Systemy operacyjne

wykład 10 - Synchronizacja procesów. Klasyczne problemy synchronizacji

dr Marcin Ziółkowski

Instytut Matematyki i Informatyki
Akademia im. Jana Długosza w Częstochowie

2 czerwca 2016 r.

PROBLEM SYNCHRONIZACJI PROCESÓW

Problem synchronizacji procesów pojawia się wszędzie tam, gdzie mamy do czynienia ze współpracującymi ze sobą współbieżnymi procesami. Oto najczęściej spotykane przyczyny, dla których konieczna jest synchronizacja współpracujących procesów:

- Procesy współdzielą pewną strukturę danych. Zwykle wykonania operacji na tej strukturze danych nie mogą dowolnie przeplatać się współbieżnie ze sobą. Pojedyncze operacje muszą być "zatomizowane", tzn. tylko jeden proces może na raz modyfikować współdzieloną strukturę danych, stąd konieczność synchronizacji procesów przy dostępie do struktury danych. Jest to przykład tzw. **problemu sekcji krytycznej**

PROBLEM SYNCHRONIZACJI PROCESÓW

- Wyniki działania jednego procesu stanowią dane dla innego procesu. Oczywiście drugi z procesów może przetwarzać dane dopiero wówczas, gdy zostaną one obliczone przez pierwszy z procesów, stąd konieczność synchronizacji działań obydwu procesów. Dodatkowo, jeżeli dane zajmują dużo pamięci i są obliczane stopniowo, to drugi z procesów może informować pierwszy, które dane zostały "zużyte" i można wykorzystać zajmowaną przez nie pamięć. Przykładem takiego współdziałania procesów jest **problem producenta-konsumenta**
- Procesy korzystają z pewnej wspólnej puli zasobów, które pobierają i zwalniają wedle potrzeb. Oczekiwanie na zasoby i przydzielanie ich wymaga również synchronizacji między procesami. Przykładem dobrze ilustrującym taki rodzaj synchronizacji jest **problem pięciu filozofów**

PROBLEM PRODUCENTA-KONSUMENTA

Rozważmy następujący problem. Mamy dane dwa procesy: producenta i konsumenta. Oba procesy działają w nieskończonej pętli. Producent wytwarza jednostki jakiegoś produktu, a konsument konsumuje je. Jednostki wyprodukowane, ale jeszcze nie skonsumowane są umieszczane w **buforze**. Bufor może pomieścić maksymalnie n jednostek produktu. Jeżeli bufor jest pusty, to konsument musi zaczekać, aż producent wyprodukuje kolejną jednostkę produktu. Jeżeli bufor jest pełny, to producent musi zaczekać, aż konsument pobierze jednostkę produktu z bufora. Jest to problem bardzo często spotykany w systemach operacyjnych. Choćby buforowanie znaków naciskanych na klawiaturze, czy wysyłanie wydruków na drukarkę to przykłady problemu producent-konsument.

PROBLEM SEKCJI KRYTYCZNEJ - CZYTELNICY I PISARZE

W problemie czytelników i pisarzy mamy dwie grupy procesów (czytelników i pisarzy) współzawodniczących o dostęp do pewnej sekcji krytycznej, którą będziemy nazywać biblioteką. Biblioteka jest o tyle specyficzną sekcją krytyczną, że może w niej przebywać równocześnie wielu czytelników. Jeżeli natomiast znajduje się w niej pisarz, to nie może w niej być nikogo innego. Powstaje pytanie, w jakiej kolejności możemy wpuszczać do biblioteki czytelników i pisarzy tak, aby nie było możliwości **zagłodzenia?**

