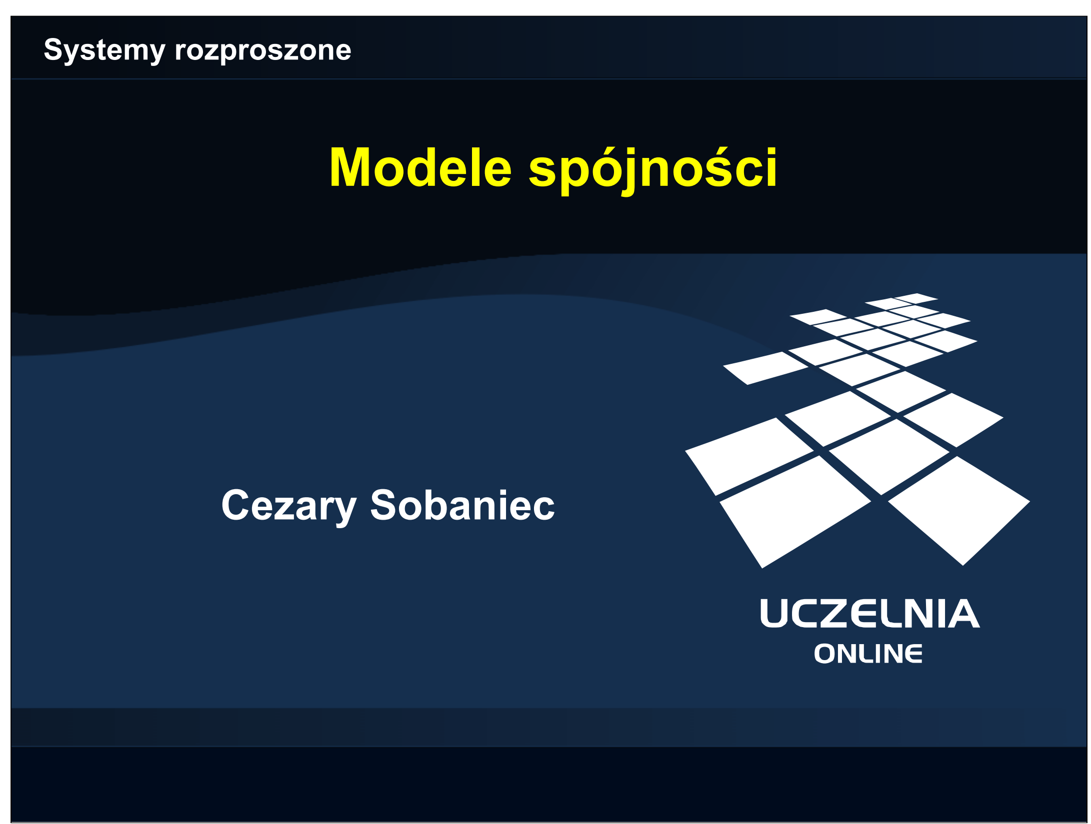

Systemy rozproszone
Modele spójności
Cezary Sobaniec
UCZELNIA
ONLINE

Model spójności

Model spójności określa gwarancje dotyczące spójności replik, dawane aplikacji (równoległej) przez system

- W jaki sposób definiować model spójności?
- W jaki sposób określić gwarancje dla aplikacji?
- Kiedy i w jaki sposób egzekwować te gwarancje?

Problem spójności danych istnieje w systemach stosujących zwielokrotnianie. Jeżeli dane są modyfikowane, to zmiana wprowadzona na pojedynczym węźle musi być przekazana do pozostałych, najlepiej zanim zlecony zostanie odczyt tych danych. W systemie rozproszonym wymiana danych na najniższym poziomie realizowana jest jednak zawsze z wykorzystaniem przesyłania komunikatów, które wprowadza opóźnienia. Propagacja modyfikacji nie może więc następować natychmiastowo, co może być wychwycone przez użytkowników obserwujących niespójności pomiędzy zawartościami poszczególnych replik. Model spójności określa „reguły gry" jakie będą stosowane przez system podczas aktualizacji danych. Są to pewne gwarancje, których udziela system w odniesieniu do uporządkowania operacji realizowanych w systemie rozproszonym. Aplikacje, dla których gwarancje te są wystarczające będą działały poprawnie pomimo istnienia zwielokrotniania, opóźnień i czasowych niespójności danych. Problem otwartym jest sposób wyrażenia tych gwarancji, czy innymi słowy tej „umowy" pomiędzy aplikacją a systemem. Mając zdefiniowany model spójności (reguły) należy skonstruować *protokół spójności*, który te reguły zrealizuje w systemie rozproszonym. Protokół określa kiedy i w jaki sposób należy ingerować w przetwarzanie, tak aby zdefiniowane na poziomie modelu własności były faktycznie zachowane.

