Komunikacja międzyprocesowa (IPC)

Inter-Process Communication

- pamięć współdzielona (*Shared Memory*)
- potoki (*pipes*) - jednokierunkowe przesyłanie strumieni danych
  - potoki nazwane (*named pipes*) - dla procesów niepowiązanych
- 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 procesów czekaj na zasoby, które są trzymane przez siebie nawzajem, co powoduje, że żaden z procesów nie może kontynuować
- 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

Komunikacja międzyprocesowa: sygnały

Sygnały są krótkimi wiadomościami, które można wysyłać do procesu, wątku lub grupy procesów. Z każdym sygnałem jest związana jego nazwa i numer (zależne od platformy)

- są prostą (ograniczoną) formą komunikacji międzyprocesowej zdefiniowaną w normie POSIX (dostepne w Unix, Linux, macOS X)
- są programowymi odpowiednikami przerwań sprzętowych, obsługa sygnału odbywa się w ramach procesu w trybie użytkownika
- niektóre wyjątki sprzętowe powodują dostarczenie sygnału do procesu (SIGBUS, SIGEMT, SIGFPE, SIGILL, SIGSEGV, SIGTRAP)
- 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)

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)
- polecenie `kill`
- funkcje systemowe `kill()`, `pthread_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 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`	nie używane (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 nieokreslona
- 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
- moga być powiązane z danymi dostarczanymi do procesu (liczba lub wskaźnik)
- umożliwiają odczytanie identyfikatora procesu i użytkownika nadawcy
- są kolejkowanie a kolejność dostarczenia jest zagwarantowana, im niższy numer sygnału tym wyższy priorytet
- minimalny i maksymalny numer sygnału zdefiniowane przez makro SIGRTMIN i SIGRTMAX (np. SIGRTMIN+1, SIGRTMAX-2)

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`

Zasoby niepodzielne: zasoby systemowe, które nie mogą być współdzielone ani używane równocześnie przez więcej niż jeden proces w danym momencie.

- większość urządzeń zewnętrznych, pliki zapisywalne, obszary danych, które ulegają zmianom

Zasoby podzielne:

- jednostki centralne, pliki tylko do czytania, obszary pamięci, gdzie znajdują się (dzielone) biblioteki, kody programów, itp.

W systemie wieloprocesowym lub wieloprocesorowym (wielowątkowym) mamy do czynienia z (pseudo)równoczesnymi wątkami wykonania.

- Jak zapewnić prawidłowy dostęp procesów do zasobów niepodzielnych?
- Jak unikać zakleszczeń przy dostępie procesów do dwóch lub więcej zasobów równocześnie?

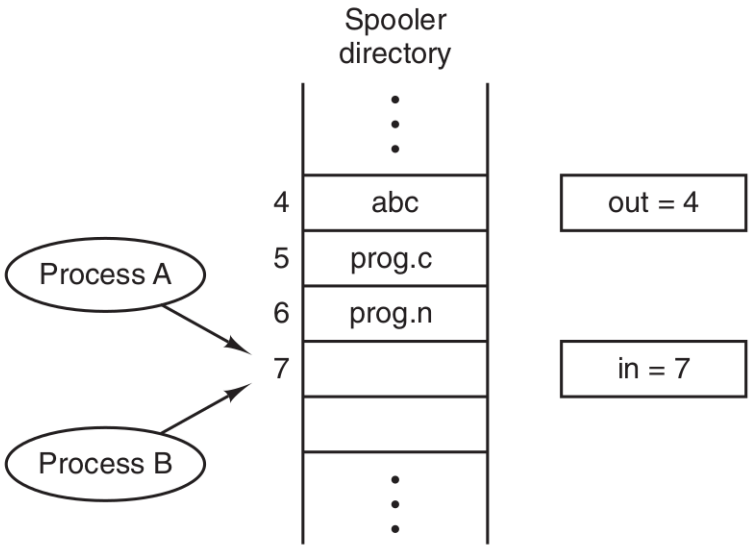
Wyścigi i sekcje krytyczne

Przykład 1: system rezerwacji biletów

Biuro A widzi, że jest wolne miejsce X i powiadamia o tym swego klienta	
	Biuro B widzi, że jest wolne miejsce X i powiadamia o tym swego klienta
⋮	
Biuro A rezerwuje miejsce X	⋮
	Biuro B rezerwuje miejsce X

