Systemy operacyjne

Współbieżność i synchronizacja procesów

Wykład prowadzą:
Jerzy Brzeziński
Dariusz Wawrzyniak

UCZELNIA
ONLINE

Celem wykładu jest przedstawienie zagadnień współbieżnego wykonywania wątków lub procesów, zwłaszcza w przypadku, gdy wchodzą one we wzajemne interakcje. Omówiony zostanie też jeden z podstawowych problemów synchronizacji — problem wzajemnego wykluczania oraz jego rozwiązania oparte na środkach dostępnych na poziomie architektury komputera.

Na potrzeby tego modułu, w celu uwypuklenia pewnych aspektów wprowadzonych wcześniej pojęć, pojęcia te zostaną ponownie zdefiniowane lub sformalizowane.

Systemy operacyjne

Plan wykładu

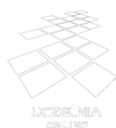
- Abstrakcja programowania współbieżnego
- Instrukcje atomowe i ich przeplot
- Istota synchronizacji
- Kryteria poprawności programów współbieżnych
- Klasyfikacja mechanizmów synchronizacji
- Wzajemne wykluczania
- Algorytmy wzajemnego wykluczania (alg. Petersona i alg. Lamporta)
- Złożone instrukcje atomowe (**`test&set`**, **`exchange`**)

Współbieżność i synchronizacja procesów (2)

Wykład obejmuje omówienie modelu systemu współbieżnego, opartego na dwóch zasadniczych pojęciach: *instrukcji atomowej* i *przeplocie*. Pokazane zostaną przykładowe przeploty operacji dwóch współbieżnych procesów, z których nie wszystkie dają wyniki, zgodne z oczekiwaniami programisty. Na bazie tych pojęć zostanie wyjaśnione, jak rozumiana jest poprawność programu współbieżnego oraz na czym polega synchronizacja. Następnie dokonana zostanie klasyfikacja mechanizmów synchronizacji procesów, z których część zostanie omówiona w niniejszym module, a pozostała część w następnym.

Dalsza część wykładu dotyczy podstawowego problemu synchronizacji procesów — wzajemnego wykluczania i poprawności jego rozwiązania. Przedstawione zostaną algorytmy oparte wyłącznie na zapisie i odczycie współdzielonych zmiennych — algorytm Petersona oraz Lamporta (tzw. algorytm piekarni). Następnie analizowane będą rozwiązania wykorzystujące złożone instrukcje atomowe: **`test&set`** oraz **`exchange`**.

Systemy operacyjne



Wprowadzenie do abstrakcji przetwarzania współbieżnego

- Realizacja przetwarzania polega na wykonywaniu instrukcji przez jednostkę przetwarzającą (procesor).
- Instrukcje wykonywane są w kontekście jakiegoś procesu.
- Wykonanie instrukcji (akcja) oznacza zajście zdarzenia w procesie, skutkiem czego jest zmiana stanu procesu.
- Zajście zdarzenia jest zdeterminowane poprzedzającym je stanem procesu.
- Na stan procesu wpływają pośrednio akcje innych procesów w systemie, przejawiające się w stanie współdzielonych zasobów.

Współbieżność i synchronizacja procesów (3)

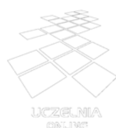
Przetwarzanie w systemie współbieżnym polega na wykonywaniu instrukcji różnych procesów. Wykonanie instrukcji (akcja) skutkuje zmianą stanu procesu i określane jest jako *zdarzenie*.

Instrukcje wykonywane w systemie, niezależnie od procesu, wpływają również na stan systemu jako całości. Stan systemu stanowi pewne otoczenie każdego procesu (jest tym samym elementem stanu każdego z procesów), gdyż niektóre zasoby systemu są przez procesy współdzielone. W najprostszym przypadku można mówić o współdzielonych zmiennych, których wartości wynikają z wykonywania instrukcji różnych procesów. W ten sposób dochodzi do interakcji pomiędzy procesami.

W celu ujednolicenia opisu stan systemu będzie występował wyłącznie w kontekście procesu, a interakcja pomiędzy procesami będzie realizowana poprzez zmienne wejściowe i wyjściowe poszczególnych procesów.

Przedstawioną abstrakcję można również odwzorować na proces wielowątkowy. Proces taki odpowiadałby wówczas systemowi, udostępniając środowisko do wykonywania wątków. Zmiana stanu wynikałaby z realizacji instrukcji związanych z wątkami tego procesu.

Systemy operacyjne



Podstawowe definicje i oznaczenia

- *C* — zbiór instrukcji (operacji), możliwych do wykonania przez jednostkę przetwarzającą (np. dodawanie, odejmowanie, skok, rozgałęzienie warunkowe)
- *D* — zbiór danych, przetwarzanych (modyfikowanych lub czytanych) w ramach wykonywania operacji przez jednostkę przetwarzającą
- $e_i(c, O)$ — wykonanie w ramach procesu P_i instrukcji atomowej $c \in C$ na operandach ze zbioru $O \subseteq D$

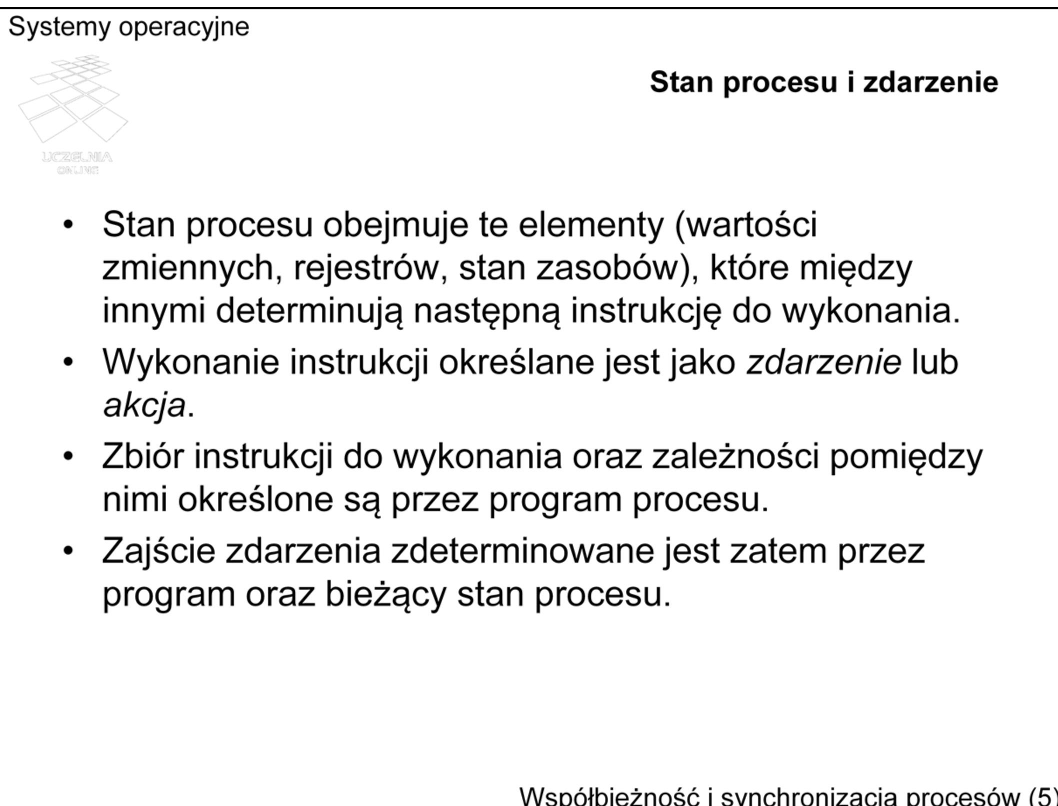
Współbieżność i synchronizacja procesów (4)

Programowanie współbieżne opiera się na dwóch zasadniczych pojęciach: *instrukcji atomowej* i *przeplocie*. Instrukcja atomowa to taka instrukcja, która wykonywana jest w sposób niepodzielny. W trakcie jej wykonywania nie są obsługiwane żadne przerwania.

W systemach z jedną jednostką przetwarzającą warunki te spełnia każdy rozkaz procesora, gdyż sprawdzanie wystąpienia przerwań oraz ewentualna ich obsługa wykonywana jest zawsze na końcu cyklu rozkazowego. Poza tym skutki wykonania takiej instrukcji są natychmiast widoczne dla instrukcji następnych.

W systemach wieloprocesorowych instrukcje mogą być wykonywane jednocześnie przez różne jednostki przetwarzające. Takie wykonanie instrukcji odbywa się w izolacji, co gwarantuje, że żadne częściowe (tymczasowe) wyniki nie są widoczne i tym samym nie wpływają na wykonanie innych instrukcji. Jeśli jednak jednocześnie wykonywane instrukcje operują na tych samych danych, konieczne jest najczęściej wykonanie ich w określonej sekwencji. W systemach wieloprocesorowych, opartych na wspólnej magistrali, efekt ten można uzyskać przez zablokowanie dostępu do magistrali systemowej na czas wykonywania instrukcji. W architekturach Intel, w tym celu przekazywany jest sygnał LOCK, który jest integralnie związany z cyklem rozkazowym w przypadku niektórych rozkazów (np. `xchg`) lub wynika z jawnego żądania ze strony wykonywanego programu, poprzez użycie prefiksu LOCK.

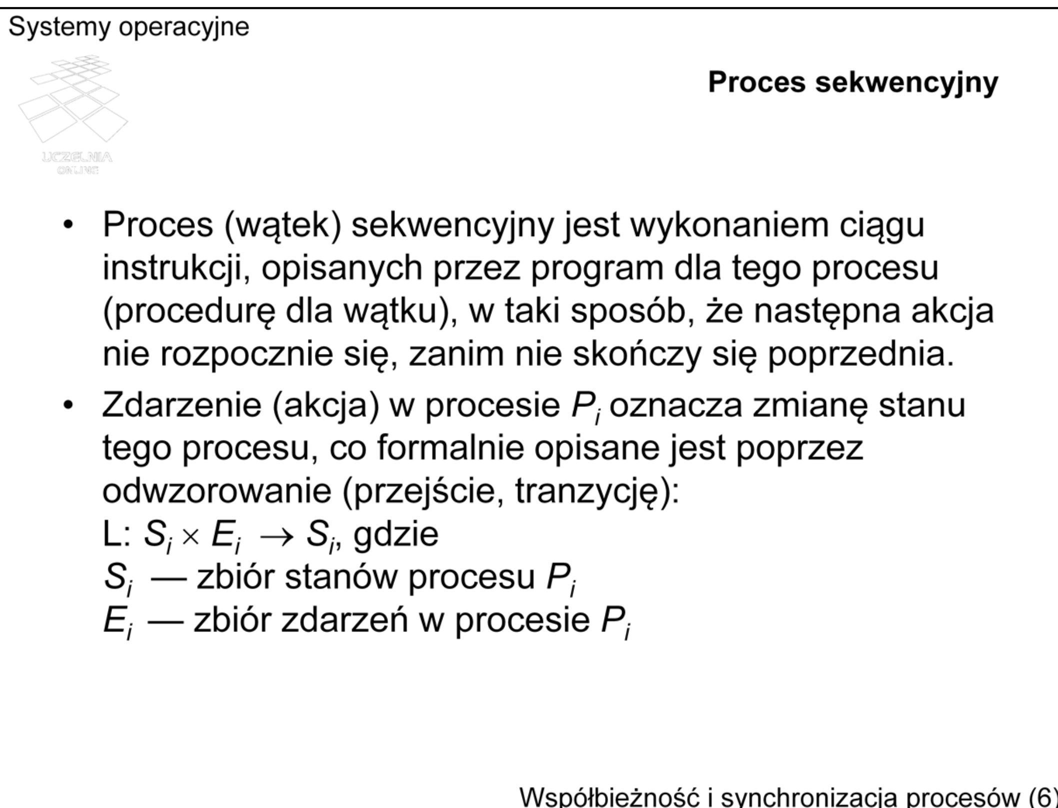
W przyjętych oznaczeniach $e_i\ (c, O)$ oznacza wykonanie w ramach procesu P_i operacji $c \in C$, w wyniku której czytane lub modyfikowane są operandy (argumenty) ze zbioru $O = \{o_1, o_2,..., o_n\} \subseteq D$. Formalnie wykonanie operacji jest zatem elementem iloczynu kartezjańskiego $C \times 2^D$.



Z punktu widzenie współbieżnej i asynchronicznej realizacji przetwarzania przez wiele procesów istotne jest, kiedy jakaś instrukcja się wykona, zwłaszcza instrukcja, która wpływa na otoczenie procesu (potencjalnie zatem na inne procesy). Dlatego najbardziej istotny jest ten aspekt stanu procesu, który dotyczy wykonywania instrukcji i pośrednio wpływa na realizację innych procesów w systemie.

Na poziomie architektury procesora następną instrukcję do wykonania bezpośrednio wskazuje rejestr w procesorze, zwany licznikiem rozkazów lub wskaźnikiem instrukcji. Jednak następna wartość tego rejestru zależy od wykonywanego rozkazu oraz ustawienia flag w rejestrze, zwanym słowem stanu programu. Zawartość tego rejestru wynika z kolei z wcześniej wykonanych instrukcji oraz wartości ich operandów. Poza tym możliwość wykonania instrukcji uwarunkowana jest dostępnością takich zasobów jak pamięć, czy procesor.

