1. Działania na procesach

(1.1) Tworzenie procesu

Pamiętamy, że proces "twórca" nazywany jest **procesem macierzystym**, zaś nowo utworzony proces to **proces potomny**. Każdy nowo utworzony proces może tworzyć kolejne procesy. Strukturą danych określająca korelacje pomiędzy procesami jest więc drzewo procesów. Dopuszczalne są dwie możliwości przydziału przestrzeni adresowej dla procesu potomnego [1]:

- proces potomny jest kopią swego rodzica;
- proces potomny otrzymuje nowy program;

Ponieważ proces do wykonania powierzonych mu informacji potrzebuje zasobów systemowych (np. czasu procesora), muszą mu one zostać przydzielone. Proces potomny utworzony przez proces macierzysty może dostać zasoby systemowe od systemu operacyjnego lub też od procesu macierzystego. Proces macierzysty natomiast, może podzielić swoje zasoby pomiędzy procesy potomne, bądź też sprawić, że niektóre zasoby będą przez potomków wykorzystywane wspólnie.

Gdy proces macierzysty "stworzy" potomka może podążyć dwoma drogami:

- może kontynuować działanie współbieżnie ze swoimi potomkami;
- może zaczekać na zakończenie działań swoich procesów potomnych (części z nich albo wszystkich)

Drugi z wyżej wymienionych przypadków pozwala na zastosowanie w środowiskach wieloprocesorowych mechanizmów synchronizacji (np. barier).

(1.2) Kończenie procesu

Koniec działania procesu następuje, gdy proces wykona swoją ostatnią instrukcję. Wówczas proces wywołuje funkcję systemową *exit* i zostaje usunięty przez system operacyjny. Przed swym wyjściem może jednak przekazać dane do procesu macierzystego. Przed wyjściem odbierane są mu również wszystkie zasoby procesu, a czyni to system operacyjny.

Zakończenie procesu może przebiegać także inaczej. Proces macierzysty może zakończyć działanie procesu potomnego używając funkcji *abort*. Proces macierzysty musi więc znać identyfikatory swych potomków, dlatego tworzenie nowego procesu zawsze wiąże się z przekazaniem jego identyfikatora do procesu macierzystego. Funkcja *abort* może zostać użyta w przypadku, gdy [1]:

- wykonanie zadania przez potomka jest już zbędne;
- potomek nadużył przydzielonych mu zasobów;
- po zakończeniu procesu macierzystego system operacyjny nie pozwala jego potomkowi na dalszą pracę;

Zakończenie działania procesu nie oznacza wcale, że proces ten nie może zostać uruchomiony ponownie z takimi samymi zasobami, jakie miał wcześniej.

NASTĘPNA

2. Współpraca między procesami

W systemie operacyjnym współbieżnym procesy, które są w trakcie wykonania, mogą współpracować ze sobą lub działać niezależnie.

Proces niezależny (ang. *independent*) to taki, który nie może oddziaływać na inne procesy wykonywane w systemie, a i inne procesy nie mogą oddziaływać na niego. Procesy są więc niezależne tylko i tylko wtedy, gdy nie dzielą żadnych danych z żadnymi procesami.

Proces współpracujący (ang. *cooperating*) to taki, który może wpływać na inne procesy lub inne procesy mogą oddziaływać na niego. Analogicznie, proces współpracujący to proces dzielący dane z innymi procesami.

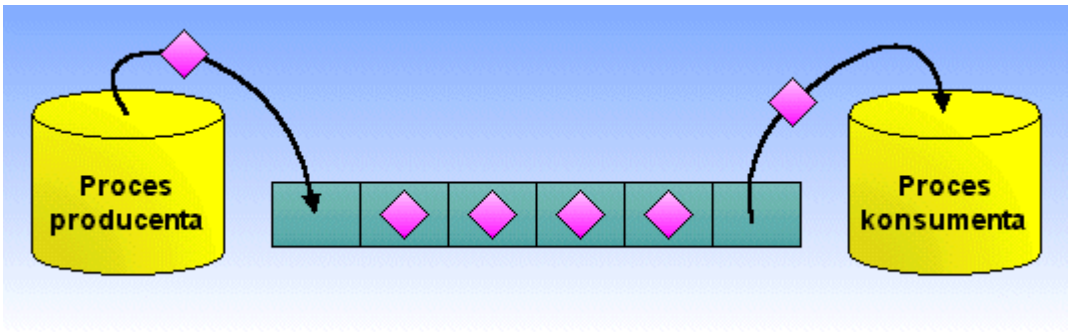
Dzięki wprowadzeniu środowiska umożliwiającego współpracę pomiędzy procesami możliwe staje się [1]:

- dzielenie informacji - możliwość dostarczania kilku użytkownikom tych samych informacji (np. kopii arkusza kalkulacyjnego);
- przyspieszenie obliczeń - duże zadania mogą być dekomponowane na podzadania, które z kolei mogą być wykonywane równolegle (o ile dysponujemy kilkoma procesorami);
- modularność - możliwość stworzenia systemu składającego się z osobnych procesów;
- wygoda - możliwość korzystania z kilku funkcji systemu równocześnie (np. redagowanie tekstu i drukowanie);

(2.1) Algorytm producenta-konsumenta

Zagadnienie producenta-konsumenta (ang. *producer-consumer problem*) stanowi model współpracy procesów.

Zakładamy, że proces producent wytwarza dane, które konsumuje proces konsumenta. Aby było to możliwe musi istnieć bufor (wyobraźmy sobie go jako tablicę) jednostek, który będzie zapełniany przez producenta i opróżniany przez konsumenta. W takim układzie, gdy producent produkuje jednostkę, konsument może wyciągnąć z bufora inną. Ważne jest jednak, aby procesy producenta i konsumenta podlegały synchronizacji tak, by konsument nie "wyciągał" jednostek jeszcze niewyprodukowanych, a w przypadku pustego bufora poczekał na wyprodukowanie czegoś do "wyciągania".