Wyścigi i sekcje krytyczne

Przykład 2: drukarka jako zasób niepodzielny



- dwa procesy chcą zmodyfikować współdzielony zasób

Wyścigi i sekcje krytyczne

Przykład 3: zmiana zmiennej współdzielonej

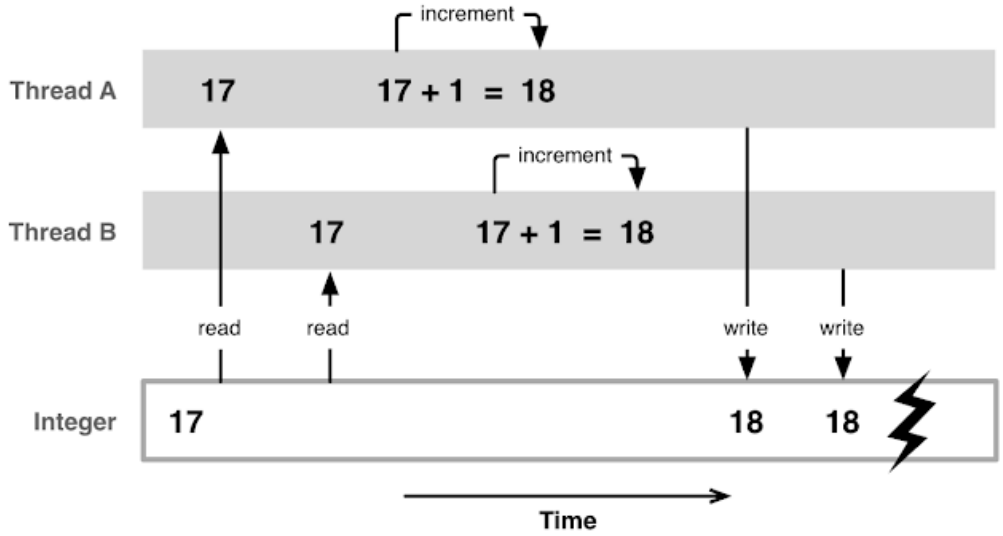
Dwa wątki współdzielą zmienną globalną typu całkowitego `n` i wykonują kod: `n++`, który oznacza