Na potrzeby dalszej analizy zakłada się, że określony stan procesu jest unikalny, tzn. ten sam stan nigdy się nie powtórzy, pomimo że stan przetwarzania (stan rejestrów procesora, pamięci) będzie dokładnie taki sam jak wcześniej. Można przyjąć, że elementem stanu procesu jest rzeczywisty czas ostatniego zdarzenia lub licznik wykonanych instrukcji, który będzie się monotonicznie zwiększał.



Przedstawiona definicja jest nieco bardziej sformalizowaną postacią definicji przedstawionej wcześniej na potrzeby zarządzania zasobami systemu.

Odwzorowanie L zdefiniowane jest przez program dla procesu, zakładając, że program jest deterministyczny. Odzwierciedla ono fakt, że zajście zdarzenia w określonym stanie prowadzi do następnego stanu. Opisując to z drugiej strony można stwierdzić, że stan następny uwarunkowany jest zajściem zdarzenia i stanem poprzedzającym to zajście. Odwzorowanie to nie jest określonej dla każdej pary ze zbioru $S_i \times E_i$. W przetwarzaniu sekwencyjnym stan determinuje następne zdarzenie, np. licznik rozkazów wskazuje następny rozkaz do wykonania.

Jak już zasygnalizowano na poprzednim slajdzie, użycie pojęcia *wątek*, obok powszechnie używanego w tym kontekście pojęcia *proces*, podkreśla fakt, że realizacja współbieżnego przetwarzania przebiega we wspólnej przestrzeni adresowej, czyli przy dostępie do współdzielonych danych. W tym module pojęcia *proces* i *wątek* będą utożsamiane.

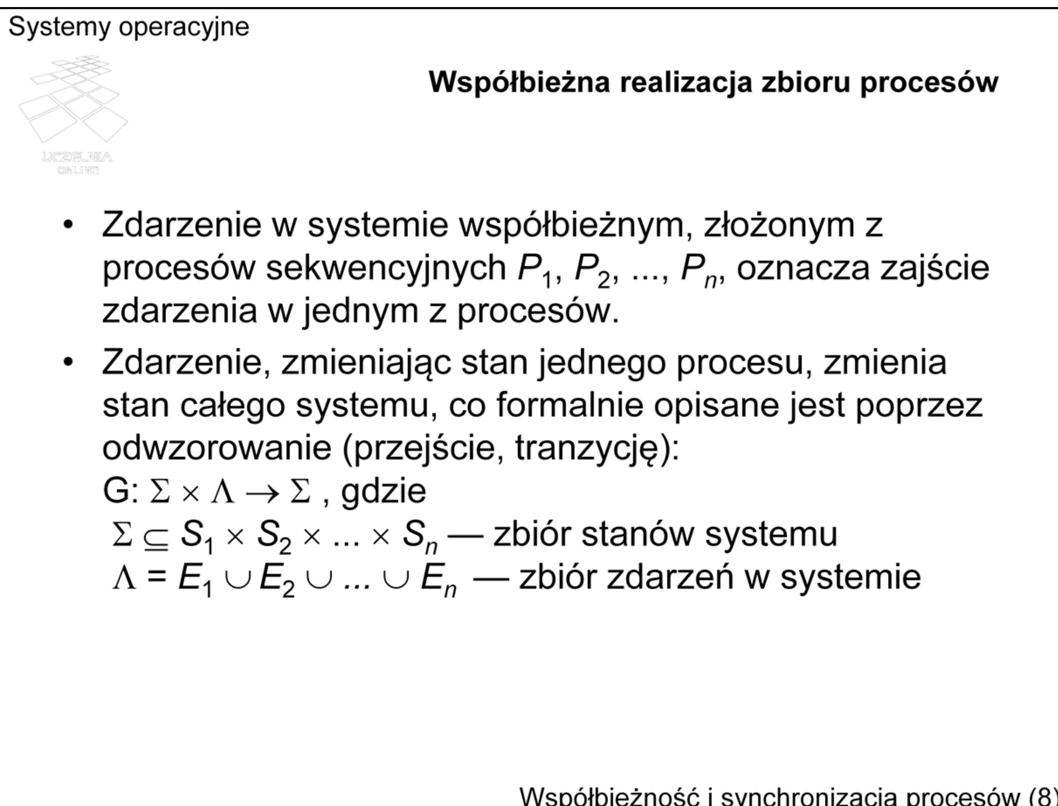
Systemy operacyjne

Relacja lokalnego porządku

- Proces P_i jest ciągiem (skończonym lub nieskończonym) następujących po sobie naprzemiennie stanów i zdarzeń s_i^0, e_i^1, s_i^1, e_i^2, s_i^2, ..., gdzie:
  s_i^0 — stan początkowy procesu P_i
  $\forall_{k\geq 0}\ s_i^k \in S_i$
  $\forall_{k>0}\ e_i^k \in E_i$
  $\forall_{k\geq 0}\ \mathrm{L}(s_i^k, e_i^{k+1}) = s_i^{k+1}$
- Relacja porządku w procesie P_i — odzwierciedlająca kolejność stanów i zdarzeń w ciągu — nazywana będzie lokalnym porządkiem i oznaczana symbolem $\rightarrow_i$

Współbieżność i synchronizacja procesów (7)

Przetwarzanie sekwencyjne oznacza następstwo stanów i zdarzeń. Zarówno zdarzenia, jak i stany w procesie sekwencyjnym są liniowo uporządkowane zgodnie z kolejnością ich wystąpienia.



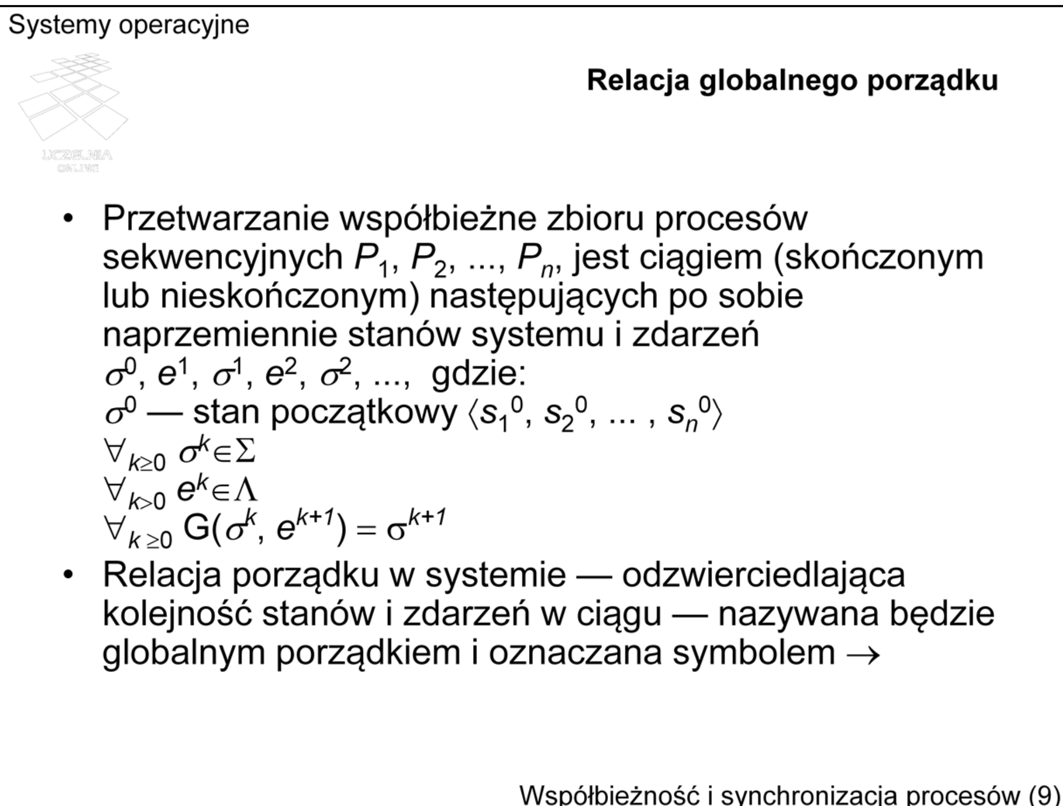
Na potrzeby analizy systemu współbieżnego przyjęte zostaje założenie, że zdarzenia zachodzą w sposób natychmiastowy i pojedynczo, stan natomiast może trwać przez pewien interwał czasu. Oznacza to, że operacje wykonywane są sekwencyjnie, co odpowiada systemowi współbieżnemu z jedną jednostką przetwarzającą.

Zakładając, że w systemie z kilkoma równolegle działającymi procesorami operacje jednoczesne (nakładające się w czasie, ang. overlapping) nie powodują konfliktu w dostępie do danych, model taki jest w dalszym ciągu adekwatny. Można pokazać, że przetwarzanie z jednoczesnym wykonaniem pewnych instrukcji jest równoważne wykonaniu sekwencyjnemu.

Jeśli występuje konflikt w dostępie do danych, tzn. dwie (lub więcej) instrukcje wykonywane jednocześnie mają wspólny operand, przy czym przynajmniej jedna z nich go modyfikuje, konieczne jest ich uszeregowanie w czasie. W wyniku tego uszeregowania instrukcje „w konflikcie” będą wykonane w pewnej sekwencji.

Niezależnie zatem od sposobu realizacji przetwarzania, instrukcje różnych procesów będą analizowane tak, jak gdyby przeplatały się one w czasie.

We współbieżnej realizacji zakłada się, że procesy działają asynchronicznie, tzn. liczba instrukcji poszczególnych procesów, wykonana w jednostce czasu, może być dla każdego z nich inna. Oznacza to, że nie ma pewności, w jakiej kolejności instrukcje różnych procesów będą następowały po sobie, chyba że kolejność tę wymuszą zastosowane mechanizmy synchronizacji.

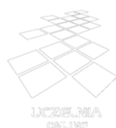


Kontynuując rozumowanie, przedstawione na poprzednich slajdach, można powiedzieć, że na stan przetwarzania współbieżnego składają się stany poszczególnych procesów, a zmiana tego stanu spowodowana jest zdarzeniem w jednym z tych procesów.

Na stan początkowy przetwarzania składają się stany początkowe poszczególnych procesów.

Podobnie jak odwzorowanie L, odwzorowanie G nie jest określone dla każdej pary ze zbioru $\Sigma \times \Lambda$. Nie każde zdarzenie może zatem wystąpić w określonym stanie przetwarzania współbieżnego. W przeciwieństwie do stanu procesu sekwencyjnego stan przetwarzania współbieżnego nie determinuje jednak jednoznacznie następnego zdarzenia, gdyż potencjalnie jakieś zdarzenie może wystąpić w każdym ze współbieżnych procesów. Liczba możliwych zdarzeń może być zredukowana poprzez zastosowanie mechanizmów synchronizacji.

Systemy operacyjne



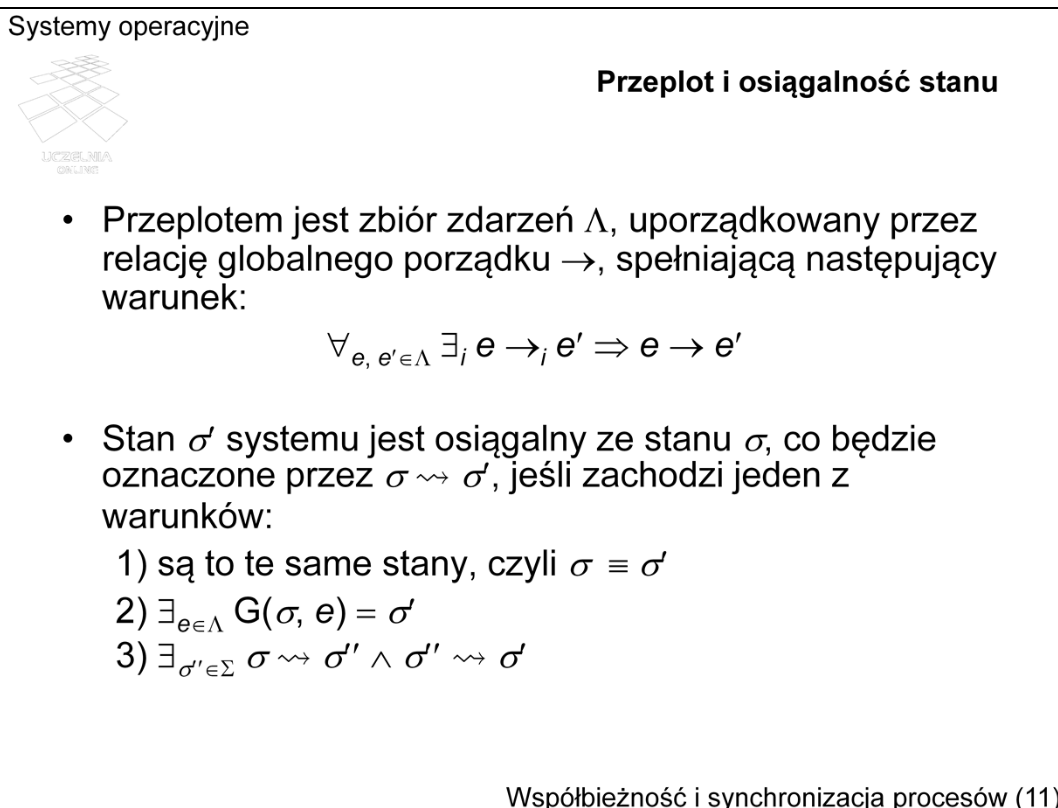
Niedeterminizm przetwarzania

- Zdarzenie, które może pojawić się w określonym stanie przetwarzania, określane jest jako zdarzenie dopuszczalne.
- W przetwarzaniu współbieżnym z każdym sekwencyjnym procesem w danym stanie związane jest jedno zdarzenie. W stanie całego przetwarzania jest zatem zbiór zdarzeń dopuszczalnych.
- W zależności od dostępności zasobów (procesora, magistrali systemowej) oraz decyzji planisty, wykonywany będzie jeden z procesów gotowych. Należy więc przyjąć, że zajście jednego ze zdarzeń dopuszczalnych ma charakter losowy.