Rysunek 5.1. Producent, bufor oraz konsument

Problem producenta-konsumenta może być dwojakiego rodzaju [1]:

- z nieograniczonym buforem (ang. *unbounded-buffer*) - możliwa jest sytuacja oczekiwania

przez konsumenta na nowe jednostki, jednakże producent produkuje je nieustannie;
- z ograniczonym buforem (ang. *bounded-buffer*) - możliwa jest sytuacja oczekiwania przez konsumenta na nowe jednostki, ale możliwe jest również, że czekał będzie producent w wyniku zapełnienia bufora;

Poniższy algorytm przedstawia rozwiązanie problemu producenta-konsumenta z ograniczonym buforem, który używa pamięci dzielonej. W algorytmie tym procesy konsumenta i producenta korzystają z następujących wspólnych zmiennych:

```
var n;                                  // rozmiar bufora
type jednostka=...;                     // jednostka do konsumpcji
var bufor: array [0..n-1] of            // bufor jako tablica cykliczna
jednostka;
       we, wy: 0..n-1;                  // wskaźniki do wejścia i wyjścia bufora
```

Zmienna *we* wskazuje na następne wolne miejsce w buforze. Gdy bufor jest pusty wskaźniki *we* i *wy* pokrywają się. Gdy bufor jest pełny *we+1* **mod** *n* = *wy* (*mod* oznacza resztę z dzielenia).

Poniżej przedstawione są kody procesów producenta i konsumenta. Wyjaśnijmy Instrukcja *nic* jest instrukcją pustą tzn. jej wykonanie nie daje żadnego efektu. Jednostki nowo produkowane znajdują się w zmiennej lokalnej *nowyprodukt*, zaś jednostki do skonsumowania są przechowywane w zmiennej *nowakonsumpcja*.

A oto i proces producenta:

```
repeat
    ...
    // produkuj jednostkę w nowyprodukt
    ...
    while we+1 mod n=wy do nic;
    bufor[we]:=nowyprodukt;
we:=we+1 mod n;
until false
```

Proces konsumenta ma natomiast taką postać:

```
repeat
    while we=wy do nic;
    nowakonsumpcja:=bufor[wy];
    wy:=wy+1 mod n;
    ...
    // konsumuj jednostkę z nowakonsumpcja
    ...
until false
```

Podany algorytm umożliwia używać co najwyżej *n-1* jednostek bufora w tym samym momencie.

NASTĘPNA

3. Wątki i ich struktura

Wątek (ang. *thread*) określany również jako **proces lekki** (ang. *lightweight process* - LWP) to podstawowa jednostka wykorzystania procesora [1]. W skład wątku wchodzą:

- licznik rozkazów
- zbiór rejestrów
- obszar stosu

Wątek współużytkuje z innymi równorzędnymi wątkami sekcję kodu i danych oraz takie zasoby systemu operacyjnego jak otwarte pliki i sygnały, co łącznie nazywane jest **zadaniem** (ang. *task*) [1].

Aby zrozumieć pojęcie wątku najlepiej porównać go z procesem tradycyjnym, tzw. *ciężkim* (ang. *heavyweight*). Proces = zadanie z jednym wątkiem. Takie właśnie równanie opisuje zależność pomiędzy procesem a wątkiem.

Wątki, w przeciwieństwie do procesów, można przełączać tanim kosztem, tj. z małym użyciem czasu procesora. Oczywiście przełączenie wątku również wymaga od systemu operacyjnego przełączenia zbioru rejestrów, jednakże nie jest konieczne zarządzanie pamięcią.

Porównując wątki i procesy, rozważymy sposób sterowania wieloma wątkami i wieloma procesami. Każdy z procesów działa niezależnie, posiada własny licznik rozkazów, wskaźnik stosu oraz przestrzeń adresową. W przypadku, gdy zadania wykonywane przez procesy nie są ze sobą powiązane wszystko jest w porządku, jednakże gdy mamy do czynienia z jednym zadaniem wykonywanym przez kilka procesów marnowane są zasoby systemu (każdy proces przechowuje osobne kopie tych samych informacji). Proces wielowątkowy natomiast, wykonujący jedno zadanie, zużywa mniej zasobów (wątki korzystają z tego samego zbioru informacji).

Wątki są podobne do procesów pod wieloma względami. Tak jak procesy, watki mogą znajdować się w jednym z określonych stanów: gotowości, zablokowania, aktywności i zakończenia. Aktywny może być tylko jeden wątek (tak jak proces). Każdy wątek ma własny licznik rozkazów i stos, a jego wykonanie w procesie przebiega sekwencyjnie. Wątki, również podobnie jak procesy, mogą tworzyć wątki potomne i mogą blokować się do czasu zakończenia wywołań systemowych - możliwa jest więc sytuacja działania jednego wątku, gdy zablokowany jest inny. Różnicą pomiędzy wątkiem a procesem jest fakt, że wątki są zależne od siebie. Każdy z wątków ma dostęp do danych innego z wątków (dostęp do każdego adresu w zadaniu). Niemożliwa jest więc ochrona na poziomie wątków. Ponieważ jednak w obrębie jednego zadania wątki mogą należeć do jednego użytkownika, ochrona taka nie jest konieczna.

Różne systemy operacyjne w różny sposób traktują wątki. Popularne są dwie koncepcje:

- wątki są obsługiwane przez jądro - system zawiera zbiór funkcji do obsługi wątków;
- wątki są tworzone powyżej jądra systemu za pomocą funkcji bibliotecznych wykonywanych z poziomu użytkownika (ang. *user-level threads*);

Wątki tworzone na poziomie użytkownika, ponieważ nie wymagają dostępu do jądra (korzysta się z nich za pomocą wywołań bibliotecznych zamiast odwołań do systemu), mogą być szybciej przełączane. Może jednak zdarzyć się sytuacja, w której po dowolnym wywołaniu systemowym cały proces musi czekać (nie jest więc realizowany), gdyż jądro planuje tylko procesy (nie wie o istnieniu wątków). Dlatego też, coraz częściej w systemach operacyjnych, stosuje się kombinację tych dwóch rozwiązań, realizując wątki w jądrze i z poziomu użytkownika.

NASTĘPNA

4. Algorytmy planowania przydziału procesora

Planowanie przydziału procesora jest to określenie kolejności przydziału jego mocy obliczeniowej procesom z *kolejki procesów gotowych do działania*. Poniżej zostaną omówione najciekawsze z algorytmów planowania przydziału procesora.

(4.1) Algorytm FCFS

FCFS (ang. *first-come, first-served*) oznacza "pierwszy zgłoszony - pierwszy obsłużony". Algorytm ten opiera się na prostej zasadzie, którą właściwie wyjaśnia jego nazwa. Najlepiej do implementacji algorytmu nadaje się kolejka FIFO, w której blok kontrolny procesu wchodzącego do kolejki dołączany jest na jej koniec, zaś wolny procesor przydzielany jest procesowi z początku kolejki.

