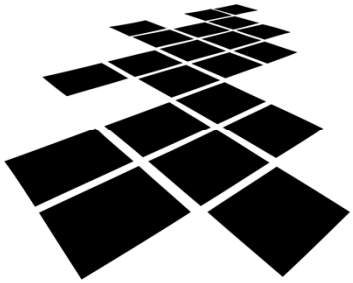
Systemy operacyjne

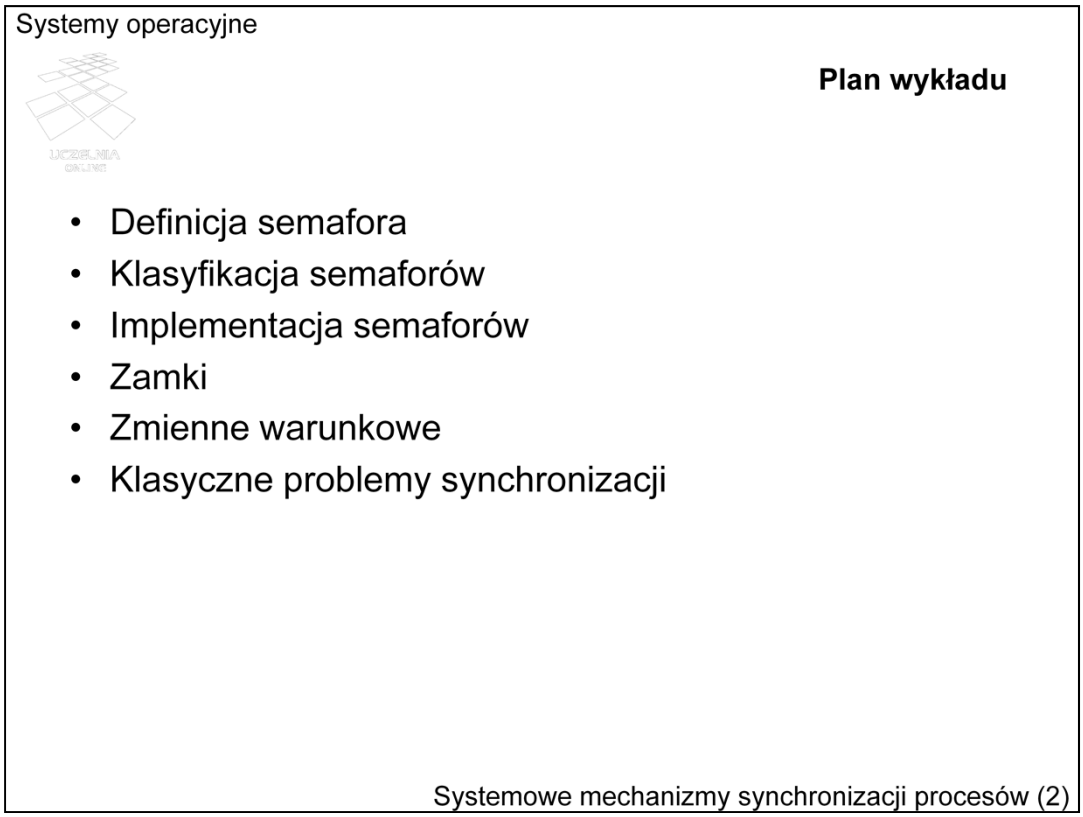
Systemowe mechanizmy synchronizacji procesów

Wykład prowadzą:
Jerzy Brzeziński
Dariusz Wawrzyniak

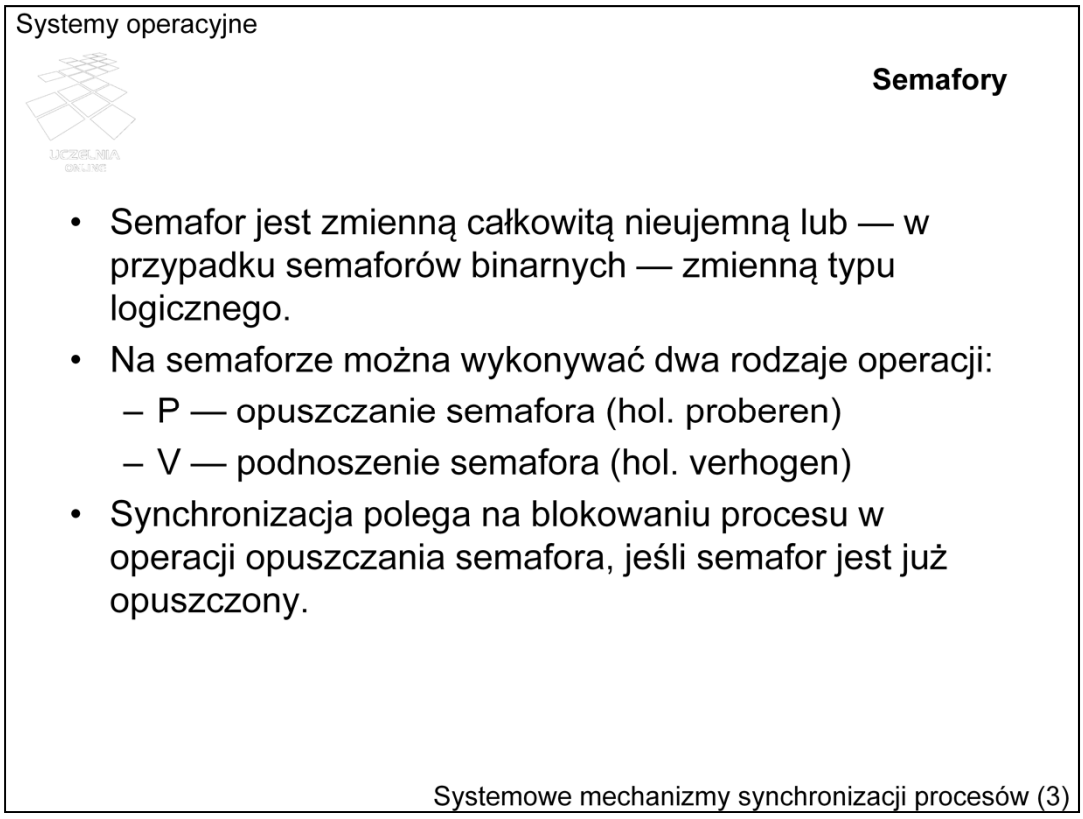


UCZELNIA
ONLINE

Celem wykładu jest przedstawienie mechanizmów synchronizacji, które mogą być implementowane zarówno na poziomie architektury komputera, jak i przy wsparciu systemu operacyjnego, polegającego na odpowiednim zarządzaniu synchronizowanymi procesami. Należą do nich semafory oraz mechanizmy zdefiniowane w ramach standardu POSIX, czyli zamki oraz zmienne warunkowe.



Wykład składa się z trzech części. Pierwsza obejmuje semafory, ich abstrakcyjną definicję, klasyfikację oraz sposoby implementacji. Druga dotyczy mechanizmów standardu POSIX i sposobu ich użycia w synchronizacji. Ostatnia z zasadniczych części wykładu obejmuje klasyczne problemy synchronizacji procesów — producenta i konsumenta, czytelników i pisarzy, pięciu filozofów oraz śpiących fryzjerów. Przedstawione zostaną rozwiązania tych problemów z użyciem semaforów. Na koniec pojawi się krótka wzmianka o strukturalnych mechanizmach synchronizacji.



Semafor, jako mechanizm synchronizacji procesów, został zaproponowany przez Dijkstrę.

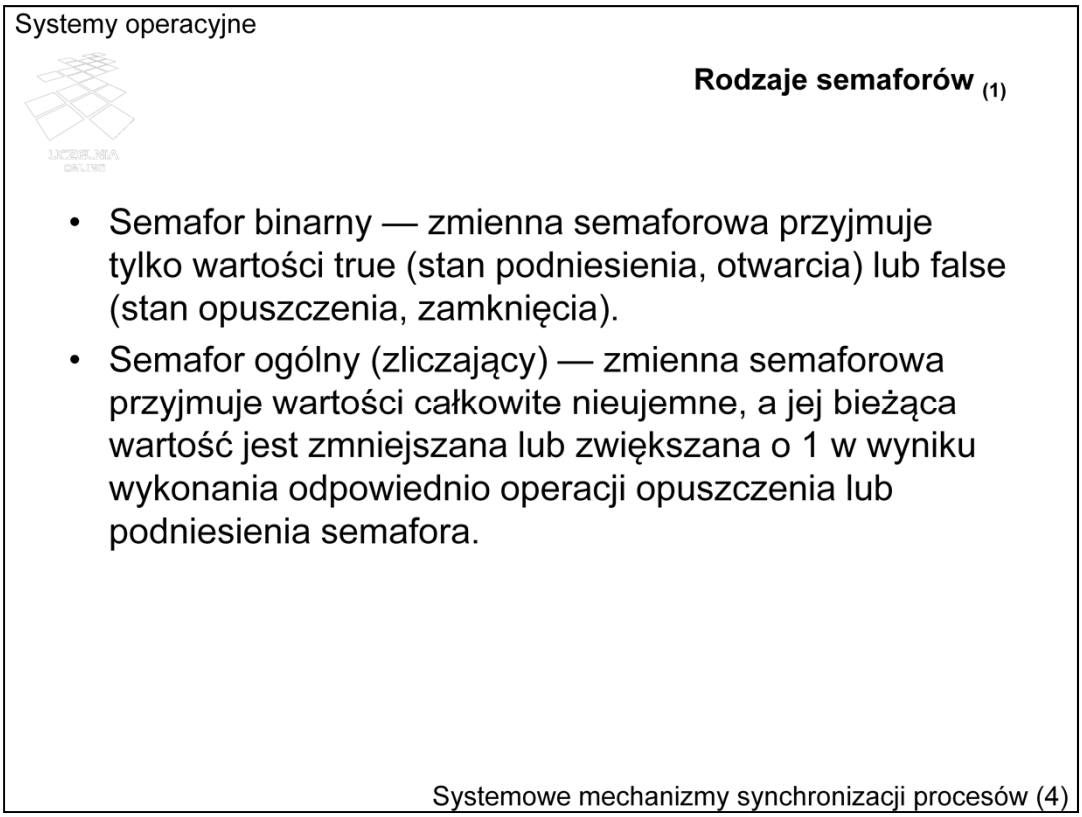
Semafor jest zmienną całkowitą, która z logicznego punktu widzenia (z punktu widzenia aplikacji) przyjmuje wartości nieujemne (≥0) lub — w przypadku semaforów binarnych — logiczne. Zmienna semaforowa musi mieć nadaną początkową wartość (oczywiście nieujemną).

Po nadaniu początkowej wartości zmiennej semaforowej można na niej wykonywać tylko dwa rodzaje operacji:

P — opuszczanie semafora (hol. proberen),

V — podnoszenie semafora (hol. verhogen).

Operacja opuszczania powoduje zmniejszenie wartości zmiennej semaforowej, a operacja podnoszenia jej zwiększenie. Wykonując operację semaforową, proces może zastać zablokowany (przejść w stan oczekiwania). Typowym przypadkiem jest blokowanie w operacji opuszczania semafora. Operacja opuszczania nie zakończy się do czasu, aż wartość zmiennej semaforowej będzie na tyle duża (być może zostanie zwiększona w międzyczasie), że zmniejszenie jej wartości w wyniku tej operacji nie spowoduje przyjęcia wartości ujemnej. W przypadku semaforów dwustronnie ograniczonych blokowanie może wystąpić również w przypadku podnoszenia semafora.



Typowym semaforem jest semafor binarny, który może mieć dwa stany: podniesiony (otwarty) i opuszczony (zamknięty). Wielokrotne podnoszenie takiego semafora nie zmieni jego stanu — skutkiem będzie stan otwarcia. W niektórych rozwiązaniach przyjmuje się, że próba podniesienia otwartego semafora sygnalizowana jest błędem.

W przeciwieństwie do semafora binarnego, semafor ogólny „pamięta” liczbę operacji podniesienia. Przy wartości początkowej 0 można zatem bez blokowania procesu wykonać tyle operacji opuszczenia semafora, ile razy został on wcześniej podniesiony. Stąd określenie — *semafor zliczający*.

Systemy operacyjne

Rodzaje semaforów (2)

- Semafor uogólniony — semafor zliczający, w przypadku którego zmienną semaforową można zwiększać lub zmniejszać o dowolną wartość, podaną jako argument operacji.
- Semafor dwustronnie ograniczony — semafor ogólny, w przypadku którego zmienna semaforowa, oprócz dolnego ograniczenia wartością 0, ma górne ograniczenie, podane przy definicji semafora.

Systemowe mechanizmy synchronizacji procesów (5)

Semafor uogólniony można zwiększać lub zmniejszać o dowolną podaną wartość pod warunkiem, że w wyniku zmniejszenia zmienna semaforowa nie osiągnie wartości ujemnej. Jeśli zatem wartość parametru, o którą ma być zmniejszona zmienna semaforowa jest większa od wartości tej zmiennej, następuje zablokowanie procesu.

Dla semafora dwustronnie ograniczonego definiuje się górne ograniczenie, po osiągnięciu którego następuje blokowanie procesu również w operacji podnoszenia.