Spójność ścisła

Spójność ścisła (ang. *strict consistency*)

- każdy odczyt zmiennej *x* zwraca wartość odpowiadającą wynikowi *ostatnio* wykonanej operacji zapisu

Systemy rozproszone

- niejednoznaczność określenia *ostatni* (brak globalnego zegara)
- duży koszt realizacji spójności ścisłej

W systemach scentralizowanych każdy odczyt zwraca wynik będący odzwierciedleniem *ostatniego* zapisu, co jest określane jako spójność ścisła (ang. *strict consistency*). W systemie rozproszonym pojęcie *ostatni* jest niejednoznaczne, ponieważ nie ma jednego, globalnego zegara synchronizującego pracę poszczególnych węzłów. W efekcie może się zdarzyć, że operacja odczytu zwróci wynik sprzed ostatniego zapisu. Realizacja modelu przetwarzania charakterystycznego dla systemów scentralizowanych jest bardzo kosztowna, a niekiedy niemożliwa do zrealizowania. Najprostsza realizacja spójności ścisłej polega na zastosowaniu pojedynczego, centralnego serwera, który przetwarza wszystkie odwołania do danych. Realizacja taka jest oczywiście wysoce nieefektywna i nie korzysta z zalet jakie potencjalnie oferuje zwielokrotnianie. Stąd poszukuje się innych modeli spójności, które będą oferowały słabsze gwarancje, ale które będą mogły być dużo bardziej efektywnie implementowane.

Modele spójności replik

Modele spójności nastawione na dane

- modele spójności przy dostępie ogólnym
  uspójnianie danych przy każdej modyfikacji
- modele spójności przy dostępie synchronizowanym
  uspójnianie danych tylko podczas wykonywania jawnych operacji synchronizujących

Modele spójności nastawione na klienta

- uwzględnienie mobilności klienta
  → zobacz następny wykład

Generalnie modele spójności można podzielić na dwie podstawowe klasy: modele spójności nastawione na dane (ang. *data-centric consistency models*) i modele spójności nastawione na klienta (*client-centric consistency models*). Modele spójności nastawione na klienta będą prezentowane w ramach następnego wykładu.

Modele spójności nastawione na dane można podzielić dalej na dwie grupy: modele spójności przy dostępie ogólnym (ang. *general access consistency models*) i modele spójności przy dostępie synchronizowanym (ang. *synchronization access consistency models*). Modele spójności przy dostępie ogólnym są bardziej „przyjazne” dla programisty, ponieważ program korzystający z tych modeli wygląda tak samo jak program pisany dla systemu scentralizowanego. Konsekwencją tego jest konieczność uspójniania danych podczas wykonywania każdej operacji modyfikującej. Rezygnację z pełnej przezroczystości dostępu do danych zastosowano w modelach przy dostępie synchronizowanym. W tej grupie modeli spójności występują specjalne operacje synchronizujące, dodawane do kodu programu, których wykonanie powoduje przeprowadzenie synchronizacji danych pomiędzy serwerami. Zwykłe dostępy do danych nie muszą powodować komunikacji z innymi serwerami. Podejście takie jest racjonalne w przypadku przeprowadzania przez aplikację serii aktualizacji, być może dotyczących tych samych zmiennych. Wyniki przeprowadzonych modyfikacji mogą w takim przypadku być przesłane łącznie.

Modele spójności nastawione na dane były rozwijane głównie w ramach badań nad rozproszoną pamięcią dzieloną (ang. *Distributed Shared Memory*).

Modele spójności przy dostępie ogólnym

- **Spójność atomowa** (*ang. atomic consistency*) **liniowość** (ang. *linearizability*)
- **Spójność sekwencyjna** (ang. *sequential consistency*)
- **Spójność przyczynowa** (ang. *causal consistency*)
- **Spójność PRAM** (ang. *pipelined RAM consistency*)
- **Spójność podręczna** (ang. *cache consistency*) **koherencja** (ang. *coherence*)
- **Spójność procesorowa** (ang. *processor consistency*)

Slajd powyższy wymienia główne modele spójności nastawione na dane przy dostępie ogólnym. Kolejność nie jest przypadkowa i (z grubsza) odzwierciedla siłę tych modeli. Spójność atomowa odpowiada działaniu systemu scentralizowanego – jest to więc najsilniejszy model, ale jednocześnie model wymuszający najmniej efektywną implementację. Spójność atomowa bywa też nazywana liniowością. Spójność sekwencyjna nadal jest bliska przetwarzaniu w systemie scentralizowanym, ale dopuszcza już pewne nakładanie się operacji w systemie, co potencjalnie może być wykorzystane do bardziej efektywnej pracy protokołu spójności. Spójność przyczynowa zachowuje porządek przyczynowy. Spójność PRAM porządkuje operacje zlecane przez poszczególne węzły. Spójność podręczna (zwana również koherencją) porządkuje operacje wykonywane na poszczególnych zmiennych. W końcu spójność procesorowa łączy w sobie własności spójności PRAM i podręcznej.

Modele spójności przy dostępie synchronizowanym

- **Spójność słaba** (ang. *weak consistency*)
- **Spójność zwalniania** (ang. *release consistency*)
- **Spójność wejścia** (ang. *entry consistency*)
- **Spójność zakresu** (ang. *scope consistency*)

Modele spójności przy dostępie ogólnym są wygodne w użyciu, gdyż nie wymagają modyfikacji przy przenoszeniu z systemu nie stosującego zwielokrotniania do systemu stosującego zwielokrotnianie. Niestety pomijając spójność atomową i sekwencyjną, pozostałe modele oferują własności, które często są niewystarczające do poprawnego działania aplikacji. Z drugiej strony realizacja nawet modelu sekwencyjnego zawsze obarczona jest bardzo dużym kosztem efektywnościowym. Poszukując innych rozwiązań zaproponowano modele o dostępie synchronizowanym, które są pewnego rodzaju kompromisem pomiędzy wygodą programisty a złożonością samego modelu. Z jednej bowiem strony programista jest zmuszany do uzupełnienia kodu programu o dodatkowe instrukcje, ale z drugiej strony te dodatkowe instrukcje „podpowiadają" systemowi kiedy jakie dane aplikacja będzie wykorzystywać, a więc kiedy należy je uspójniać. W efekcie protokoły spójności dla modeli przy dostępie synchronizowanym oferują efektywnie modele spójności zgodne z modelem sekwencyjnym (lub niekiedy atomowym), a jednocześnie charakteryzują się dużą efektywnością pozwalającą na praktyczne wdrożenie koncepcji DSM.

