Komunikacja międzyprocesowa (IPC)

Inter-Process Communication

- pamięć współdzielona (*Shared Memory*)
- potoki (*pipes*) - jednokierunkowe przesyłanie strumieni danych
  - potoki nienazwane (*unnamed pipes*) - bezpośrednie połączenie (`stdout-stdin`)
  - potoki nazwane (*named pipes*) - powiązanie przez pliki `FIFO`
- sygnały
- gniazda (*sockets*)
- semafory
- pliki
- kolejki komunikatów (*Message Queues*) - przesyłanie asynchroniczne komunikatów za pomocą buforowanych kolejek
- zdalne wywołanie procedur (*remote procedure call*, RPC)

Problemy IPC

- wyścigi (*race conditions*) - wynik działania zależy od nieprzewidywalnego przeplatania się operacji wykonywanych przez różne procesy
- zakleszczenie (*deadlock*) - dwa (lub więcej) procesy czekają na zasoby, które są trzymane przez siebie nawzajem, co powoduje, że żaden z procesów nie może kontynuować pracy
- zablokowanie (*livelock*) procesy ciągle zmieniają swoje stany w odpowiedzi na działania innych procesów, nie wykonując jednak żadnej efektywnej pracy
- głodzenie (*starvation*) - proces nie otrzymuje zasobów, których potrzebuje do kontynuowania pracy, z powodu nieustannego przydzielania tych zasobów innym procesom
- problem odwróconych priorytetów (*priority inversion*) - proces o wyższym priorytecie jest blokowany przez proces o niższym priorytecie, który trzyma zasób

Sygnały

Sygnały są krótkimi wiadomościami, które można wysyłać do procesu, wątku lub grupy procesów.

- z każdym sygnałem jest związana jego nazwa i numer, np. `SIGINT` (2)
- są programowymi odpowiednikami przerwań sprzętowych, obsługa sygnału odbywa się w ramach procesu w trybie użytkownika
- informują proces lub wątek o wystąpieniu określonego zdarzenia
- zmuszają proces do przerwania aktualnej pracy i wykonania procedury obsługującej sygnał
- jeżeli proces nie zarejestrował funkcji obsługi sygnału to uruchamiana jest domyślna funkcja obsługi (zazwyczaj zamknięcie procesu)
- wyjątki procesora powodują dostarczenie sygnału do procesu (SIGBUS, SIGEMT, SIGFPE, SIGILL, SIGSEGV, SIGTRAP)
- zdefiniowane w normie POSIX (dostępne w Unix, Linux, macOS X, w ograniczony sposób w Windows)

Sygnały

Jądro używa sygnałów, żeby powiadomić proces (zadanie) o wystąpieniu błędów lub asynchronicznych zdarzeń.

Przykłady:

- naruszenie ochrony pamięci (SIGSEGV, *segmentation violation signal*)
- przerwanie klawiaturowe Ctrl-C (SIGINT), Ctrl-Z (SIGSTOP)
- funkcje systemowe `kill()`, `pthread_kill()`
- funkcje biblioteki standardowej C (`signal.h`): `signal()`, `raise()`
- polecenie `kill`

Obsługa sygnału

Działania podejmowane przez proces po otrzymaniu sygnału:

1. wyłapanie sygnału i uruchomienie procedury obsługi
2. zignorowanie sygnału
3. działanie domyślne, gdy nie zdefiniowano procedury obsługi
   - przerwanie (`Term`) - proces jest niszczony
   - zrzut (`Core`) - tworzony jest plik `core`[1] i proces jest niszczony
   - ignorowanie (`Ign`) - sygnał jest pomijany
   - zatrzymanie (`Stop`) - proces jest zatrzymywany (przechodzi w stan TASK_STOPPED)
   - kontynuacja (`Cont`) - proces ze stanu TASK_STOPPED przechodzi w stan TASK_RUNNING

[1] Także `core.$$` lub `/var/lib/systemd/coredump/core.sar.1000...`

Standardowe sygnały

Numer	Nazwa	Domyślnie	Opis
1	**SIGHUP**	`Term`	zamknięcie terminala kontrolującego
2	**SIGINT**	`Term`	przerwanie z klawiatury
3	SIGQUIT	`Core`	zamknięcie z klawiatury
4	SIGILL	`Core`	nieprawidłowa instrukcja
5	SIGTRAP	`Core`	pułapka dla programu śledzącego
6	SIGABRT, SIGIOT	`Core`	nieprawidłowe zakończenie
7	SIGBUS	`Core`	błąd szyny (np. niepoprawny adres fizyczny)
8	SIGFPE	`Core`	wyjątek operacji zmiennoprzecinkowej
9	**SIGKILL**	`Term`	wymuszone zakończenie procesu
10	SIGUSR1	`Term`	do wykorzystania przez proces
11	**SIGSEGV**	`Core`	nieprawidłowe odwołanie do pamięci
12	SIGUSR2	`Term`	do wykorzystania przez proces
13	SIGPIPE	`Term`	zapis do potoku, którego nikt nie czyta
14	SIGALRM	`Term`	alarm upłynięcia czasu z funkcji `alarm()`
15	**SIGTERM**	`Term`	zakończenie procesu

Zobacz: `man 7 signal` (`/usr/include/asm/signal.h`)