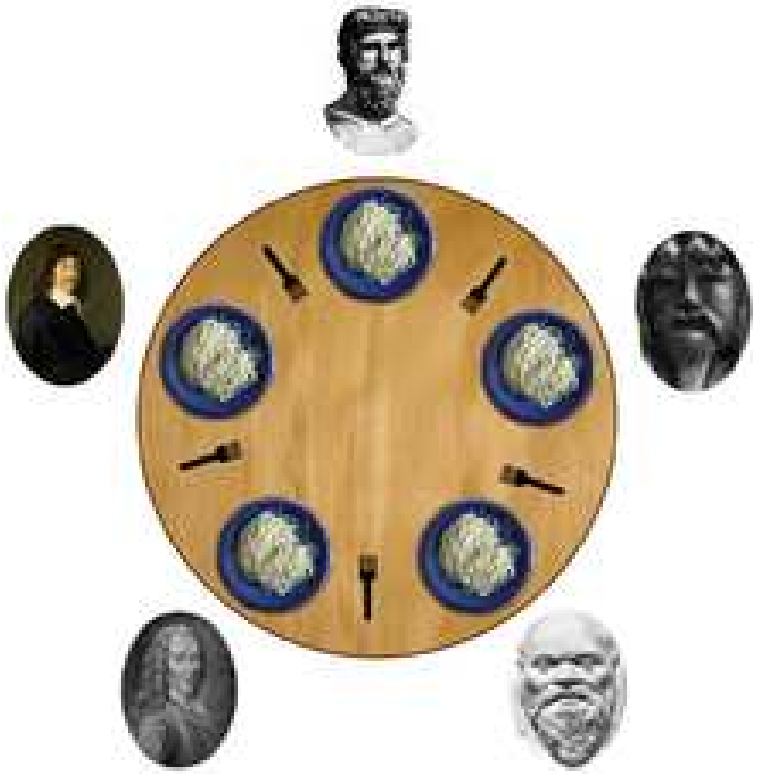
PROBLEM SEKCJI KRYTYCZNEJ - CZYTELNICY I PISARZE

Rozwiązanie problemu może być następujące: Jeżeli czytelnik chce wejść do biblioteki, a nie ma w niej pisarzy, ani też żaden pisarz nie czeka na wejście do biblioteki, to po prostu do niej wchodzi. W przeciwnym przypadku staje w kolejce oczekujących na wejście do biblioteki. Jeżeli pisarz chce wejść do biblioteki, a nie jest ona pusta lub ktoś już czeka na wejście do niej, to staje w kolejce oczekujących na wejście do biblioteki. Gdy pisarz opuszcza bibliotekę, to wchodzą do niej wszyscy oczekujący czytelnicy, lub jeśli ich brak, to pierwszy pisarz z kolejki. Gdy ostatni czytelnik wychodzi z biblioteki, to wchodzi do niej pierwszy pisarz z kolejki (o ile taki jest). W ten sposób, nawet jeżeli cały czas będą pojawiać się pisarze i czytelnicy chętni do skorzystania z biblioteki, to wchodzić do niej będą na zmianę: grupa czytelników, pisarz, grupa czytelników, pisarz itd., co gwarantuje, że nie nastąpi zagłodzenie.

PROBLEM PIĘCIU FILOZOFÓW

Problem pięciu filozofów, to akademicki problem synchronizacji, który okazał się bardzo dobrym sprawdzianem dla różnych metod synchronizacji. Mamy pięciu filozofów. Każdy z nich rozmyśla. Gdy zgłodnieje, to idzie jeść, a gdy się naje, to kontynuuje rozmyślania, aż znowu nie zgłodnieje itd. Filozofowie jedzą przy okrągłym stole przedstawionym na poniższym rysunku, przy czym każdy z nich ma swoje miejsce. Każdy z nich najpierw nakłada sobie jedzenie z miski stojącej na środku stołu, bierze dwie pałeczki leżące po lewej i prawej stronie talerza i je. Po zjedzeniu odkłada pałeczki na miejsce. Rzecz w tym, że na stole jest tylko pięć pałeczek, po jednej między dwoma sąsiednimi talerzami. Problem polega na takim zsynchronizowaniu poczynań filozofów, aby wykluczyć możliwość **blokady i zagłodzenia**.

PROBLEM PIĘCIU FILOZOFÓW



PROBLEM PIĘCIU FILOZOFÓW

Czy to faktycznie jest problem? Czy nie wystarczy, aby każdy filozof siadał na swoim miejscu, brał do ręki jedną pałeczkę (np. lewą), potem drugą i jadł? Oczywiście jeżeli z którejś pałeczki korzysta akurat jego kolega, to z filozoficznym spokojem poczeka aż on zje. Okazuje się, że przy takim rozwiązaniu możliwa jest **blokada (zakleszczenie)**. Przypuśćmy, że wszyscy filozofowie robią dokładnie to samo. Wszyscy na raz siadają przy stole, biorą do ręki pałeczkę z lewej strony i ... mamy blokadę. Każdy z nich czeka na pałeczkę leżącą z prawej strony, ale nigdy się jej nie doczeka.