Systemy rozproszone

Podstawowe założenia

- W skład systemu DSM wchodzą:
  - zbiór sekwencyjnych procesów $P = \{p_1, p_2, \ldots, p_n\}$
  - zbiór współdzielonych zmiennych $X = \{x_1, x_2, \ldots\}$
- Każdy proces ma własną replikę całego zbioru X
- Proces p_i może realizować na zmiennej $x \in X$ operacje:
  - zapisu wartości v, oznaczane jako $w_i(x)v$
  - odczytu wartości v, oznaczane jako $r_i(x)v$
- Realizacja operacji przebiega w dwóch fazach:
  - żądanie operacji (ang. *operation issue*)
  - wykonanie operacji (ang. *operation execution*)

Modele spójności (7)

Rozważamy system składający się z n procesów, każdy pracujący na oddzielnym węźle. Procesy odwołują się do pamięci zorganizowanej w postaci zmiennych, współdzielonych przez wszystkie procesy. Dla uproszczenia rozważań zakładamy, że w systemie stosowana jest pełna replikacja, co oznacza, że każdy węzeł posiada pełną kopię całego zbioru zmiennych współdzielonych. Założenie to nie zmniejsza ogólności rozważań, bo nie wpływa na definicję modelu spójności, a jedynie na organizację rozproszonej pamięci.

Operacje realizowane na zmiennych współdzielonych są bądź odczytami bądź zapisami. Odczyt wartości v ze zmiennej x realizowany przez proces p_i oznaczany będzie jako $r_i(x)v$. Zapis wartości v do zmiennej x realizowany przez proces p_i oznaczany będzie jako $w_i(x)v$. Jeżeli kontekst stosowania wymienionych oznaczeń będzie jednoznaczny, to indeksy identyfikujące procesy zostaną pominięte.

Operacja odczytu bądź zapisu nie jest operacją atomową. Jest to szczególnie wyraziste w systemie rozproszonym, gdzie wykonanie operacji może oznaczać potrzebę komunikacji. W związku z tym w niektórych przypadkach będziemy rozważać jawne rozpoczęcie wykonywania operacji (zgłoszenie jej przez proces) i zakończenie wykonywania operacji (zwrócenie wyniku do procesu).

UCZELNIA ONLINE

Oznaczenia

$w_i(x)v$	zapis wartości v do zmiennej x wykonany przez proces p_i
$r_i(x)v$	odczyt wartości v ze zmiennej x wykonany przez proces p_i
O	zbiór wszystkich operacji w systemie
O_i	zbiór operacji procesu p_i (żądanych przez p_i)
OW	zbiór wszystkich operacji zapisu w systemie
$O\|x$	zbiór wszystkich operacji na zmiennej x
$\rightarrow_i$	lokalny porządek operacji procesu p_i
$\rightarrow$	porządek przyczynowy
$\mapsto_i$	uszeregowanie operacje postrzegane są przez proces p_i

W definicjach modeli spójności zostaną użyte wymienione na slajdzie oznaczenia. Zapisy i odczyty zostały przedstawione na poprzednim slajdzie. Zbiór wszystkich operacji w systemie, zarówno zapisy jak i odczyty, będzie oznaczany przez O. Zbiór wszystkich operacji procesu p_i będzie oznaczany przez O_i. Z punktu widzenia zarządzania spójnością najważniejsze są operacje zapisu *(OW),* bo to one muszą być powielone na wszystkich serwerach. Operacje wykonane na zmiennej x będą oznaczane jako $O|x$.

Procesy zlecają wykonywanie operacji sekwencyjnie. Liniowy porządek operacji zlecanych przez proces p_i oznaczany będzie jako $\rightarrow_i$. Porządek przyczynowy operacji wykonywanych w systemie będzie oznaczany zwykłą strzałką $\rightarrow$.

Symbol $\mapsto_i$ będzie oznaczać uporządkowanie operacji postrzeganych przez proces p_i. Proces „widzi” swoje operacje, ale i również operacje innych procesów jeśli np. czyta wyniki zapisów tych procesów. Odczyt wartości zapisanej przez inny proces oznacza, że przed tym odczytem musi być zaszeregowany zapis tego procesu. Uporządkowanie operacji (uszeregowanie) postrzeganych przez poszczególne procesy może być różne, w zależności od ograniczeń nakładanych przez model spójności. Uszeregowanie określa porządek wykonywania operacji na zmiennych w konkretnym węźle.