Systemy operacyjne

Implementacja semafora ogólnego na poziomie maszynowym

- Opuszczanie semafora

```
procedure P(var s: Semaphore)
   begin
      while s = 0 do nic;   } instrukcje muszą być
      s := s - 1;           } wykonane atomowo
   end;
```

- Podnoszenie semafora

```
procedure V(var s: Semaphore)
   begin
      s := s + 1;           } instrukcja musi być
   end;                     } wykonana atomowo
```

Systemowe mechanizmy synchronizacji procesów (6)

W implementacji na poziomie maszynowym semafor ogólny jest zmienną całkowitą nieujemną (teoretycznie nieograniczoną od góry), na której wykonywane są operacje *P* i *V*. Takie podejście do implementacji wynika zatem bezpośrednio z definicji.

W celu wyeksponowania przepływu sterowania i wskazania instrukcji, które muszą być wykonane niepodzielnie, implementację operacji *P* można przedstawić następująco:

```
                procedure P(var s: Semaphore)
                                  begin
      próbuj:
      zablokuj_obsługę_przerwań;
      if s = 0 then
                                     begin
            odblokuj_obsługę_przerwań;
            goto próbuj
         end;
                                  else
                                   begin
            s := s - 1;
            odblokuj_obsługę_przerwań
                                     end;
   end;
```

Ciąg instrukcji pomiędzy zablokowaniem, a odblokowaniem przerwań wykonywany jest niepodzielnie. Istotne jest zatem niepodzielne sprawdzenie i zmniejszenie wartości zmiennej *s* (pod warunkiem, że jest większa od 0).

Skok do linii zaetykietowanej jako *próbuj* oznacza aktywne czekanie.

Systemy operacyjne

Implementacja semafora binarnego na poziomie maszynowym

- Opuszczanie semafora

```
procedure P(var s: Binary_Semaphore)
   begin
      while not s do nic;   } instrukcje muszą być
      s := false;           } wykonane atomowo
   end;
```

- Podnoszenie semafora

```
procedure V(var s: Binary_Semaphore)
   begin
     s := true;             } instrukcja musi być
   end;                     } wykonana atomowo
```

Systemowe mechanizmy synchronizacji procesów (7)

Zasada implementacji jest dokładnie taka sama, jak w przypadku semafora ogólnego, różnią się tylko wartości zmiennej *s*.

Systemy operacyjne

Implementacja semafora ogólnego na poziomie systemu operacyjnego (1)

- Struktury danych

```
type Semaphore = record
    wartość: Integer;
    L: list of Proces;
end;
```

- Opuszczanie semafora

```
procedure P(var s: Semaphore) begin
    s.wartość := s.wartość - 1;
    if s.wartość < 0 then begin
        dołącz dany proces do s.L
        zmień stan danego procesu na „oczekujący”
    end;
end;
```

Systemowe mechanizmy synchronizacji procesów (8)

Celem implementacji na poziomie systemu operacyjnego jest zlikwidowanie aktywnego czekania i związanego z tym marnotrawstwa czasu procesora.

Zamiast permanentnego testowania zmiennej semaforowej, stan procesu zmieniany jest na *oczekujący*, w związku z czym planista przydziału procesora nie uwzględnia go, wybierając proces do wykonania. Z semaforem wiąże się kolejka procesów oczekujących na jego podniesienie. W definicji struktury danych na potrzeby semafora użyto abstrakcyjnej konstrukcji **`list of`**. Kolejkę taką można zbudować w oparciu o odpowiednie pola do wskazywania procesów, przechowywane w deskryptorze procesu. W strukturze semaforowej jest wówczas tzw. głowa listy, czyli wskaźnik na deskryptor pierwszego z oczekujących procesów.

W samej implementacji operacji opuszczania interesujący jest sposób modyfikacji pola *wartość* struktury semaforowej. Jest ono zmniejszane bezwarunkowo i może osiągnąć wartość ujemną. Interpretacja wartości tego pola jest następująca:

- wartość dodatnia oznacza, że semafor jest podniesiony i przy takim stanie proces nie jest blokowany w operacji opuszczania,
- wartość ujemna oznacza, że semafor jest opuszczony, są procesy oczekujące na podniesienie semafora, a ich liczba jest równa wartości bezwzględnej pola *wartość*,
- wartość 0 oznacza, że semafor jest opuszczony ale nie ma procesów oczekujących na jego podniesienie (jest to szczególny przypadek poprzedniego punktu).

Systemy operacyjne

Implementacja semafora ogólnego na poziomie systemu operacyjnego (2)

- Podnoszenie semafora

```
procedure V(var s: Semaphore)
  begin
    s.wartość := s.wartość + 1;
    if s.wartość ≤ 0 then begin
      wybierz i usuń jakiś/kolejny proces z kolejki s.L
      zmień stan wybranego procesu na „gotowy”
    end;
  end;
```

Systemowe mechanizmy synchronizacji procesów (9)

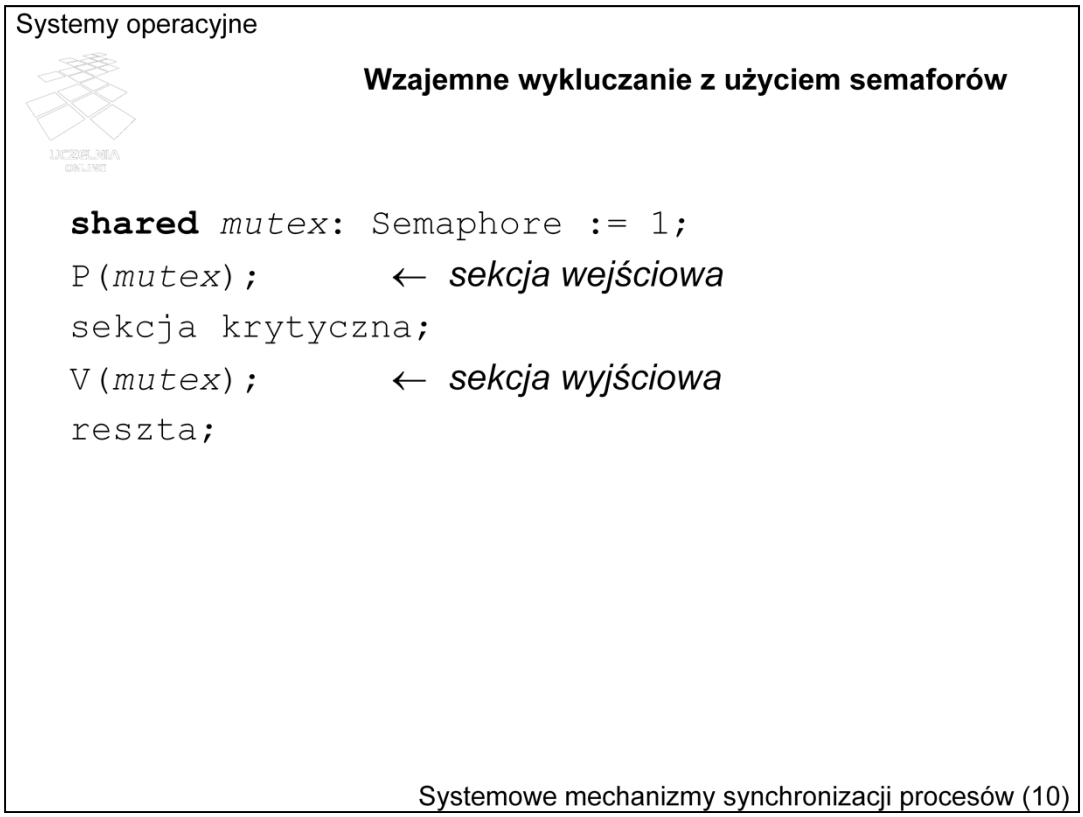
Zwiększenie wartości zmiennej semaforowej (pole *wartość* struktury semafora) jest operacją oczywistą, ale wartość ta mogła być ujemna, co oznacza, że w kolejce są procesy oczekujące na podniesienie semafora. Wówczas z kolejki wybierany jest jeden z procesów, który jest z niej usuwany, a następnie jest budzony (jego stan z *oczekujący* zmienia się na *gotowy*).

W podobny sposób można zrealizować semafor binarny, z tą różnicą, że jeśli pole *wartość* jest równe 1 (co oznacza otwarcie), a wykonywana jest operacja podnoszenia, to stan semafora nie ulega zmianie, czyli:

```
procedure V(var s: Semaphore)
                begin
              if s.wartość < 1 then
          s.wartość := s.wartość + 1;
        if s.wartość ≤ 0 then begin
    wybierz i usuń jakiś/kolejny proces z kolejki s.L
        zmień stan wybranego procesu na „gotowy”
                end;
  end;
```

Nieco bardziej skomplikowany jest przypadek implementacji semaforów uogólnionych. Pozostawia się to jako ćwiczenie. Warto mieć jednak na uwadze dwa kryteria przy podnoszeniu semafora:

- zwiększanie przepustowości i budzenie procesów, dla których zmienna semaforowa osiągnęła wystarczającą wartość, nawet jeśli nie znajdują się na czele kolejki (ryzyko głodzenia procesów, które chcą opuścić semafor o relatywnie dużą wartość),
- zachowanie sprawiedliwości i budzenie procesów w kolejności ich kolejkowania.



Wzajemne wykluczanie z użyciem semafora polega na opuszczaniu tego semafora w sekcji wejściowej i podnoszeniu w sekcji wyjściowej. Jeśli wartość początkowa semafora jest równa 1 (lub jest to semafor binarny, ustawiony początkowo na true), po wykonaniu jednej operacji opuszczania wartość zmiennej semaforowej osiągnie 0 i kolejne operacje opuszczania spowodują zablokowanie procesów. W ten sposób pozostałe procesy utkną w sekcji wejściowej do czasu, aż nie nastąpi podniesienie semafora. Podniesienie z kolei nastąpi dopiero po wyjściu z sekcji krytycznej i umożliwi jednemu z oczekujących procesów zakończenie operacji opuszczania.

Algorytm taki można zastosować dla dowolnego zbioru procesów, ale gwarancja spełnienia warunku ograniczonego czekania zależy od sposobu implementacji operacji semaforowych. Jeśli procesy, zablokowane pod semaforem, budzone są w kolejności FIFO, warunek ograniczonego czekania jest spełniony. Gdyby semafor implementowany był na poziomie maszynowym, jak to przedstawiono wcześniej, nie ma gwarancji ograniczonego czekania.

Systemy operacyjne

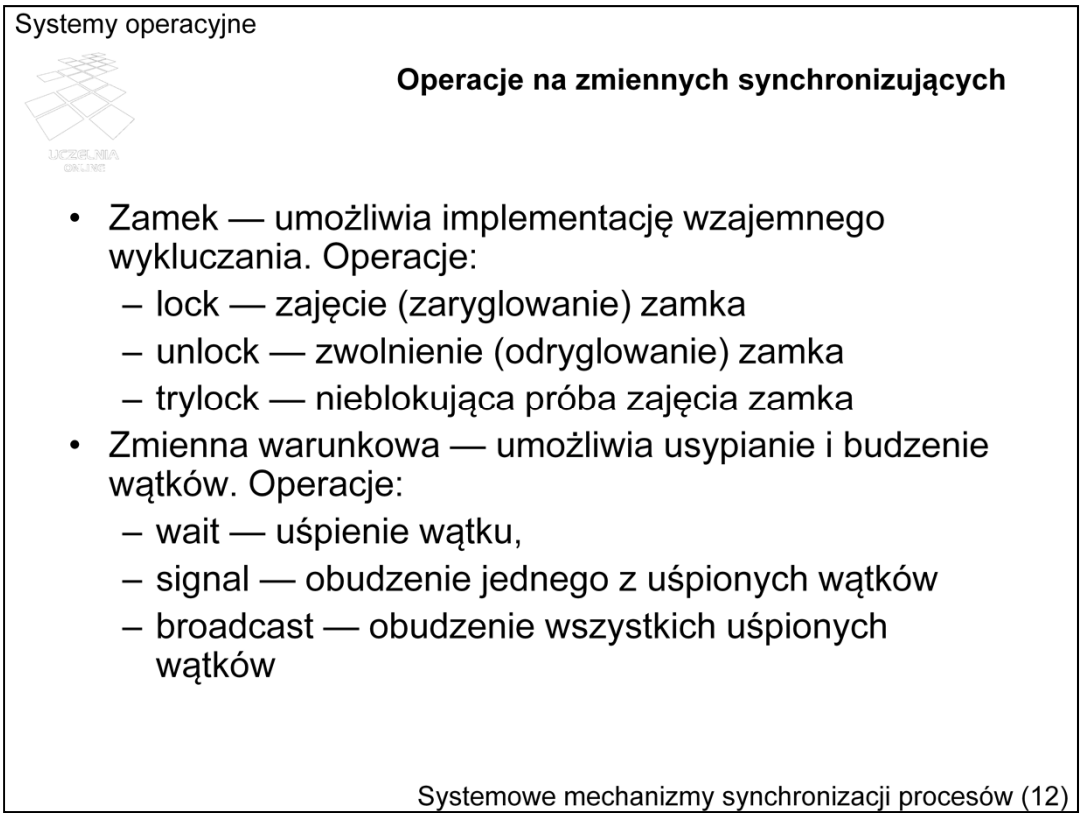
Mechanizmy synchronizacji POSIX — zmienne synchronizujące

- Rodzaje zmiennych synchronizujących:
  - zamek — umożliwiająca implementację wzajemnego wykluczania,
  - zmienna warunkowa — umożliwia usypianie i budzenie wątków.
- Zmienne synchronizujące muszą być współdzielone przez synchronizowane wątki.
- Zanim zmienna zostanie wykorzystana do synchronizacji musi zostać zainicjalizowana.