PROBLEM PIĘCIU FILOZOFÓW

Czy można by uniknąć blokady, gdyby filozofowie brali pałeczki tylko po dwie naraz? Jeżeli jedna z pałeczek jest używana przez innego filozofa, to czeka on, aż obydwie będą leżeć na stole. Okazuje się, że wówczas możliwe jest zagłodzenie. Przypuśćmy, że jeden z filozofów siada do stołu i czeka na pałeczkę z prawej strony. W tym czasie z jego lewej strony siada jego kolega i zaczyna jeść, zabierając pałeczkę z lewej strony. Po chwili filozof z prawej strony odkłada pałeczkę, ale nasz filozof nie może jej wziąć, bo brak pałeczki po lewej stronie. Po chwili filozof z prawej strony wraca i zaczyna jeść, zabierając pałeczkę z prawej strony. Filozof z lewej strony kończy jeść i odkłada pałeczkę, ale teraz nie ma pałeczki z prawej strony, itd. Widzimy, że nasz filozof będzie czekał raz na pałeczkę z lewej, raz prawej strony, ulegając (dosłownie) **zagłodzeniu**. Istnieje wiele rozwiązań tego problemu. Jedno z nich polega na ograniczeniu liczby równocześnie jedzących filozofów do czterech. Powiedzmy, że każdy z filozofów zanim usiądzie, bierze talerz. **Wystarczy, że w jadalni będą tylko 4 talerze**.

DZIAŁANIE PROCESÓW WSPÓŁBIEŻNYCH

Analizując programy współbieżne zakładamy, że współbieżne wykonanie procesów może polegać na dowolnym przepleceniu wykonań poszczególnych procesów. Dodatkowo musimy pamiętać, że przeplatane nie są instrukcje języka programowania wysokiego poziomu, lecz instrukcje procesora w skompilowanym programie. Jeśli nasz program jest poprawny przy takim założeniu, to mamy pewność, że będzie działał poprawnie (niezależnie od realizacji systemu operacyjnego, parametrów technicznych komputera, zachodzących przerwań i innych procesów działających w systemie).

DZIAŁANIE PROCESÓW WSPÓŁBIEŻNYCH

W przypadku programów współbieżnych testowanie programów ma bardzo ograniczone zastosowanie. Nie jesteśmy w stanie przetestować wszystkich możliwych przeplotów wykonań procesów. Siłą rzeczy testy są przeprowadzane na konkretnej platformie i konkretnym komputerze. Tak więc pomyślne wyniki testów wcale nie gwarantują, że program będzie działał poprawnie na innej platformie, czy np. na szybszym komputerze. Jesteśmy więc skazani na weryfikowanie poprawności programów współbieżnych. Już w przypadku programów sekwencyjnych jest to trudne zadanie, a w przypadku programów współbieżnych jest to bardzo trudne.

PROBLEM SEKCJI KRYTYCZNEJ

Sekcja krytyczna – to fragment(y) kodu lub operacje, które nie mogą być wykonywane współbieżnie. Sekcja krytyczna jest otoczona dodatkowym kodem synchronizującym przebywanie procesów wejściowych – przed wejściem do sekcji krytycznej każdy proces musi przejść przez sekcję wejściową, a wychodząc przechodzi przez sekcję wyjściową. Dopóki jakiś proces przebywający w sekcji krytycznej nie opuści jej, inne procesy chcące wejść do sekcji krytycznej są wstrzymywane w sekcji wejściowej. Przechodząc przez sekcję wyjściową, proces opuszczający sekcję krytyczną wpuszcza jeden z procesów oczekujących (jeśli takowe są).

PROBLEM SEKCJI KRYTYCZNEJ

Zakładamy tutaj, że każdy proces, który wejdzie do sekcji krytycznej kiedyś ją opuści. Podstawowa własność, jaką powinna spełniać sekcja krytyczna, to **wzajemne wykluczanie**: tylko jeden proces na raz może przebywać w sekcji krytycznej. Gdyby było to jedyne wymaganie wobec sekcji krytycznej, to można by ją bardzo prosto zaimplementować: nie wpuszczać żadnych procesów do sekcji krytycznej. Oczywiście nie o to chodzi! Stąd też druga wymagana własność, to **wykorzystanie sekcji krytycznej**: jeśli jakiś proces chce wejść do sekcji krytycznej, to nie może ona pozostawać pusta.