Wadą wspomnianej metody przydziału procesora jest to, że średni czas oczekiwania może być bardzo długi. Jeżeli rozważymy przyjście w danym momencie czasu trzech procesów, których długości faz procesora wynoszą:

Proces	Czas trwania fazy
P_1	21 ms
P_2	6 ms
P_3	3 ms

przekonamy się, że nadejście procesów w kolejności P_1,P_2,P_3 wyglądać będzie na *diagramie Gantta* następująco dla algorytmu FCFS:

Dla procesu P_1 czas oczekiwania wynosi więc 0 ms, dla procesu P_2 21 ms, dla procesu P_3 zaś 27 ms. Średni czas oczekiwania wynosi więc: (0+21+27)/3 = 16 ms. Dla sytuacji, w której procesy nadeszłyby w kolejności P_3, P_2, P_1 diagram Gantta wyglądałby następująco:

Dla takiego przypadku średni czas oczekiwania wyniósłby: (0+3+9)/3 = 4 ms. Widzimy więc, że zastosowanie algorytmu niesie ze sobą pewne ryzyko.

W warunkach dynamicznych działanie algorytmu również nie jest zadowalające. W przypadku, gdy mamy do czynienia z dużym procesem ograniczonym przez procesor oraz małymi (szybciej się wykonującymi) procesami ograniczonymi przez wejście-wyjście i proces ograniczony przez procesor jako pierwszy uzyska dostęp do procesora, procesy wejścia-wyjścia oczekiwać będą w kolejce procesów gotowych stosunkowo długo, a urządzenia wejścia wyjścia w tym czasie nie będą mogły być używane. Po zakończeniu wykonania procesu ograniczonego przez procesor, proces ten przejdzie do kolejki oczekującej na wejście-wyjście, natomiast małe procesy ograniczone przez

wejście-wyjście wykonają się w krótkim czasie i zwolnią procesor, który może wtedy być bezczynny. Następnie powróci nasz duży proces, który ponownie będzie obsługiwany przez procesor, a procesy mniejszych rozmiarów będą musiały czekać. Sytuację wyżej opisaną, w której procesy oczekują na zwolnienie procesora przez jeden duży proces, nazywamy **efektem konwoju** (ang. *convoy effect*).

Algorytm ten, z uwagi na to, że pozwala procesowi na zajmowanie procesora przez dłuższy czas, nie nadaje się do stosowania w środowiskach z podziałem czasu, w których każdy użytkownik powinien dostawać przydział procesora w równych odstępach czasu. Taki algorytm, który po objęciu kontroli nad procesorem nie umożliwia do niego dostępu, nazywamy niewywłaszczającym.

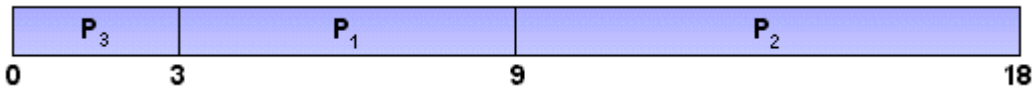
(4.2) Algorytm SJF

Algorytm SJF (ang. *shortest-job-first*), czyli "najkrótsze zadanie najpierw", opiera swe działanie na powiązaniu procesu z długością jego najbliższej fazy. Procesor zostaje przydzielony procesowi, który ma najkrótszą następną fazę procesora. W przypadku, gdy dwa procesy mają następne fazy procesora równe, stosuje się algorytm FCFS.

Porównajmy działanie tego algorytmu do algorytmu FCFS. Załóżmy więc, że mamy 3 następujące procesy:

Proces	Czas trwania fazy
P_1	6 ms
P_2	9 ms
P_3	3 ms

Diagram Gantta dla tych procesów wygląda następująco:



Jeżeli procesy przychodzą w kolejności P_1, P_2, P_3 to dla algorytmu SJF średni czas oczekiwania wynosi (3+9+0)/3 = 4ms, zaś dla algorytmu FCFS wyniósłby (0+6+15)/3 = 7 ms.

Algorytm SJF daje minimalny średni czas dla danego zbioru procesów. Mówimy, że algorytm SJF jest **optymalny**.

Dużym problemem w algorytmie SJF jest określenie długości następnego zamówienia na przydział procesora. Przy planowaniu długoterminowym (zadań), za czynnik ten możemy przyjąć limit czasu procesu, określony przez użytkownika, który zadanie przedłożył. Algorytm SJF jest więc chętnie używany w planowaniu długoterminowym. Natomiast w przypadku planowania krótkoterminowego, nie ma sposobu na określenie długości następnej fazy procesora. Dlatego też oszacowuje się tę wartość, obliczając ją na ogół jako średnią wykładniczą pomiarów długości poprzednich faz procesora [1].

Możliwe są dwa sposoby realizacji algorytmu SJF - wywłaszczająca (ang. *shortest-remaining-time-first*) oraz niewywłaszczająca. Różnica pomiędzy tymi realizacjami widoczna jest w sytuacji, gdy w *kolejce procesów gotowych* pojawia się nowy proces o krótszej następnej fazie procesora niż

pozostała faza "resztki" procesu aktualnie wykonywanego. W takim układzie algorytm wywłaszczający usuwa aktualny proces z procesora, zaś algorytm niewywłaszczający pozwala procesowi bieżącemu na zakończenie fazy procesora. Forma wywłaszczająca algorytmu SJF jest więc szybsza.

(4.3) Algorytm priorytetowy

Algorytm planowania priorytetowego jest ogólnym przypadkiem algorytm SJF. Zamiast długości następnej fazy z algorytmu SJF, procesowi przypisuje się priorytet. Procesy o wyższym proiorytecie mają pierwszeństwo w dostępie do procesora, przed procesami o priorytecie niższym. Procesom o równych priorytetach przydziela się procesor zgodnie z algorytmem FCFS.

Priorytety definiuje się wewnętrznie lub zewnętrznie. Wewnętrzna definicja priorytetu opiera się na własnościach procesu, np.: stosunek średniej fazy we/wy do średniej fazy procesora, wielkość obszaru wymaganej pamięci, liczba otwartych plików, itp. Do określenia priorytetów zewnętrznych używa się kryteriów spoza wnętrza systemu, np.: kwota opłat za użytkowanie komputera, znaczenie procesu dla użytkownika, itp.