Współbieżność i synchronizacja procesów (10)

Analizując konkretną realizację przetwarzania można określić jakie stany i jakie zdarzenia miały miejsce, kiedy miały miejsce (w jakiej kolejności). Analizując program dla przetwarzania współbieżnego, pewne stany i zdarzenia należy przewidzieć, gdyż każde wykonanie takiego programu może przebiegać nieco inaczej. Ponieważ w większości przypadków dopuszczalnych jest wiele zdarzeń różnych procesów, w przypadku przetwarzania asynchronicznego nie wiadomo, które dokładnie zdarzenie zajdzie jako kolejne. Stąd niedeterminizm takiego przetwarzania.

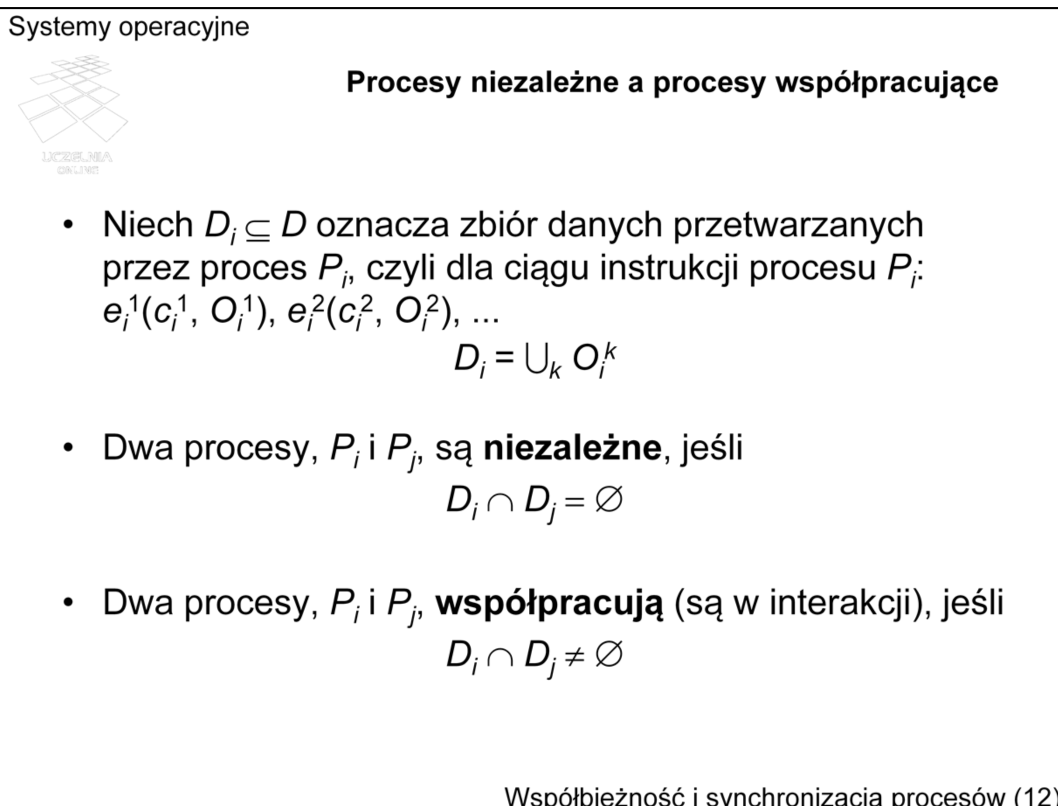
W niektórych przypadkach kolejność wystąpienia zdarzeń nie ma znaczenia, osiągamy ostatecznie taki sam stan. Są jednak przypadki, w których wystąpienie określonego zdarzenia na tyle istotnie wpływa na stan przetwarzania współbieżnego, że sterowanie w niektórych procesach może przebiegać zupełnie inną ścieżką. Zjawisko, w którym w zależności od kolejności pewnych zdarzeń system osiąga różne stany, określa się jako *hazard* (ang. race condition).



Przeplot jest takim globalnym uporządkowaniem akcji w systemie, które zachowuje porządek wynikający z programu każdego ze współbieżnych procesów. Używając zależności teoriomnogościowych, można stwierdzić, że $\bigcup_{1\leq i\leq n} \rightarrow_i \subseteq \rightarrow$ lub dokładniej, że relacja $\rightarrow$ jest liniowym rozszerzeniem przechodniego domknięcia sumy mnogościowej $\bigcup_{1\leq i\leq n} \rightarrow_i$.

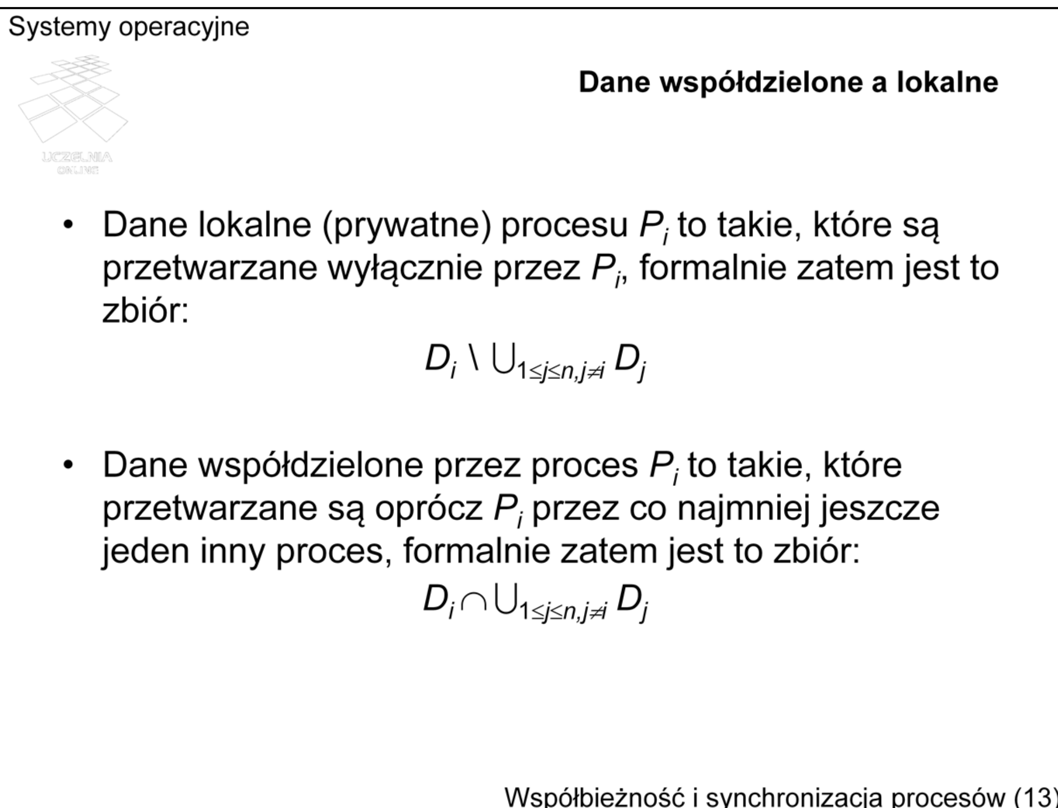
Przeplot może być analizowany w kontekście zrealizowanego już przetwarzania, a może być rozważany potencjalnie, jako ciąg dopuszczalnych zdarzeń i wynikających z nich stanów, na potrzeby weryfikacji poprawności lub innych własności. W tym drugim przypadku, uwzględniając niedeterminizm, należałoby raczej mówić o pewnym zbiorze możliwych przeplotów, czyli różnych uporządkowaniach tego samego lub zbliżonego zbioru zdarzeń. Różnice w samym zbiorze zdarzeń mogą wynikać z faktu, że w zależności od stanu przetwarzania, przebieg sterowania w poszczególnych procesach może być nieco inny, w związku z czym pewne instrukcje mogą zostać pominięte.

Z punktu widzenia analizy określonych własności, typu bezpieczeństwo, żywotność, zakleszczenie, istotny jest nie tyle przeplot ile stan systemu, który powstanie w wyniku zajścia zdarzeń w przeplocie. Kluczowe w tym kontekście jest pojęcie osiągalności stanów. Osiągalność jakiegoś stanu z innego stanu zachodzi wówczas, gdy istnieje przeplot, który prowadzi z jednego stanu do drugiego. Wyraża to formalnie definicja rekurencyjna, przedstawiona na slajdzie.



Procesy niezależne to takie, w przypadku których nie można mówić o bezpośrednim wpływie jednego z nich na stan innego. Procesy takie mogą oczywiście rywalizować o zasoby, zarządzane przez system operacyjny. Niedostatek tych zasobów (np. czasu procesora, pamięci fizycznej, dostępności drukarki itp.) może spowodować spowolnienie przetwarzania jedno z nich na rzecz drugiego. **W każdym z procesów wystąpi jednak prędzej czy później określona sekwencja stanów, niezależnie od przeplotu operacji tych procesów.**

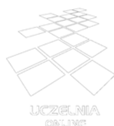
Procesy współpracujące ze sobą mogą się komunikować i rywalizować o dostępność zasobów, a z faktu wystąpienia interakcji między nimi może wynikać taka lub inna sekwencja osiąganych stanów a nawet realizowanych instrukcji (taki lub inny przepływ sterowania). Zdarzenie związane z jednym z procesów może mieć zatem wpływ na wybór instrukcji do wykonania w innym. Na przykład, jeden z procesów modyfikuje współdzieloną zmienną, od wartości której zależy spełnienie warunku wykonania instrukcji lub pętli w innym procesie.



Niektóre zmienne mogą być dostępne w obrębie tylko jednego procesu. Nie mogą one być oni czytane ani modyfikowane przez inne procesy. Takie zmienne będą określane jako *lokalne*, a w algorytmach ich definicje poprzedzane będą modyfikatorem **`local`**.

Zmienne współdzielone z kolei dostępne są dla kilku (w szczególności wszystkich) procesów. Można rozważać różne schematy dostępności zmiennych współdzielonych, np. dostępność do zapisu w jednym procesie, a do odczytu w pozostałych procesach. W prezentowanych algorytmach własności takie nie będą wyrażane explicite, mogą jednak wynikać konstrukcji algorytmu. W algorytmach definicje zmiennych współdzielonych będą poprzedzane modyfikatorem **`shared`**.

Systemy operacyjne



Dane wejściowy i wyjściowe

- Niech $D_i^R \subseteq D_i$ oznacza zbiór danych czytanych przez proces P_i, a $D_i^M \subseteq D_i$ oznacza zbiór danych modyfikowanych przez proces P_i
- Dane wejściowe procesu P_i to takie dane współdzielone, które są czytane przez proces P_i, formalnie zatem jest to zbiór:

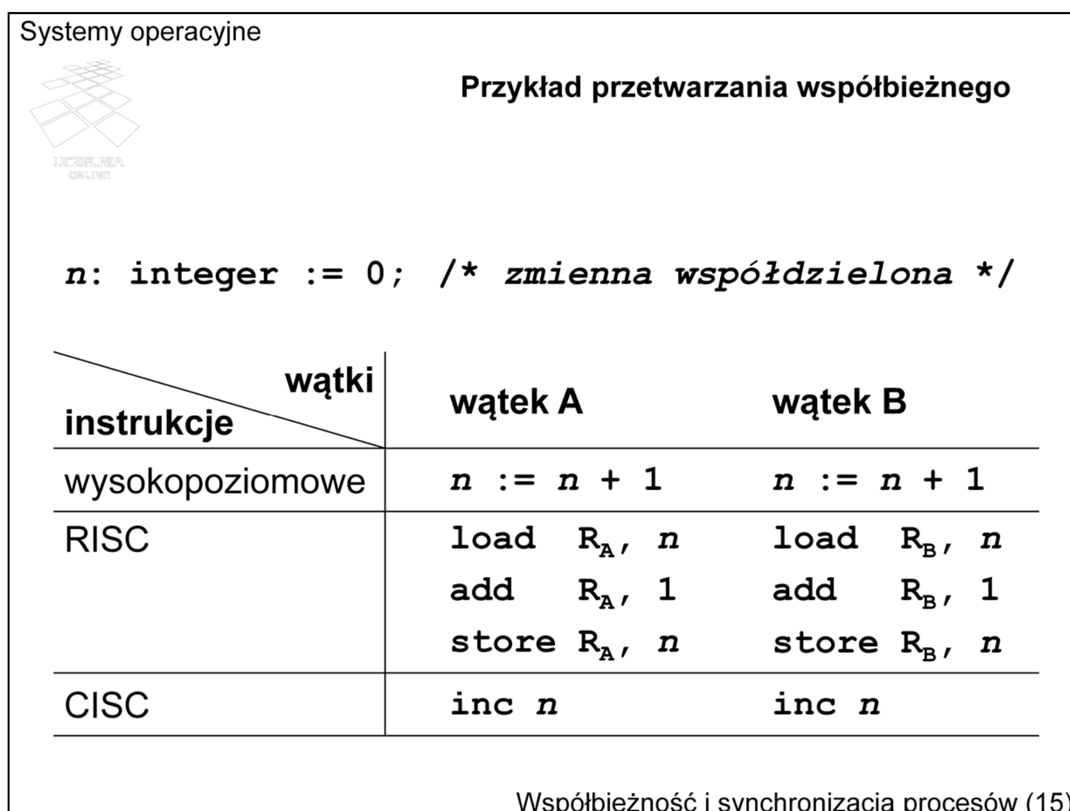
$$D_i^R \cap \bigcup_{1 \leq j \leq n, j \neq i} D_j$$

- Dane wyjściowe procesu P_i to takie dane współdzielone, które są modyfikowane przez proces P_i, formalnie zatem jest to zbiór:

$$D_i^M \cap \bigcup_{1 \leq j \leq n, j \neq i} D_j$$

Współbieżność i synchronizacja procesów (14)

Za pośrednictwem danych (zmiennych) wejściowych proces uzyskuje informacje od innych procesów z jego otoczenia. Za pośrednictwem danych wyjściowych z kolei proces przekazuje informację do innych procesów z otoczenia. Zbiory danych wejściowych i wyjściowych nie muszą być rozłączne — pewne zmienne mogą być zarówno wejściowymi dla danego procesu, jak i wyjściowymi. Takie dane będą określane jako wejściowo-wyjściowe.