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

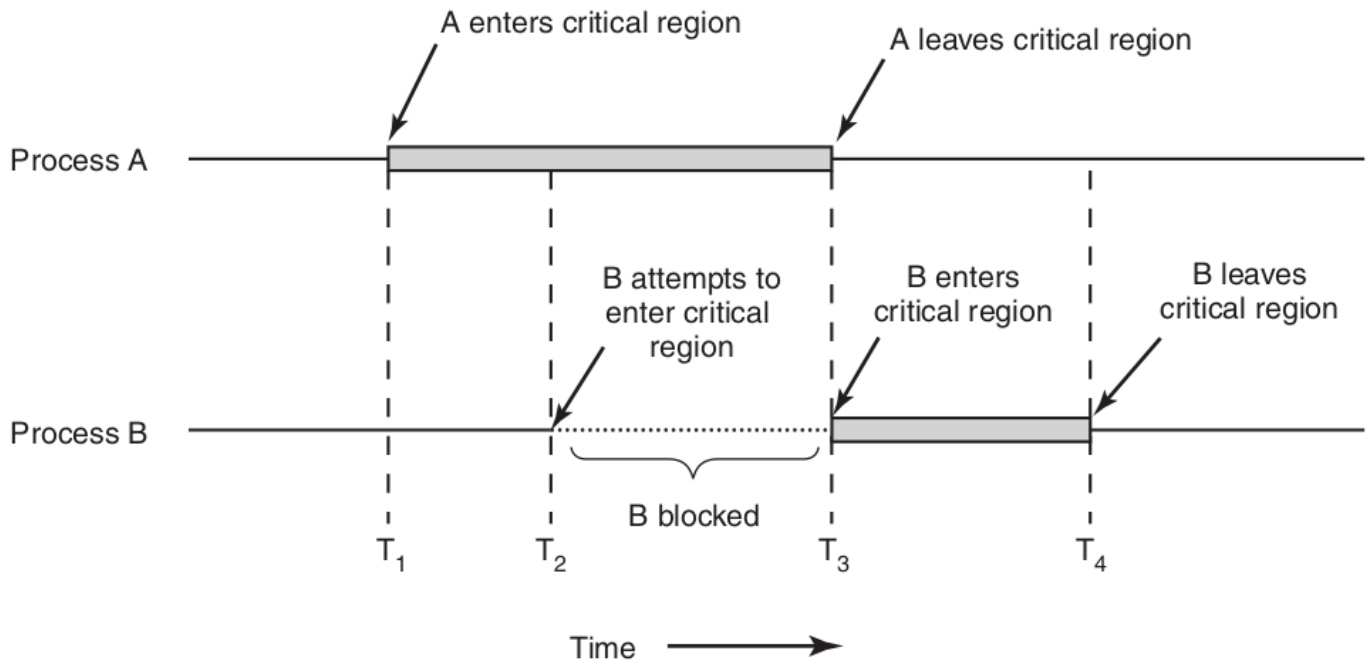


- zwiększanie wartości zmiennej globalnej powinno być wykonywane w sposób niepodzielny (atomowy)
- **operacje atomowe** (*atomic operation*) to niepodzielna operacja, która jest wykonywana w całości albo wcale, zapewnia poprawne wykonanie operacji na zasobach współdzielonych

Wyścigi i sekcje krytyczne

- wspólnie użytkowane zasoby wymagają ochrony przed współbieżnym dostępem, gdyż współbieżność wątków wykonania może prowadzić do powstanie niespójności w danych
- sytuacje, w których procesy (wątki) czytają oraz modyfikują pewne dzielone dane i rezultat końcowy zależy od tego, kiedy dokładnie każdy z tych procesów (wątków) będzie je czytał/modyfikował, nazywamy **wyścigiem** (*race condition*), hazardem lub przeplotem operacji.
- **sekcja krytyczna** (*critical section*) - część programu, w której następuje czytanie i modyfikowanie dzielonych danych
- aby zapobiec wyścigom musimy zapewnić **wzajemne wykluczenie**

Wzajemne wykluczenie



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

Wymogi dotyczące synchronizacji procesów

1. żadne dwa procesy nie mogą przebywać jednocześnie w swojej sekcji krytycznej
2. nie można czynić żadnych założeń, co do szybkości i liczby CPU
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.

Jak zapewnić wzajemne wykluczanie?

- jądro bez wywłaszczenia
- wyłączanie przerwań
- zmienne blokujące
- blokady wirujące (*spin lock*)
- semafory, muttexy
- monitory

Wyłączanie przerwań

Proces wyłącza przerwania (także zegarowe), kiedy wchodzi do swojej sekcji krytycznej i włącza je, kiedy ją opuszcza.

Wady:

- proces użytkownika nie powinien mieć możliwości wyłączania przerwań, technika przydatna tylko gdy używane przez jądro
- w systemach wieloprocesorowych wyłączanie przerwań dotyczy tylko jednego procesora

Zalety:

- łatwy sposób modyfikowania struktur jądra w spójny sposób

Aktywne czekanie (*busy waiting*) - wątek oczekujący na zwolnienie zasobu pozostaje w pętli, stale sprawdzając warunek synchronizacji, aż zostanie spełniony. Jest to tzw. **blokada wirująca** (*spin lock*)

Wada: procesor jest w pełni obciążony wykonaniem pętli, może być nieefektywne w kontekście zużycia zasobów procesora