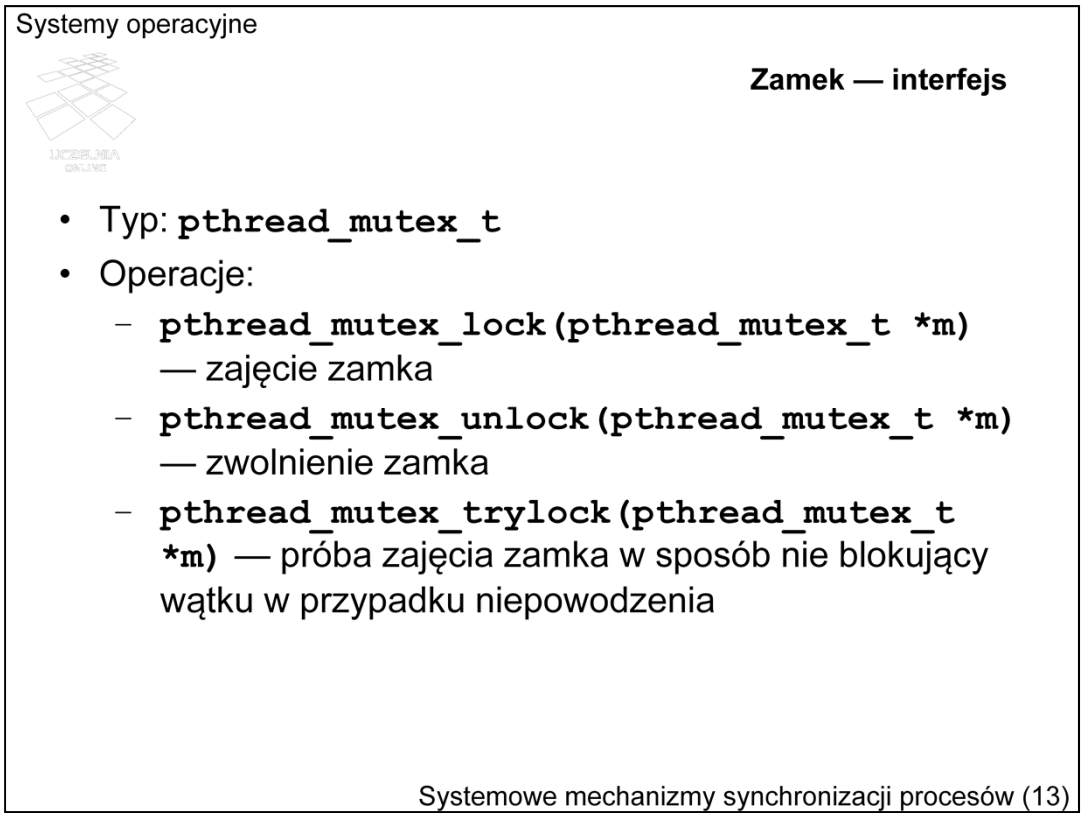
Systemowe mechanizmy synchronizacji procesów (11)

Mechanizmy synchronizacji są częścią standardu POSIX związaną z obsługą wielowątkowości. Mechanizmy te powstały zatem na potrzeby synchronizacji wątków i w takim kontekście będą rozważane w dalszej części.

Jeśli zmienne mają być użyte do synchronizacji, muszą znajdować się w obszarze pamięci dostępnym dla synchronizowanych wątków. We wszystkich operacjach zmienne synchronizujące udostępniane są przez wskaźniki. Użycie zmiennej synchronizującej musi być poprzedzone jej inicjalizacją.



Zamki są podobne do semaforów binarnych i używane są do zapewnienie wzajemnego wykluczania. Zmienne warunkowe używane są wówczas, gdy stan przetwarzania uniemożliwia wątkowi dalsze działanie i wątek musi wejść w stan oczekiwania.



Ponieważ interfejs operacji synchronizujących w standardzie POSIX wyspecyfikowany jest w języku C, w takiej samej formie będzie zaprezentowany na kolejnych slajdach.

Zamek jest obiektem typu `pthread_mutex_t` i do wszystkich operacji na nim jest przekazywany przez wskaźnik. Przed użyciem zamek powinien zostać zainicjalizowany. Najprostszy sposób, to podstawienie odpowiedniej stałej inicjalizującej przy jego definicji. Nieco bardziej złożone jest użycie odpowiedniej funkcji.

Jak już wspomniano, działanie zamka zbliżone jest do działania semafor binarnego. Operacja zaryglowania (zajmowania zamka, ang. lock) jest odpowiednikiem opuszczania semafora, a operacja odryglowania (zwalniania zamka, ang. unlock) jest odpowiednikiem podnoszenia semafora. Jednak z zamkiem, w przeciwieństwie do semafora binarnego, wiąże się zasada własności. Ten wątek, który zajmie zamek (zarygluje, wykona lock), musi go zwolnić (odryglować, wykonać unlock). Podnieść i opuścić semafor może natomiast dowolny proces.

Biorąc pod uwagę te analogie funkcja `pthread_mutex_lock` powoduje zajęcie (zaryglowanie) zamka, wskazywanego przez parametr, jeśli zamek ten jest zwolniony. W przeciwnym przypadku następuje zawieszenie wątku w oczekiwaniu na zwolnienie (odryglowanie). Zwolnienie zamka z kolei następuje w wyniku wywołania funkcji `pthread_mutex_unlock` przez ten wątek, który wcześniej zamek zajął. Funkcja `pthread_mutex_trylock` służy również do zajmowania zamka, ale nie powoduje zablokowania wątku w przypadku, gdy zamek jest zajęty. Zwracany jest wówczas tylko odpowiedni status zakończenia operacji, po którym można rozpoznać efekt.

Systemy operacyjne

Zamek — implementacja

- **`pthread_mutex_lock`**
  - zajęcie zamka, jeśli jest wolny
  - ustawienie stanu wątku na oczekujący i umieszczenie w kolejce, jeśli zamek jest zajęty
- **`pthread_mutex_unlock`**
  - ustawienie zamka na wolny, jeśli kolejka oczekujących wątków jest pusta
  - wybranie wątku z niepustej kolejki wątków oczekujących i ustawienie jego stanu na gotowy.
- **`pthread_mutex_trylock`**
  - zajęcie zamka lub kontynuacja przetwarzania

Systemowe mechanizmy synchronizacji procesów (14)

Implementacja zamka jest zbliżona do implementacji semafora binarnego z tą różnicą, że w celu weryfikacji własności w odpowiedniej strukturze musiałby być przechowywany identyfikator wątku, który zajął zamek. Próba zwolnienia zamka poprzez wywołanie funkcji `pthread_mutex_unlock` wymagałaby zatem weryfikacji właściciela i ewentualnego zwrócenia błędu.

Systemy operacyjne

Zmienna warunkowa — interfejs

- Typ: **`pthread_cond_t`**
- Operacje:
  - **`pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`** — uśpienie wątku na zmiennej warunkowej,
  - **`pthread_cond_signal(pthread_cond_t *c)`** — obudzenie jednego z wątków uśpionych na zmiennej warunkowej,
  - **`pthread_cond_broadcast(pthread_cond_t *c)`** — obudzenie wszystkich wątków uśpionych na zmiennej warunkowej.

Systemowe mechanizmy synchronizacji procesów (15)

Zmienna warunkowa jest obiektem typu `pthread_cond_t` i podobnie, jak w przypadku zamaka:

- do wszystkich operacji jest przekazywana przez wskaźnik,
- przed użyciem powinna zostać zainicjalizowana (przez podstawienie właściwej stałej lub użycie odpowiedniej funkcji).

Zmienna warunkowa nie jest używana samodzielnie. Ze względu na ryzyko hazardu (różnicy w działaniu w zależności od kolejności operacji w przeplocie) pewne operacje na zmiennej warunkowej muszą być wykonywane sekcji krytycznej, chronionej przez zamek.

Wywołanie funkcji `pthread_cond_wait` powoduje wejście wątku w stan oczekiwania na sygnał, który z kolei musi wysłać inny wątek, wywołując funkcję `pthread_cond_signal` lub `pthread_cond_broadcast`. Na czas oczekiwania następuje zwolnienie zamka, do którego wskaźnik przekazywany jest jako drugi parametr funkcji `pthread_cond_wait`. Jak można się domyślać funkcja ta wywoływana jest w sekcji krytycznej. Dokładniej zostanie to omówiony przy schematach synchronizacji w dalszej części.

Funkcja `pthread_cond_signal` budzi jeden z oczekujących wątków, a `pthread_cond_broadcast` budzi wszystkie oczekujące wątki. Obudzenie wątku nie musi oznaczać natychmiastowego uzyskania stanu gotowości, podobnie jak sygnał budzika nie oznacza natychmiastowego wstania z łóżka. Wątek budzony musi jeszcze ponownie zająć zamek, który zwolnił na czas oczekiwania. Jeśli zamek jest zajęty musi poczekać na jego zwolnienie.

Systemy operacyjne

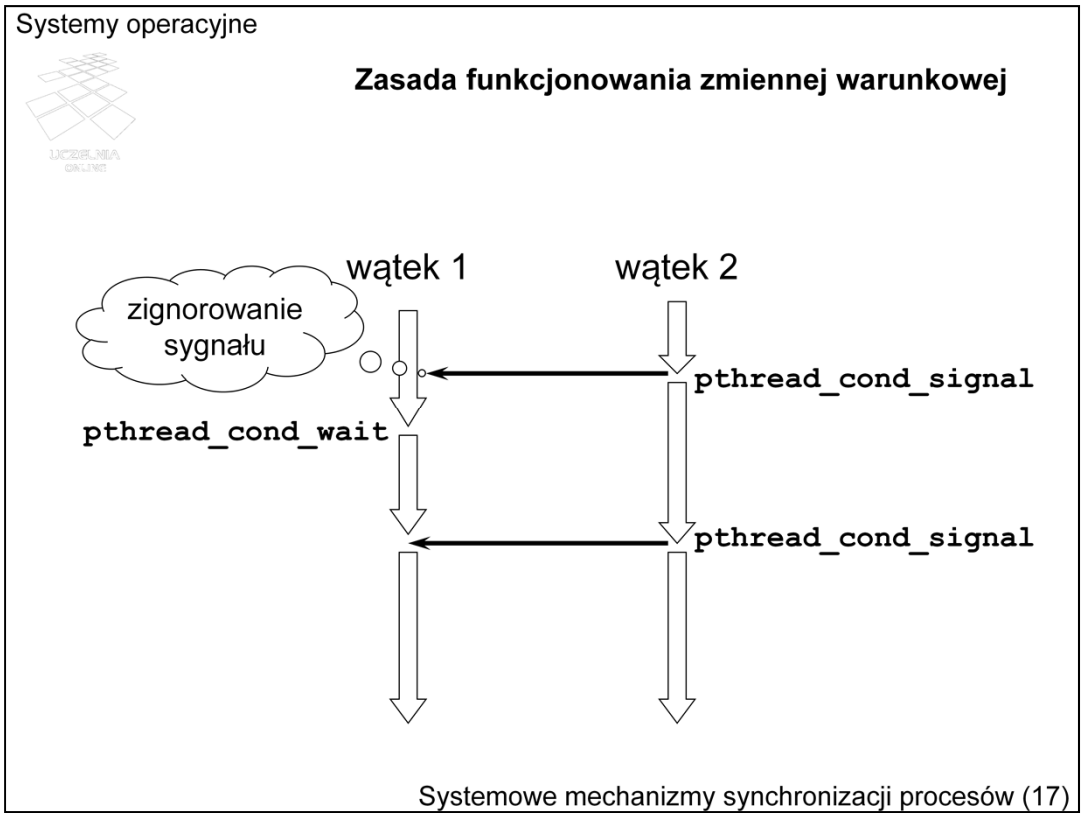
Zmienna warunkowa — implementacja

- **`pthread_cond_wait`**
  - ustawienie stanu wątku na oczekujący i umieszczenie go w kolejce
- **`pthread_cond_signal`**
  - wybranie jednego wątku z kolejki i postępowanie takie, jak przy zajęciu zamka
  - zignorowanie sygnału, jeśli kolejka jest pusta
- **`pthread_cond_broadcast`**
  - ustawienie wszystkich wątków oczekujących na zmiennej warunkowej w kolejce do zajęcia zamka, a jeśli zamek jest wolny zmiana stanu jednego z nich na gotowy.