Systemy operacyjne

Przykład przetwarzania współbieżnego

```
n: integer := 0; /* zmienna współdzielona */
```

instrukcje \ wątki	wątek A	wątek B
wysokopoziomowe	`n := n + 1`	`n := n + 1`
RISC	load R_A, n add R_A, 1 store R_A, n	load R_B, n add R_B, 1 store R_B, n
CISC	`inc n`	`inc n`

Współbieżność i synchronizacja procesów (15)

W przedstawionym programie instrukcja podstawienia *n* := *n* + 1 wykonywana jest współbieżnie przez 2 wątki: A oraz B, a *n* jest zmienną współdzieloną przez te wątki. Innymi słowy, zmienna *n* znajduje się w obszarze pamięci współdzielonym przez te wątki i jest dla nich zmienną wejściowo-wyjściową. Instrukcja podstawienia wymaga wykonania operacji arytmetycznej, w związku z czym może być różnie przetłumaczona na kod maszynowy, czyli na sekwencję instrukcji wykonywanych atomowo.

W procesorach o architekturze RISC operacje arytmetyczne wykonywane są wyłącznie na rejestrach procesora, wobec czego podstawienie takie wymaga wcześniejszego załadowania zawartości komórki pamięci, przechowującej wartość zmiennej *n*, do odpowiedniego rejestru, dodania wartości 1 do zawartości tego rejestru, a następnie umieszczenia zmodyfikowanej wartości ponownie w pamięci pod adresem przypisanym zmiennej *n*. W tym celu wątek A korzysta z jakiegoś rejestru procesora, oznaczonego R_A, a wątek B z rejestru R_B. W szczególności może to być ten sam rejestr, ale raz występujący w kontekście wątku A, a raz w kontekście wątku B.

W procesorach o architekturze CISC powszechne są rozkazy typu *odczytmodyfikacja-zapis* (ang. read-modify-write). Przykładem może być 16- lub 32-bitowa architektura intelowska z rozkazem **`inc`**, którego operand może być w pamięci. Rozkaz ten może być zatem użyty w przekładzie na kod maszynowy wysokopoziomowej instrukcja podstawienia *n* := *n* + 1.

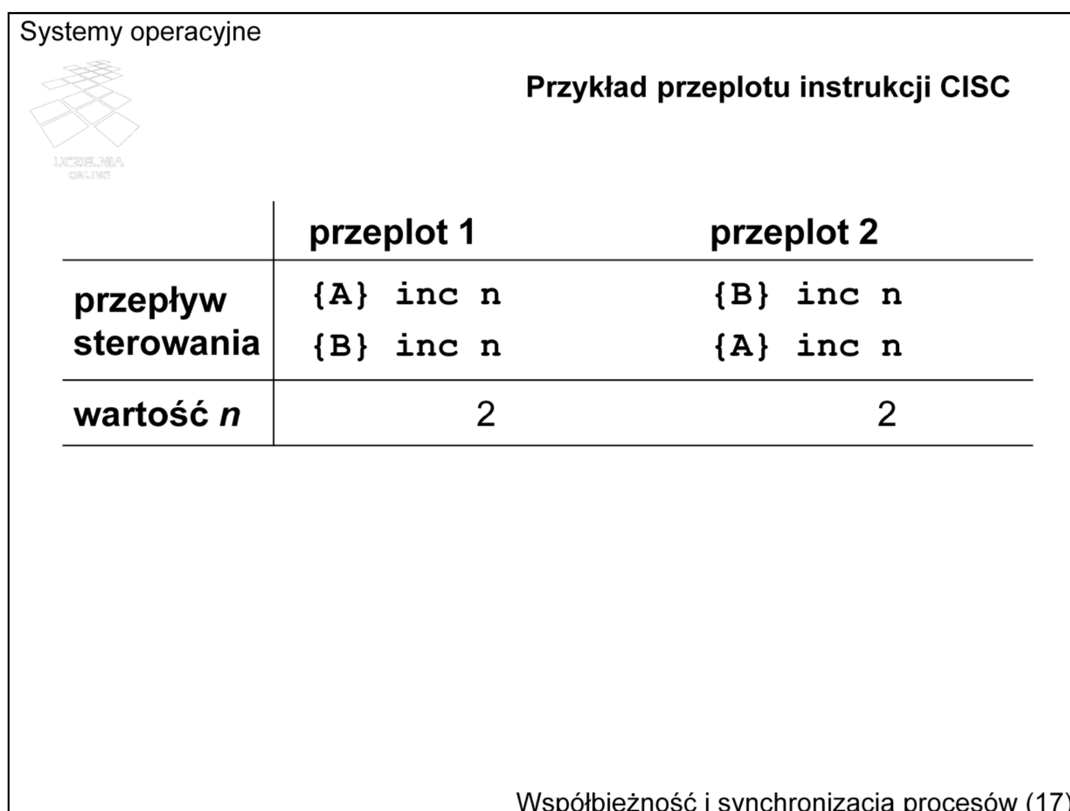
Systemy operacyjne

Przykład przeplotu instrukcji RISC

	przeplot 1	przeplot 2
przepływ sterowania	{A} load R_A, *n*	{A} load R_A, *n*
	{A} add R_A, 1	{A} add R_A, 1
	{A} store R_A, *n*	// *n = 0*
	// *n = 1*	{B} load R_B, *n*
	{B} load R_B, *n*	{B} add R_B, 1
	{B} add R_B, 1	{A} store R_A, *n*
	{B} store R_B, *n*	{B} store R_B, *n*
wartość *n*	2	1

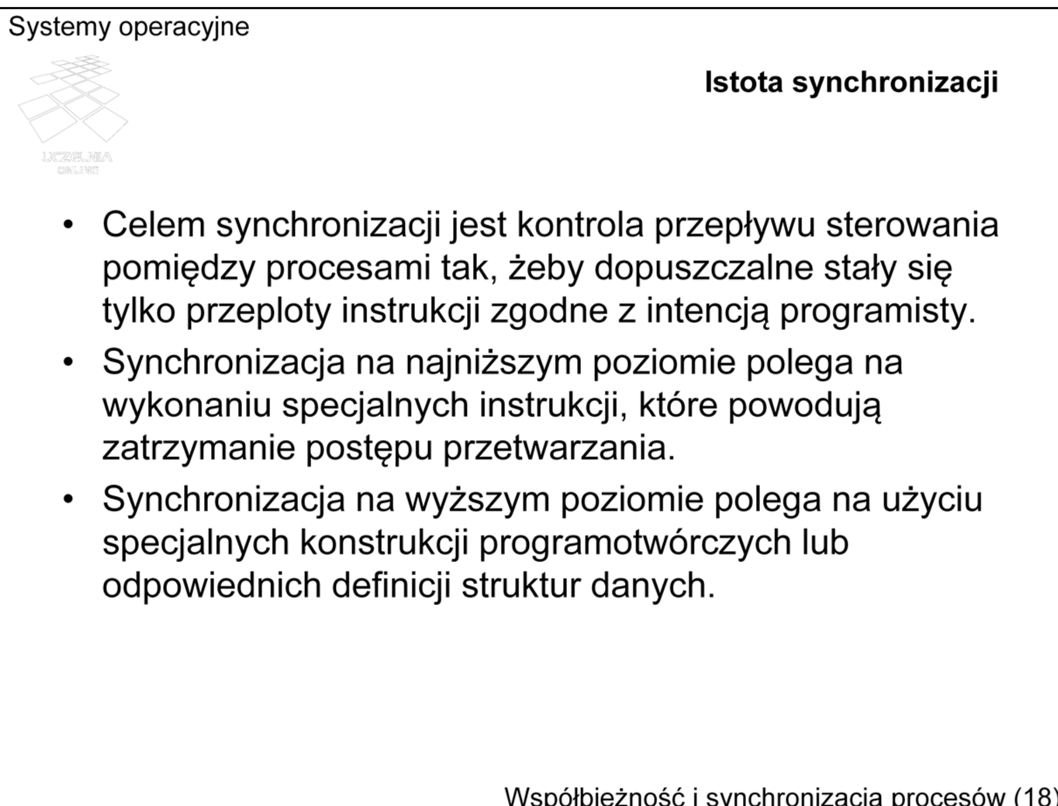
Współbieżność i synchronizacja procesów (16)

Dwa różne przeploty operacji wątków A i B prowadzą do innych wyników. Przeplot pierwszy daje wynik zgodny z oczekiwaniami, a w przeplocie drugim wątek B czyta zmienną *n* po jej modyfikacji w wątku A, ale przed udostępnieniem zmodyfikowanej wartości dla otoczenia. Można zatem powiedzieć, że w momencie pobrania wartości zmiennej *n* przez wątek B modyfikacja miała lokalny charakter w wątku A. Innymi słowy, przedstawiony sposób modyfikacji nie jest wykonaniem instrukcji atomowej i po dwóch pierwszych operacjach wątku A nastąpiło przełączenie kontekstu na wątek B.



	przeplot 1	przeplot 2
przepływ sterowania	{A} inc n {B} inc n	{B} inc n {A} inc n
wartość *n*	2	2

W przypadku procesora typu CISC można wykonać atomowo operację zwiększenia o 1 bez ryzyka niepożądanego przeplotu. Niezależnie od kolejności wykonania operacji przez oba procesy wynik jest zgodny z oczekiwaniami. Wykonanie takiej operacji w systemie wieloprocesorowym wymagałoby jednak użycia prefiksu LOCK przed instrukcją inkrementacji w programie dla wątków. W przeciwnym przypadku istnieje ryzyko, że w trakcji realizacji operacji przez jeden procesor, drugi procesor rozpocznie wykonywanie konfliktowej operacji.

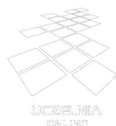


Jak wydać na przedstawionym wcześniej przykładzie, nie wszystkie przeploty operacji współbieżnych procesów (wątków) są dopuszczalne z punktu widzenie oczekiwań programisty. Swobodę przeplotu należy zatem czasami ograniczyć poprzez zastosowanie mechanizmów synchronizacji w celu kontroli przepływu sterowania pomiędzy współbieżnymi procesami.

Synchronizacja na najniższym poziomie polega na wykonaniu określonych (często specjalnych) instrukcji, które powodują zablokowanie postępu przetwarzania do czasu wystąpienia określonego zdarzenia w systemie, związanego również z instrukcją synchronizującą, ale w innym wątku.

Synchronizacja na wyższym poziomie polega na użyciu w programie specjalnych konstrukcji lub odpowiednim zdefiniowaniu struktur danych, które kompilator zamienia na właściwe instrukcje synchronizujące, udostępniane przez system operacyjny lub architekturę procesora.

Systemy operacyjne



Poprawność programów współbieżnych

- Własność bezpieczeństwa (ang. safety) — w każdym osiągalnym stanie przetwarzania muszą być spełnione pewne warunki.
- Własność żywotności (ang. liveness) — w wyniku przetwarzania muszą w końcu zajść pewne warunki.

Współbieżność i synchronizacja procesów (19)

Problem wyspecyfikowania intencji programisty odnośnie dopuszczalnych przeplotów wiąże się z warunkami poprawności programów współbieżnych. Ogólnie formułuje się dwie podstawowe własności:

- bezpieczeństwo — w każdym stanie przetwarzania współbieżnego (niezależnie od przeplotu) spełniony będzie pewien warunek, zwany od nazwy własności *warunkiem bezpieczeństwa*,
- żywotność — w wyniku przetwarzania, po skończonej liczbie zdarzeń, zajdzie określony (oczekiwany) warunek.