Wzajemne wykluczenie z aktywnym czekaniem

- zmienne blokujące (*lock variables*)
  proste w użyciu ale nie rozwiązują problemu wyścigu, zmienne blokujące również są zasobem współdzielonym
- ścisła zmienność (*strict alternation*)
- algorytm Petersona
- instrukcja TSL (*Test and Set Lock*)

Ścisła zmienność

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

Początkowo `turn = 0`

proces A

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

proces B

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

Problem:

- gdy procesy różnią się szybkością, może dojść do blokowania przez proces wykonujący się poza sekcją krytyczną

Algorytm Petersona

```
#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;                          /* number of the other process */
  other = 1 - process;                /* the opposite of process */
  interested[process] = TRUE;         /* show that you are interested */
  turn = process;                     /* set flag */
  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 */
}
```

Instrukcja TSL

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

```
enter_region:
  tsl register,lock   | copy lock to register and set lock to 1
  cmp register,#0     | was lock zero?
  jne enter_region    | if it was not zero, lock was set, so loop
  ret                 | return to caller; critical region entered

leave_region:
  mov lock,#0         | store a 0 in lock
  ret                 | return to caller
```

Wymagana jest kooperacja procesów tak aby w odpowiednich momentach wołały procedury *enter_region* oraz *leave_region*.

Instrukcja TSL: zalety

- nadają się dla dowolnej liczby procesów (maszyny jedno- i wieloprocesorowe (SMP))
- są proste i łatwe w weryfikacji
- nadają się do kontroli wielu sekcji krytycznych

Instrukcja TSL: wady

- aktywne czekanie marnuje czas CPU
- możliwość wystąpienia zagłodzeń (*starvation*)
- możliwość wystąpienia zakleszczeń
- problem odwróconego priorytetu - proces aktywnie czekający o wyższym priorytecie, może nie dopuścić do przełączenia CPU na proces o niższym priorytecie przebywającym w sekcji krytycznej, powodując zakleszczenie

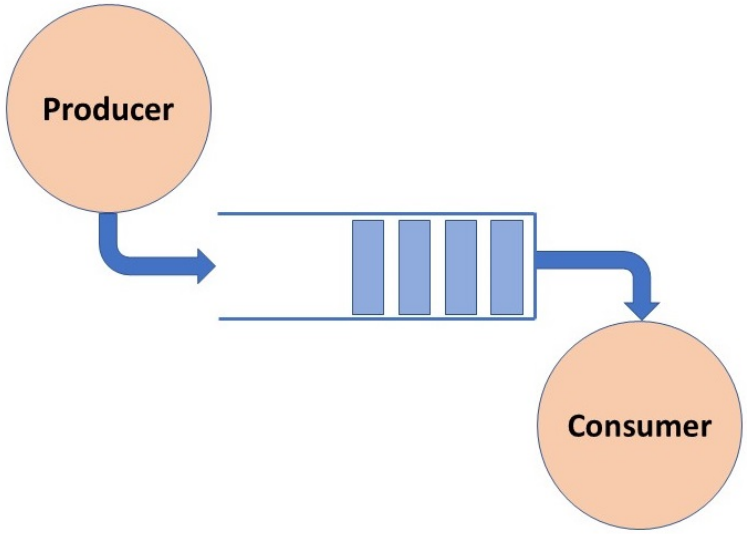
Report_MarsPathFinder.pdf

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 tworzyć przedmiotu, jeśli bufor współdzielony jest pełny
- proces konsumenta nie może zużywać przedmiotu, jeśli współdzielony bufor jest pusty
- dostęp do współdzielonego bufora musi się wzajemnie wykluczać

Problem producenta i konsumenta: przykład

```
#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 */
    if (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 == N) wakeup(consumer);     /* was buffer empty? */
  }
}
```