Numer	Nazwa	Domyślnie	Opis
16	SIGSTKFLT	`Term`	błąd stosu
17	SIGCHLD	`Ign`	proces potomny zakończony lub zatrzymany
18	**SIGCONT**	`Cont`	wznawianie zatrzymanego procesu
19	**SIGSTOP**	`Stop`	zatrzymanie wykonywania procesu
20	**SIGTSTP**	`Stop`	zatrzymanie procesu z terminala (`Ctrl+Z`)
21	SIGTTIN	`Stop`	program w tle żąda wejścia z terminala
22	SIGTTOU	`Stop`	program w tle żąda wyjścia na terminal
23	SIGURG	`Ign`	pilne dane w gnieździe
24	SIGXCPU	`Core`	przekroczenie limitu czasu procesora
25	SIGXFSZ	`Core`	przekroczenie limitu wielkości pliku
26	SIGVTALRM	`Term`	alarm zegara wirtualnego
27	SIGPROF	`Term`	alarm zegara programu profilującego
28	SIGWINCH	`Ign`	zmiana rozmiaru okna (terminala)
29	SIGIO, SIGPOL	`Term`	gotowość do operacji I/O
30	SIGPWR, SIGLOST	`Term`	awaria źródła zasilania
31	SIGUNUSED, SIGSYS	`Core`	nieużywane (dla wstecznej kompatybilności)

Sygnały `SIGKILL` i `SIGSTOP` nie mogą zostać przechwycone, zablokowane ani ignorowane.

Ryzyka sygnałów

- sygnały są asynchroniczne, obsługa sygnałów może powodować wystąpienie wyścigu (*race conditions*)
- inny sygnał może zostać dostarczony do procesu podczas wykonywania procedury obsługi sygnału
- sygnały mogą spowodować przerwanie trwającego (np. zablokowanego) wywołania systemowego
- procedury obsługi sygnałów powinny być napisane w sposób nie powodujący niepożądanych skutków ubocznych (np. zmiana globalnych zmiennych procesu, użycie funkcji niebezpiecznych z punktu widzenia asynchronicznego wykonania)
- z punktu widzenia jądra wykonanie kodu obsługi sygnału jest dokładnie takie samo, jak wykonanie dowolnego innego kodu przestrzeni użytkownika (odpowiedzialność programisty)

Maskowanie sygnałów

- proces ma możliwość zamaskować (zablokować) czasowo wybrane sygnały w celu wykonania krytycznych operacji (maska jest zasobem procesu)
- sygnały standardowe nie są kolejkowane, tylko jeden sygnał danego typu dociera po odblokowaniu
- przy odblokowaniu kilku sygnałów różnego typu kolejność obsługi jest nieokreślona
- proces może zablokować się w oczekiwaniu na sygnał (obsługa synchroniczna)

Sygnały czasu rzeczywistego w Linux

Linux używa 31 standardowych sygnałów i do 32 sygnałów czasu rzeczywistego (od 32 do 64)

Sygnały czasu rzeczywistego:

- nie mają zdefiniowanego zastosowania, programiści mogą ich używać według własnego uznania
- domyślne działanie to przerwanie wykonywania procesu
- mogą być powiązane z danymi dostarczanymi do procesu (liczba lub wskaźnik)
- umożliwiają odczytanie identyfikatora procesu i użytkownika nadawcy
- są kolejkowane, kolejność dostarczenia jest zagwarantowana, im niższy numer sygnału, tym wyższy priorytet

Wysyłanie sygnałów z poziomu powłoki

- wysyłanie sygnałów do procesów
  `kill -SIGNAL PID`
- zatrzymywanie i wznawianie procesów
  `kill -STOP PID`
  `kill -CONT PID`
- zamknięcie procesu (domyślnie sygnał `SIGTERM`)
  `kill PID`
- zabicie procesu
  `pkill -9 top`
- zabicie wszystkiego co się da
  `kill -1 -9`

Sygnał `SIGKILL/9` nie trafia do wskazanego procesu, ale do procesu `init/systemd`, dlatego nie może być przechwycony ani zignorowany przez proces docelowy.

Zasoby podzielne i niepodzielne

Zasoby niepodzielne: zasoby, które w danym sposobie użycia wymagają wyłącznego dostępu, w danej chwili korzysta z nich co najwyżej jeden proces lub wątek

- niektóre urządzenia zewnętrzne (np. drukarka), pliki do zapisu, kolidujące obszary pamięci współdzielonej (np. zmienne globalne aplikacji wielowątkowej), itp.

Zasoby podzielne: zasoby, z których wiele procesów lub wątków może korzystać współbieżnie bez ryzyka powstania błędów lub niespójności danych

- pliki tylko do odczytu, niemodyfikowalny kod biblioteki współdzielonej, kody programów, itp.

Wyścigi i sekcje krytyczne

- Współbieżny dostęp do wspólnych danych lub zasobów może wymagać synchronizacji, jeżeli operacje procesów lub wątków mogą ze sobą kolidować
- **Wyścig** (*race condition*) występuje, gdy poprawność działania zależy od niekontrolowanej kolejności lub czasu wykonania współbieżnych operacji. Np. dwa procesy próbujące jednocześnie zaktualizować tę samą zmienną współdzieloną.
  Wyścigi nazywane są również **hazardem** lub **przeplotem** operacji.
- **Sekcja krytyczna** (*critical section*) - fragment programu, który korzysta ze współdzielonych danych lub zasobów i którego wykonanie wymaga ochrony przed kolidującymi operacjami innych procesów lub wątków
- Aby zapobiec wyścigom musimy zapewnić **wzajemne wykluczenie**