Systemy operacyjne

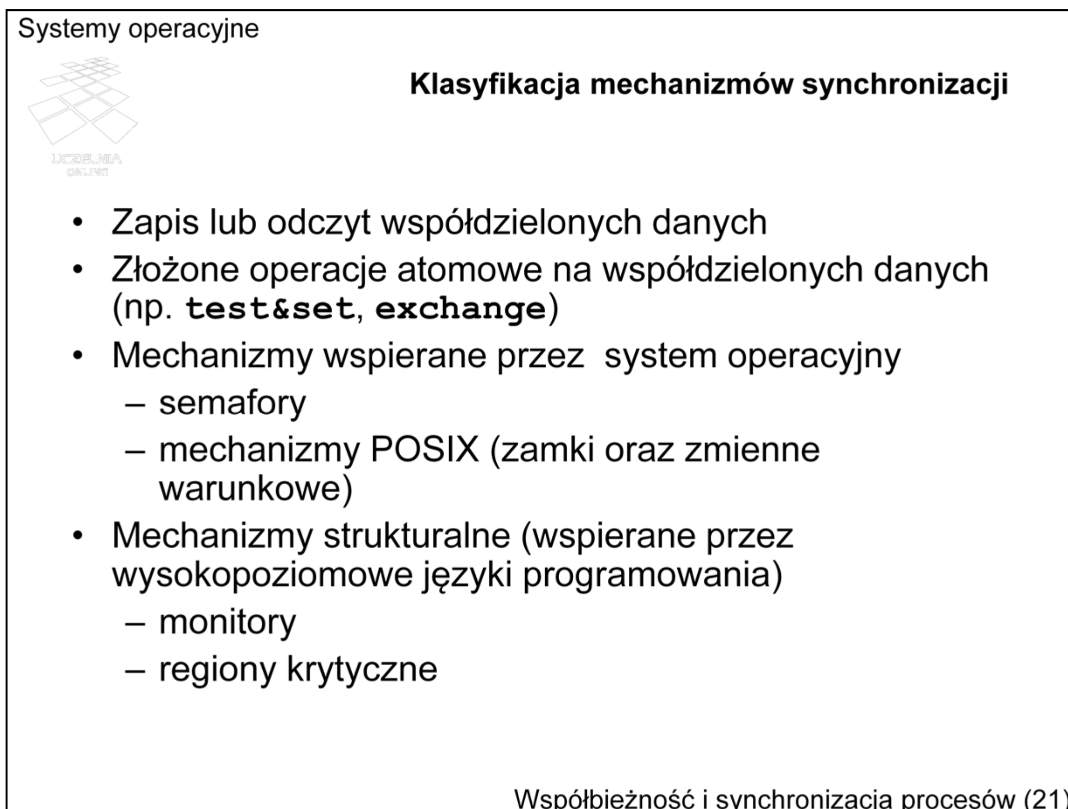
Własność uczciwości programów współbieżnych

- Uczciwość słaba — nieprzerwanie zgłaszane żądanie procesu będzie kiedyś obsłużone.
- Uczciwość mocna — nieskończenie wiele razy zgłaszane żądanie procesu będzie kiedyś obsłużone.
- Oczekiwanie liniowe — żądanie procesu będzie obsłużone po najwyżej jednokrotnym obsłużeniu żądań innych procesów.
- FIFO — żądania będą realizowane w kolejności zgłoszeń.

Współbieżność i synchronizacja procesów (20)

Dodatkowo sformułować można własność uczciwości (lub inaczej sprawiedliwości, ang. fairness) programów współbieżnych. Własność ta jest uszczegółowieniem własności żywotności i precyzuje czas oczekiwania na wystąpienie określonego stanu.

Własność uczciwości zostaje tylko zasygnalizowana i nie będzie w dalszej części analizowana w prezentowanych algorytmach.



Wśród mechanizmów synchronizacji można wyodrębnić dwie zasadnicze klasy:

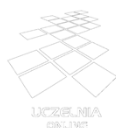
- mechanizmy sprzętowe — wspierane przez rozwiązania na poziomie maszynowym procesora (lub architektury komputera), związane z listą rozkazów i obsługą przerwań,
- mechanizmy systemowe — zintegrowane z systemem operacyjnym i związane z odpowiednim zarządzaniem procesami.

Środki udostępniane przez poziom maszynowy procesora to głównie atomowy zapis oraz odczyt współdzielonych danych, określanych również jako współdzielone rejestry, czyli współdzielone komórki pamięci. Na poziomie architektury komputera mamy gwarancję atomowego transferu danych 8-, 16-, 32-bitowych itd. pomiędzy rejestrami procesora a pamięcią, co daje pewność, że dane są zapisywane lub odczytywane „w całości". Procesor może też udostępnić bardziej złożone operacje atomowe na współdzielonych rejestrach lub bitach rejestrów. Do takiej grupy należą np. instrukcje **`test&set`** oraz **`exchange`**.

Synchronizacja za pomocą odpowiednich instrukcji opartych na rozwiązaniach w architekturze procesora oznacza konieczność permanentnego wykonywania określonej instrukcji, aż do uzyskania oczekiwanego efektu (odpowiednik odpytywania w interakcji jednostki centralnej z urządzeniem wejścia-wyjścia). Takie podejście przy przedłużającym się oczekiwaniu oznacza najczęściej marnowania czasu procesora, chyba że procesor jest dedykowany wyłącznie do wykonywania danego procesu. W systemach ogólnych, gdy liczba zadań znacznie przekracza liczbę jednostek przetwarzających, lepszym rozwiązaniem jest uśpienie procesu do czasu zajścia oczekiwanego zdarzeniu lub osiągnięcia określonego stanu. Wykonanie instrukcji synchronizującej oznacza odpowiednią zmianę stanu procesu, co jest z kolei sygnałem dla planisty, że proces nie jest gotowy i nie jest rozważany jako kandydat do przydziału procesora. Do tego typu mechanizmów należą semafory oraz mechanizmy standardu POSIX — zamki (inaczej rygle, muteksy) i zmienne warunkowe.

Ponadto języki programowania wysokiego poziomu dostarczają konstrukcji do wyrażania oczekiwań odnośnie sposobu współbieżnej realizacji instrukcji lub dostępu do współdzielonych danych. Jako przykłady mechanizmów z tej grupy podać można monitory, regiony krytyczne, obiekty chronione (w języku Ada 95), spotkania symetryczne (w języku CSP lub Occam), spotkania asymetryczne (w języku Ada-83).

Systemy operacyjne



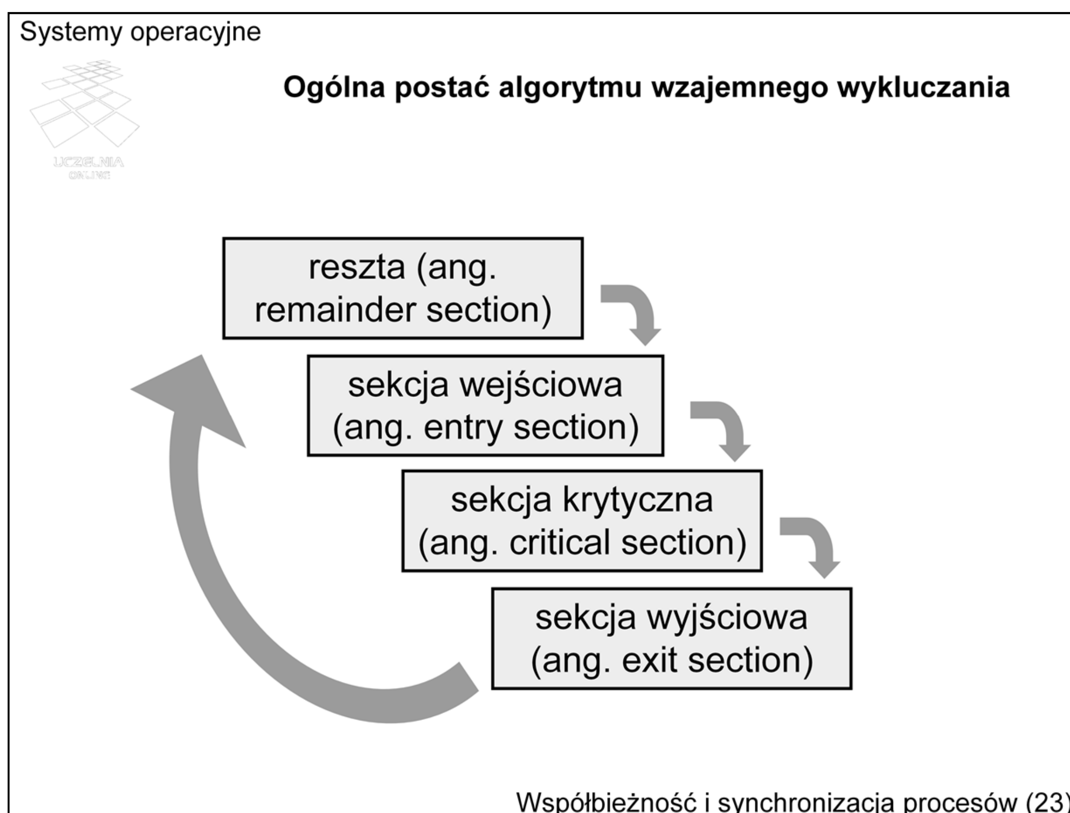
Wzajemne wykluczanie — sformułowanie problemu

- W systemie działa *n* procesów P_0, P_1, ..., P_{n-1}.
- W programie każdego procesu znajduje się fragment kodu zwany sekcją krytyczną (ang. critical section).
- Sekcja krytyczna wykonywana jest w danej chwili przez co najwyżej jeden proces.

Współbieżność i synchronizacja procesów (22)

Problem wzajemnego wykluczania (ang. mutual execlusion) w ogólności formułowany jest dla *n* procesów, przy czym dla wygody prezentacji niektórych algorytmów numeracja przyjęta jest od 0 do *n*-1, a nie od 1 do *n*.

Sekcja krytyczna jest fragmentem kodu w programie każdego z procesów, który ze względu na poprawność nie może być wykonywany współbieżnie. Wykonywanie sekcji krytycznej przez jeden proces wyklucza możliwość wykonywania swoich sekcji krytycznych przez inne procesy (stąd nazwa wzajemne wykluczanie). Sekcja krytyczna każdego procesu może być inna. Najczęściej jest to fragment kodu związany z modyfikacją jakieś współdzielonej zmiennej lub z dostępem do jakiegoś zasobu, który może być używany w trybie wyłącznym. Przykładem operacji, która powinna być wykonywana w sekcji krytycznej jest zwiększanie o 1 wartości zmiennej *n* w przykładzie przedstawionym wcześniej. Podobnym przykładem, z różnym kodem w sekcji krytycznej jest zwiększanie licznika o 1 przez jeden proces, a zmniejszanie o 1 przez drugi proces. Z tego typu przypadkiem w praktyce można mieć do czynienia w niektórych rozwiązaniach jednego z klasycznym problemów synchronizacji — problemu producenta i konsumenta.

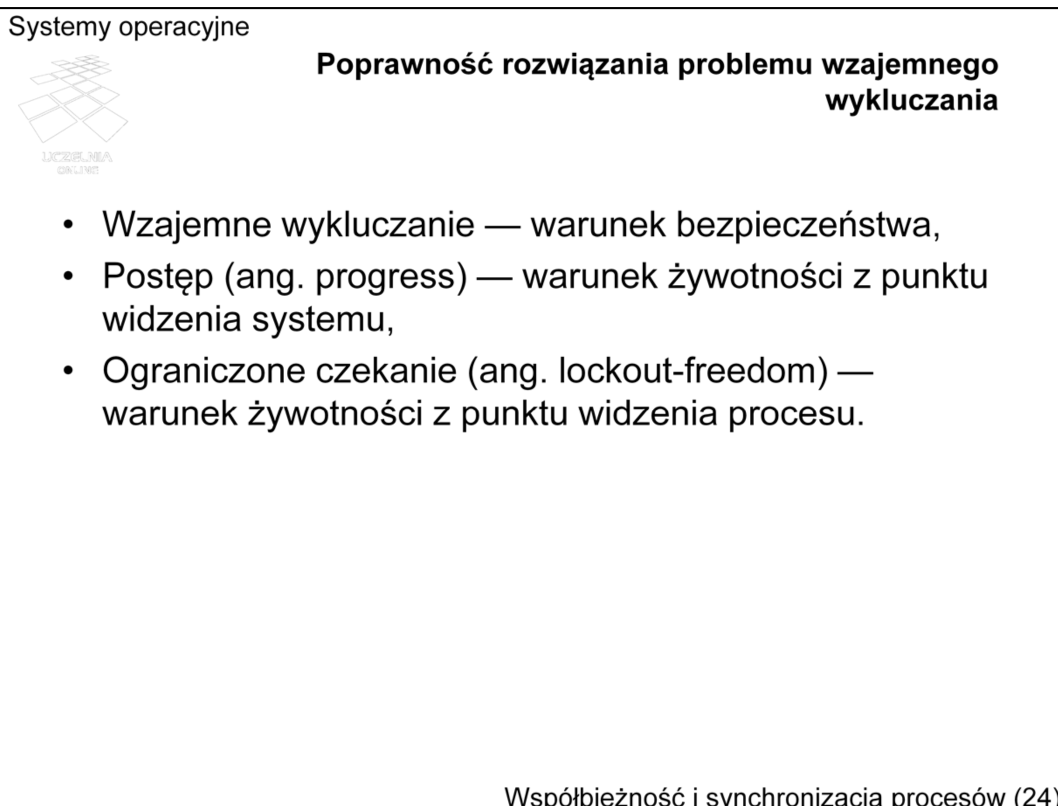


W ogólnej strukturze algorytmu wzajemnego wykluczania wyróżnia się 4 część:

- resztę — część nie związaną w żaden sposób z realizacją sekcji krytycznej,
- sekcję wejściową — w której proces sygnalizuje swoje zamiary wejścia do sekcji krytycznej oraz sprawdza zamiary innych procesów, następnie podejmuje decyzję co do wejścia do sekcji krytycznej,
- sekcję krytyczną — fragment kodu wykonywany w trybie wyłącznym,
- sekcję wyjściową — w której proces informuje o wyjściu z sekcji krytycznej i daje tym samym sygnał do wejścia następnemu procesowi lub sygnał do wznowienia rywalizacji procesem przebywającym w swoich sekcja wejściowych.