PROBLEM SEKCJI KRYTYCZNEJ

Jeśli wiele procesów oczekuje na wejście do sekcji krytycznej, to nie jest określone w jakiej kolejności wejdą one do niej. Nie ma tu jednak zupełnej dowolności. Przyjęto najmniejsze wymagania zapewniające sensowne działanie sekcji krytycznej, brak zagłodzenia oczekujących procesów: każdy proces, który chce wejść do sekcji krytycznej ma gwarancję, że kiedyś do niej wejdzie. Rozwiązanie problemu sekcji krytycznej polega na podaniu implementacji sekcji wejściowej i wyjściowej.

ALGORYTM DEKKERA

Algorytm Dekkera to implementacja sekcji krytycznej dla dwóch procesów korzystająca ze zmiennych współdzielonych przez te procesy. Algorytm ten wykorzystuje trzy zmienne: tablicę dwuelementową k oraz zmienną **czyjakolej**. Początkowo komórki tablicy k są równe 1. Każdy z procesów kontroluje jedną komórkę tej tablicy. Gdy proces chce wejść do sekcji krytycznej, to ustawia komórkę na 0, aż do chwili wyjścia z sekcji krytycznej. Zmienna **czyjakolej** jest początkowo równa 1. Łamie ona symetrię między procesami. W sytuacji, gdy obydwa procesy chcą wejść do sekcji krytycznej zmienna ta rozstrzyga, który powinien ustąpić drugiemu. Zakładamy, że procesy mają numery 1 i 2, i każdy proces pamięta swój numer na zmiennej lokalnej p.

ALGORYTM DEKKERA - implementacja

sekcja wejściowa

```
k [p] := 0;
 while k [3 - p] = 0 do begin
   if czyja_kolej ? p then begin
     (* Ustąp drugiemu procesowi. *)
     k [p] := 1;
     while czyja_kolej ? p do;
     k [p] := 0
   end
 end
```

sekcja wyjściowa

```
k [p] := 1;
 czyja_kolej := 3 - p;
```

ALGORYTM DEKKERA - WADY

Rozwiązanie to jest poprawne, jednak ma pewne wady:

- jest ograniczone do dwóch procesów,
- proces oczekujący oczekuje aktywnie, tzn. czekając na jakieś zdarzenie sprawdza ciągle warunek określający, czy dane zdarzenie już zaszło.

Pierwszą z tych wad można usunąć – za chwilę zobaczymy jak. Druga wada jest jednak poważna. Oczekiwanie aktywne oznacza niepotrzebne zużycie czasu procesora. Nie można jej usunąć bez wprowadzenia dodatkowych mechanizmów programistycznych. Mechanizmy takie poznamy dalej.

ALGORYTM PIEKARNIANY

Algorytm piekarniany stanowi rozwiązanie problemu sekcji krytycznej dla n procesów. W polskich realiach algorytm ten mógłby nazywać się pocztowy, gdyż przypomina system numerków wydawanych oczekującym klientom w urzędach pocztowych. Algorytm ten korzysta z następujących zmiennych współdzielonych: tablicy logicznej $n-$ elementowej **wybieranie**, tablicy całkowitej $n-$ elementowej **numerek**. Początkowo elementy tych tablic są równe odpowiednio **false** i 0. Procesy są ponumerowane od 1 do n. Zakładamy, że każdy proces pamięta swój numer na zmiennej lokalnej p. Procesy chcące wejść do sekcji krytycznej "pobierają numerki". W tablicy wybieranie proces zaznacza, iż jest właśnie w trakcie pobierania numerka, a pobrany numerek zapamiętuje w tablicy numerek. Proces wchodzi do sekcji krytycznej, gdy upewni się, że jest pierwszy w kolejce. Po wyjściu z sekcji krytycznej proces "wyrzuca numerek", tzn. zapisuje w tablicy numerek wartość 0.

ALGORYTM PIEKARNIANY

Jest pewna różnica między algorytmem piekarnianym, a systemem numerków stosowanym w urzędach pocztowych. Może się zdarzyć, że dwa procesy otrzymają ten sam numerek. Mamy gwarancję, że kolejne numerki tworzą ciąg niemalejący, możliwe są w nim jednak powtórzenia. Jeżeli dwa procesy otrzymają ten sam numerek, to pierwszeństwo ma proces o niższym numerze (p). Możemy więc powiedzieć, że procesy wchodzą do sekcji krytycznej w porządku leksykograficznym par (numerek, numer procesu). Para (a, b) poprzedza w porządku leksykograficznym parę (c, d), co oznaczamy przez $(a, b) < (c, d)$, gdy $a < c$, lub $a = c$ oraz $b < d$.