Definicja porządku przyczynowego

$$(1) \quad \forall_{o1,o2 \in O_i} (o1 \rightarrow_i o2 \Rightarrow o1 \rightarrow o2)$$

$$(2) \quad \forall_{x \in X} w(x)v \rightarrow r(x)v$$

$$(3) \quad \forall_{o1,o2,o \in H} \left[(o1 \rightarrow o \wedge o \rightarrow o2) \Rightarrow o1 \rightarrow o2 \right]$$

Model spójności przyczynowej odwołuje się do definicji porządku przyczynowego. Porządek przyczynowy jest z kolei definiowany wymienionymi 3 warunkami. Po pierwsze: lokalne uporządkowanie operacji wykonywanych przez poszczególne procesy jest zachowywane przez porządek przyczynowy. Po drugie: jeżeli w jednym procesie następuje odczyt *r*(*x*)*v*, to oznacza, że zapis, który wprowadził do zmiennej *x* wartość *v*, czyli *w*(*x*)*v* musi poprzedzać ten odczyt w porządku przyczynowym. Co ważne: nie jest istotne na jakim węźle wykonywane są operacje odczytu i zapisu, a więc mogą być realizowane na różnych węzłach. Odczyt wartości zapisanej na innym węźle jest możliwy, jeżeli protokół aktualizacji danych prześle w międzyczasie komunikat aktualizujący. Ostatni warunek porządku przyczynowego jest domknięciem przechodnim relacji, mówiącym, że jeżeli operacja *o1* poprzedza przyczynowo jakąś operację *o*, a operacja *o* poprzedza przyczynowo *o2*, to wtedy *o1* poprzedza przyczynowo *o2*.

Definicja uszeregowania legalnego

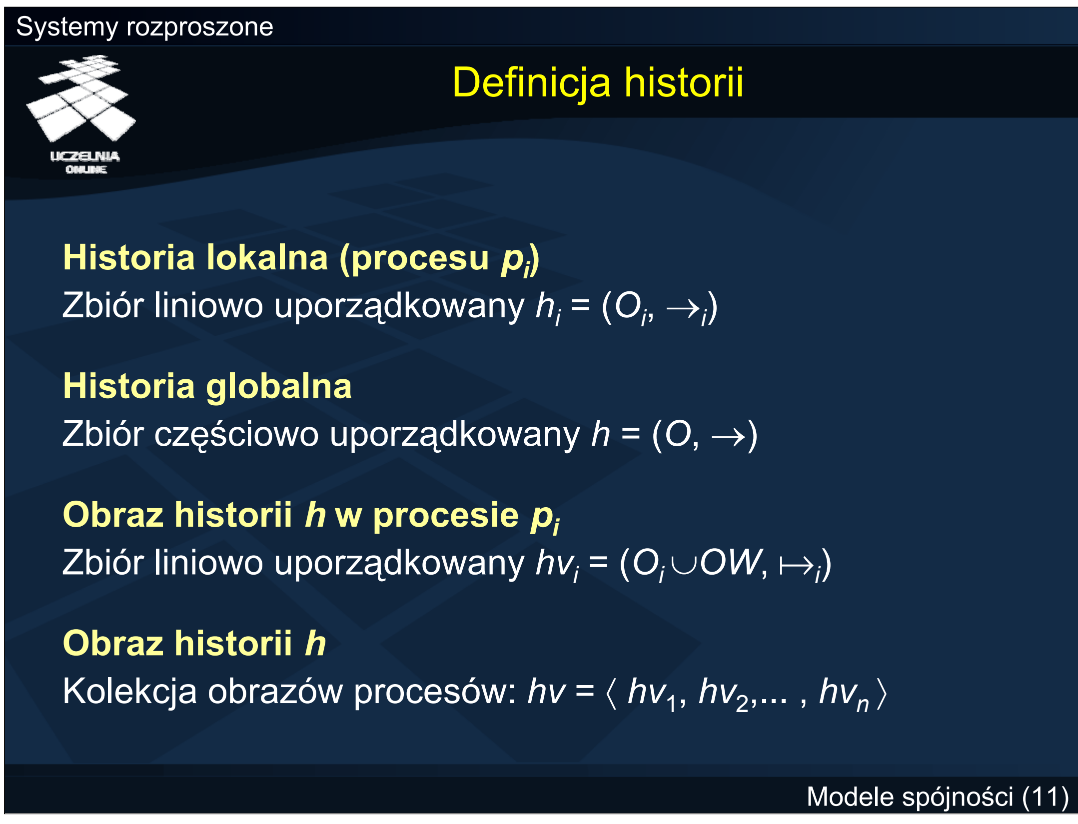
- Uszeregowanie $\mapsto_i$ jest legalne $\Leftrightarrow$

$$\underset{\substack{w(x)v \in OW \\ r(x)v \in O_i}}{\forall} \left(\begin{array}{l} w(x)v \mapsto_i r(x)v \\ \wedge \underset{o(x)u \in O_i \cup OW}{\nexists} \left[u \neq v \wedge w(x)v \mapsto_i o(x)u \mapsto_i r(x)v \right] \end{array} \right)$$

UWAGA
W celu uproszczenia **definicji** zakłada się, że każda operacja zapisu danej zmiennej zapisuje unikalną wartość, co umożliwia identyfikowanie operacji zapisu poprzez tą wartość

Nie każde uszeregowanie operacji wykonanych w systemie rozproszonym jest możliwe do zrealizowania w praktyce. Będziemy rozważać tylko **uszeregowania legalne**. Legalność oznacza, że odczyt wartości *v* ze zmiennej *x* jest możliwy tylko wtedy, gdy nie było żadnego innego zapisu do zmiennej *x* wartości różnej od *v*, który znajdowałby się w uszeregowaniu procesu czytającego po zapisie wartości *v* a przed odczytem wartości *v*. Co więcej: nie może być miedzy zapisem wartości *v* a jej odczytem również operacji odczytu wartości innej niż *v*. Założenia te są dość intuicyjne jeżeli weźmiemy pod uwagę działanie pamięci lokalnej w węźle. Każda zmienna ma w pamięci swoją pojedynczą lokalizację i odczyt zwraca zawsze ostatnio zapisaną wartość.