W celu wyeksponowania protokołu dostępu, algorytmy prezentowane na kolejnych slajdach rozpoczynają się od sekcji wejściowej. Ponadto, abstrahuje się od kwestii aplikacyjnych, z których mogłoby wynikać, że w procesie jest kilka niezależnych sekcji krytycznych, związanych z dostępem do różnych zmiennych współdzielonych lub zasobów. **Istotą jest realizacja protokołu dostępu!**

W algorytmach tych nie wyróżnia się również procesu, który byłby arbitrem dla procesów rywalizujących o sekcję krytyczną. Wszystkie decyzje odnośnie wejścia do sekcji krytycznej podejmowane są na podstawie informacji, znajdujących się we współdzielonym obszarze pamięci. Podejścia z arbitrem wbrew pozorom nie ułatwiają rozwiązania problemu, gdyż wymagają pewnych środków komunikacji międzyprocesowej, których implementacja na bazie pamięci współdzielonej wymaga z kolei odpowiednich mechanizmów synchronizacji, w szczególności gwarancji wzajemnego wykluczania. W pewnym sensie jednak rozwiązania bazujące na mechanizmach systemowych (np. semaforach) opierają się na arbitrażu ze strony jądra systemu operacyjnego, które decyduje o przejściu procesu zablokowanego w stan gotowości.



Zasadnicze założenie na potrzeby analizy poprawności rozwiązania problemu wzajemnego wykluczania to **skończony czas przebywania procesu w swojej sekcji krytycznej**.

Warunkiem bezpieczeństwa jest wzajemne wykluczanie, co oznacza, że nigdy w systemie nie może zaistnieć stan, w którym dwa (lub więcej) procesy byłyby w swojej sekcji krytycznej.

Warunek postępu oznacza, że jeśli nie ma żadnego procesu w sekcji krytycznej, a są procesy w sekcji wejściowej, to jeden z nich w skończonym czasie (po zajściu skończonej liczby zdarzeń w systemie) wejdzie do sekcji krytycznej.

Warunek postępu nie gwarantuje, że konkretny proces wejdzie do sekcji krytycznej. Może się zdarzyć tak, że w momencie przejścia do sekcji wyjściowej, proces opuszczający sekcję krytyczną do sygnał do wejścia procesom oczekującym w sekcji wejściowej, w wyniku którego jakiś proces wejdzie do sekcji krytycznej, a inny (przy zachowaniu warunku bezpieczeństwa) oczywiście nie wejdzie. Przy kolejnym sygnale ze strony procesu wychodzącego z sekcji krytycznej pominięty poprzednio proces ponownie może nie uzyskać prawa wejścia, podczas gdy inny proces wykonujący swoją sekcję wejściową prawo takie dostanie. Sytuacja może się powtarzać w nieskończoność. Postęp jest zachowany bo jakiś proces wchodzi do sekcji krytycznej, ale istnieje proces permanentnie pomijany.

Warunek ograniczonego czekania gwarantuje właśnie, że każdy proces ubiegający się o wejście do sekcji krytycznej w końcu (w skończonym czasie, po skończonej liczbie zdarzeń w systemie) uzyska prawo wejścia do niej. Warto podkreślić, że nie wszystkie algorytmy gwarantuję tę własność.

Systemy operacyjne

Wzajemne wykluczanie 2 procesów — podejście 1

Algorytm dla procesu P_i przy dwóch procesach P_i i P_j

```
shared numer: Integer := i;

while numer ≠ i do      } sekcja wejściowa
   nic;
sekcja krytyczna_i;
numer := j;            ← sekcja wyjściowa
reszta_i;
```

Współbieżność i synchronizacja procesów (25)

Pierwsze podejście do rozwiązania problemu wzajemnego wykluczania w oparciu o pamięć współdzieloną polega na wykorzystaniu współdzielonej zmiennej wejściowo-wyjściowej *numer*, wskazującej numer (identyfikator) procesu, który ma prawo wejść do sekcji krytycznej. Podejście rozważane jest w kontekście dwóch procesów, mogłoby jednak być uogólnione na większą ich liczbę. Drugi z procesów wykonuje analogiczne operacje, przy czym w miejscu numeru *i* występuje *j* i odwrotnie. Kod dla P_j wygląda zatem następująco:

```
while numer ≠ j do
   nic;
sekcja krytyczna_j;
numer := i;
reszta_j;
```

Zmienna *numer*, ze względu na fakt współdzielenia, jest inicjalizowana globalnie wartością, która jest numerem jednego z procesów. Proces o tym numerze będzie mógł jako pierwszy wejść do sekcji krytycznej, podczas gdy wszystkie pozostałe procesy, ubiegające się o wejście, utkną w pętli **`while`**.

Ponieważ dopiero w sekcji wyjściowej proces ustawia numer następnego procesu do wejścia do sekcji krytycznej, nie ma ryzyka naruszenie warunku bezpieczeństwa. Podejście to wymusza jednak naprzemienność zajmowania sekcji krytycznej przez dwa procesy. Nie jest zatem spełniony warunek postępu, gdyż proces P_i, wychodząc z sekcji krytycznej, nie może zająć jej ponownie, zanim nie zrobi tego proces P_j. Może więc dojść do takiego stanu, w którym proces P_i po opuszczeniu sekcji krytycznej ponownie wchodzi do sekcji wejściowej i nie może zająć sekcji krytycznej. Jeśli z programu procesu P_j wynika, że nie będzie on już wchodził do sekcji krytycznej, proces P_i nie wejdzie tam nigdy.

Systemy operacyjne

Wzajemne wykluczanie 2 procesów — podejście 2

Algorytm dla procesu P_i przy dwóch procesach P_i i P_j
(dla uproszczenia prezentacji założono, że $i = 0$, a $j = 1$)

```
shared znacznik: array [0..1] of
                 Boolean := false;

znacznik[i] := true;          }
while znacznik[j] do          } sekcja wejściowa
   nic;                       }
sekcja krytycznai;
znacznik[i] := false;   ←  sekcja wyjściowa
resztai;
```

Współbieżność i synchronizacja procesów (26)

Podejście bazuje na współdzielonej tablicy *znacznik*, przy czy każdy z dwóch procesów modyfikuje w niej pozycję odpowiadającą swojemu numerowi. Traktując poszczególne pozycje tablicy *znacznik* w odseparowaniu można stwierdzić, że dla procesu P_i *znacznik*[*i*] jest zmienną wyjściową, a dla P_j jest zmienną wejściową. Odwrotna zależność dotyczy oczywiście zmiennej *znacznik*[*j*].

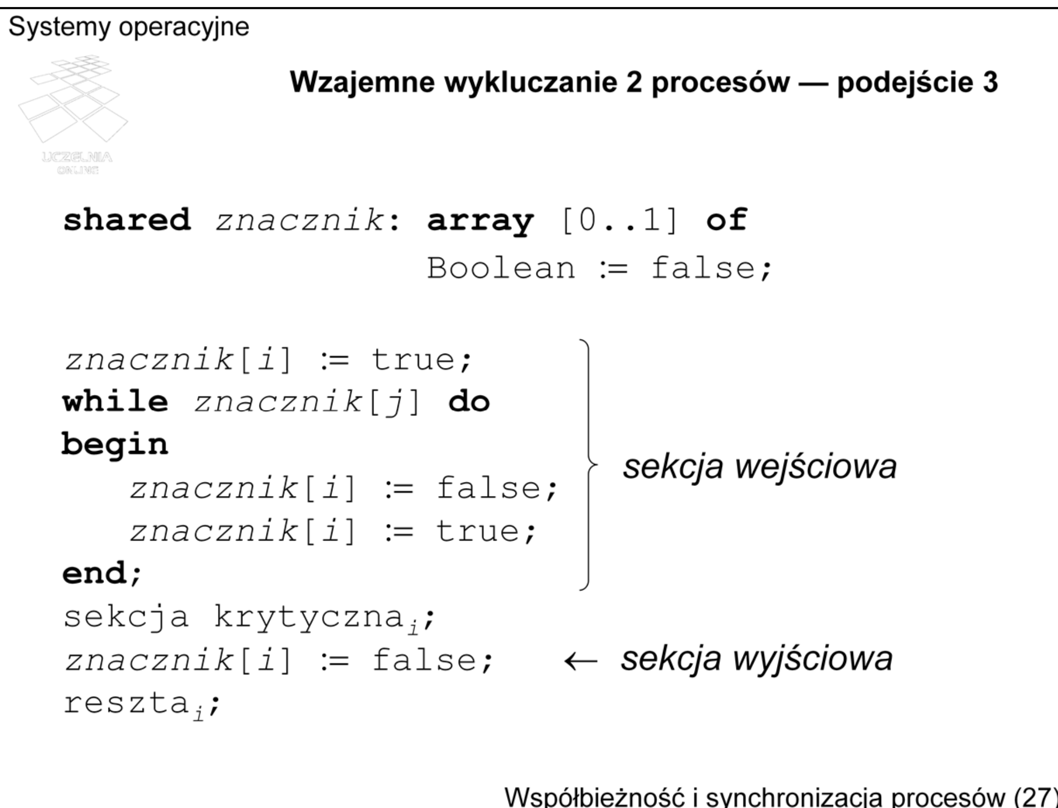
W rozwiązaniu tym proces sygnalizuje zamiar lub docelowo fakt wejścia do sekcji krytycznej, ustawiając *znacznik* na swojej pozycji na true. W celu stwierdzenia dostępności sekcji krytycznej sprawdza wartość znacznika na pozycji odpowiadającej rywalowi (dla P_i rywalem jest P_j i odwrotnie). Jeśli *znacznik* na pozycji rywala ma wartość false, proces przerywa pętlę `while` i tym samym wchodzi do sekcji krytycznej.

Własność bezpieczeństwa łatwo można wykazać metodą *nie wprost*. Zakładając, że dwa procesy mogą jednocześnie wykonywać sekcję krytyczną, każdy z nich musiał odczytać wartość false z pozycji tablicy *znacznik*, odpowiadającej rywalowi. Wcześniej jednak każdy z nich ustawił wartość true na swojej pozycji. Przeplot z punktu widzenia każdego z procesów musi zatem uwzględniać fakt, że nie została ustawiona wartość true na pozycji rywala. Dla P_i oznacza to, że podstawienie true pod *znacznik*[*j*] wykonało się po zakończeniu przez niego pętli `while`. Dla P_j sytuacja jest odwrotna, więc P_i musiałby wykonać pętlę po podstawieniu true pod *znacznik*[*j*].

Własność postępu nie jest spełniona, gdyż mogą nastąpić podstawienia true pod odpowiednie pozycje znacznika, czyli:

```
znacznik[i] := true;
znacznik[j] := true;
```

W takim stanie systemu oba procesy utkną w pętli `while` w swoich sekcjach wejściowych i żaden nie wejdzie do sekcji krytycznej. Stan taki będzie stabilny, tzn. nie zmieni się, jeśli nie nastąpi jakaś interwencja z zewnątrz (spoza zbioru procesów). Jest to przykład *zakleszczenia*.



Przedstawione podejście jest podobne do poprzedniego. Różnica polega na tym, że zamiast tylko testować *znacznik* na pozycji odpowiadającej procesowi rywalizującemu po ustawieniu wartości true na własnej pozycji, proces dodatkowo wykonuje kolejno dwa podstawienia — wartości false oraz true — na swojej pozycji w tablicy *znacznik*. Pomiędzy tymi podstawieniami może być pewna zwłoka czasowa. W ten sposób stwarza szansę rywalowi, że „wplecie" się z odczytem znacznika (w nagłówku pętli **while**) pomiędzy te dwa podstawienia, odczyta false i wejdzie do sekcji krytycznej. Przykładowy przeplot w najprostszym przypadku byłby następujący:

$\{P_i\}$ *znacznik*[*i*] := true;

$\{P_j\}$ *znacznik*[*j*] := true;

$\{P_i\}$ **while** *znacznik*[*j*] **do**

$\{P_i\}$ *znacznik*[*i*] := false;

$\{P_j\}$ **while** *znacznik*[*i*] **do ...** // *wyjście z pętli*

$\{P_j\}$ sekcja krytyczna$_j$

Nie ma tu zatem zakleszczenia, gdyż przy asynchroniczności przetwarzania cały czas istnieje potencjalna szansa, że jeden z procesów opuści sekcję wejściową i wejdzie do sekcji krytycznej. Z drugiej strony nie ma pewności, że tak się kiedyś stanie. Stan taki nie jest stabilny — może (ale nie musi) się zmienić — i nazywany jest *uwięzieniem* (ang. livelock). Pojęcie uwięzienia można kojarzyć z głodzeniem procesu. Głodzenie dotyczy jednak określonego procesu, którego obsługa — najczęściej ze względu na niski priorytet — jest odkładana na dalszy plan, przy czym w systemie ciągle jest rywal o wyższym priorytecie. Uwięzienie z kolei dotyczy ogółu rywalizujących procesów, można zatem powiedzieć, że jest to głodzenie wszystkich współpracujących procesów.