Problem związany z planowaniem priorytetowym to **nieskończone blokowanie** (ang. *indefinite blocking*), nazywane **głodzeniem** (ang. *starvation*). Problem wiąże się z odkładaniem możliwości obsługi przez procesor tych procesów, które mają niski priorytet. W sytuacji, w której system jest bardzo obciążony, może zdarzyć się, że proces o niskim priorytecie będzie czekał w kolejce procesów gotowych w nieskończoność. Powoduje to duże opóźnienia wykonania procesu o niskim priorytecie, a może doprowadzić do niewykonania tego procesu.

Problem ten jest rozwiązywany poprzez **postarzanie** (ang. *aging*) procesów o niskich priorytetach. Postarzanie procesów o niskim priorytecie, które długo oczekują na obsługę procesora odbywa się co pewien okres czasu. W ten sposób, z procesu o niskim priorytecie, tworzy się proces o wysokim priorytecie, który na pewno zostanie obsłużony w pierwszej kolejności.

Algorytmy planowania priorytetowego mogą być postaci wywłaszczającej, bądź niewywłaszczającej. Działanie jest analogiczne jak dla algorytmu SJF, z tą różnicą, że kryterium wyboru jest priorytet, a nie czas trwania następnej fazy.

(4.4) Algorytm RR

Algorytm planowania rotacyjnego (ang. *round-robin* - RR) jest oparty na algorytmie FCFS, jednakże różni się od FCFS faktem, że możliwe jest wywłaszczanie. Algorytm ten jest chętnie stosowany w systemach z podziałem czasu.

Zasada działania algorytmu jest następująca. Z każdym procesem powiązany jest kwant czasu, najczęściej 10 do 100 ms. Procesom znajdującym się w kolejce (traktowanej jako cykliczna) procesów gotowych do wykonania, przydzielane są odpowiednie odcinki czasu, nie dłuższe niż jeden kwant czasu.

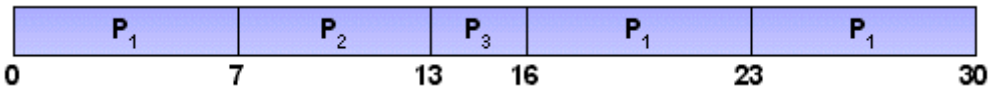
Do implementacji algorytmu, podobnie jak w przypadku FCFS, używa się kolejki FIFO - brany jest pierwszy proces z wyjścia kolejki, ustawiany jest timer na przerwanie po upływie 1 kwantu czasu i proces jest wysyłany do procesora. Jeżeli czas trwania fazy procesu jest krótszy od 1 kwantu czasu, proces sam zwolni procesor i rozpocznie się obsługiwanie kolejnego procesu z kolejki. Jeżeli proces ma dłuższą fazę od 1 kwantu czasu, to nastąpi przerwanie zegarowe systemu operacyjnego, nastąpi przełączenie kontekstu, a proces do niedawna wykonywany, trafi na koniec kolejki procesów gotowych do wykonania.

Zobaczmy, ile wynosi średni czas oczekiwania w porównaniu z metodą FCFS dla następujących

danych:

Proces	Czas trwania fazy
P_1	21 ms
P_2	6 ms
P_3	3 ms

Dla kroku 7 ms diagram Gantta dla powyższych procesów wyglądał będzie następująco:



Śreni czas oczekiwania wynosi więc: (0+13+16)/3 = 9,67 ms.

Wydajność algorytmu rotacyjnego zależy od kwantu czasu. W przypadku, gdy kwant czasu jest bardzo długi (dąży do nieskończoności) metoda RR działa jak FCFS. Natomiast w przypadku, gdy kwant czasu jest bardzo krótki (dąży do zera) planowanie RR nazywamy **dzieleniem procesora** (ang. *processor sharing*), gdyż zachodzi złudzenie, że każdy z n procesów ma własny procesor działający z prędkością 1/n prędkości prawdziwego procesora [1].

W przypadku algorytmu RR, podczas mówienia o wydajności, należy zastanowić się nad zagadnieniem przełączania kontekstu. Im mniejszy jest kwant czasu, tym system bardziej jest narażony na częste przełączenia kontekstu. Jeżeli czas przełączania kontekstu wynosi 10 procent kwantu czasu, to niemalże 10 procent czasu procesora jest tracone w wyniku przełączania kontekstu.

NASTĘPNA

5. Problemy synchronizacji procesów

(5.1) Problem ograniczonego buforowania

Zagadnienie ograniczonego buforowania powinno być już znane. Mówiliśmy o tym w segmencie 2 tej lekcji.

Jeżeli przyjmiemy, że mamy *n* buforów, a każdy z nich mieści 1 jednostkę, oraz że semafor *mutex* (umożliwiający wzajemne wykluczanie dostępu do buforów) jest na starcie równy 1, zaś semafory *pusty* i *pełny* zawierają informację o liczbie pustych (wartość początkowa n) i pełnych (wartość początkowa 0) buforów, to ogólna wersja algorytmu wygląda tak [1]:

```
repeat
    ...
    produkuj jednostkę w nowyprodukt
    ...
    poczekaj(pusty);
    poczekaj(mutex);
    ...
    dodaj jednostkę nowyprodukt do bufora bufor
    ...
    zasygnalizuj(mutex);
    zasygnalizuj(pełny);
until false
```

Proces producenta ...

```
repeat
    poczekaj(pełny);
    poczekaj(mutex);
    ...
    wyjmij jednostkę z bufora bufor do nowakonsumpcja
    ...
    zasygnalizuj(mutex);
    zasygnalizuj(pusty);
    ...
    konsumuj jednostkę z nowakonsumpcja
    ...
until false
```

... i proces konsumenta.

Instrukcje *poczekaj* oznaczają oczekiwanie z powodu faktu, że bufor jest pusty lub pełny. Instrukcja *zasygnalizuj* opisuje sytuację, w której bufor staje się pusty lub pełny, a także uruchomienie semafora *mutex*.