Przykład: zmiana zmiennej współdzielonej

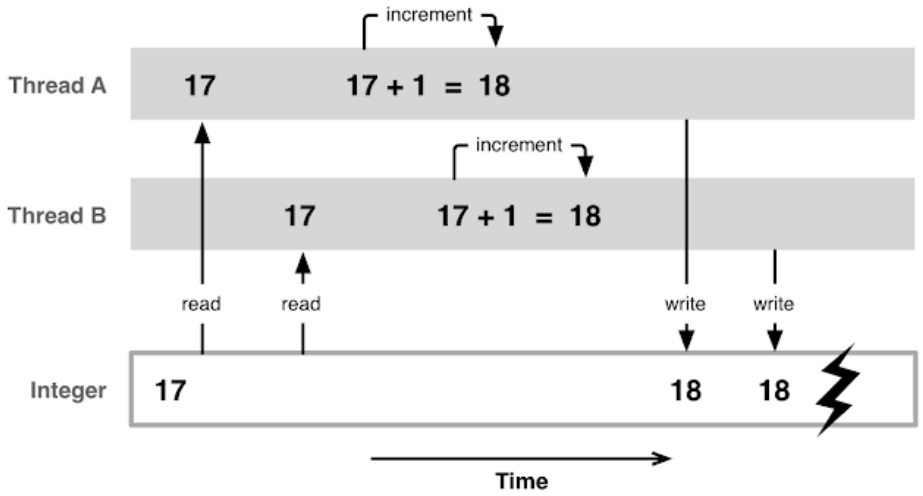
Dwa wątki współdzielą zmienną globalną typu całkowitego `n` i wykonują kod `n++`, co oznacza

1. `pobierz aktualną wartość n i umieść ją w rejestrze`
2. `dodaj 1 do wartości w rejestrze`
3. `zapisz nową wartość n do pamięci`

```
Wątek A                              Wątek B
------------------------             ------------------------
pobierz n (17)                        ...
zwiększ wartość n (17->18)             ...
zapisz n (18)                         ...
...                                  pobierz n (18)
...                                  zwiększ wartość n (18->19)
...                                  zapisz n (19)
```

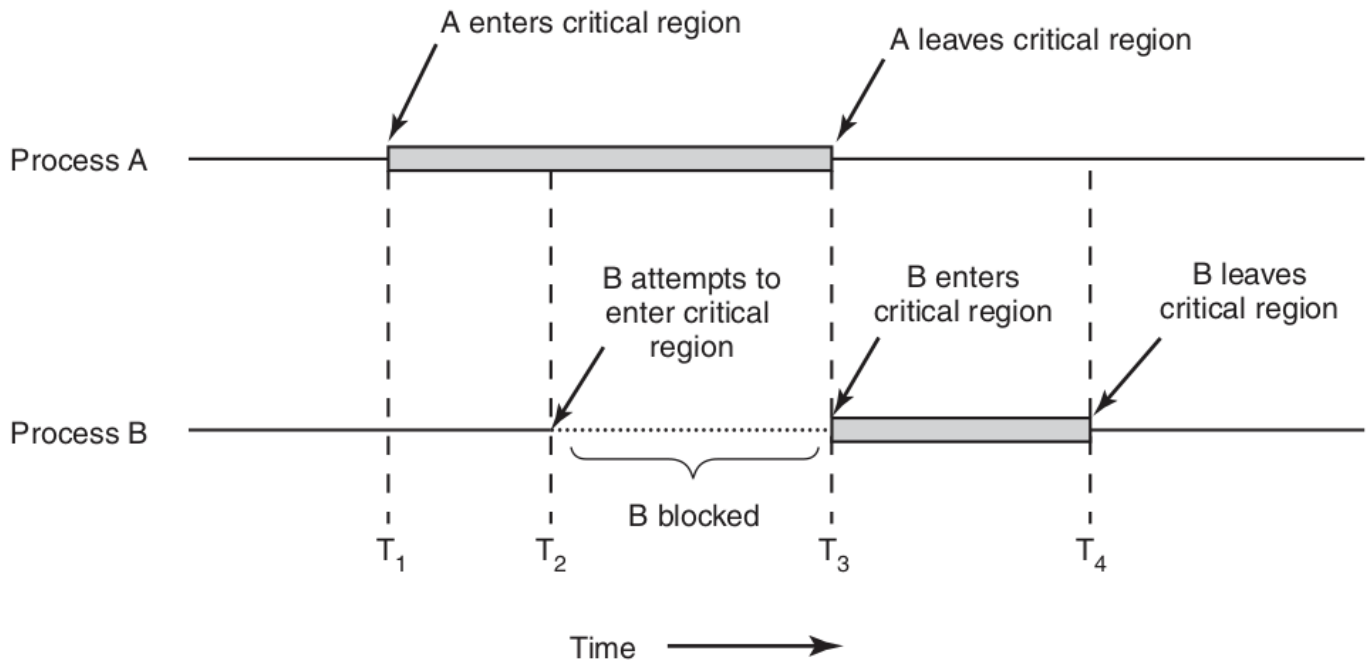
```
Wątek A                              Wątek B
------------------------             ------------------------
pobierz n (17)                        ...
...                                  pobierz n (17)
zwiększ wartość n (17->18)             ...
...                                  zwiększ wartość n (17->18)
zapisz n (18)                         ...
...                                  zapisz n (18)
```

Wyścigi i operacje atomowe



- Zwiększanie wartości zmiennej globalnej powinno być wykonywane w sposób niepodzielny (atomowy) względem innych operacji
- **Operacja atomowa** (*atomic operation*) to niepodzielna operacja, która jest wykonywana w całości, nie można zaobserwować jej częściowego wykonania ani wstawić kolidującej operacji pomiędzy jej etapy. Zapewnia poprawne wykonanie operacji na zasobach współdzielonych

Wzajemne wykluczenie



Wzajemne wykluczenie (*mutual exclusion*) polega na zapewnieniu takich warunków działania systemu, by w danej chwili co najwyżej jeden proces lub wątek mógł korzystać z zasobów niepodzielnych, tj. tylko jeden proces mógł przebywać w sekcji krytycznej

Wymagania dotyczące wzajemnego wykluczenia

1. żadne dwa procesy nie mogą przebywać jednocześnie w sekcji krytycznej chroniącej dany zasób
2. nie można czynić żadnych założeń, co do szybkości procesów i liczby procesorów
3. żaden proces wykonujący się poza sekcją krytyczną nie może blokować innych procesów
4. żaden proces nie może czekać w nieskończoność na wejście do swojej sekcji krytycznej
5. jeśli żaden proces nie przebywa w sekcji krytycznej, to dowolny proces, który chce do niej wejść, powinien otrzymać na to natychmiastową zgodę
6. proces pozostaje w sekcji krytycznej przez skończony czas

Jak zapewnić wzajemne wykluczanie?