Systemy operacyjne

Wzajemne wykluczanie 2 procesów — podejście 4

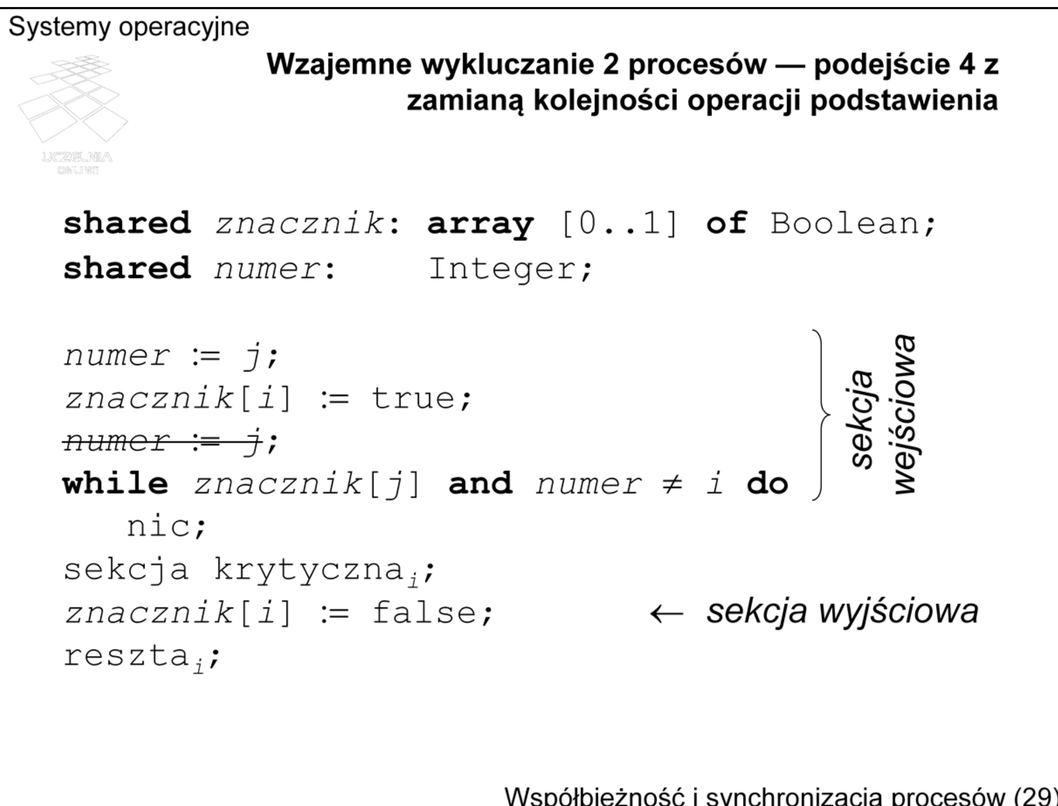
```
shared znacznik: array [0..1] of Boolean;
shared numer:    Integer;

znacznik[i] := true;                            } sekcja
numer := j;                                     } wejściowa
while znacznik[j] and numer ≠ i do              }
   nic;                                         }
sekcja krytyczna_i;
znacznik[i] := false;         ← sekcja wyjściowa
reszta_i;
```

Współbieżność i synchronizacja procesów (28)

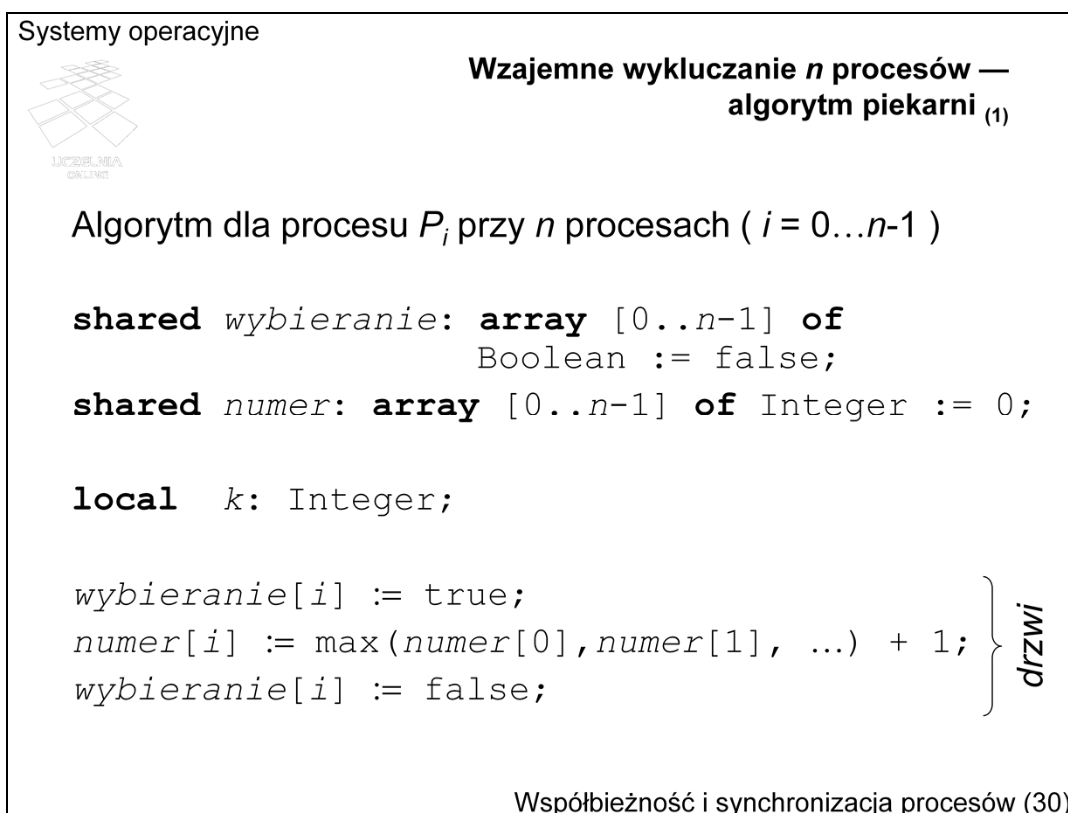
Rozwiązanie to jest wynikiem połączenia podejścia 1 i 2. Za pośrednictwem tablicy *znacznik* procesy informują się nawzajem o swoim stanie, a za pośrednictwem zmiennej *numer* rozstrzygają ewentualny konflikt. Jeśli zatem wartość w tablicy *znacznik* na pozycji odpowiadającej procesowi rywalizującemu jest ustawiona na false, to nie ubiega się on o sekcję krytyczną. Następuje wówczas opuszczenie pętli **while**, tym samym sekcji wejściowej i wejście do sekcji krytycznej. W przypadku, gdy dwa rywalizujące procesu ustawią wartość true na swoich pozycjach w tablicy *znacznik*, rozstrzygnięcie sporu zależy od wartości zmiennej *numer*. Ten z procesów, który później ustawi w niej numer rywala, ten musi poczekać, aż rywal wyjdzie z sekcji krytycznej.

Przedstawione rozwiązanie znane jest pod nazwą *algorytmu Petersona*. Algorytm ten można uogólnić na *n* procesów, stosując podejście „wieloetapowe". Na każdym etapie eliminowany jest jeden proces. Zmienna *numer* musi być wówczas tablicą *n*1-elementową, a tablica *znacznik* przechowuje numer etapu, na którym jest dany proces.



Rozwiązanie to jest wynikiem połączenia podejścia 1 i 2. Za pośrednictwem tablicy *znacznik* procesy informują się nawzajem o swoim stanie, a za pośrednictwem zmiennej *numer* rozstrzygają ewentualny konflikt. Jeśli zatem wartość w tablicy *znacznik* na pozycji odpowiadającej procesowi rywalizującemu jest ustawiona na false, to nie ubiega się on o sekcję krytyczną. Następuje wówczas opuszczenie pętli **while**, tym samym sekcji wejściowej i wejście do sekcji krytycznej. W przypadku, gdy dwa rywalizujące procesu ustawią wartość true na swoich pozycjach w tablicy *znacznik*, rozstrzygnięcie sporu zależy od wartości zmiennej *numer*. Ten z procesów, który później ustawi w niej numer rywala, ten musi poczekać, aż rywal wyjdzie z sekcji krytycznej.

Przedstawione rozwiązanie znane jest pod nazwą *algorytmu Petersona*. Algorytm ten można uogólnić na *n* procesów, stosując podejście „wieloetapowe”. Na każdym etapie eliminowany jest jeden proces. Zmienna *numer* musi być wówczas tablicą *n*1-elementową, a tablica *znacznik* przechowuje numer etapu, na którym jest dany proces.



Idea algorytm piekarni, zaproponowanego przez Lamporta, opiera się na przydzielaniu kolejnego numeru w kolejce oczekujących petentów i wpuszczaniu petenta z najniższym numerem. Algorytm stosowany jest w niektórych urzędach, bankach oraz przychodniach lekarskich.

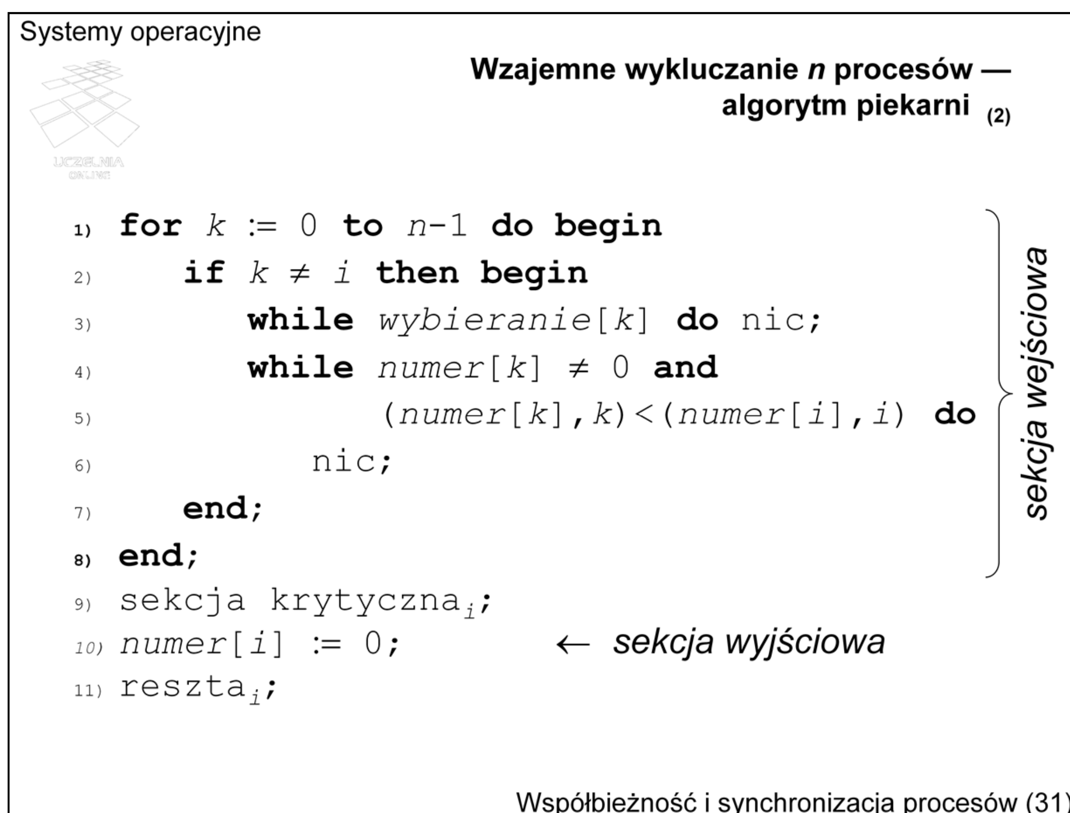
Etap algorytmu, w którym przydzielany jest numer, nazywany jest przejściem przez drzwi. Jest to część sekcji wejściowej. Proces, przechodząc przez drzwi, odczytuje numer wszystkich pozostałych, wybiera maksymalny z nich, zwiększa go o 1 i w ten sposób ustala swój własny numer. Proces, który wykonuje resztę, ma numer 0. Numery przydzielone procesom przechowywane są w tablicy współdzielonej *numer*. Pozycja *i*-ta tej tablicy jest zmienną wyjściową procesu P_i, a wszystkie pozostałe pozycje tablicy są dla niego zmiennymi wejściowymi. Wynika to z rozwinięcia operacji max, która w pełnej postaci mogłaby wyglądać następująco:

```
tmp := numer[0];
for k := 1 to n-1 do
   if numer[k] > tmp then tmp := numer[k];
numer[i] := tmp;
```

Dodatkowo można by jeszcze wykluczyć przypadek $k = i$ w pętli. Początkowo tablica *numer* wypełniona jest oczywiście wartościami 0.

W celu kontroli przydziału numeru każdy proces ustawia na swojej pozycji w tablicy *wybieranie* wartość true na czas ustalania swojego numeru. Początkowo tablica wypełniona jest oczywiście wartościami false.

Analizując szczegóły operacji na zmiennych współdzielonych, można zauważyć, że wszystkie zmienne są modyfikowane przez 1 proces, a czytane przez pozostałe. Są to tzw. współdzielone rejestry typu „jeden zapisujący wielu odczytujących” (ang. single-writer-multiple-readers shared registers).