ALGORYTM PIEKARNIANY - IMPLEMENTACJA

sekcja wejściowa

```
wybieranie [p] := true;
 numerek [p] := max (numerek [1], ..., numerek [n]) + 1;
 wybieranie [p] := false;
 for j := 1 to n do begin
   while wybieranie [j] do;
   while numerek [j]?0 and (numerek [j],j)<(numerek [p],p) do;
 end
```

sekcja wyjściowa

```
numerek [p] := 0;
```

Proces czekający na wejście do sekcji krytycznej z każdym krokiem pętli for upewnia się/czeka aż proces nr j nie stoi przed nim w kolejce do sekcji krytycznej. Algorytm piekarniany jest bardziej ogólny niż algorytm Dekkera, lecz również nie jest wolny od wad: nadal mamy do czynienia z aktywnym oczekiwaniem, ponadto liczba procesów jest ograniczona przez stałą n.

MECHANIZMY SYNCHRONIZACJI - SEMAFORY

Nazwa dobrze oddaje naturę semaforów – jest to mechanizm synchronizacji idealnie pasujący do rozwiązywania problemu sekcji krytycznej, a jego działanie przypomina działanie semafora kolejowego. Wyobraźmy sobie wielotorową magistralę kolejową, która na pewnym odcinku zwęża się do k torów. Pociągi jeżdżące magistralą to procesy. Zwężenie to uogólnienie sekcji krytycznej. Na zwężonym odcinku może znajdować się na raz maksymalnie k pociągów, każdy na oddzielnym torze. Podobnie jak w przypadku sekcji krytycznej, każdy oczekujący pociąg powinien kiedyś przejechać i żaden pociąg nie powinien stać, jeżeli jest wolny tor. Semafor to specjalna zmienna całkowita, początkowo równa k. Specjalna, ponieważ (po zainicjowaniu) można na niej wykonywać tylko dwie operacje:

MECHANIZMY SYNCHRONIZACJI - SEMAFORY

- P – Jeżeli semafor jest równy 0 (semafor jest opuszczony), to proces wykonujący tę operację zostaje wstrzymany, dopóki wartość semafora nie zwiększy się (nie zostanie on podniesiony). Gdy wartość semafora jest dodatnia (semafor jest podniesiony), to zmniejsza się o 1 (pociąg zajmuje jeden z k wolnych torów).
- V – Wartość semafora jest zwiększana o 1 (podniesienie semafora). Jeżeli jakiś proces oczekuje na podniesienie semafora, to jest on wznawiany, a semafor natychmiast jest zmniejszany.

MECHANIZMY SYNCHRONIZACJI - SEMAFORY

Istotną cechą operacji na semaforach jest ich niepodzielność. Jeżeli jeden z procesów wykonuje operację na semaforze, to (z wyjątkiem wstrzymania procesu, gdy semafor jest opuszczony) żaden inny proces nie wykonuje w tym samym czasie operacji na danym semaforze. Aby zapewnić taką niepodzielność, potrzebne jest specjalne wsparcie systemowe lub sprzętowe. Rozwiązanie problemu sekcji krytycznej za pomocą semaforów jest banalnie proste. Wystarczy dla każdej sekcji krytycznej utworzyć odrębny semafor, nadać mu na początku wartość 1, w sekcji wejściowej każdy proces wykonuje na semaforze operację P, a w sekcji wyjściowej operację V. Wymaga to jednak od programisty pewnej dyscypliny: Każdej operacji P musi odpowiadać operacja V i to na tym samym semaforze. Pomyłka może spowodować złe działanie sekcji krytycznej. Semafory są tak często stosowane do rozwiązywania problemu sekcji krytycznej, że spotyka się ich uproszczoną wersję: semafory binarne. Semafor binarny, to taki semafor, który może przyjmować tylko dwie wartości: 0 i 1. Próba podniesienia semafora równego 1, w zależności od implementacji, nie zmienia jego wartości lub powoduje błąd.