- **wsparcie sprzętowe** dla operacji niepodzielnych
  - wyłączanie przerwań
  - atomowe instrukcje procesora: testuj i załóż rygiel TSL (*Test and Set Lock*), zamiana XCHG (*exchange*)
- **aktywne czekanie**
  - zmienne blokujące
  - blokady wirujące (*spin locks*)
  - algorytm Petersona
- **blokowanie oczekujących procesów** (bez aktywnego czekania)
  - semafory, muteksy,
  - monitory
- inne: synchronizacja za pomocą komunikatów, koordynowanie dostępu przez właściciela zasobu

Wyłączanie przerwań

Kod jądra może na krótko wyłączyć lokalne przerwania maskowalne przed sekcją krytyczną, a po jej zakończeniu przywrócić ich poprzedni stan. Zapobiega to przerwaniu tej sekcji przez lokalną obsługę takich przerwań.

Ograniczenia:

- proces użytkownika nie powinien mieć możliwości wyłączania przerwań, technika przydatna tylko dla kodu jądra
- w systemach wieloprocesorowych wyłączanie przerwań dotyczy tylko jednego procesora, nie chroni przed dostępem z innych procesorów
- opóźnia obsługę przerwań, co może wpływać na czas reakcji systemu
- wymaga ostrożności wobec wywłaszczania i blokowania procesów w sekcji krytycznej

Aktywne czekanie (*busy waiting*) - proces lub wątek wielokrotnie sprawdza w pętli, czy warunek synchronizacji został spełniony, zamiast blokować się na czas oczekiwania.

Zalety:

- proste w implementacji
- możliwość szybkiej reakcji bez kosztu uśpienia i wznowienia wątku, przydatne, gdy oczekiwanie na zdarzenie jest krótkie i nieopłacalne jest przełączanie kontekstu

Wady:

- oczekujący wątek zużywa czas procesora i energię, co jest szczególnie niekorzystne przy długim oczekiwaniu

Wzajemne wykluczenie z aktywnym czekaniem

- **zmienne blokujące** (*lock variables*) - wskazują, czy blokada jest wolna, czy zajęta (np. przez inny proces w sekcji krytycznej); proste w użyciu, ale bez operacji niepodzielnych wciąż podatne na wyścigi
- **blokada wirująca** (*spinlock*) to blokada zapewniająca wzajemne wykluczanie, na której zwolnienie wątek oczekuje aktywnie, sprawdzając w pętli stan blokady. Po uzyskaniu dostępu wątek atomowo przejmuje blokadę
- **ścisła zmienność** (*strict alternation*) - proste rozwiązanie wzajemnego wykluczenia dla dwóch wątków z użyciem zmiennej określającej dostęp do sekcji krytycznej na zmianę
- **algorytm Petersona** - programowe rozwiązanie wzajemnego wykluczania

Ścisła zmienność

Procesy uzyskują dostęp naprzemiennie, wartość zmiennej współdzielonej `turn` określa, który proces może wejść do sekcji krytycznej

```
          proces A                                proces B
-------------------------------      ---------------------------------
while (TRUE) {                        while (TRUE) {
  while (turn != 0) /* wait */;         while (turn != 1)  /* wait */;
  critical section();                   critical section();
  turn = 1;                             turn = 0;
  noncritical_section();                noncritical_section();
}                                     }
```

Problem:

- wykonanie sekcji krytycznej jest naprzemienne - tempo zależy od wolniejszego procesu
- gdy A zakończy sekcję i ustawi `turn = 1`, nie może wejść ponownie, dopóki B nie wykona swojej sekcji, nawet jeśli B nie chce z niej korzystać, a sekcja jest wolna.
- w przypadku awarii jednego z procesów drugi proces może zostać zablokowany na zawsze

Algorytm Petersona dla dwóch procesów

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */
int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially zero (FALSE) */

void enter_region (int process)       /* process: who is entering (0 or 1) */
{
  int other = 1 - process;            /* number the opposite of process */
  interested[process] = TRUE;         /* show that you are interested */
  turn = process;
  while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)        /* who is leaving (0 or 1) */
{
  interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Algorytm Petersona

- spełnia wszystkie wymogi wzajemnego wykluczania
- globalna tablica `interested[]` wskazuje, czy dany proces chce wejść do sekcji krytycznej
- zmienna `turn` wskazuje, który proces ma pierwszeństwo wejścia do sekcji krytycznej, w sytuacji konfliktu, gdy oba procesy chcą wejść jednocześnie
- zapobiega wzajemnemu blokowaniu się procesów
- algorytm może być uogólniony na dowolną liczbę procesów

Instrukcja maszynowa TSL

- rozkaz `tsl` kopiuje wartość zmiennej blokującej do rejestru i jednocześnie zmienia jej wartość; jest to gwarantowana sprzętowo **operacja niepodzielna**
- sprzęt zapewnia atomowośc operacji; przykładowo na czas wykonywania instrukcji `tsl` procesor może zablokować szynę pamięci, aby żaden inny proces nie miał do niej dostępu

Przykład blokady wirującej (*spin lock*) z użyciem instrukcji TSL:

```
enter_region:
  tsl register,lock   | skopiuj lock do rejestru i ustaw lock na 1
  cmp register,#0     | czy lock było równe 0?
  jne enter_region    | jeśli nie, blokada była zajęta - ponów próbę
  ret                 | powrót; wejście do sekcji krytycznej

leave_region:
  mov lock,#0         | ustaw lock na 0
  ret                 | powrót do procedury wywołującej
```

