRUE);
```

Czytelnik

Problem ucztujących filozofów - definicja



- Rozważmy pięciu filozofów siedzących przy stole jak na obrazku obok
- Na środku stołu jest miska ryżu, wokół pięć talerzy, po jednym dla każdego filozofa
- Pomiedzy talerzami jest pięć sztućców
- Od czasu do czasu dany filozof chce zjeść
- Aby zjeść muszą być wolne dwa sztućce
- Jedząc filozof ma wyłączność na 2 sztućce
- Po skończeniu jedzenia zwalnia sztućce



Problem ucztujących filozofów - rozwiązanie

- Filozof próbuje wziąć sztućce wywołując:
wait()
- Filozof odkłada sztućce wywołując:
signal()
- Filozofowie współdzielą sztućce:

semaphore chopstick[5]; // init: 1

- Rozwiązania:
 1. Max 4-ch filozofów przy stole
 2. Można podnieść sztućce tylko wtedy, jeśli oba są wolne (podnieść w sekcji krytycznej)
 3. Parzyści filozofowie podnoszą najpierw lewy, potem prawy sztuciec, a nieparzyści odwrotnie: najpierw prawy, potem lewy

```
do {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    // eat  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    // think  
    . . .  
} while (TRUE);
```



Kudłacz