(5.2) Problem czytelników i pisarzy

Problem ten odnosi się do środowisk, w których zasoby mogą być dzielone. W takich środowiskach może zachodzić sytuacja dostępu do danego zasobu przez kilka procesów jednocześnie. Niektóre z tych procesów odczytują tylko zawartość obiektu dzielonego, inne zapisują do niego informacje (aktualizacja). Jak już Procesy odczytujące dane z obiektu dzielonego to czytelnicy, zaś zapisujące

dane to pisarze.

Problem czytelników i pisarzy (ang. *readers-writers problem*) polega na zapewnieniu "prywatności" pisarzowi. Nie może zdarzyć się sytuacja, w której dany obiekt współdzielony jest przez pisarza i inny proces, czy to czytelnika, czy to innego pisarza. Problem ten ma kilka odmian. Najłatwiejsza z nich nazywana jest pierwszym problemem czytelników i pisarzy. Metoda ta zakłada, że żaden czytelnik nie musi czekać, pomimo tego że pisarz jest gotowy, na zakończenie pracy pozostałych czytelników. Kolejna odmiana problemu czytelników i pisarzy zakłada, że gotowość pisarza oznacza, że żaden nowy czytelnik nie będzie mógł współpracować z obiektem.

Ponieważ oba przedstawione powyżej schematy mogą powodować głodzenie, stosuje się również inne wersje rozwiązania tego problemu.

Poniżej zostanie opisane rozwiązanie pierwszego problemu czytelników i pisarzy. Załóżmy, że wszystkie z procesów mają wspólne następujące zmienne [1]:

```
var mutex, pisarz: semaphore;        // wartość początkowa "1" dla obu
    liczba_czytelnikow:              semaforów
integer;                             // wartość początkowa "0"
```

Przed napiseniem struktury procesu czytelnika warto jeszcze nadmienić, że semafor *mutex* zajmuje się wzajemnym wykluczaniem procesów podczas nadpisywania zmiennej *liczba_czytelnikow*. Semafor *pisarz* daje gwarancję wzajemnego wykluczania pisarzy, a ponadto jest używany przez pierwszy proces wchodzący lub ostatni proces wychodzący z sekcji krytycznej czytelnika. Semafor ten nie jest natomiast używany przez czytelników wchodzących/wychodzących podczas, gdy inni czytelnicy znajdują się w sekcjach krytycznych. A oto i rozwiązanie naszego problemu [1].

Tak wygląda struktura procesu pisarza:

```
poczekaj(pisrz)
    ...
    teraz proces sobie pisze i pisze, i pisze ...

    ...
zasygnalizuj(pisarz);
```

Struktura procesu czytelnika wygląda następująco:

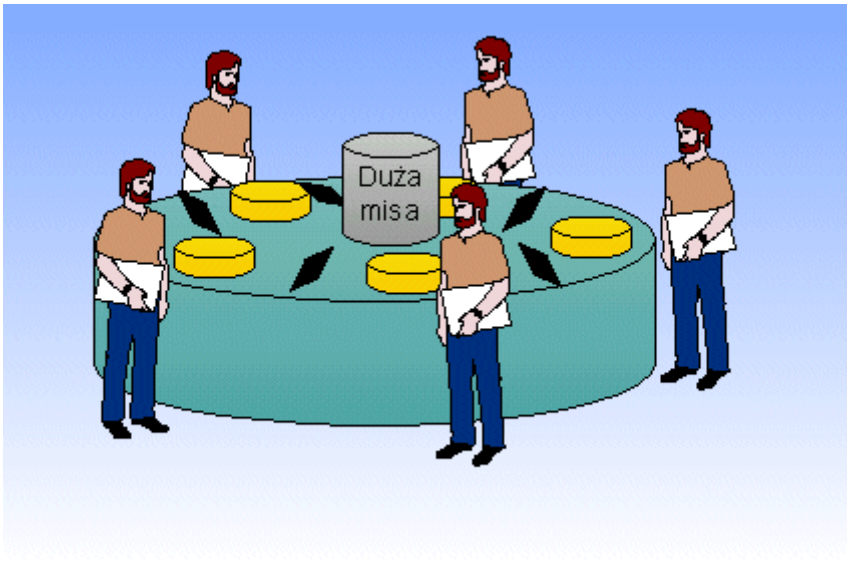
```
poczekaj(mutex);
    liczba_czytelnikow:=liczba_czytelnikow+1;
    if liczba_czytelnikow=1 then poczekaj(pisarz);
zasygnalizuj(mutex);
    ...
    proces czyta sobie i czyta, i czyta ...
    ...
poczekaj(mutex);
    liczba_czytelnikow:=liczba_czytelnikow-1;
    if liczba_czytelnikow=0 then zasygnalizuj(pisarz);
zasygnalizuj(mutex);
```

Po wykonaniu przez proces pisarza operacji *zasygnalizuj(pisarz)* możliwe są dwie drogi: można wznowić działanie pojedynczego pisarza lub czekających czytelników, zależnie od wyboru planisty.

(5.3) Problem obiadujących filozofów

Uwaga, wybieramy się na wycieczkę. Znajdujemy się teraz w miejscu bliżej nieokreślonym, w którym jest tylko jeden mebel - stół. Przy tymże stole siedzi pięć osób. Wszyscy mają przed sobą miseczki, a na środku stołu stoi misa z ryżem. Ponadto, na stole znajduje się pięć pałeczek rozmieszczonych pomiędzy miseczkami nieznanych nam osób. Po dokładniejszych oględzinach okazuje się, że ci ludzie są filozofami. A wiecie dlaczego? Ponieważ przez cały czas myślą, a gdy są głodni sięgają po pałeczki, aby zjeść trochę ryżu, potem znowu myślą. I tu dochodzimy do naszego problemu. Jak nasz drogi filozof ma wziąć dwie pałeczki, skoro sąsiad też może chcieć coś zjeść? Przecież filozof nie będzie wyrywał siłą koledze pałeczki z ręki.

Można by przypuszczać, że problem filozofów nie ma wiele wspólnego z synchronizacją. To nieprawda. Problem obiadujących filozofów jest klasycznym problemem konieczności przydziału wielu zasobów do wielu procesów w warunkach zagrożenia głodzeniem i zakleszczeniem [1].