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

  while (TRUE) {                          /* repeat forever */
    if (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 */
  }
}
```

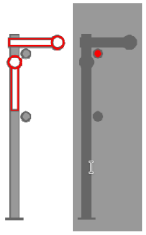
Uwaga! Zmienna count odzwierciedlająca liczbę przedmiotów w buforze, nie jest chroniona, dostęp do niej nie jest niepodzielny. W przypadku wyścigu może dojść do zablokowania producenta i konsumenta

Semafory

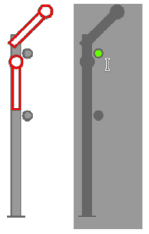
Konieczność zapewnienia niepodzielności wykonywania operacji prymitywnych SLEEP i WAKEUP doprowadziła do powstania semaforów (E.W.Dijkstra, 1965).

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

`SLEEP` ⟶ `DOWN`



`WAKEUP` ⟶ `UP`



Operacje `UP` i `DOWN` muszą być **niepodzielne** (atomowe), tj. w dowolnej chwili tylko jeden proces/wątek może je wykonywać.

[2]Oryginalne nazwy operacji używane przez Dijkstra to `P` (*Proberen*, sprawdź) i `V` (*Vorhogen*, podnieś)

Semafory

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 (rozwiązanie z semaforami)

```
#define N     100                    /* number of slots in the buffer */
typdef int semaphore;                /* semaphores are a special kind of int */
semaphore mutex = 1;                 /* controles 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 */
  }
}
```

Przeznaczenie semaforów:

- semafor binarny otrzymuje początkową wartość 1 i jest używany przez dwa lub więcej procesów, aby zapewnić że tylko jeden z nich może w danej chwili przebywać w swojej sekcji krytycznej
  - `mutex` jest **semaforem binarnym**
  - `mutex` zapewnienia wzajemne wykluczanie się procesów (*MUTual EXclusion*)
- semafory `empty` oraz `full` służą do **synchronizacji**, zapewniają, że zdarzenia będą zachodziły we właściwej kolejności

Semafory i przerwania

Jak można wykorzystać semafory przy obsłudze przerwań?

- z każdym urządzeniem wej/wyj wiąże się semafor, którego początkowa wartość wynosi zero
- po zainicjowaniu operacji wej/wyj dla danego urządzenia proces wykonuje `DOWN` na semaforze i przechodzi w stan uśpienia
- po nadejściu przerwania proces obsługujący to przerwanie wykonuje `UP` na semaforze
- proces, który zgłosił żądanie wej/wyj staje się gotowy do wykonania

Wady semaforów

- semafor jest wysokopoziomową abstrakcją opartą na niskopoziomowych mechanizmach elementarnych, które zapewniają niepodzielność i dostarczają mechanizmów wstrzymywania
- wstrzymywanie i wznawianie wymaga przełączania kontekstu i zmian w kolejkach modułu szeregowania i kolejk wątków wstrzymanych; operacje na semaforze są **powolne**

Blokady pętlowe, wirujące (*spinlocks*)