Systemowe mechanizmy synchronizacji procesów (16)

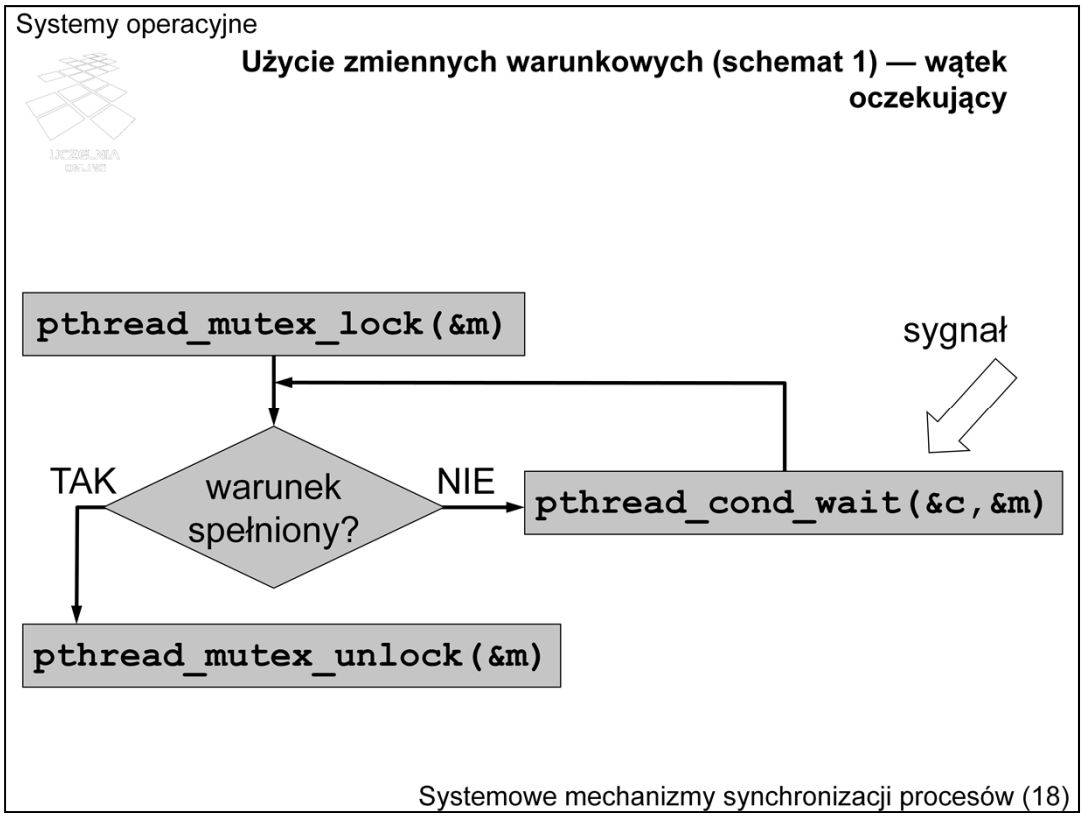
W kontekście przedstawionych wcześniej implementacji semaforów i zamków powyższy opis jest dość oczywisty. Warto tylko zwrócić uwagę na dwie sprawy:

1. Wejście w stan oczekiwania po wywołaniu funkcji `pthread_cond_wait` jest bezwarunkowe.
2. Opuszczenie funkcji `pthread_cond_wait` zależy od możliwości zajęcia zamka. Po otrzymaniu sygnału wątek zachowuje się tak, jak w przypadku wywołania funkcji `pthread_mutex_lock`. Wybór następnego wątku do zajęcia zamka po jego zwolnieniu w innym wątku (w wyniku wywołania `pthread_mutex_unlock` lub `pthread_cond_wait`) zależy od polityki szeregowania.



Działanie zmiennej warunkowej kojarzone jest niekiedy z semaforem. W operacjach semaforowych używa się czasami takich samych nazw, czyli *wait* — opuszczanie oraz *signal* — podnoszenie.

Zasadnicza różnica pomiędzy zmienna warunkową a semaforem polega na tym, że sygnał na zmiennej warunkowej budzi oczekujący wątek lub jest ignorowany, jeśli żaden wątek nie oczekuje na tej zmiennej. Efekt operacji podnoszenia semafora jest natomiast odzwierciedlany w stanie semafora i może być „skonsumowany” przez późniejsza operację opuszczenia. W przypadku zmiennej warunkowej istnieje zatem ryzyko hazardu — jeśli wątek nie zdąży usnąć na zmiennej warunkowej, można stracić sygnał, który miał go obudzić.

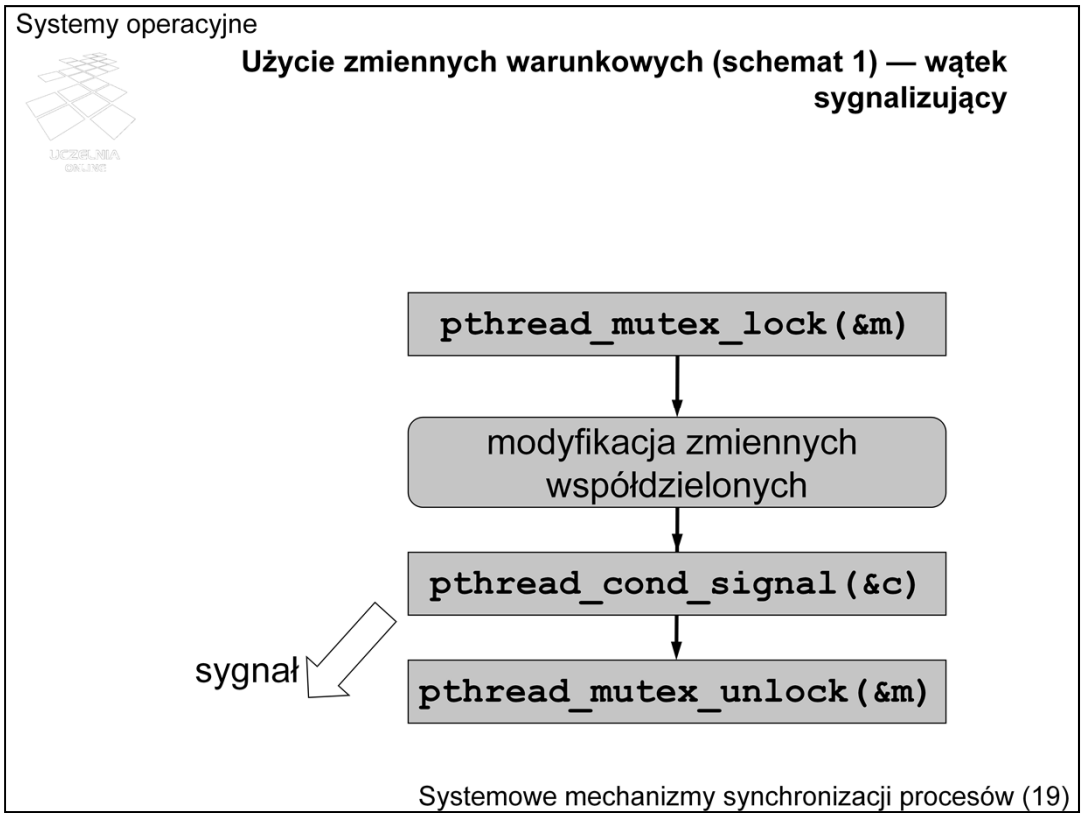


Można rozważać dwa typowe schematy synchronizacji z użyciem zmiennych warunkowych:

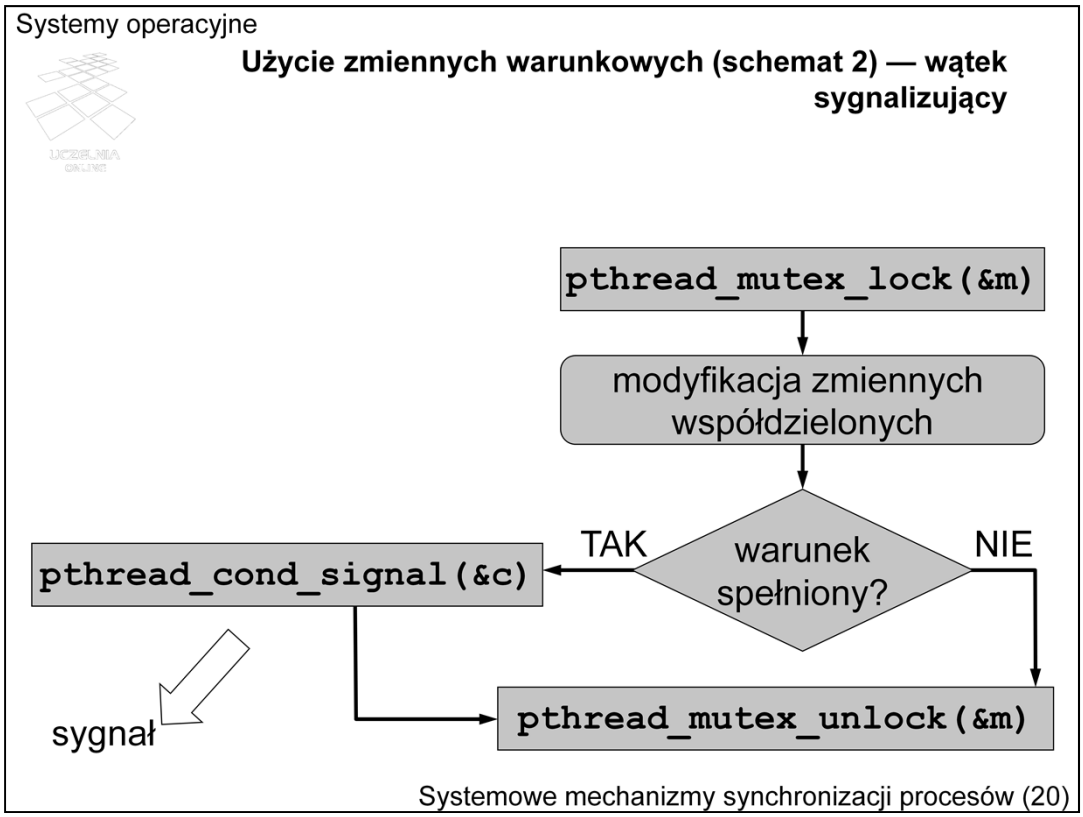
- schemat 1 — tylko wątek oczekujący wie, na jaki stan czeka, a wątek sygnalizujący budzi go w przypadku każdej zmiany tego stanu,
- schemat 2 — zarówno wątek oczekujący jak i sygnalizujący wiedzą, jaki stan jest oczekiwany, w związku z czym możliwa jest optymalizacja — wątek sygnalizujący budzi wątek oczekujący dopiero po osiągnięciu tego stanu.

W schemacie 1 wątek sprawdza warunek kontynuacji przetwarzania i jeśli jest niespełniony, wchodzi w stan oczekiwania. Jakakolwiek zmiana interesujących go aspektów stanu, będąca skutkiem aktywności innych wątków, powinna spowodować jego obudzenie. Obudzenie nie oznacza jednak, że oczekiwany stan musi wystąpić, w związku z czym wątek ponownie sprawdza warunek kontynuacji przetwarzania. Cała pętla wykonuje się w sekcji krytycznej, chronionej przez zamek, który zwalniany jest na czas oczekiwania. Z tego samego zamka korzysta wątek sygnalizujący, zatem zwolnienie jest konieczne w celu umożliwienia mu dostępu do fragmentów kodu, związanych z modyfikacją stanu przetwarzania.

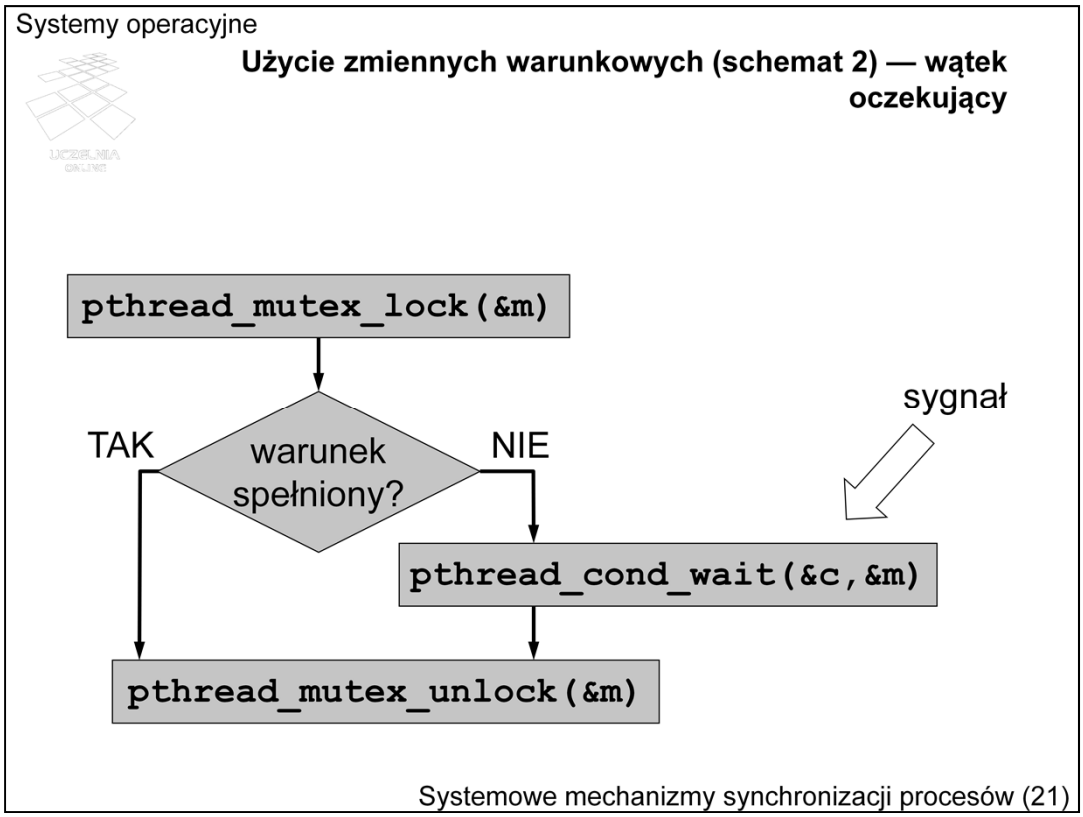
Brak sekcji krytycznej podczas sprawdzania warunku kontynuacji przetwarzania, nawet jeśli sprawdzanie nie wiąże się ze modyfikacją współdzielonych zmiennych, mógłby z kolei prowadzić do hazardu i utraty sygnału. Wykazanie niepoprawności takiego podejścia pozostawia się jako ćwiczenie.