Dla uproszczenia rozważań (nie redukując jednak ogólności rozważań) będziemy zakładać, że każdy zapis i odczyt dotyczy unikalnej wartości *v*. Innymi słowy, wartość zapisywana będzie jednoznacznie identyfikować konkretną operację zapisu.



Analiza modeli spójności odwołuje się do pojęcia historii. Historia reprezentuje zrealizowane wcześniej przetwarzanie, a więc uporządkowany zbiór wszystkich operacji, które zostały wykonane.

Przetwarzanie każdego z procesów jest reprezentowane jego lokalną **historią** h_i, która jest liniowo uporządkowanym zbiorem operacji zleconych przez ten proces. Relacją porządkującą jest lokalny porządek zleceń wykonania kolejnych operacji.

Złożenie historii wszystkich procesów daje **historię globalną**, gdzie relacją porządkującą jest relacja zależności przyczynowej (zawierająca w sobie lokalne porządki poszczególnych procesów). Zbiór ten jest częściowo uporządkowany, ponieważ niektóre zdarzenia realizowane są współbieżnie.

Każdy z procesów postrzega w jakiś sposób zapisy, które zostały zgłoszone na innych węzłach. Uszeregowanie tych zapisów daje **obraz historii w procesie**. Obraz historii jest uporządkowany liniowo relacją lokalnego uszeregowania, które musi być legalne.

Złożenie wszystkich lokalnych obrazów historii daje **obraz historii**, który jest kolekcją zbiorów liniowo uporządkowanych.

Spójność sekwencyjna

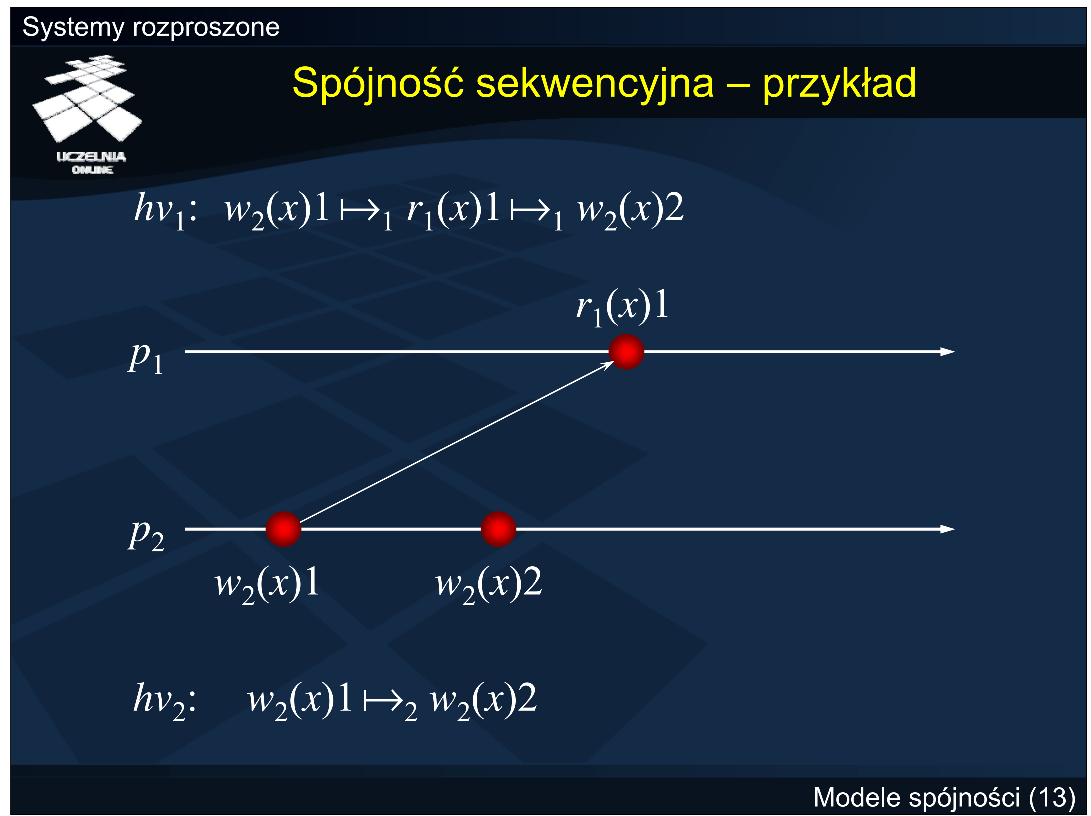
Obraz *hv* historii *h* musi spełniać następujące warunki:

$$\underset{o1,o2\in O_i\cup OW}{\forall}\left(\left(\underset{j=1..n}{\exists} o1 \rightarrow_j o2\right) \Rightarrow o1 \mapsto_i o2\right)$$

$$\underset{w1,w2\in OW}{\forall}\left(\underset{i=1..n}{\forall} w1 \mapsto_i w2 \vee \underset{i=1..n}{\forall} w2 \mapsto_i w1\right)$$

Przetwarzanie jest zgodne z modelem spójności sekwencyjnej, jeżeli obraz historii w każdym z procesów spełnia oba zaprezentowane warunki. Pierwszy warunek mówi o tym, że lokalne uporządkowanie operacji powinno być zachowane w obrazie przetwarzania w każdym procesie. Dotyczy to oczywiście operacji wykonywanych w danym węźle oraz wszystkich zapisów, które zostały zgłoszone. Odczyty wykonywane na innych węzłach nie są brane pod uwagę, ponieważ są one realizowane jedynie lokalnie. Zapisy natomiast muszą być propagowane do wszystkich węzłów.