Rys. 5.2. Pięciu współczesnych obiadujących filozofów

Przedstawimy proste rozwiązanie tego problemu, oparte na semaforach, nie wolne jednak od niebezpieczeństw. Operacja *poczekaj* oznacza podniesienie przez filozofa pałeczki, zaś operacja *zasygnalizuj* oznacza pałeczki odłożenie. Nasza pałeczka wygląda natomiast następująco [1]:

```
var paleczka: array[0..4] of semaphore;
                                        // wartość początkowa=1
```

Po wielu zmaganiach umysłowych będziemy wreszcie mogli zobaczyć jak wygląda struktura n-tego filozofa.

```
repeat
    poczekaj(pałeczka[n]);
    poczekaj(pałeczka[n+1 mod 5]);
    ...
        jedzenie, jedzenie, jedzenie, np. ryżu
    ...
    zasygnalizuj(pałeczka[n]);
    zasygnalizuj(pałeczka[n+1 mod 5]);
    ...
    myślnie nie boli, więc filozof myśli
    ...
until false
```

Struktura procesu n-tego filozofa [1]

Rozwiązanie powyżej przedstawione nie jest najlepsze. Zapewnia co prawda gwarancję, że dwaj sąsiedzi nie będą jedli jednocześnie, jednakże w przypadku gdy każdy z filozofów w tym samym czasie weźmie pałeczkę (by było to możliwe każdy z filozofów musi wziąć pałeczkę leżącą po tej samej stronie, np. prawej) tak, że wszystkie pałeczki zostaną podniesione (elementy tablicy pałeczka są równe "0"), nie będzie możliwe podniesienie przez któregokolwiek z filozofów pałeczki drugą ręką. Zatrzymamy się w pętli nieskończonej.

Naturalnie możliwa jest eliminacja takich zakleszczeń. A oto kilka na to sposobów:

- do stołu zasiada czterech filozofów, podczas gdy jest wolnych pięć miejsc;
- filozof może podnosić pałeczki wtedy i tylko wtedy, gdy po prawej i po lewej stronie są one dostępne;
- wprowadzenie asymetrii - filozofowie nieparzyści zaczynają podnosić pałeczki z lewej strony, zaś parzyści ze strony prawej.

NASTĘPNA

6. Wykorzystanie semaforów

Przy omawianiu złożonych zagadnień synchronizacji wygodnie jest posługiwać się narzędziem synchronizacji nazywanym **semaforem** (ang. *semaphore*). Semafor jest strukturą zawierającą zmienną typu całkowitego, której można nadać wartość początkową i do której można się odwoływać tylko za pośrednictwem dwu niepodzielnych operacji: P (hol. *proberen*, ang. *wait*, pol. *czekaj*) oraz V (hol. *verhogen*, ang. *signal*, pol. *sygnalizuj*). Definicja semafora przedstawia się nastepująco [1]:

poczekaj(S): `while S<=0 do nic;`

`S:=S-1;`

zasygnalizuj(S): `S:=S+1;`

Ważne jest, aby zmiany wartości semafora wykonywane za pomocą operacji *poczekaj* i *zasygnalizuj* uniemożliwiały sytuację, w której dwa procesy próbowałyby modyfikować wartość semafora. Dodatkowo, podczas operacji *poczekaj(S)*, nie może wystąpić przerwanie w trakcie sprawdzania wartości zmiennej S (dopóki S<=0) oraz jej aktualizacji.

(6.1) Sposób używania

Semafory stosuje się często przy rozwiązywaniu problemów sekcji krytycznej, której przydzielone jest kilka procesorów. Procesory te obsługują wspólny semafor **mutex** (ang. *mutual exclusion*), który zapewnia wzajemne wykluczanie kilku procesów tak, aby zapewnić dostęp tylko jednemu spośród nich. Organizacja każdego z procesów korzystających z semafora przedstawia się następująco [1]:

```
repeat
   poczekaj(mutex);
      ...
      sekcja krytyczna
      ...
   zasygnalizuj(mutex);
      ...
      reszta działań procesu
      ...
until false;
```

Semafory stosujemy także przy rozwiązywaniu problemów związanych z synchronizacją. Jeżeli na przykład chcielibyśmy, aby dwa procesy P-X i P-Y, zawierające odpowiednio rozkazy S-X i S-Y, wykonały się sekwencyjnie, tzn. w kolejności X, potem Y. Możliwa jest prosta realizacja takiego problemu, gdy wprowadzimy wspólną dla obu procesów zmienną *synchronizacja*, nadawszy jej wartość początkową "0". Zakładając, że dołączymy do procesów P-X i P-Y instrukcje:

P-X:

```
S-X;
sygnalizuj(synchronizacja);
```

P-Y:

```
czekaj(synchronizacja);
S-Y;
```

możliwe stanie się wykonanie najpierw kodu S-X, później S-Y, ponieważ dopóki zmienna *synchronizacja* wynosi "0", semafor *czekaj* blokuje wykonanie instrukcji S-Y. Wykonanie S-Y możliwe staje się dopiero po wykonaniu rozkazu *sygnalizuj(synchronizacja)*.

(6.2) Implementacja

Obsługa semaforów wymaga **aktywnego czekania** (ang. *busy waiting*). Polega to na tym, że procesy chcąc dostać się do sekcji krytycznej, zajmowanej przez jakiś proces, zmuszone są do wykonywania pętli w sekcji wejściowej. Aktywne czekanie marnuje czas procesora, który mógłby wykonywać nieco ważniejsze zadania. Semafory tego typu nazywa się też **wirującą blokadą** (ang. *spinlock*).

Wirujące blokady w przypadku założenia, że czas oczekiwania procesu pod blokadą jest bardzo krótki, stają się bardzo użyteczne w środowiskach wieloprocesorowych. Ich zaletą jest fakt, że postój procesu pod semaforem nie wymaga przełączania kontekstu, który może trwać długo.

Nie trzeba jednak traktować aktywnego czekania jako niezniszczalnego. Aktywne czekanie można bowiem wyeliminować. Aby było to możliwe proces, który musiałby czekać aktywnie, powinien mieć możliwość zablokowania się (ang. *block*). Zablokowanie się procesu powoduje umieszczenie go w kolejce do danego semafora oraz zmianę stanu procesu na "czekanie", po czym wybierany jest do wykonania kolejny z procesów. Wznowienie działania procesu zablokowanego powoduje sygnał *zasygnalizuj* pochodzący od innego procesu. Wykonywana jest wtedy operacja budzenia (ang. *wakeup*), która zmienia stan procesu na "gotowość" [1]. Powoduje to przejście procesu do kolejki procesów gotowych do wykonania przez procesor.