Wątek sygnalizujący w schemacie 1 budzi wątek oczekujący każdorazowo, gdy dokona modyfikacji tych aspektów stanu (tych zmiennych współdzielonych), które interesują watek oczekujący. Obudzenie może okazać się niepotrzebne, gdyż pomimo dokonanych modyfikacji, nie jest to jeszcze taki stan, na jaki czeka wątek budzony. Tę wiedzę ma jednak tylko wątek oczekujący i to on musi zweryfikować stan, ewentualnie ponownie wejść w stan oczekiwania.



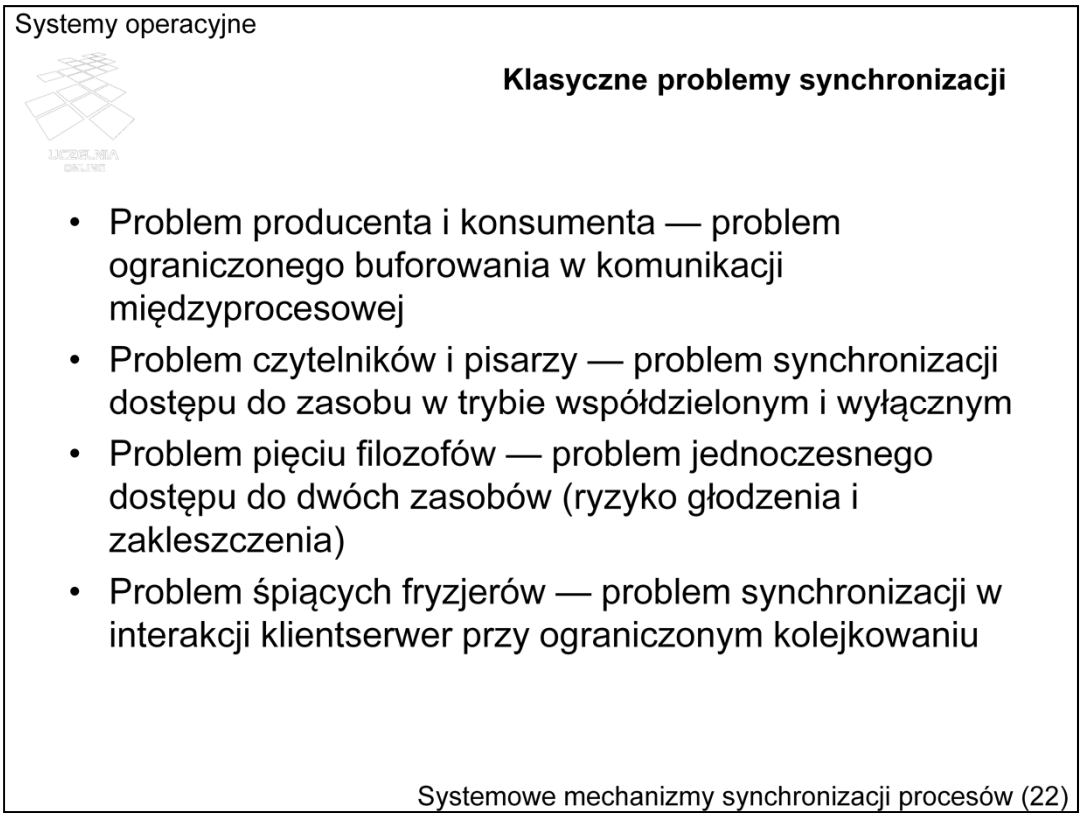
Omówienie schematu 2 rozpoczynamy od wątku sygnalizującego. Znając oczekiwania wątku współpracującego, uśpionego na zmiennej warunkowej, wątek sygnalizujący może mu wysłać sygnał wówczas, gdy sprawdzi, że oczekiwany stan zaistniał.



Wątek oczekujący ma pewność, że zostanie obudzony dopiero wówczas, gdy przetwarzanie osiągnie oczekiwany stan. Po opuszczeniu funkcji `pthread_cond_wait` nie musi on ponownie sprawdzać warunku. Musi natomiast wykonać `pthread_mutex_unlock`, gdyż wychodząc z `pthread_cond_wait` zajmuje ponownie zamek. Schemat ten jest bardziej ryzykowny w użyciu, gdyż ten sam warunek sprawdzany jest w dwóch lub większej liczbie miejsc. Jakiekolwiek korekty w programie, dotyczące warunku, wymagają zatem modyfikacji kodu w kilku miejscach i któreś z tych miejsc można przeoczyć. Poza tym przy większej liczbie współpracujących wątków kolejność zajmowania sekcji krytycznej może być nieokreślona i narazić na zmianę stanu przetwarzania pomiędzy obudzeniem wątku, a ponownym zajęciem przez niego sekcji krytycznej. Z tego powodu w ogólnym przypadku ten schemat nie jest zalecany. Może on być stosowany w prostych rozwiązaniach z dość precyzyjnie przewidywalnym przeplotem operacji współbieżnych wątków.

Wysłanie sygnału przez wątek sygnalizujący (wywołanie `pthread_cond_signal`) na ogół odbywa się w sekcji krytycznej i najczęściej jest ostatnią operacją wykonywaną w tej sekcji. Warto zwrócić uwagę, że dopiero opuszczenie sekcji krytycznej (wywołanie funkcji `pthread_mutex_unlock`) umożliwia przejście wątku budzonego w stan gotowości, czyli właściwe opuszczenie funkcji `pthread_cond_wait`. Wątek sygnalizujący nie musi jednak natychmiast wyjść z sekcji krytycznej. Potencjalnie nawet mógłby dokonać kolejnych modyfikacji współdzielonych zmiennych, w wyniku których oczekiwany warunek byłby ponownie niespełniony, co kwestionowałoby zasadność schematu 2. Szerszą dyskusję na ten temat można znaleźć w literaturze przy okazji omawiania monitorów.

Można również rozważać wysłanie sygnału po wyjściu z sekcji krytycznej. Nie jest to błąd, ale zmniejsza przewidywalność decyzji planisty przydziału procesora.



Problem producenta i konsumenta dotyczy przekazywania danych — w szczególności strumienia danych — pomiędzy procesami z wykorzystaniem bufora o ograniczonej pojemności. Bufor może być zapełniony, co zmusza producenta do ograniczenia swojej aktywności, lub może być pusty, co blokuje konsumenta.

Problem czytelników i pisarzy jest ilustracją synchronizacji dostępu do współdzielonych danych, które mogą być czytane lub modyfikowane. Modyfikacja danych wymaga wyłączności dostępu do danych, podczas gdy ich odczyt może być wykonywany współbieżnie. Jest to namiastka problemu, z jakim stykają się twórcy systemów zarządzania bazami danych, projektując mechanizmy współbieżnego wykonywania transakcji.

Problem pięciu filozofów wiąże się z dostępem procesu do wielu różnych zasobów (w tym przypadku dwóch) w tym samym czasie. Skutkiem braku odpowiedniej koordynacji może być zakleszczenie (ang. deadlock), zagłodzenie któregoś procesu (ang. starvation) lub uwięzienie (ang. livelock).

Problem śpiących fryzjerów jest odzwierciedleniem zagadnień interakcji klienta z serwerem (np. w komunikacji sieciowej), w której początkowo aktywną stroną jest klient żądający obsługi, a po zaakceptowaniu żądania występuje aktywność zarówno po stronie klienta jaki i serwera. Ze względu na ograniczoną pojemność kolejki oczekujących żądań, próba nawiązania interakcji ze strony klienta może być odrzucona.

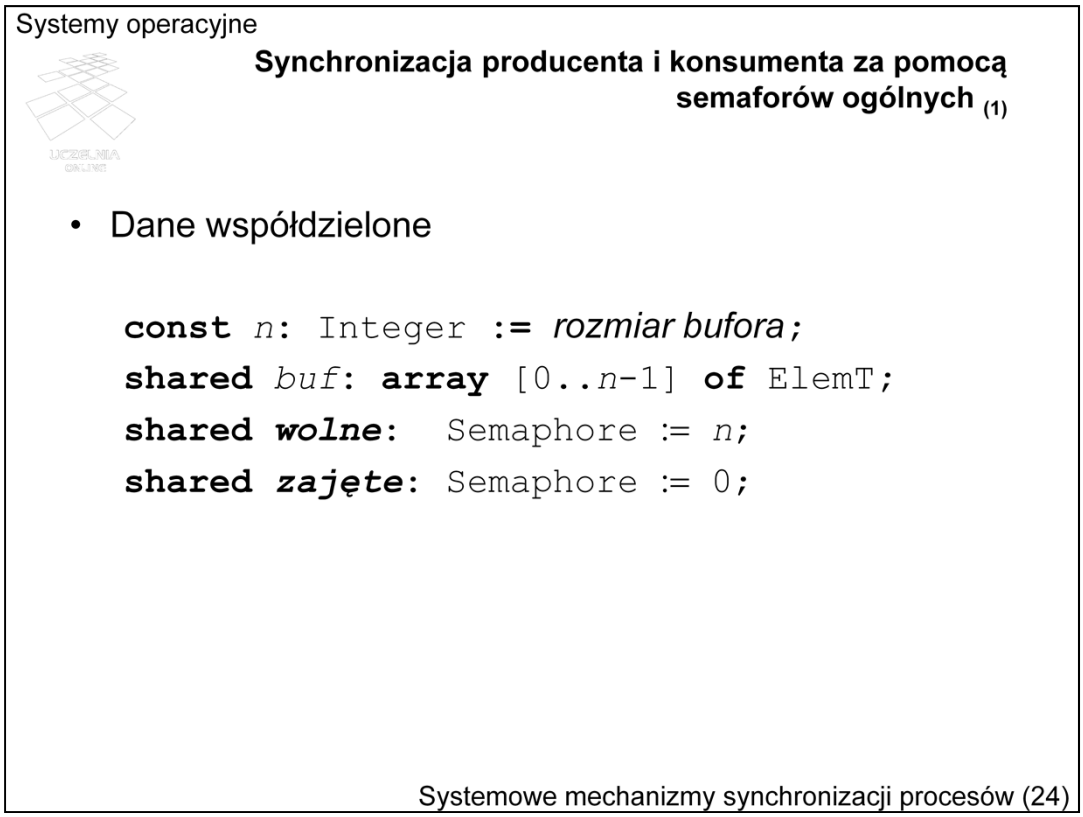
Systemy operacyjne

Problem producenta i konsumenta

- Producent produkuje jednostki określonego produktu i umieszcza je w buforze o ograniczonym rozmiarze.
- Konsument pobiera jednostki produktu z bufora i konsumuje je.
- Z punktu widzenia producenta problem synchronizacji polega na tym, że nie może on umieścić kolejnej jednostki, jeśli bufor jest pełny.
- Z punktu widzenia konsumenta problem synchronizacji polega na tym, że nie powinien on mięć dostępu do bufora, jeśli nie ma tam żadnego elementu do pobrania.

Systemowe mechanizmy synchronizacji procesów (23)

Bufor dla współpracy producenta i konsumenta ma pojemność ograniczoną do pewnej (ustalonej) liczby jednostek przekazywanego produktu. W literaturze spotkać można również nieco uproszczoną wersję tego problemu, w którym nie ma ograniczenia na rozmiar bufora. Nie ma wówczas potrzeby blokowania producenta, konieczne jest tylko blokowanie konsumenta, żeby „nie wyprzedził” producenta.



W przypadku jednego producenta i jednego konsumenta rozwiązanie wymaga dwóch semaforów ogólnych. Stan semafora *wolne* określa liczbę wolnych pozycji w buforze, a stan komplementarnego semafora *zajęte* określa liczbę pozycji zajętych. Przy założeniu, że początkowo bufor jest pusty, wartość semafora wolne wynosi *n* (*n* wolnych pozycji), a semafora zajęte 0 (żadna pozycja w buforze nie jest zajęta).

Systemy operacyjne

Synchronizacja producenta i konsumenta za pomocą semaforów ogólnych (2)

- Producent

```
local i: Integer := 0;
local elem: ElemT;
while ... do begin
   produkuj(elem);
   P(wolne);
   buf[i] := elem;
   i := (i+1) mod n;
   V(zajęte);
end;
```

Systemowe mechanizmy synchronizacji procesów (25)

Producent ma lokalną zmienną *i*, która wskazuje kolejną pozycję do zapełnienia w buforze. W przypadku wielu producentów zmienna ta musiałaby być przez nich współdzielona. Zmienna zwiększana jest cyklicznie (modulo *n*) po każdym wstawieniu elementu do bufora. Tak funkcjonujący bufor określą się jako *ograniczony bufor cykliczny*.