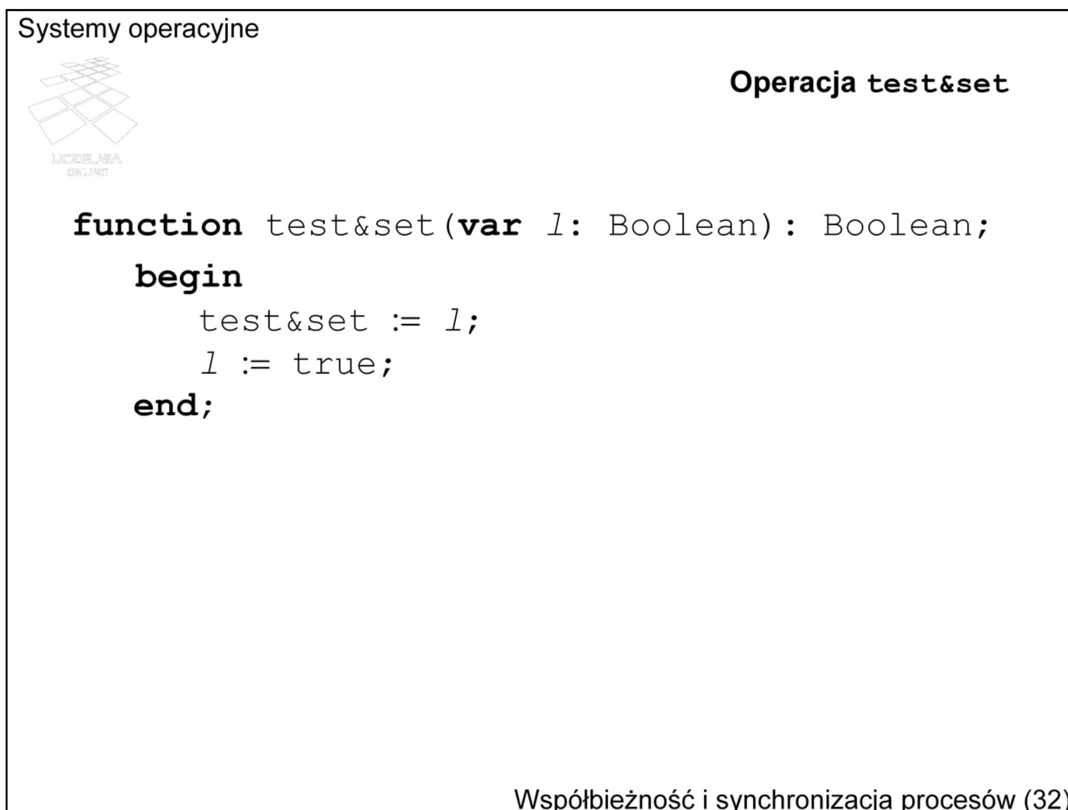


Po ustaleniu numeru zaczyna się właściwa sekcje wejściowa, w której dokonuje się rozstrzygnięcie odnośnie zajęcia sekcji krytycznej. W tym celu proces P_i sprawdza zamiary kolejnych rywali, analizując współdzielone tablice począwszy od pozycji 0 z pominięciem własnej pozycji. Jeśli analizowana pozycja odnosi się do procesu, który jest w trakcie wybierania numeru, wykonywana jest pętla oczekiwania na zakończenie wyboru (linia 3). Wykazanie konieczności takiego oczekiwania jest jednym z zadań ćwiczeniowych.

Po ewentualnym zakończeniu wyboru następuje sprawdzenie stanu potencjalnego rywala. Jeśli ma on przydzielony numer 0, to znaczy, że wykonuje resztę i nie jest zainteresowany sekcją krytyczną — pętla **`while`** w linii 4 kończy się i rozpoczyna się kolejna iteracji pętli **`for`**. To samo dzieje się, gdy numer rywala jest większy. Możliwy jest jednak przypadek, gdy dwa procesu uzyskają te same numery. Rozstrzygający jest wówczas numer procesu, który z założenia jest unikalny. Stąd porównanie: $(numer[k],k) < (numer[i],i) \equiv numer[k] < numer[i] \vee (numer[k] = numer[i] \wedge k < i)$, które można by również zapisać jako $(numer[k] \cdot n + k) < (numer[i] \cdot n + i)$.

Jeśli w ten sposób procesowi P_i uda się zakończyć pętlę **`for`**, to znaczy, że nie ma w systemie procesu ubiegającego się o sekcję krytyczną, którego numer byłby mniejszy (lub równy przy mniejszym identyfikatorze) od P_i i możne on opuścić sekcję wejściową, zajmując tym samym sekcję krytyczną.

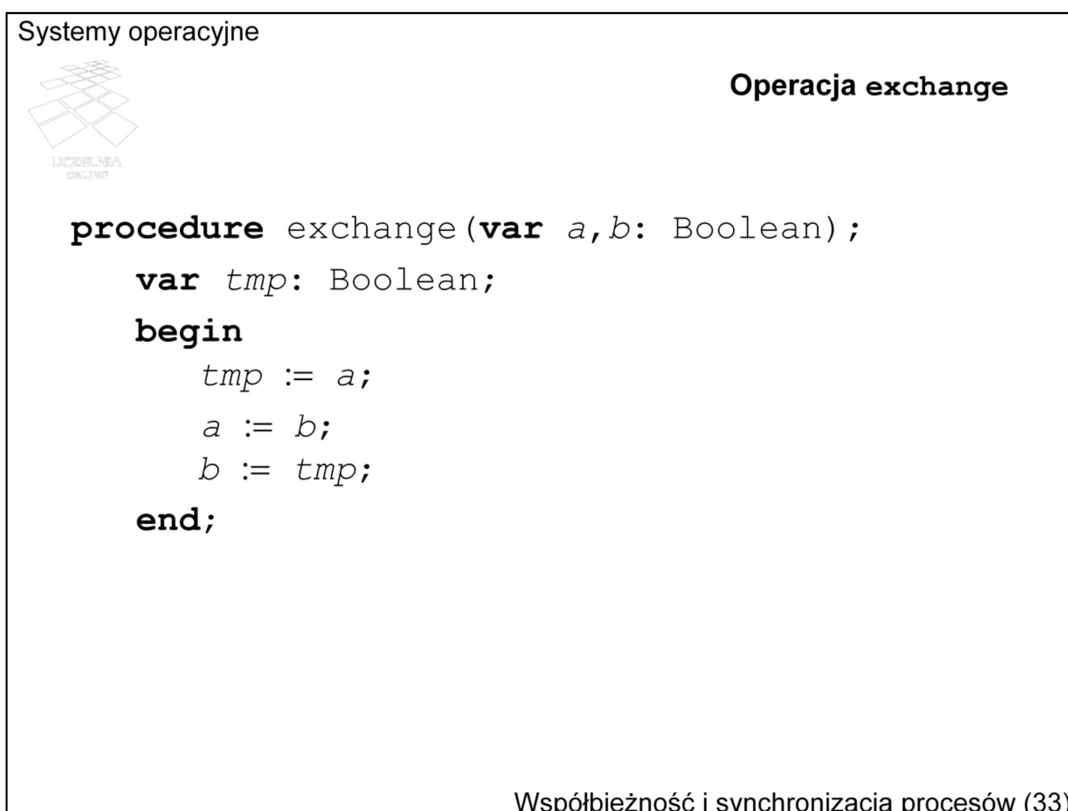
Po wyjściu z sekcji krytycznej, proces sygnalizuje brak zainteresowania rywalizacją, wpisując wartość 0 jako swój numer.



Atomowo wykonywana operacja **test&set** polega na odczytaniu dotychczasowej wartości zmiennej logicznej (w praktyce jakiegoś bitu), a następnie ustawieniu wartości tej zmiennej na true. Jeśli wartość była już true, wykonanie operacji niczego nie zmieni.

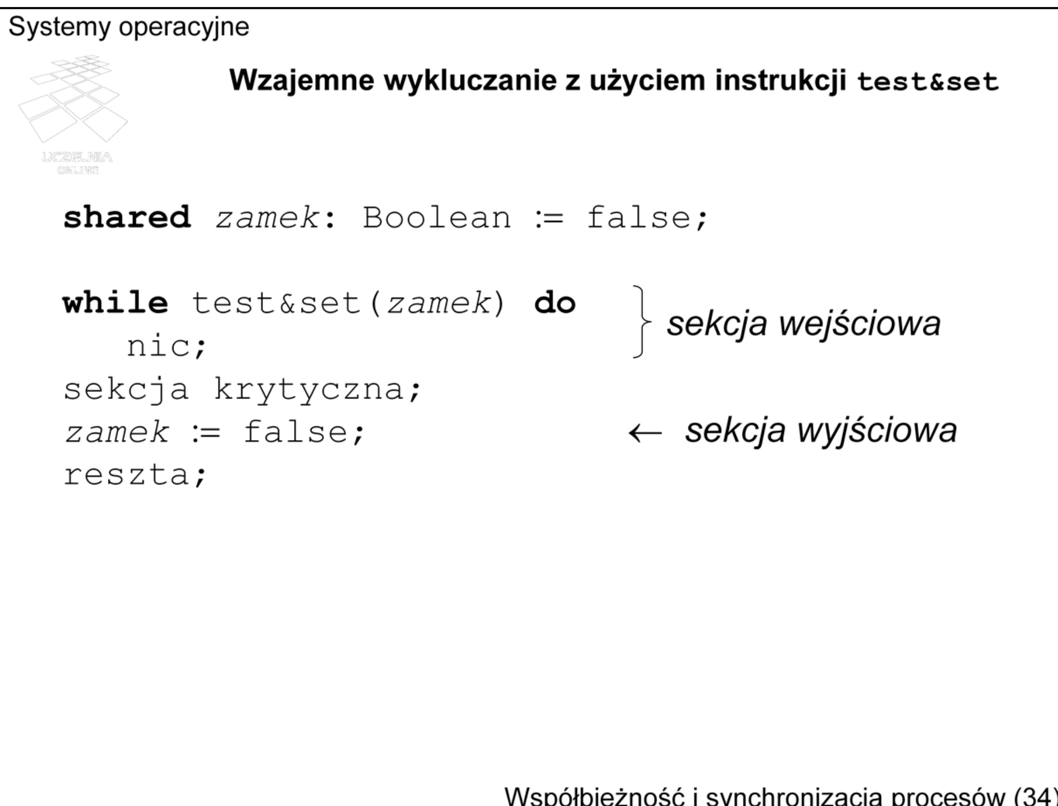
W systemie jednoprocesorowym efekt atomowości (niepodzielności) można uzyskać poprzez zablokowanie przerwań na czasy wykonywania operacji. W implementacji musiałyby się zatem znaleźć rozkazy blokowania przerwań na początku i odblokowania na końcu, właściwe dla danej architektury.

W architekturze IA-32 (Intel) tego typu operacje realizowane są na poziomie maszynowym przez rozkazy: **bts**, **btr**, **btc**, poprzedzone ewentualnie prefiksem `lock`.



Atomowo wykonywana operacja **`exchange`** polega na zamianie wartości dwóch zmiennych logicznych. Podobnie, jak w przypadku **`test&set`**, w systemie jednoprocesorowym efekt atomowości operacji można uzyskać poprzez zablokowanie przerwań na czas jej wykonywania.

W architekturze IA-32 (Intel) operacja **`exchange`** realizowana jest na poziomie maszynowym przez rozkaz: **`xchg`**, dotyczy jednak nie bitów a zawartości całych rejestrów. Pewnym ograniczeniem jest fakt, że jeden z operandów musi być w rejestrze procesora, ale nie przeszkadza to np. w zastosowaniu tego. rozkazu do rozwiązania problemu wzajemnego wykluczania. Jeśli któryś z operandów rozkazu **`xchg`** jest w pamięci, następuje blokada magistrali niezależnie od użycia prefiksu `lock`.



Rozwiązanie polega na wykorzystaniu współdzielonej zmiennej *zamek*, której wartość true oznacza zajętość sekcji krytycznej.

Na zmiennej wykonywana jest operacja **`test&set`**, która ustawia true (być może ponownie) i zwraca poprzednią wartość zmiennej. Jeśli poprzednia wartość była false, to znaczy, że w sekcji krytycznej nie było żadnego procesu i może ona zostać zajęta. W przeciwnym przypadku nie nastąpiła zmiana wartości zmiennej *zamek* (było true i jest nadal true), a jedynie ustalenie wartości tej zmiennej. Z wartości tej wynika oczywiście, że sekcja krytyczna jest zajęta i należy dalej kontynuować wykonywanie pętli **`while`**.

Taki algorytm może być stosowany dla dowolnej liczby współpracujących procesów. Nie gwarantuje on jednak ograniczonego czekania, gdyż nie ma żadnej pewności, że konkretny proces będzie mógł wykonać na swoje potrzeby operację **`test&set`** akurat wówczas, gdy zmienna zamek będzie miała wartość false.

Systemy operacyjne

Wzajemne wykluczanie z użyciem instrukcji exchange

```
shared zamek: Boolean := false;
local klucz: Boolean;

klucz := true;                         } sekcja
repeat exchange(zamek, klucz);         } wejściowa
until klucz = false;
sekcja krytyczna;
zamek := false;          ← sekcja wyjściowa
reszta;
```

Współbieżność i synchronizacja procesów (35)

Rozwiązanie z użyciem procedury **`exchange`** jest bardzo podobne. Wykorzystywana jest jednak dodatkowa zmienna lokalna *klucz*. Wartość zmiennej *klucz* zamieniana jest z wartością zmiennej *zamek*. Ponieważ na początku sekcji wejściowej pod *klucz* podstawiana jest wartość true, ta wartość trafia następnie do zmiennej *zamek*. Z zamka do klucza trafia z kolei dotychczasowa wartość zamka. Jeśli wartość ta jest false, można przerwać pętlę **`repeat-until`** i wejść do sekcji krytycznej. Jeśli wartością tą jest true, wykonanie operacji **`exchange`** niczego nie zmieni, a za pośrednictwem klucza proces dowie się, że sekcja krytyczna jest niedostępna.

Z realizacją wzajemnego wykluczania w taki sposób wiążą się te same kwestie, które poruszono przy rozwiązaniu z użyciem **`test&set`**.