Semafor, o którym tak już dużo napisaliśmy można zrealizować jako prosty rekord [1]:

```
type semaphore = record
                    wartosc: integer;
                    L: list of proces;
                 end;
```

Jak widzimy semafor składa się z dwóch pól: wartości całkowitej i listy procesów, które muszą czekać pod semaforem. Operacje na tak zdefiniowanym semaforze można przedstawić następująco [1]:

poczekaj(S):
```
S.wartosc:=S.wartosc-1;
if S.wartosc<0 then begin
                       dodaj proces do S.L;
                       blokuj;
                    end;
```

zasygnalizuj(S):
```
S.wartosc:=S.wartosc+1;
if S.wartosc<=0 then begin
                        uzuń proces P z S.L;
                        obudz(P);
                     end;
```

Tak zaimplementowane operacje *poczekaj* i *zasygnalizuj* nie eliminują całkowicie efektu aktywnego czekania, jednakże zapewniają usunięcie tego zjawiska z wejść do sekcji krytycznych programów użytkowych. Ma to ogromne znaczenie, gdyż sekcje krytyczne tych programów mogą zajmować dużo czasu lub być często zajęte. Aktywne czekanie w takich warunkach mogłoby być fatalnym posunięciem i zakończyłoby się dużym spadkiem wydajności systemu.

(6.3) Głodzenie i zakleszczenia

Mówimy, że zbiór procesów jest w stanie zakleszczenia (ang. *deadlock*), gdy każdy proces w tym zbiorze oczekuje na zdarzenie, które może być spowodowane tylko przez inny proces z tego zbioru [1]. Dla semaforów zdarzeniem tym jest operacja *zasygnalizuj*.

Przykładem takiego zdarzenia są procesy Px i Py, które korzystają z semaforów A i B, o wartościach "1", których postać przedstawia się następująco:

Px:	Py:
`poczekaj(A);`	`poczekaj(B);`
`poczekaj(B);`	`poczekaj(A);`
.	.
.	.
.	.
`zasygnalizuj(A);`	`zasygnalizuj(B);`
`zasygnalizuj(B);`	`zasygnalizuj(A);`

W takiej sytuacji powstaje zakleszczenie, gdyż proces Px musi czekać na wykonanie operacji *zasygnalizuj(A)* przez proces Py, zaś Proces Py czeka na wykonanie przez Px rozkazu *zasygnalizuj (B)*.

Kolejnym problemem związanym z semaforami jest głodzenie, czyli konieczność nieskończonego oczekiwania przez proces pod semaforem. Problem ten jest wynikiem niskiego priorytetu danego procesu. Proces jest ignorowany np. w wyniku dużego natłoku innych procesów, które mają wyższe przerwania. Najczęściej głodzenie pojawia się w przypadku użycia kolejki LIFO (ang. *Last In First Out*) jako listy procesów oczekujących do semafora [1].

NASTĘPNA

7. Przydziały zasobów i zapobieganie zakleszczeniom

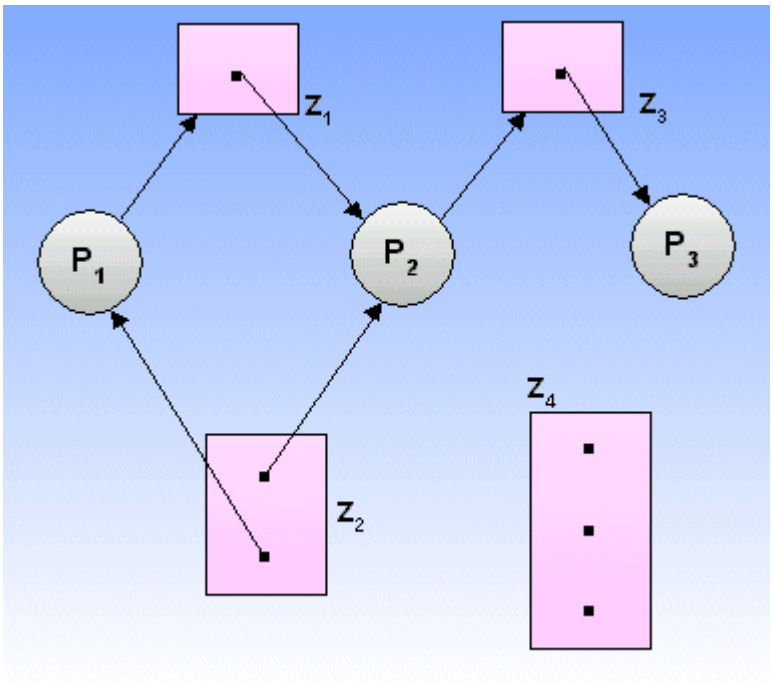
(7.1) Graf przydziału zasobów

Graf przydziału zasobów (ang. *system resource-allocation graph*) bardzo dobrze nadaje się do opisywania zakleszczeń. Składnikami tego skierowanego grafu są [1]:

1. wierzchołki W, składające się z węzłów:
   - $P=\{P_1, P_2,..., P_n\}$ - wszystkie procesy systemu;
   - $Z=\{Z_1, Z_2,..., Z_n\}$ - wszystkie typy zasobów systemowych;

2. krawędzie K, które dzielimy na:
   - krawędzie skierowane od procesu P_x do zasobu Z_y (zapisujemy $P_x \rightarrow Z_y$) - oznacza to zamówienie zasobu Z_y przez proces P_x;
     krawędzie zwane są *krawędziami zamówienia* (ang. *request edge*);
   - krawędzie skierowane od zasobu Z_y do procesu P_x (zapisujemy $Z_y \rightarrow P_x$) - oznacza to przydzielenie zasobu Z_y do procesu P_x;
     krawędzie te nazywa się *krawędziami przydziału* (ang. *assignment edge*);

Ponadto zbiór procesów P na grafie przedstawia się w postaci kółek, zaś zbiór typów zasobów Z w postaci prostokątów. Każdy z typów zasobów Z_y może mieć więcej niż jeden egzemplarz, zaznaczany jako kropka w prostokącie zasobu. Podczas gdy krawędź zamówienia dochodzi tylko do boku prostokąta zasobu, krawędź przydziału rozpoczyna się od konkretnego egzemplarza (czytaj kropki) typu zasobu wewnątrz prostokąta. Ilustruje to rysunek 5.3.