Wzajemne wykluczenie instrukcją TSL lub XCHG

- eliminuje problem wyścigu przy dostępie do zmiennej blokującej
- nadaje się dla dowolnej liczby procesów (maszyny jedno- i wieloprocesorowe (SMP))
- proste i łatwe w weryfikacji
- nadaje się do kontroli wielu sekcji krytycznych, każda z własną blokadą
- wykorzystywane w implementacjach blokad w systemach operacyjnych (np. semaforów) i mechanizmów synchronizacji procesów

Ograniczenia:

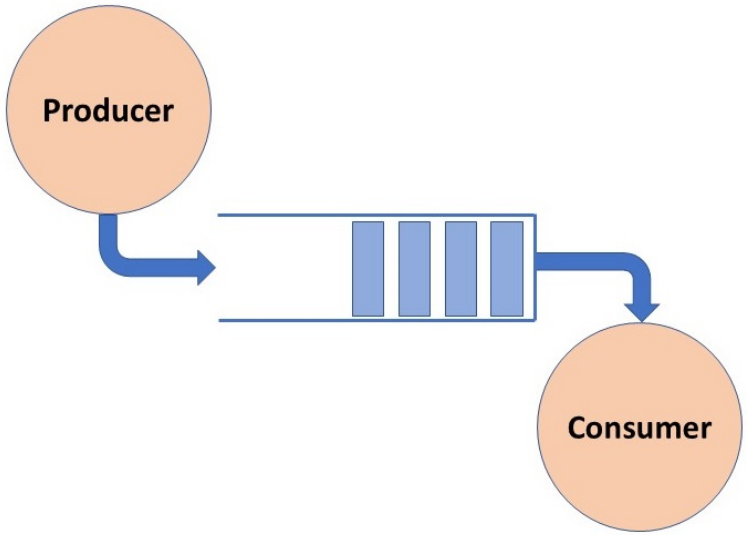
- blokada wirująca wykorzystuje aktywne czekanie, a to marnuje czas CPU
- możliwość wystąpienia głodzenia (*starvation*), np. gdy procesy o wysokim priorytecie ciągle przejmują dostęp do sekcji krytycznej
- możliwość wystąpienia zakleszczeń (*deadlock*) oraz problemu odwróconego priorytetu

Wzajemne wykluczenie bez aktywnego czekania

Można uniknąć marnowania czasu jednostki centralnej na jałowe sprawdzanie warunku, jeśli system udostępnia dwie operacje do synchronizacji pracy procesów: `SLEEP` oraz `WAKEUP`.

- `SLEEP` – odwołanie do systemu, które powoduje zablokowanie procesu wywołującego tę funkcję do czasu, aż nie nadejdzie sygnał budzenia
- `WAKEUP` – odwołanie do systemu służące do budzenia procesów

Problem producenta i konsumenta (ograniczonego buforu)



- proces producenta nie może wstawić przedmiotu, jeśli współdzielony bufor jest pełny
- proces konsumenta nie może pobrać przedmiotu, jeśli współdzielony bufor jest pusty
- dostęp do współdzielonego bufora musi być zabezpieczony wzajemnym wykluczeniem

Próba rozwiązania problemu producenta i konsumenta

```
#define N     100                       /* number of slots in the buffer */

int count = 0;                          /* number of items in the buffer */

void producer(void)
{
  int item;

  while (TRUE) {                          /* repeat forever */
    produce_item(&item);                  /* generate next item */
    while (count == N) sleep();              /* if buffer is full go to sleep */
    enter_item(item);                     /* put item in buffer */
    count = count + 1;                    /* increment number of items in buffer */
    if (count == 1) wakeup(consumer);     /* was buffer empty? */
  }
}
```

```
void consumer(void)
{
  int item;

  while (TRUE) {                          /* repeat forever */
    while (count == 0) sleep();              /* if buffer is empty go to sleep */
    remove_item(&item);                   /* take item out of buffer */
    count = count - 1;                    /* decrement number of items in buffer */
    if (count == N-1) wakeup(producer); /* was buffer full? */
    consume_item(item);                   /* print item */
  }
}
```

Problemy:

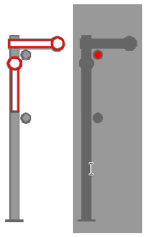
- zmienna `count` nie jest chroniona, w przypadku wyścigu może dojść do zablokowania producenta i konsumenta
- możliwe jest utracone przebudzenie: konsument sprawdza `count == 0`, producent wstawia przedmiot i wywołuje `wakeup`, zanim konsument zdąży zasnąć. Konsument następnie zasypia mimo niepustego bufora.

Potrzebne są mechanizmy synchronizacji, takie jak semafory.

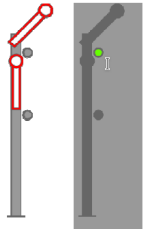
Semafory

Semafor to specjalna zmienna całkowita, na której są wykonywane operacje niepodzielne `DOWN` i `UP`[2]

`SLEEP` ⟶ `DOWN`



`WAKEUP` ⟶ `UP`



- DOWN (opuszczenie, oczekiwanie) - zmniejsza wartość semafora o 1, jeśli wartość semafora jest większa od 0, w przeciwnym razie blokuje proces wywołujący tę operację
- UP (podniesienie, sygnalizacja) - zwiększa wartość semafora o 1 i budzi procesy oczekujące na semaforze

[2] Oryginalne nazwy operacji używane przez Edsgera Dijkstrę to `P` (*Proberen*, sprawdź) i `V` (*Vorhogen*, podnieś). W literaturze używa się też nazw `wait` i `signal`.