Wstawienie elementu do bufora poprzedzone jest operacją opuszczenia semafora *wolne*. Brak wolnego miejsca oznacza wartość 0 zmiennej semaforowej *wolne* i tym samym uniemożliwia opuszczenie. W ten sposób producent blokowany jest w dostępie do bufora, co chroni bufor przed przepełnieniem. Semafor *wolne* zostanie podniesiony przez konsumenta, gdy zwolni on miejsce w buforze.

Jeśli producentowi uda się umieścić kolejny element w buforze, sygnalizuje to przez podniesienie semafora *zajęte*. Ile razy podniesie go producent, tyle razy będzie mógł go opuścić konsument.

Systemy operacyjne

Synchronizacja producenta i konsumenta za pomocą semaforów ogólnych (3)

- Konsument

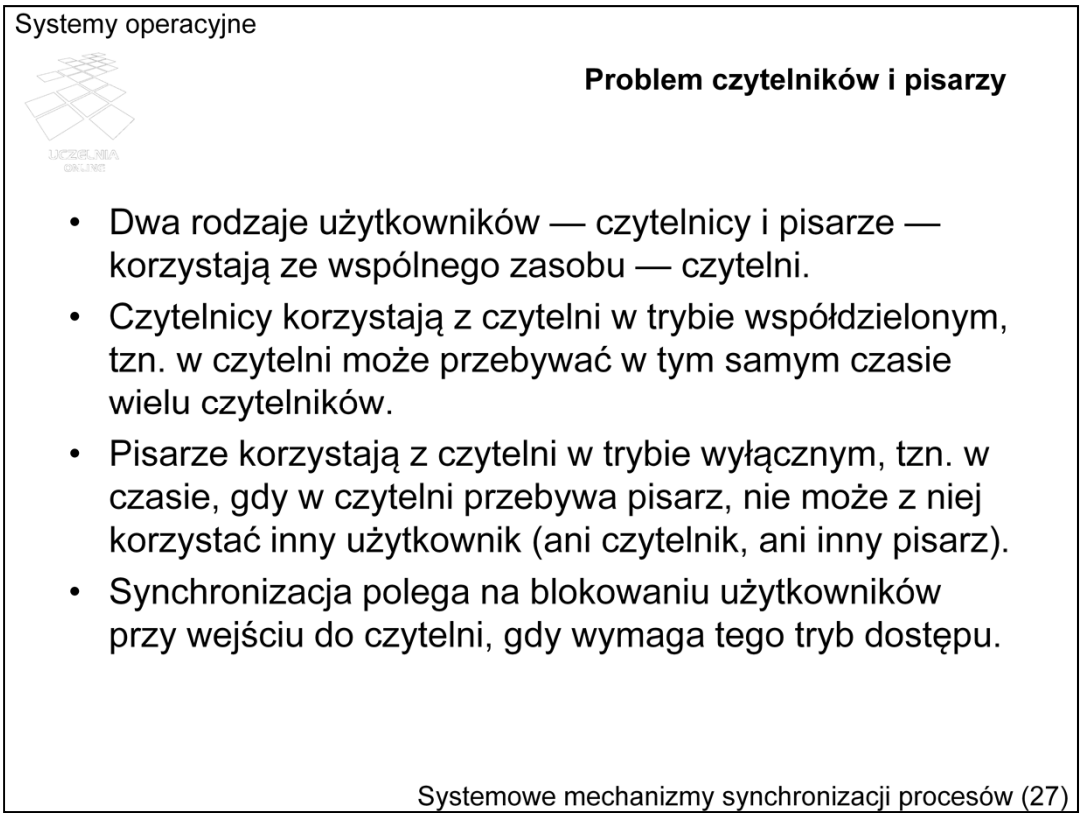
```
local i: Integer := 0;
local elem: ElemT;
while ... do begin
   P(zajęte);
   elem := buf[i];
   i := (i+1) mod n;
   V(wolne);
   konsumuj(elem);
end;
```

Systemowe mechanizmy synchronizacji procesów (26)

Konsument działa symetrycznie w stosunku do producenta. Podobnie jak producent, utrzymuje on lokalną zmienną *i*, która wskazuje mu pozycję z kolejnym elementem do pobrania. Zmienna ta musiałaby być współdzielona w przypadku wielu konsumentów.

Przed uzyskaniem dostępu do bufora konsument wykonuje operację opuszczenia semafora *zajęte*, który zwiększa producent po umieszczeniu w buforze kolejnego elementu. Jeśli semafor *zajęte* jest równy 0, bufor jest pusty i konsument nie ma tam czego szukać. Utknie on zatem w operacji opuszczania.

Jeśli konsument uzyska dostęp do bufora, pobierze element i tym samym zwolni miejsce. Fakt ten zasygnalizuje poprzez podniesieni semafora *wolne*, co z kolei umożliwi wykonanie kolejnego kroku producentowi.



W problemie można rozważać dwa rodzaje użytkowników (procesów) — czytelników i pisarzy — lub użytkowników wcielających się w rolę czytelnika lub pisarza. Rozróżnienie takie jest o tyle istotne, że w niektórych rozwiązaniach (zwłaszcza w środowisku rozproszonym) przyjęcie stałej liczby czytelników i pisarzy ma istotny wpływ na konstrukcję protokołu synchronizacji.

Można rozważać dwa warianty problemu:

- czytelnia ma nieograniczoną pojemność, co oznacza, że może z niej jednocześnie (współbieżnie) korzystać nieograniczona liczba czytelników,
- czytelnia ma tylko n miejsc, co oznacza, że liczba jednocześnie korzystających czytelników nie może być większa niż n.

Systemy operacyjne

Synchronizacja czytelników i pisarzy za pomocą semaforów binarnych (1)

- Dane współdzielone

```
shared l_czyt: Integer := 0;
shared mutex_r: Binary_Semaphore := true;
shared mutex_w: Binary_Semaphore := true;
```

Systemowe mechanizmy synchronizacji procesów (28)

Zaprezentowane rozwiązanie faworyzuje czytelników, a dopuszcza głodzenie pisarzy. Synchronizacja opiera się na dwóch binarnych semaforach i współdzielonej zmiennej *l_czyt* (liczba czytelników w czytelni). Semafor *mutex_r* służy do synchronizacji dostępu do zmiennej *l_czyt*. Semafor *mutex_w* służy z kolei do blokowania pisarzy.

Systemy operacyjne

Synchronizacja czytelników i pisarzy za pomocą semaforów binarnych (2)

- Czytelnik

```
while ... do begin
   P(mutex_r);
   l_czyt := l_czyt + 1;
   if l_czyt = 1 then P(mutex_w);
   V(mutex_r);
   czytanie;
   P(mutex_r);
   l_czyt := l_czyt - 1;
   if l_czyt = 0 then V(mutex_w);
   V(mutex_r);
end;
```

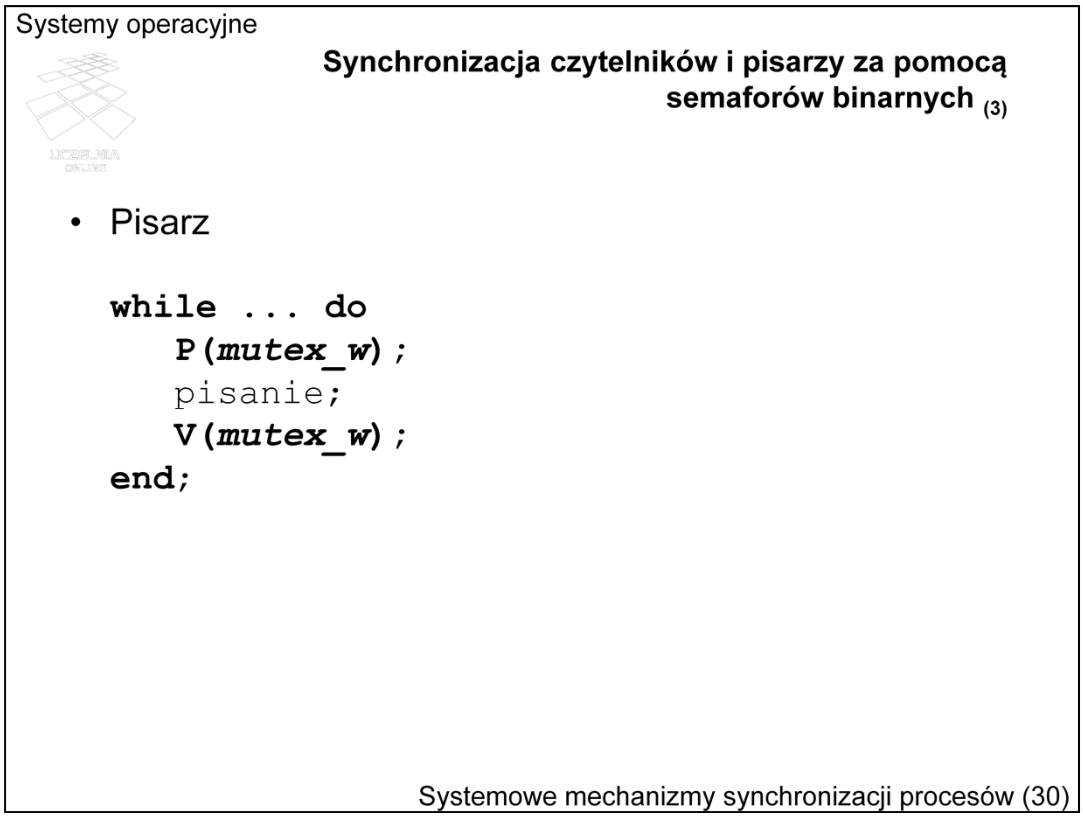
Systemowe mechanizmy synchronizacji procesów (29)

W prezentowanym rozwiązaniu czytelnik może wejść do czytelni zawsze, gdy nie ma w niej pisarza. Czytelnik zamyka zatem semafor *mutex_r*, chroniący zmienną *l_czyt*, po czym zwiększa ją o 1. Jeśli po zwiększeniu o 1 zmienna równa jest 1, to znaczy, że jest on pierwszym czytelnikiem, który zajął czytelnię. Musi on zatem zamknąć wejście pisarzom poprzez opuszczenie semafora *mutex_w*. Mogłoby się okazać jednak, że w czytelni nie było wprawdzie czytelników, ale był właśnie pisarz. Operacja opuszczenia semafora *mutex_w* powstrzyma czytelnika przed dostępem do czytelni do czasu wyjścia z niej pisarza (patrz następny slajd).

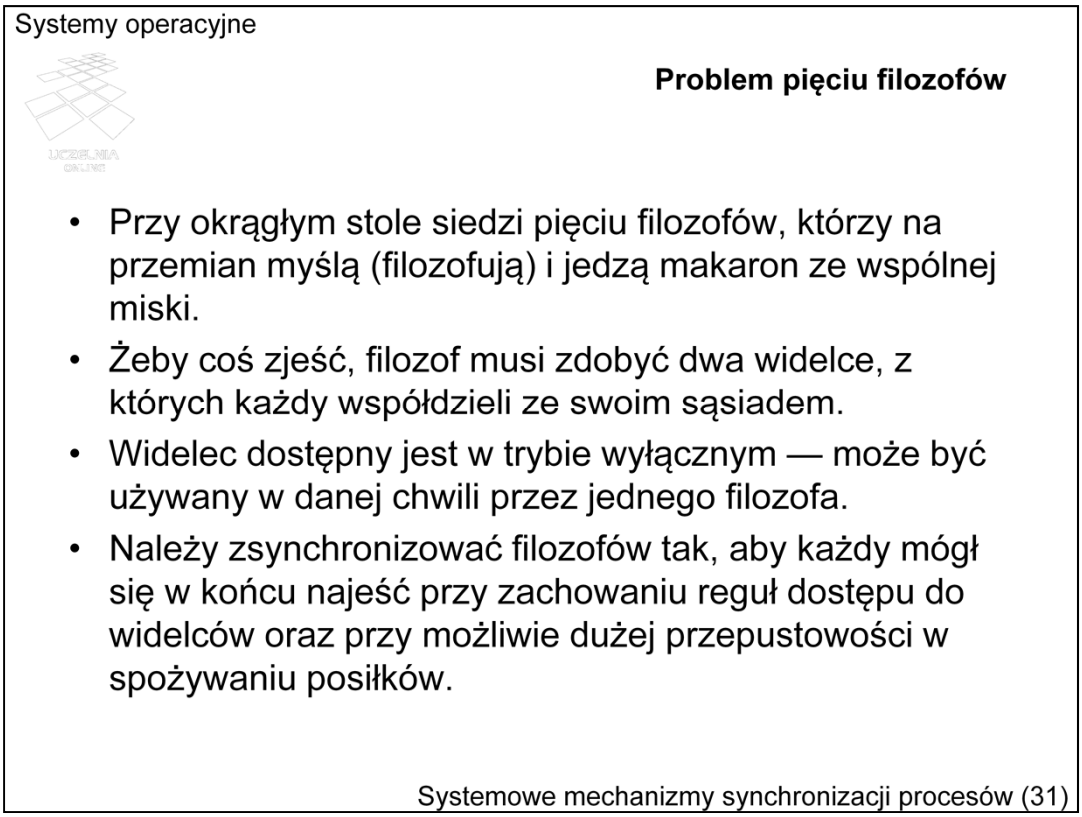
Jeśli do czytelni wchodzą kolejni czytelnicy, nie wykonują już operacji na semaforze *mutex_w*. Zamknął go już pierwszy czytelnik.