MECHANIZMY SYNCHRONIZACJI - KOLEJKI KOMUNIKATÓW

Kolejki komunikatów przypominają uporządkowane skrzynki na listy. Podstawowe dwie operacje, jakie można wykonywać na kolejce, to wkładanie i wyjmowanie komunikatów (pakietów informacji) z kolejki. Komunikaty są uporządkowane w kolejce na zasadzie FIFO (ang. first in, first out). Włożenie powoduje dołączenie komunikatu na koniec kolejki, a pobranie powoduje wyjęcie pierwszego komunikatu z kolejki. Jeżeli kolejka jest pusta, to próba pobrania komunikatu powoduje wstrzymanie procesu w oczekiwaniu na komunikat. Możliwych jest wiele różnych implementacji kolejek komunikatów, różniących się zestawem dostępnych operacji i ew. ograniczeniami implementacyjnymi. Pojemność kolejki może być ograniczona. Wówczas próba włożenia komunikatu do kolejki powoduje wstrzymanie procesu w oczekiwaniu na pobranie komunikatów. Niektóre implementacje mogą udostępniać nieblokujące operacje wkładania/pobierania komunikatów – wówczas zamiast wstrzymywania procesu przekazywana jest informacja o niemożności natychmiastowego wykonania operacji. Możliwe jest też selektywne pobieranie komunikatów.

MECHANIZMY SYNCHRONIZACJI - KOLEJKI KOMUNIKATÓW

Kolejki komunikatów są równie silnym mechanizmem synchronizacji, co semafory. Mając do dyspozycji semafory można zaimplementować kolejki komunikatów. Odwrotna konstrukcja jest również możliwa. Rozważmy kolejkę komunikatów, do której wkładamy tylko puste, nic nie znaczące komunikaty. Wówczas kolejka taka będzie zachowywać się jak semafor. Chcąc zainicjować ją na określoną wartość, należy na początku włożyć do niej określoną ilość pustych komunikatów.

MECHANIZMY SYNCHRONIZACJI - MONITORY

Monitory to przykład strukturalnego mechanizmu synchronizacji wbudowanego w języki programowania wysokiego poziomu. Monitor to rodzaj klasy, której metody stanowią sekcję krytyczną. (Mówiąc o klasach, pomijamy tutaj zjawisko dziedziczenia.) Atrybuty monitora są dostępne jedynie wewnątrz (metod) monitora. Konstrukcja monitora zapewnia, że tylko jeden proces na raz może znajdować się w monitorze. Pozostałe procesy oczekują w kolejce (FIFO). Jeśli jakiś proces chce wejść do monitora, który jest właśnie zajęty, to jest on wstawiany na koniec kolejki procesów oczekujących na wejście do monitora. Jeśli proces opuszcza monitor, a inne procesy czekają w kolejce na wejście do monitora, to pierwszy proces z kolejki wchodzi do monitora.

MECHANIZMY SYNCHRONIZACJI - MONITORY

Monitory stanowią wygodniejsze rozwiązanie problemu sekcji krytycznej niż semafory, ze względu na ich strukturalny charakter. Stosując semafory można popełnić szereg błędów programistycznych: zapomnieć o podniesieniu czy opuszczeniu semafora, pomylić semafory, czy np. pomylić operacje P i V. W przypadku monitorów odpowiednie instrukcje synchronizujące są realizowane przez język programowania. Monitory nie sprowadzają się tylko do strukturalnej sekcji krytycznej. Udostępniają one dodatkowo kolejki procesów typu condition. Kolejki takie mogą być używane tylko wewnątrz monitorów. Są to kolejki FIFO. Kolejki condition udostępniają dwie operacje:

- **wait** – powoduje wstrzymanie procesu i wstawienie go na koniec kolejki,
- **signal** – powoduje wznowienie pierwszego procesu z kolejki, o ile kolejka nie jest pusta.

Jeżeli w momencie wstrzymania procesu znajdującego się w monitorze jakiekolwiek procesy oczekują na wejście do monitora, to pierwszy z oczekujących procesów wchodzi do monitora. W momencie, gdy proces znajdujący się w monitorze wykona operację signal na (niepustej) kolejce, to nagle w monitorze mamy dwa procesy. Z drugiej strony, w monitorze może przebywać tylko jeden proces na raz. Istnieją dwa rozwiązania tego problemu:

- Operacja signal może być wykonywana tylko jako ostatnia w monitorze. Po jej wykonaniu proces natychmiast opuszcza monitor. Rozwiązanie to jest o tyle ograniczone, że jeden proces może wznowić tylko jeden inny proces.
- Wznawiany proces "wypycha" z monitora proces, który wykonał operację signal. Jednak wypychany proces jest wstawiany na początek kolejki procesów oczekujących na wejście do monitora. Dzięki temu jesteśmy w stanie prześledzić co się będzie działo w monitorze do momentu ponownego wejścia wypchniętego procesu.