- Wartość semafora oznacza liczbę procesów, które mogą wejść do sekcji krytycznej (liczba pozwoleń)
- Operacje `UP` i `DOWN` muszą być **niepodzielne** (atomowe).
  Realizowane jest to w postaci wywołań systemowych blokujących przerwania i instrukcją TSL dla systemów wieloprocesorowych
- semafor jest wysokopoziomową abstrakcją opartą na niskopoziomowych mechanizmach elementarnych, które zapewniają niepodzielność i dostarczają mechanizmów wstrzymywania

Wady:

- wstrzymywanie i wznawianie wymaga przełączania kontekstu i zmian w kolejkach modułu szeregowania i kolejek wątków wstrzymanych; operacje na semaforze są **powolne**
- trudne do debugowania

Operacje UP i DOWN na semaforze

Definicje operacji opuszczania (DOWN) i podnoszenia (UP) semafora:

```
void DOWN(semaphore) {
   if (semaphore > 0 ) {
       semaphore = semaphore - 1;
       continue();
   } else {
       sleep();  /* add the process to semaphore queue and block it */
   }
}

void UP(semaphore) {
   semaphore = semaphore + 1;
   wakeup(); /* wakeup all blocked processes and make them runnable */
             /* scheduler selects one to run */
}
```

Problem producenta i konsumenta z semaforami

```
#define N     100                    /* number of slots in the buffer */
                                     /* semaphores are a special kind of int */
semaphore mutex = 1;                 /* controls access to critical section */
semaphore empty = N;                 /* count empty buffer slots */
semaphore full  = 0;                 /* count full buffer slots */

void producer(void)
{
  int item;
  while (TRUE) {                     /* TRUE is the constant 1 */
    produce_item(&item);             /* generate next item */
    down(&empty);                    /* decrement empty count */
    down(&mutex);                    /* enter critical region */
    enter_item(&item);               /* put new item in buffer */
    up(&mutex);                      /* leave critical region */
    up(&full);                       /* increment count of full slots */
  }
}
```

```
void consumer(void)
{
  int item;
  while (TRUE) {                    /* infinite loop */
    down(&full);                    /* decrement full count */
    down(&mutex);                   /* enter critical region */
    remove_item(&item);             /* take item from buffer */
    up(&mutex);                     /* leave critical area */
    up(&empty);                     /* increment count of empty slots */
    consume_item(&item);            /* do sth with the item */
  }
}
```

- semafory `empty` oraz `full` służą do **synchronizacji**, zapewniają, że zdarzenia będą zachodziły we właściwej kolejności
- semafor `mutex` przyjmuje wartości 0 lub 1, jest używany przez dwa lub więcej procesów, aby zapewnić że tylko jeden z nich może w danej chwili przebywać w swojej sekcji krytycznej
- `empty` i `full` reprezentują dostępne pozwolenia na wstawienie i pobranie przedmiotu; oczekiwanie na `empty` lub `full` odbywa się przed zajęciem `mutex`

Rodzaje semaforów

- **semafor zliczający (ogólny)** - wartość całkowita służąca do sygnalizacji między procesami (synchronizacja procesów, zarządzanie ograniczonymi zasobami)
- **semafor binarny** - semafor, który przyjmuje tylko wartości 0 lub 1
- **mutex** - semafor binarny, który zapewnia wzajemne wykluczanie procesów (ang. *mutual exclusion*), proces zamykający mutex jest tym samym, który go odblokuje

W semaforach jest utrzymywana kolejka procesów oczekujących na semafor:

- **semafor silny** (Dijkstry) - kolejka FIFO, procesy oczekujące na semafor są budzone w kolejności ich przybycia (sprawiedliwy, zapobiega głodzeniu)
- **słaby semafor** - procesy mogą być budzone w dowolnej kolejności

Monitor to mechanizm synchronizacji, który łączy dane współdzielone z operacjami na tych danych i zapewnia wzajemne wykluczanie podczas ich wykonywania.

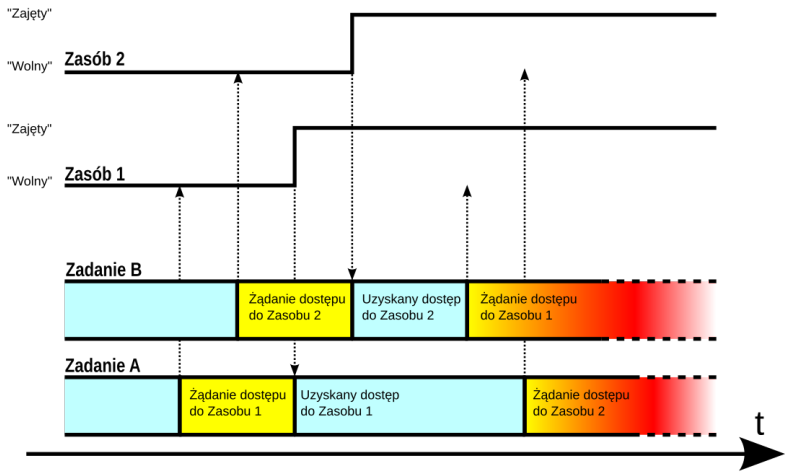
- Dane są dostępne przez procedury monitora; **tylko jeden wątek** może wykonywać jego kod, posiadając blokadę monitora.
- **Zmienne warunkowe** (*condition variables*) umożliwiają oczekiwanie na zmianę stanu monitora.
- **Oczekiwanie** (`wait`) zwalnia blokadę i usypia wątek; przed powrotem wątek ponownie przejmuje blokadę.
- **Sygnalizowanie** (`signal`) powiadamia oczekujący wątek o możliwej zmianie stanu. Po wznowieniu należy ponownie sprawdzić warunek.
- Implementowane często w postaci klas lub obiektów, które enkapsulują dane i metody synchronizujące dostęp do nich.
- W języku Java każdy obiekt ma powiązany monitor udostępniający metody `wait`, `notify` i `notifyAll`, którego blokadę można przejąć za pomocą `synchronized`.