Przy wyjściu czytelnika z czytelni zmniejszana jest wartość zmiennej *l_czyt*. Operacja chroniona jest oczywiście przez semafor *mutex_r*, gwarantujący realizację w sekcji krytycznej. Gdyby po zmniejszeniu zmiennej *l_czyt* okazało się, że jest ona równa 0, to znaczy, że z czytelni wyszedł ostatni czytelnik i można dać szansę pisarzowi. Podnoszony jest zatem semafor *mutex_w*.

Może się jednak tak zdarzyć, że w czytelni zawsze będzie jakiś czytelnik. Zanim wyjdzie jeden czytelnik, następny może już wejść do czytelni. Można w ten sposób doprowadzić do głodzenia pisarzy.



Jednym zadaniem pisarza jest zakończyć sukcesem operację opuszczania semafora *mutex_w*. Jeśli pisarzowi się to uda, na czas pisania semafor będzie opuszczony. Kolejny pisarz utknie oczywiście na semaforze *mutex_w*. Pierwszy z czytelników, próbujących wejść do czytelni, również utknie na semaforze *mutex_w*. W wyniku zablokowania w opuszczaniu semafora *mutex*_w, do czasu odblokowania nie zostanie podniesiony semafor *mutex_r*, co spowoduje zablokowanie na nim pozostałych czytelników.



Synchronizacja w problemie pięciu filozofów odbywa się na dwóch poziomach:

- lokalnym — synchronizacja dostępu do konkretnego widelca,
- globalnym — koordynacja działań wszystkich pięciu filozofów w taki sposób, aby nie dopuścić do zakleszczenie, uwięzienia lub zagłodzenia któregoś z filozofów.

Potrzeba synchronizacji na poziomie lokalnym wynika z faktu, że każdy widelec jest współwłasnością dwóch filozofów, a używany w danej chwili może być tylko przez co najwyżej jednego z nich. Globalna koordynacja z kolei jest wymagana, gdyż każdy filozof w celu przejścia do stanu „jedzenie” potrzebuje dwóch widelców. Uzyskanie tylko jednego z nich może oznaczać przetrzymywanie zasobu bez wyraźnej perspektywy zwolnienia go po osiągnięciu stanu „sytości”, do czego potrzeba drugiego.

Systemy operacyjne

Synchronizacja 5 filozofów za pomocą semaforów binarnych (1)

```
shared dopuść: Semaphore := 4;
shared widelec: array[0..4] of
                Binary_Semaphore := true;
```

Systemowe mechanizmy synchronizacji procesów (32)

Problem dostępu do widelców w synchronizacji pięciu filozofów można rozwiązać za pomocą tablicy semaforów binarnych — *widelec*. Semafor na każdej pozycji tablicy reprezentuje jeden widelec. Zmiana stanu z myślenia na jedzenie wymaga jednak dwóch widelców. Istnieje zatem ryzyko zakleszczenia lub uwięzienia, co jest istotą problemu pięciu filozofów. Problem zakleszczenia omawiany będzie w następnym module, ale uprzedzając całościowe omówienie zagadnień zakleszczenia, można powiedzieć, że najprostszym podejściem jest niedopuszczeni do powstania cyklu w oczekiwaniu na widelce. Można w tym celu stosować różne techniki:

- dopuścić do rywalizacji o widelce w danej chwili co najwyżej 4 procesy,
- stosować różną kolejność uzyskiwania widelców — np. filozof o numerze parzystym bierze najpierw lewy, a później prawy widelec, natomiast filozof o numerze nieparzystym bierze najpierw lewy widelec,
- stosować podejście priorytetowe, oparte np. na liczbie już zjedzonych posiłków — im więcej posiłków tym mniejszy priorytet.

W przedstawionym w dalszej części rozwiązaniu przyjęto pierwsze z wymienionych podejść, w związku z czym potrzebny jest semafor ogólny dopuść o wartości początkowej 4.

Systemy operacyjne

Synchronizacja 5 filozofów za pomocą semaforów binarnych (2)

- Filozof nr *i* (*i* = 0 ... 4)

```
while ... do begin
    myślenie;
    P(dopuść);
    P(widelec[i]);
    P(widelec[(i+1) mod 5]);
    jedzenie;
    V(widelec[i]);
    V(widelec[(i+1) mod 5]);
    V(dopuść);
end;
```

Systemowe mechanizmy synchronizacji procesów (33)

Przed podjęciem próby zdobycia widelców filozof opuszcza semafor *dopuść*. Jeśli byłby piątym procesem, dopuszczonym do rywalizacji, jest ryzyko powstania cyklu i zakleszczenia. Semafor *dopuść* dopuszcza jednak tylko czterech filozofów, w związku z czym nie ma ryzyka cyklu.

Systemy operacyjne

Problem śpiących fryzjerów

- W salonie fryzjerskim jest poczekalnia z *p* miejscami oraz *n* foteli, obsługiwanych przez fryzjerów.
- Do salonu przychodzi klient, budzi fryzjera, po czym fryzjer znajduje wolny fotel i obsługuje klienta.
- Jeśli nie ma wolnego fotela, klient zajmuje jedno z wolnych miejsc w poczekalni.
- Jeśli nie ma miejsca w poczekalni, klient odchodzi.
- Problem polega na zsynchronizowaniu fryzjerów oraz klientów w taki sposób, aby jeden fryzjer w danej chwili obsługiwał jednego klienta i w tym samym czasie klient był obsługiwany przez jednego fryzjera.

Systemowe mechanizmy synchronizacji procesów (34)

Fotel w salonie fryzjerskim jest stanowiskiem, obsługiwanym przez fryzjera. Zakłada się, że fryzjerów jest nie mniej niż foteli. W przypadku braku klienta fryzjer śpi. Jeśli jednak kolejka klientów nie jest pusta, fryzjerzy obsługują kolejnych klientów w miarę dostępnych foteli. Sama obsługa oznacza jednoczesną aktywność zarówno klienta, jak i fryzjera.

Problem można rozważać w kilku wariantach:

- grupa „jednorodnych” fryzjerów,
- kilka grup fryzjerów specjalistów — klient (klientka) zgłasza się do specjalisty od danego rodzaju usługi (np. modelowanie, farbowanie, pasemka itp.),
- fryzjerzy indywidualni — klient zgłasza się do konkretnego fryzjera.

Systemy operacyjne

Synchronizacja śpiących fryzjerów za pomocą semaforów (1)

- Dane współdzielone

```
const  p: Integer := pojemność poczekalni;
const  n: Integer := liczba foteli;
shared l_czek:  Integer := 0;
shared mutex:   Binary_Semaphore := true;
shared klient:  Semaphore := 0;
shared fryzjer: Semaphore := 0;
shared fotel:   Semaphore := n;
```

Systemowe mechanizmy synchronizacji procesów (35)

Rozwiązanie opiera się na jednym semaforze binarnym i trzech ogólnych. Poza tym, w zmiennej *l_czek* pamiętana jest liczba klientów oczekujących na obsługę i zajmujących miejsca w poczekalni. Semafor binarny *mutex* służy głównie do synchronizacji dostępu do zmiennej *l_czek*.

Systemy operacyjne

Synchronizacja śpiących fryzjerów za pomocą semaforów (2)

- Klient

```
while ... do begin
   P(mutex);
   if l_czek < p then begin
      l_czek := l_czek + 1;
      V(klient);
      V(mutex);
      P(fryzjer);
      strzyżenie;
   end
   else V(mutex);
end;
```

Systemowe mechanizmy synchronizacji procesów (36)

Klient zamyka semafor *mutex* w celu ochrony operacji na zmiennej *l_czek*, po czym sprawdza, czy jest wolne miejsce w poczekalni. Jeśli nie ma miejsca, otwiera semafor *mutex* i kończy niepowodzeniem próbę skorzystania z usługi. Jeśli natomiast jest wolne miejsce w poczekalni, to będąc cały czas w sekcji krytycznej, chronionej przez *mutex,* zmniejsza o 1 zmienną *l_czek* (zajmuje miejsce w poczekalni), podnosi semafor *klient*, dając w ten sposób sygnał fryzjerowi i opuszcza sekcję krytyczną. Po opuszczeniu sekcji krytycznej czeka na wolnego fryzjera na semaforze *fryzjer* — czeka na sygnał na tym semaforze, czyli na jego podniesienie. Kiedy pojawi się wolny fryzjer, podniesie on ten semafor i klient przejdzie do fazy strzyżenia.

Systemy operacyjne

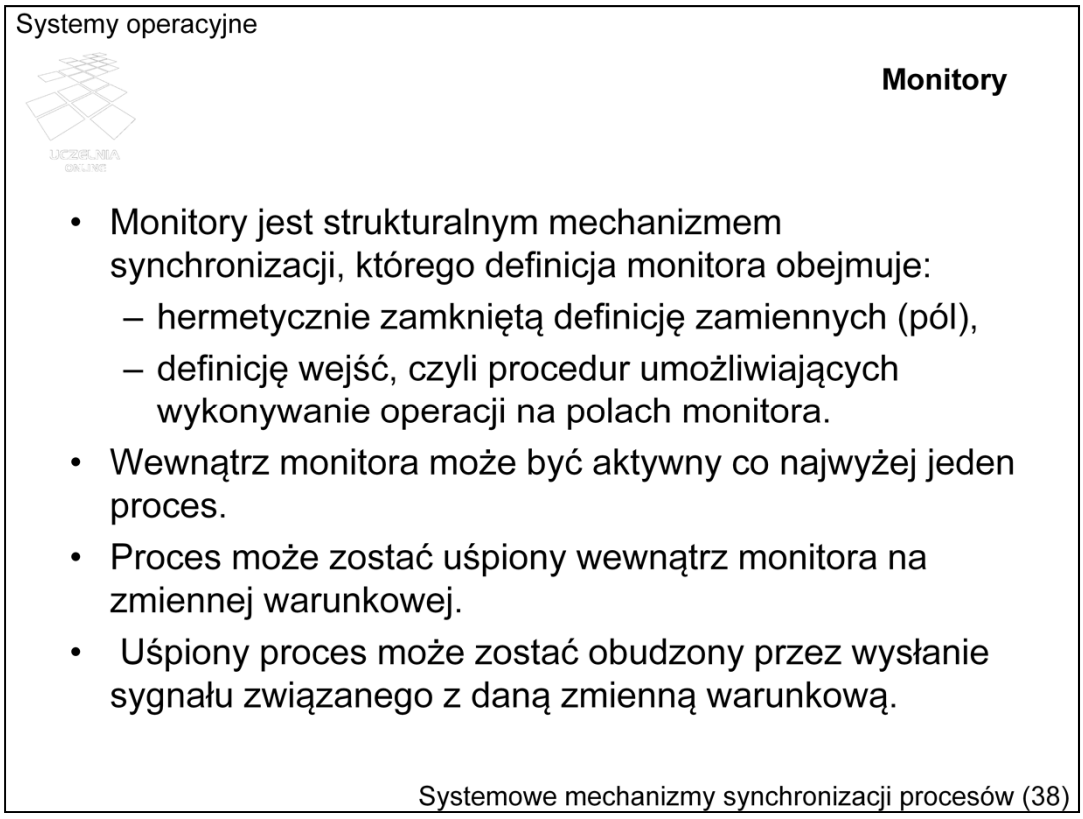
Synchronizacja śpiących fryzjerów za pomocą semaforów (3)

- Fryzjer

```
while ... do begin
   P(klient);
   P(fotel);
   P(mutex);
   l_czek := l_czek - 1;
   V(fryzjer);
   V(mutex);
   strzyżenie;
   V(fotel);
end;
```

Systemowe mechanizmy synchronizacji procesów (37)

Fryzjer czeka na klienta na semaforze *klient*, który jest podnoszony przez klienta po uzyskaniu miejsca w poczekalni. Po zakończeniu tej operacji fryzjer wie, że ma klienta więc czeka na wolny fotel opuszczając semafor *fotel*. Jeśli operacja się zakończy to jest fotel dla klienta i można przejść do obsługi. Najpierw zmniejszana jest zmienna *l_czek*, bo zwalnia się miejsce w poczekalni. Następnie poprzez podniesienie semafora *fryzjer*, przekazywany jest klientowi sygnał, że może wyjść ze stanu czekania i przejść do właściwej obsługi. Po zwolnieniu semafora *mutex* fryzjer też przechodzi do obsługi, po zakończeniu której zwalnia fotel (podnosi semafor *fotel*).



Zasada funkcjonowania monitora podobna jest do mechanizmów POSIX. Właściwie raczej mechanizmy POSIX wzorowane są na monitorach, gdyż koncepcja monitora została zaproponowana przez Hoare'a wcześniej, niż powstał standard POSIX.