Drugi warunek mówi o tym, że wszystkie zapisy w systemie muszą być globalnie uszeregowane. Innymi słowy: istnieje jedna, globalnie ustalona kolejność wykonywania *wszystkich* zapisów, które zostały zlecone w systemie. Zostało to zdefiniowane w ten sposób, że dla każdej pary operacji zapisu *w1* i *w2* wszystkie węzły albo realizują najpierw *w1* a później *w2*, albo najpierw *w2* a później *w1*. W skrócie można więc powiedzieć, że spójność sekwencyjna wymusza globalne uporządkowanie zapisów.



Przykład przedstawia przetwarzanie w systemie, który gwarantuje spójność sekwencyjną. Ostateczna postać obrazów historii w procesach p_1 i p_2 spełnia oba warunki modelu. W szczególności warto zwrócić uwagę na globalne uporządkowanie operacji zapisu: oba procesy widzą najpierw zapis wartości 1 do zmiennej *x*, a później wartości 2. W obrazach historii w procesie p_1 występuje dodatkowo odczyt, który nie występuje w obrazie historii procesu p_2. Komunikat przesyłany z procesu p_2 do p_1 oznacza aktualizację danych.

Algorytm *fast-read*

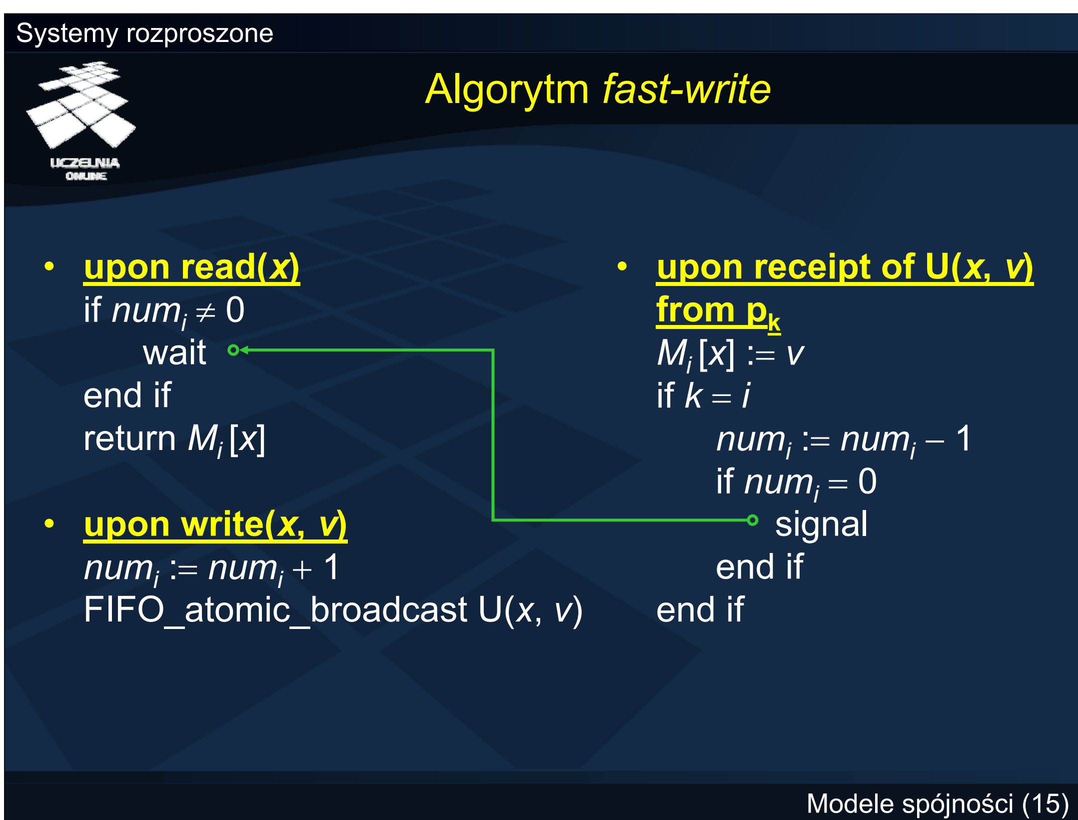
- Protokół spójności dla modelu sekwencyjnego
- Algorytm dla procesu p_i

- **upon read(x)**
  return $M_i[x]$

- **upon write(x, v)**
  atomic_broadcast U(x, v)
  wait

- **upon receipt of U(x, v) from p_k**
  $M_i[x] := v$
  if $k = i$
  signal
  end if