Rys. 5.3. Graf przydziału zasobów [1]

Sposób działań systemu odpowiadających zmianom krawędzi jest następujący:

1. Zamówienie przez proces P_x zasobu Z_y powoduje pojawienie się krawędzi zamówienia.

2. Realizacja zamówienia powoduje zamianę krawędzi zamówienia w krawędź przydziału.
3. Brak jakiejkolwiek krawędzi oznacza, że proces zwolnił już niepotrzebny mu zasób.

Czytając o grafie przydziału zasobów, na pewno zastanawiacie się: co to ma wspólnego z zakleszczeniami? Jeśli graf zawiera cykle (pętle) może dojść do zakleszczeń, jeżeli nie zawiera cykli sytuacja taka jest niemożliwa. A czym jest cykl? Na rysunku 5.3. nie występuje żaden cykl. Jeżeli jednak wyobrazilibyśmy sobie sytuację, w której proces P3 zamawia zasób Z2 (pojawia się wtedy krawędź zamówienia P3 - Z2), to w tak zmodyfikowanym układzie występują dwa cykle:

- $P_1 \rightarrow Z_1 \rightarrow P_2 \rightarrow Z_3 \rightarrow P_3 \rightarrow Z_2 \rightarrow P_1$
- $P_2 \rightarrow Z_3 \rightarrow P_3 \rightarrow Z_2 \rightarrow P_2$.

Zakleszczeniem możemy określić sytuację, w której każdy z zasobów w cyklu ma tylko jeden egzemplarz. W przypadku natomiast, gdy któryś z zasobów w cyklu ma większą liczbę egzemplarzy niż 1 nie przesądza to o występowaniu zakleszczenia.

W przedstawionym powyżej grafie systemu, do zakleszczenia dojdzie na pewno, gdy proces P_3 zechce skorzystać z zasobu Z_2.

(7.2) Wzajemne wykluczanie

Wzajemne wykluczanie to sytuacja, w której istnieje przynajmniej jeden zasób systemu, którego używa w określonym czasie tylko jeden proces.

Warunek negacji wzajemnego wykluczania byłby spełniony, a tym samym możliwe byłoby wykluczenie zakleszczenia, gdyby w systemie możliwe było stworzenie sytuacji, w której wszystkie zasoby byłyby dzielone [1]. Sytuacja taka jest jednak niemożliwa, gdyż niektóre z zasobów są niepodzielne z natury, np. skaner nie może być jednocześnie użytkowany przez kilka procesów.

(7.3) Przetrzymywanie i oczekiwanie

Warunek przetrzymywania i oczekiwania jest spełniony, gdy istnieje proces, któremu został przydzielony co najmniej jeden zasób. Ponadto proces ten oczekuje na przydział kolejnego zasobu, który jest używany przez inny proces [1].

Zapobiegniemy zakleszczeniu wtedy i tylko wtedy, gdy uda nam się dokonać negacji powyższego warunku. Musimy więc doprowadzić do sytuacji, w której proces zamawiający zasób nie posiada sam żadnych zasobów. Opiszemy pokrótce dwa protokoły realizujące takie założenie [1]:

1. Należy umieścić wywołania funkcji systemowych, opisujących zamówienia zasobów dla procesu, za wywołaniami wszystkich innych funkcji systemu.

2. Umożliwić procesowi zamawianie zasobów wtedy i tylko wtedy, gdy proces nie posiada żadnych zasobów (przed zamówieniem nowych zasobów proces musi oddać wszystkie zasoby, z których w danej chwili korzysta).

Podstawową wadą obu powyższych rozwiązań jest to, że *wykorzystanie zasobów* może być zaskakująco małe, gdyż niemożliwe będzie korzystanie z dużej części przydzielonych zasobów przez długi czas. Drugą z wad opisanych protokołów jest możliwość wystąpienia głodzenia. Proces, który chce skorzystać z "rozchwytywanego" zasobu może czekać do nieskończenie długo tylko dlatego, że zasób ten będzie ciągle w posiadaniu innego procesu.

(7.4) Brak wywłaszczeń

Z brakiem wywłaszczeń mamy do czynienia wtedy, gdy zasób może zostać zwolniony tylko w wyniku działań procesu, który go przetrzymuje.

Zapobieżenie wystąpieniu braku wywłaszczeń jest realizowalne jedynie w przypadku, gdy możliwe stanie się zwolnienie zasobu (niekoniecznie przez sam proces) w wyniku zakończenia działania procesu. Rozwiązanie problemu przynosi jeden z dostępnych protokołów. W protokole tym zgłoszenie przez proces zapotrzebowania na zasób, który nie może zostać procesowi przydzielony, powoduje utratę wszystkich dotychczasowych zasobów tego procesu [1] (zasoby te są niejawnie dopisywane do listy innych zasobów, na które dany proces czeka). Wznowienie procesu, nastąpi dopiero wówczas, gdy możliwe stanie się przywrócenie dawnych zasobów oraz dodanie tych, które zamawiał.

(7.5) Czekanie cykliczne

Warunek czekania cyklicznego występuje wtedy i tylko wtedy, gdy istnieje zbiór procesów czekających $P=\{P_0, P_1, ..., P_n\}$, takich że P_0 oczekuje na zasób będący w posiadaniu procesu P_1, P_1 zaś czeka na zasób, który jest w posiadaniu procesu P_2, ..., a P_n oczekuje na zwolnienie zasobu przez proces P_0 [1].

Najbardziej popularnym sposobem uniknięcia czekania cyklicznego jest uporządkowanie wszystkich typów zasobów i dopilnowanie, by każdy z procesów otrzymywał te zasoby rosnąco. Przyporządkowujemy więc jednoznacznie każdemu zasobowi należącemu do zbioru $Z=\{Z_0, Z_1, ..., Z_i\}$ liczbę całkowitą (w celu określenia rosnącego porządku procesów) [1]. Na początku proces może zamówić dowolną liczbę egzemplarzy wybranego typu zasobu. Potem proces może zamawiać jedynie egzemplarze tych typów, które mają większe numery. Jeśli na przykład na początku proces zamówił zasób Z_2, to później musi zamawiać zasoby $Z_x > Z_2$. Należy zaznaczyć przy tym, że możliwe jest zamówienie kilku egzemplarzy tego samego typu z użyciem jednego tylko zamówienia.

NASTĘPNA