Synchronizacja związana jest w tym przypadku z definicją określonej struktury, która (podobnie jak definicja klasy w podejściach obiektowych) obejmuje deklaracje zmiennych instancji oraz implementacje metod dostępu — wejść. Zmienne dostępne są tylko wewnątrz monitora, tzn. operacje na nich mogą być wykonywane tylko w ramach wejść. Wejścia z kolei stanowią publiczny interfejs dostępu do monitora. W ich implementacji można się odwoływać tylko do zmiennych monitora oraz parametrów.

Wejście wykonywane jest przy wyłącznym dostępie do monitora — wyklucza zatem aktywność jakiegokolwiek innego procesu wewnątrz monitora, czyli wyklucza możliwość współbieżnego wykonywana kodu monitora przez kilka procesów. Możliwe jest jednak udostępnienie monitora w przypadku, gdy proces wejdzie w stan oczekiwania na zmiennej warunkowej.

Częścią definicji monitora są zmienne warunkowe, na których proces może jawnie usnąć, a następnie zostać obudzony. Podobnie jak w przypadku mechanizmów POSIX, na czas uśpienia monitor jest zwalniany. W ten sposób uśpiony proces może kiedyś zostać obudzony przez inny proces, który wyśle sygnał na danej zmiennej. Warto jednak podkreślić, że zmienne warunkowe, jako wewnętrzne zmienne monitora, dostępne są tylko w ramach wejść. Zarówno usypianie jak i budzenie procesu jest częścią implementacji jakiegoś wejścia.

Systemy operacyjne

Ogólny schemat definicji monitora

```
type nazwa_monitora = monitor
  deklaracje zmiennych
  procedure entry proc_1(...);
     begin ... end;
     ⋮
  procedure entry proc_n(...);
     begin ... end;
  begin
     kod inicjalizujący
  end.
```

Systemowe mechanizmy synchronizacji procesów (39)

Każde wejście w definicji monitora może mieć parametry tak, jak normalna procedura. Wśród deklaracji zmiennych mogą się pojawić szczególne zmienne — warunkowe, na których w implementacji poszczególnych wejść można wykonywać operacja **`wait`** i **`signal`**. Podobnie jak w mechanizmach POSIX, **`wait`** powoduje uśpienie procesu w oczekiwaniu na sygnał i udostępnienie monitora innym procesom, a **`signal`** budzi jeden z procesów, śpiących na zmiennej warunkowej lub jest ignorowany, jeśli nie ma procesu do obudzenia.

Różnica w stosunku do mechanizmów standardu POSIX jest zatem taka, że zajęcie i zwolnienie zamaka wykonywane jest niejawnie. Sam zamek w związku z tym również nie jest jawnie deklarowany. W ten sposób można się ustrzec błędów związanych z pominięciem tego typu operacji synchronizującej. Należy jedynie zadbać o poprawność definicji struktury danych, a w zakresie tym można liczyć na wsparcie ze strony narzędzi programistycznych, np. kompilatora.

Systemy operacyjne

Ograniczony bufor cykliczny — definicja oparta na monitorze (1)

```
type Buffer = monitor
   pula : array [0..n-1] of ElemT;
   wej, wyj, licz : Integer;
   pusty, pełny : Condition;
```

Systemowe mechanizmy synchronizacji procesów (40)

Podstawą rozwiązania problemu czytelników i pisarzy za pomocą monitora jest odpowiednio zdefiniowany bufor cykliczny. Bufor ten w prezentowanym rozwiązaniu chroniony jest przez monitor, tzn. właściwy bufor — określony jako *pula* miejsc — stanowi część monitora, w związku z czym dostępny jest bezpośrednio tylko w ramach wejść. Poza tym w monitorze zadeklarowane są zmienne na potrzeby obsługi dostępu do puli:

- *wej* — indeks pozycji w puli, na której umieszczony zostanie kolejny wstawiany element,
- *wyj* — indeks pozycji w puli, z której pobierany będzie kolejny element,
- *licz* — liczba elementów w buforze.

Synchronizacja dostępu wymaga jeszcze zmiennych warunkowych, na których byłyby usypiane procesy w przypadku zapełniania lub całkowitego opróżnienia bufora. W rozwiązaniu wykorzystano dwie zmienne warunkowe:

- *pusty* — do usypiania konsumentów, gdy bufor jest pusty,
- *pełny* — do usypiania producentów, gdy bufor jest w całości wypełniony.

Systemy operacyjne

Ograniczony bufor cykliczny — definicja oparta na monitorze (2)

```
procedure entry wstaw(elem: ElemT);
   begin
      if licz = n then pełny.wait;
      pula[wej] := elem;
      wej := (wej + 1) mod n;
      licz := licz + 1;
      pusty.signal;
   end;
```

Systemowe mechanizmy synchronizacji procesów (41)

Wejście *wstaw* wywoływane jest przez producenta. Przed umieszczeniem elementu w puli danego bufora należy sprawdzić, czy jest wolne miejsce, czyli czy bieżące wypełnienie bufora — *licz* — nie jest takie, jak jego pojemność — *n*. Jeśli wartości są równe, bufor jest całkowicie zapełniony i proces producenta usypiany jest na zmiennej warunkowej *pełny*. Obudzony zostanie dopiero przez konsumenta po pobraniu elementu.

Jeśli w buforze jest wolne miejsce, na pozycji wskazanej przez zmienną *wej* umieszczany jest element, wstawiany przez producenta, po czym zmienna *wej* jest zwiększana cyklicznie o 1. Zwiększana jest też o 1 zmienna *licz*, gdyż przybył jeden element.

Na końcu wysyłany jest sygnał dla (być może) oczekującego konsumenta, że pojawiło się coś w buforze, co można skonsumować. Jeśli nikt nie czekan na zmiennej warunkowej *pusty*, sygnał zostanie zignorowany.

Systemy operacyjne

Ograniczony bufor cykliczny — definicja oparta na monitorze (3)

```
procedure entry pobierz(var elem: ElemT);
   begin
      if licz = 0 then pusty.wait;
      elem := pula[wyj];
      wyj := (wyj + 1) mod n;
      licz := licz - 1;
      pełny.signal;
   end;
```

Systemowe mechanizmy synchronizacji procesów (42)

Implementacja wejścia *pobierz* jest analogiczna. Sprawdzane jest, czy bufor nie jest pusty, czyli czy liczba elementów — *licz* — nie jest równa 0. Jeśli *licz* jest równe 0, następuje uśpienie konsumenta.

Jeśli bufor nie jest pusty, konsument pobiera z pozycji, indeksowanej przez *wyj* element, który staje się wartością parametru wyjściowego. Zmienna *wyj* zwiększana jest cyklicznie, żeby wskazywała kolejny element, po czym następuje zmniejszenie zmiennej *licz*, gdyż w buforze zwolniła się miejsce po pobranym elemencie.

Na końcu wysyłany jest sygnał do producentów, którzy mogą czekać na zwolnienie miejsca w buforze.

Systemy operacyjne

Ograniczony bufor cykliczny — definicja oparta na monitorze (4)

```
begin
    wej := 0;
    wyj := 0;
    licz := 0;
end.
```

Systemowe mechanizmy synchronizacji procesów (43)

Przyjmując, że początkowy stanu bufor jest pusty, w ramach inicjalizacji następuje wyzerowanie wszystkich zmiennych, związanych z kontrolą dostępu do puli.

Systemy operacyjne

Synchronizacja producenta i konsumenta za pomocą monitora

- Dane współdzielone

```
shared buf: Buffer;
```

- Producent

```
local
   elem: ElemType;
while ... do begin
   produkuj(elem);
   buf.wstaw(elem);
end;
```

- Konsument

```
local
   elem: ElemType;
while ... do begin
   buf.pobierz(elem);
   konsumuj(elem);
end;
```

Systemowe mechanizmy synchronizacji procesów (44)

Użycie monitora odpowiedzialnego za buforowanie w samym kodzie producenta lub konsumenta jest już bardzo proste. Wystarczy po wyprodukowaniu elementu wywołać odpowiednie wejście — *wstaw*, a chcąc skonsumować kolejny element, wywołać *pobierz*. Ponieważ bufor jest chroniony przez monitor, cała odpowiedzialność za synchronizację spoczywa na twórcy tego monitora, minimalizując ryzyko błędu w programie dla producenta i konsumenta.

Systemy operacyjne

Synchronizacja czytelników i pisarzy za pomocą monitora

- Dane współdzielone
```
shared czytelnia: Monitor;
```
- Czytelnik
```
czytelnia.wejście_czytelnika;
czytanie;
czytelnia.wyjście_czytelnika;
```
- Pisarz
```
czytelnia.wejście_pisarza;
pisanie;
czytelnia.wyjście_pisarza;
```

Systemowe mechanizmy synchronizacji procesów (45)

W przedstawionym schemacie synchronizacji monitor *czytelnia* **nie chroni** czytelni. Monitor w tym przypadku pełni rolę koordynatora dostępu do czytelni. Jeśli w programie któregoś procesu przed wejściem do czytelni nie znalazłyby się operacje na monitorze, będące w rzeczywistości pytaniami o pozwolenie na wejście, to czytelnia i tak stałaby się dostępna, tylko w sposób nie skoordynowany, zatem z ryzykiem naruszeniem poprawności.

Gdyby czytelnia była **chroniona** wewnątrz monitora i dostępna byłaby poprzez wejścia typu: *czytaj*, *pisz*, to nadmiernie ograniczona byłaby współbieżność dostępu dla czytelników, gdyż wykluczaliby się oni wzajemnie.

W rozwiązaniu pominięto implementację monitora *czytelnia*. Implementację tę pozostawia się jako ćwiczenie.

Systemy operacyjne

Regiony krytyczne

- Region krytyczny jest fragmentem programu — oznaczonym jako *S*, wykonywanym przy wyłącznym dostępie do pewnej zmiennej współdzielonej, wskazanej w jego definicji — oznaczonej jako *v*.
- Wykonanie regionu krytycznego uzależnione jest od wyrażenia logicznego — *B*, a przetwarzanie blokowane jest do momentu, aż wyrażenie będzie prawdziwe.

```
shared v: T;
region v when B do S;
```

Systemowe mechanizmy synchronizacji procesów (46)

Przedstawiona definicja dotyczy właściwie warunkowego regionu krytycznego. Blokowanie i wznawianie procesów w przypadku regionów krytycznych odbywa się na podstawie warunku logicznego, w przeciwieństwie do monitora, gdzie jawnie wysyłany jest sygnał. Jeśli więc warunek nie jest spełniony, proces jest usypiany tak samo, jak przez jawne wywołanie **`wait`** w przypadku monitora. Również w przypadku, gdy warunek stanie się prawdziwy, proces zostanie obudzony. W przypadku monitora wymagane jest jawne wykonanie **`signal`**. Oczywiście w czasie oczekiwania na spełnienie warunku, podanego po frazie **`when`**, region krytyczny jest dostępny dla innych procesów.

Systemy operacyjne

Synchronizacja producenta i konsumenta za pomocą regionu krytycznego (1)

- Dane współdzielone

```
shared buf: record
     pula : array [0..n-1] of ElemT;
     wej, wyj, licz : Integer;
end;
```

Systemowe mechanizmy synchronizacji procesów (47)

W przedstawionym rozwiązaniu *buf* jest zmienną typu rekord, obejmującą pulę pozycji i zmienne do jej obsługi (podobnie, jak w przypadku monitora). Na zmiennej *buf* będzie wykonywany region krytyczny.

Systemy operacyjne

Synchronizacja producenta i konsumenta za pomocą regionu krytycznego (2)

- Producent

```
local elem: ElemT;
while ... do
   begin
      produkuj(elem);
      region buf when buf.licz < n do
         begin
            buf.pula[wej]:= elem;
            buf.wej := (buf.wej+1) mod n;
            buf.licz := buf.licz + 1;
         end;
   end;
```

Systemowe mechanizmy synchronizacji procesów (48)

Zasada działania tego rozwiązania jest zbliżona do monitora. Jak już wspomniano przy ogólnej definicji regionu krytycznego, różnicą jest tylko sposób opisu fragmentu kodu, wykonywanego w trybie wyłącznym oraz sposób blokowania producentów i konsumentów w dostępie do bufora.

Wzajemne wykluczanie dotyczy fragmentu kodu objętego konstrukcją **`region`**, zatem trzech operacji podstawienia, modyfikujących składowe rekordu *buf*. Takie same operacje w przypadku monitora znajdowały się w implementacji wejścia *wstaw*.

Usypianie i budzenie odbywa się niejawnie w zależności od wypełnienia bufora. Taki sam warunek sprawdzany był w przypadku monitora, ale usypianie a następnie budzenie wymagało jawnego użycia w kodzie instrukcji **`wait`** i **`signal`**.

Systemy operacyjne

Synchronizacja producenta i konsumenta za pomocą regionu krytycznego (3)

- Konsument

```
local elem: ElemT;
while ... do
   begin
      region buf when buf.licz > 0 do
         begin
            elem := buf.pula[wyj];
            buf.wyj := (buf.wyj+1) mod n;
            buf.licz := buf.licz - 1;
         end;
      konsumuj(elem);
   end;
```

Systemowe mechanizmy synchronizacji procesów (49)

Działanie konsumenta jest analogiczne. Warto zwrócić uwagę, że konsument wykonuje inny region krytyczny (z innym warunkiem i innym kodem wykonywanym w trybie wyłączny) na tej samej zmiennej — *buf*. Wykonywanie kodu konsumenta wyklucza zatem wykonywanie kodu producenta i odwrotnie.