- najprostszy elementarny mechanizm blokowania
- główna zaleta: niski koszt
- wątek próbujący pozyskać zasób, aktywnie oczekuje aż do odblokowania zasobu
- wątek aktywnie oczekuje na zasób na jednym procesorze, podczas gdy inny wątek korzysta z tego zasobu na innym procesorze
- wątki nie oddają procesora, jeśli nałożyły blokadę wirującą

blokady wirujące = blokady proste

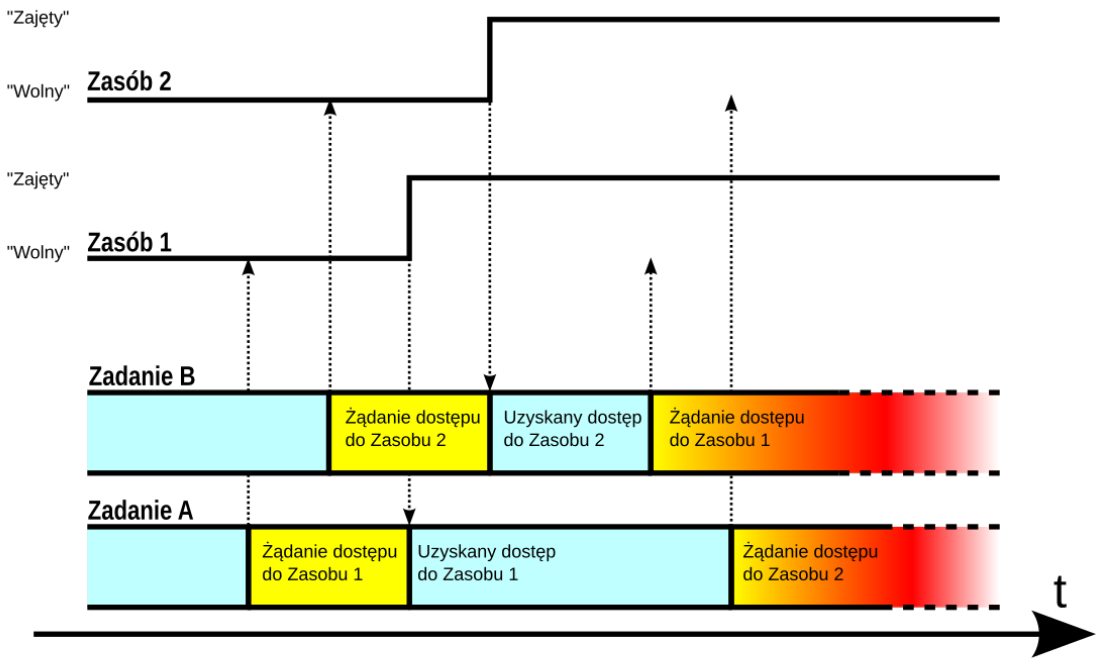
Monitory

Monitor to programistyczne narzędzie wymuszające wzajemne wykluczanie i koordynację procesów. Realizację struktury monitorów zawierają takie języki programowania jak Concurrent Pascal, Pascal-Plus, Modula, Java.

Monitorem nazywamy moduł programowy składający się z jednej lub kilku procedur, ciągu inicjującego i danych lokalnych o następujących cechach:

1. Zmienne danych lokalnych są dostępne wyłącznie za pośrednictwem procedur monitora (żadna procedura zewnętrzna nie może udostępniać zmiennych lokalnych).
2. Proces uruchamia monitor, wywołując jedną z jego procedur.
3. W tej samej chwili pod kontrolą monitora może działać tylko jeden proces; każdy inny proces, wywołujący w tym samym czasie monitor, musi zostać zawieszony do czasu zwolnienia monitora.

Zakleszczenia



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

Zakleszczenia (*deadlocks*, *deadly embraces*)

Zbiór procesów jest w stanie zakleszczenia, kiedy każdy z procesów ze zbioru czeka na zdarzenie, które może spowodować inny z procesów.

Zakleszczenie zdarza się, gdy każdy z procesów otrzymał wyłączne prawo do używania jakiegoś zasobu i dodatkowo żąda dostępu do zasobu już zajętego.

Przykłady:

- Proces A otrzymuje dostęp do drukarki, a proces B do dysku, a następnie proces A żąda dostępu do dysku, a proces B do drukarki.
- Proces A otrzymał prawo dostępu do dwóch (lub więcej) rekordów bazy danych, na których operacje mają być przeprowadzane w tym samym czasie przez proces B.

Zakleszczenia

Jak unikać zakleszczeń?

1. Jeśli to tylko możliwe, to należy unikać stosowania blokad (zamiast blokad można stosować operacje atomowe lub używać struktur danych nie wymagających zakładania blokad).
2. Jeśli blokada jest niezbędna, to należy stosować blokadę pojedynczą.
3. Jeśli potrzebnych jest kilka blokad, to należy zdefiniować (małą) lokalną hierarchię blokad lub zastosować blokowanie stochastyczne.

Klasyczne problemy komunikacji międzyprocesowej

- problem producenta-konsumenta – modelowanie procesów synchronizacji między procesami (problem ograniczonego buforu)
- problem ucztujących filozofów – modelowania 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

Problemy ucztujących filozofów i śpiącego fryzjera