Przykład monitora w C# (producent-konsument)

```
using System.Threading;
class Bufor {
    private readonly object _lock = new object();    // Obiekt używany do synchronizacji
    private int[] _dane = new int[3];
    private int _licznik = 0;

    public void Produkuj(int element) {
        lock (_lock) {
            while (_licznik == _dane.Length) Monitor.Wait(_lock);  // Czekaj, jeśli bufor pełny
            _dane[_licznik++] = element;
            Monitor.PulseAll(_lock);                               // Powiadom oczekujące wątki
        }
    }
    public int Konsumuj() {
        lock (_lock) {
            while (_licznik == 0) Monitor.Wait(_lock);             // Czekaj, jeśli bufor pusty
            int element = _dane[--_licznik];
            Monitor.PulseAll(_lock);                               // Powiadom oczekujące wątki
            return element;
        }
    }
}
```

```
class Program {
    static void Main() {
        Bufor bufor = new Bufor();

        // Uruchom producenta
        new Thread(() => {
            for (int i = 0; i < 5; i++) {
                bufor.Produkuj(i);
                Thread.Sleep(300); // Symulacja pracy
            }
        }).Start();

        // Uruchom konsumenta
        new Thread(() => {
            for (int i = 0; i < 5; i++) {
                bufor.Konsumuj();
                Thread.Sleep(500); // Symulacja pracy
            }
        }).Start();
    }
}
```

Zakleszczenia (*deadlocks*, *deadly embraces*)



Zakleszczenie to trwałe zablokowanie zbioru procesów, spowodowane wzajemnym oczekiwaniem na zasoby zajęte przez inne procesy z tego zbioru.

Przykład: proces A otrzymuje dostęp do drukarki, a proces B do dysku, a następnie proces A żąda dostępu do dysku, a proces B do drukarki.

https://pl.wikipedia.org/wiki/Zakleszczenie

Unikanie zakleszczeń

W odróżnieniu od innych problemów zarządzania procesami współbieżnymi nie istnieje skuteczne i ogólne rozwiązanie problemu zakleszczeń.

Jak unikać zakleszczeń?

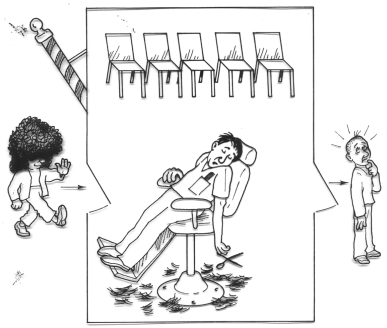
1. Jeśli to możliwe, **należy unikać stosowania blokad** (używaj operacji atomowych lub struktur danych nie wymagających zakładania blokad).
2. Jeśli blokada jest niezbędna, to **należy stosować blokadę pojedynczą**
3. Jeśli potrzebnych jest kilka blokad należy:
   - zdefiniować (małą) lokalną **hierarchię blokad** (blokowanie w ustalonej kolejności)
   - ustawić limity czasu blokady (`Monitor.TryEnter()` w C#)
   - zastosować **blokowanie stochastyczne** (losowa kolejność blokowania zmniejszy prawdopodobieństwo zakleszczenia)

Klasyczne problemy komunikacji międzyprocesowej

- problem producenta-konsumenta – modelowanie procesów synchronizacji między procesami (problem ograniczonego buforu)
- problem ucztujących filozofów – modelowanie procesów, które współzawodniczą w dostępie do ograniczonych zasobów, np. urządzeń wej/wyj
- problem pisarzy i czytelników – zagadnienie zapewnienia prawidłowego dostępu do bazy danych (w trybie odczytu i zapisu)
- problem śpiącego fryzjera – modelowanie procesów synchronizacji między procesami (problem ograniczonej liczby zasobów usługowych)

Problem śpiącego fryzjera

- W salonie fryzjerskim jest jeden fryzjer, jedno krzesło do strzyżenia i kilka krzeseł dla oczekujących klientów.
- Jeśli nie ma klientów, fryzjer zasypia na swoim krześle.
- Gdy przychodzi klient, budzi fryzjera, jeśli ten śpi. Jeśli fryzjer jest zajęty, a są wolne krzesła dla oczekujących, klient siada i czeka. Jeśli nie ma wolnych krzeseł, klient odchodzi.
- Fryzjer strzyże jednego klienta naraz i po skończeniu strzyżenia sprawdza, czy są jacyś oczekujący klienci. Jeśli tak, woła następnego klienta. W przeciwnym razie zasypia.



Problemy do rozwiązania:

- Zapewnienie wzajemnego wykluczania dostępu do krzesła fryzjera
- Zapobieganie głodzeniu klientów (jeśli ciągle przychodzą nowi klienci, ci, którzy już czekają, mogą nigdy nie dostać dostępu do fryzjera)

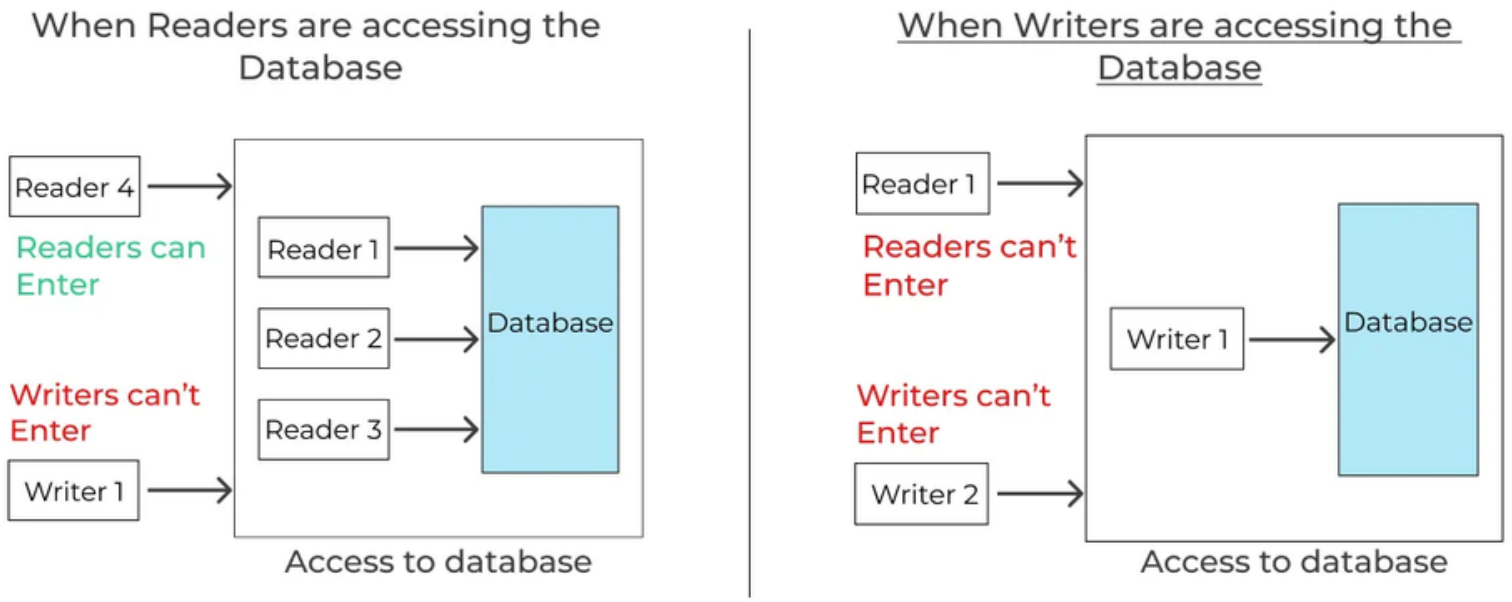
Problem czytelników i pisarzy

- Obszar danych współdzielonych (pamięć, plik, baza danych) jest użytkowany przez procesy czytające (czytelników) i procesy modyfikujące (pisarzy).
- Dane mogą być jednocześnie odczytywane przez wielu czytelników.
- Dane mogą być modyfikowane tylko przez jednego pisarza w danym czasie.
- Jeśli pisarz modyfikuje bazę danych, żaden czytelnik nie może jej czytać.

Problemy do rozwiązania:

- Zapewnienie wzajemnego wykluczania dostępu do bazy danych
- Zapobieganie głodzeniu pisarzy (jeśli ciągle przychodzą czytelnicy, pisarz może nigdy nie dostać dostępu do bazy danych)

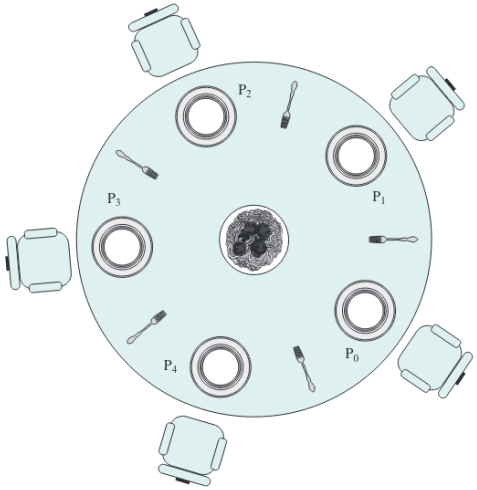
Problem czytelników i pisarzy



https://prepinsta.com/operating-systems/readers-writers-problem/

Problem ucztujących filozofów

- Pięciu filozofów siedzi przy okrągłym stole. Każdy filozof może myśleć lub jeść.
- Między sąsiadującymi filozofami znajduje się jeden widelec. Aby jeść, filozof musi mieć oba widelce znajdujące się po jego lewej i prawej stronie.
- Problem polega na zaprojektowaniu protokołu, który pozwoli filozofom jeść i myśleć bez ryzyka zakleszczenia (gdy każdy filozof trzyma jeden widelec i czeka na drugi) lub głodzenia (gdy jeden lub więcej filozofów nigdy nie dostaje szansy na jedzenie).



Przykład próby rozwiązania problemu semaforami

```
semaphore fork[5] = {1};

void philosopher(int i) {
    while (true) {
        think();
        down(fork[i]);
        down(fork[(i+1)%5]);
        eat();
        up(fork[i]);
        up(fork[(i+1)%5]);
    }
}
```

Problem: możliwe zakleszczenie, gdy każdy filozof trzyma jeden widelec i czeka na drugi.

Rozwiązanie problemu filozofów semaforami

```
semaphore fork[5] = {1};
semaphore room = 4;

void philosopher(int i) {
    while (true) {
        think();
        down(room);
        down(fork[i]);
        down(fork[(i+1)%5]);
        eat();
        up(fork[i]);
        up(fork[(i+1)%5]);
        up(room);
    }
}
```

Semafor `room` ogranicza liczbę filozofów, którzy mogą jednocześnie próbować podnieść widelce. Dzięki temu zawsze przynajmniej jeden filozof może zdobyć oba widelce i rozpocząć jedzenie.

Pytania sprawdzające

- Jaką sytuację nazywamy wyścigiem (*race condition*)?
- Co to jest semafor? Wymień operacje, które można wykonać na semaforach.
- Do czego służy algorytm Petersona?
- Jaka jest różnica między mutexem a semaforem binarnym?
- Na czym polega aktywne czekanie? Podaj główną wadę takiego rozwiązania.