Realizacja modelu spójności wymaga zaprojektowania odpowiedniego protokołu spójności. Przytoczony na przykładzie algorytm jest jednym z możliwych protokołów gwarantujących zachowanie spójności sekwencyjnej. Jak sama nazwa wskazuje preferuje on odczyty. Preferencja przejawia się tym, że odczyt nigdy nie jest blokowany – od razu zwracana jest lokalnie przechowywana wartość zmiennej.
Zapis algorytmu ma postać procedur obsługi zdarzeń. Zawartość pamięci reprezentowana jest w algorytmie jako tablica *M*, zwielokrotniona w postaci lokalnych tablic M_i w procesie p_i.
Odczyt zwraca po prostu wartość przechowywaną na odpowiedniej pozycji w tablicy *M*. Zlecenie zapisu jest rozgłaszane (lub rozsyłane) do wszystkich procesów. Rozgłaszanie jest rozgłaszaniem niepodzielnym (ang. *atomic broadcast*), co oznacza, że komunikat musi dotrzeć do wszystkich i to w tej samej kolejności. Warstwa komunikacji grupowej zajmie się więc w tym przypadku zapewnieniem uporządkowania operacji zapisu. Jest to wygodna (choć nie zawsze najbardziej efektywna) metoda implementacji protokołu spójności. Właściwa operacja modyfikacji zmiennej realizowana jest w procedurze obsługi odbioru wiadomości aktualizacyjnej. Dotyczy to również węzła, który inicjuje zapis. Jest to ważne, ponieważ zapisy mają być wszędzie realizowane w tej samej kolejności. Zapis powoduje wstrzymanie przetwarzania procesu, który go zleca, do czasu zakończenia niepodzielnego rozgłaszania. Wykonanie bieżącego zapisu może być więc poprzedzone wcześniejszym obsłużeniem zapisów zleconych na innych węzłach – o kolejności zadecyduje warstwa komunikacyjna. W procedurze obsługi odbioru wiadomości aktualizacyjnej, jeżeli komunikat dociera do procesu, który inicjował rozgłoszenie, następuje obudzenie procesu zlecającego (funkcja *signal*). Proces był zawieszony w procedurze zapisu (funkcją *wait*).



Protokół *fast-write* dla spójności sekwencyjnej optymalizuje operacje zapisu wykonywane w systemie. Ich wykonanie nigdy nie jest blokujące, co pozwala na natychmiastową kontynuację obliczeń, być może powodującą wykonanie kolejnego zapisu. Blokowaniu natomiast podlega operacja odczytu. Sprawdza ona czy licznik *num*, inkrementowany przy każdym zapisie osiągnął wartość 0, co oznacza poprawne zakończenie wszystkich operacji zapisu zleconych lokalnie. Proces może zlecić wiele zapisów, jeden za drugim, zwiększając w ten sposób zmienną *num*. Warstwa komunikacyjna będzie próbowała dostarczyć odpowiednie komunikaty do wszystkich serwerów, a proces w tym czasie może realizować dalsze zadania. Odczyt wymaga zakończenia wykonywania wszystkich dotychczasowych operacji zapisu zleconych lokalnie. Bez tego oczekiwania mogłoby dojść do wygenerowania uszeregowania nielegalnego, np.: w(x)5 → r(x)0. Obsługa zapisu wymaga w tym przypadku rozgłaszania niepodzielnego zachowującego porządek FIFO, gdyż w przeciwnym wypadku mogłoby dość do przestawienia kolejności zapisów zleconych przez pojedynczy proces, a więc doszłoby do naruszenia lokalnego uporządkowania. Protokół *fast-read* nie wymagał stosowania niepodzielnego rozgłaszania FIFO, ponieważ przed zleceniem kolejnego zapisu poprzedni musiał być zakończony, co wymuszało zachowanie lokalnego porządku.

Przedstawione protokoły dla spójności sekwencyjnej pokazują, że jej realizacja jest kosztowna. Koszt objawia się w tym przypadku opóźnieniami, które wynikają z koniecznej synchronizacji procesów. W zależności od potrzeb można optymalizować operacje odczytu bądź zapisu, ale zawsze takie optymalizacje odbywają się kosztem drugiej operacji.

Spójność atomowa

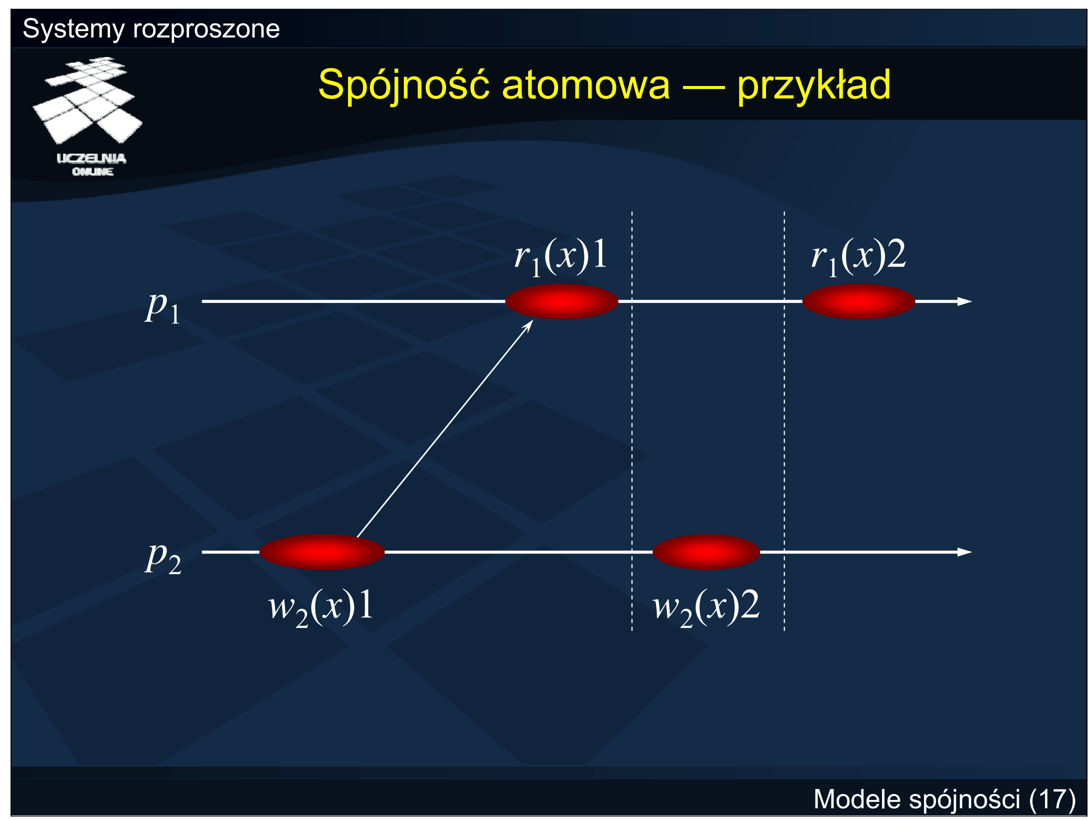
Obraz *hv* historii *h* musi spełniać następujące warunki:

$$\forall_{o1,o2 \in O_i \cup OW} \left(\left(\exists_{j=1..n} o1 \rightarrow_{RT} o2 \right) \Rightarrow o1 \mapsto_i o2 \right)$$

$$\forall_{w1,w2 \in OW} \left(\forall_{i=1..n} w1 \mapsto_i w2 \vee \forall_{i=1..n} w2 \mapsto_i w1 \right)$$

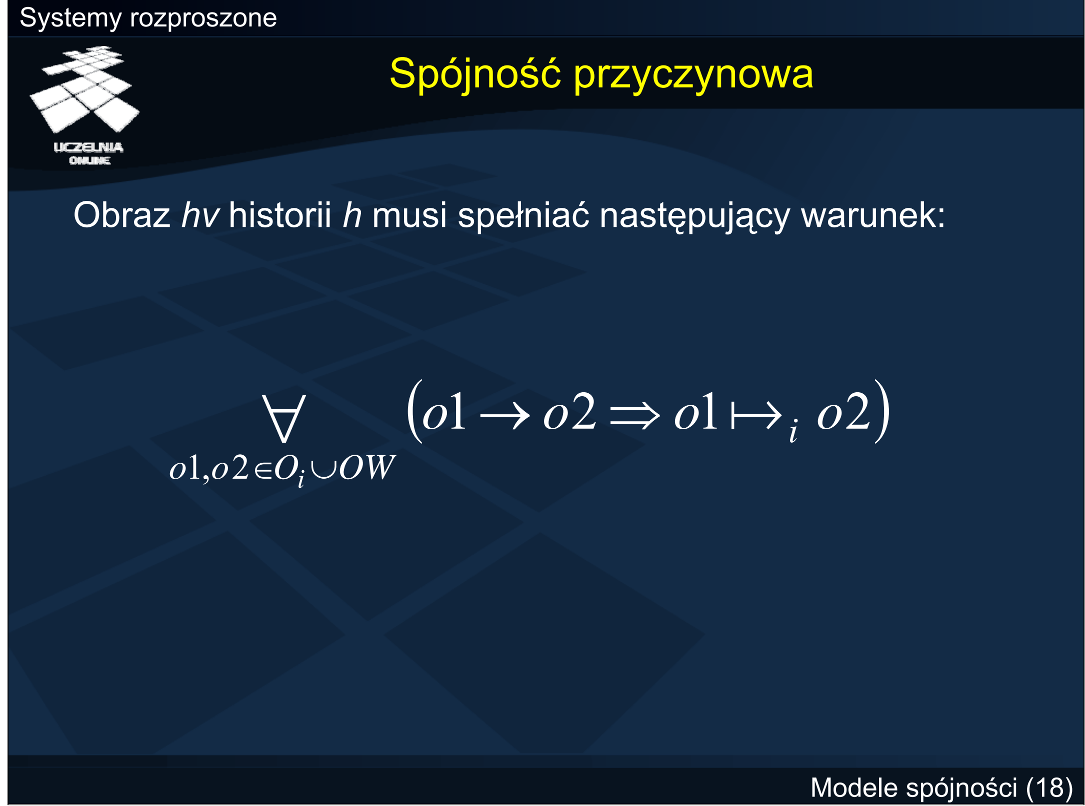
$o1 \rightarrow_{RT} o2$ *o1* kończy się **w czasie rzeczywistym**, zanim zaczyna się *o2*

Spójność atomowa jest silniejszym modelem niż spójność sekwencyjna. W jej przypadku brane są bowiem pod uwagę zależności czasowe pomiędzy zdarzeniami na różnych węzłach. Warunki, które muszą spełniać obrazy historii przetwarzania na poszczególnych węzłach są bardzo podobne do tych dla spójności sekwencyjnej. Drugi warunek jest identyczny. W pierwszym zamiast lokalnego porządku występuje zależność czasowa pomiędzy zdarzeniami. Oczywiście synchronizacja przetwarzania z uwzględnieniem zależności czasowych jest dużo bardziej kosztowna i wymaga dodatkowej komunikacji.



Rysunek przedstawia tą samą sytuację, która była prezentowana w przypadku spójności sekwencyjnej. W tym przypadku jednak uzyskanie takich samych obrazów historii przetwarzania wymaga „przesunięcia" zapisu w(x)2 tak, aby w czasie rzeczywistym występował po odczycie *r*(*x*)1 w procesie p_1.

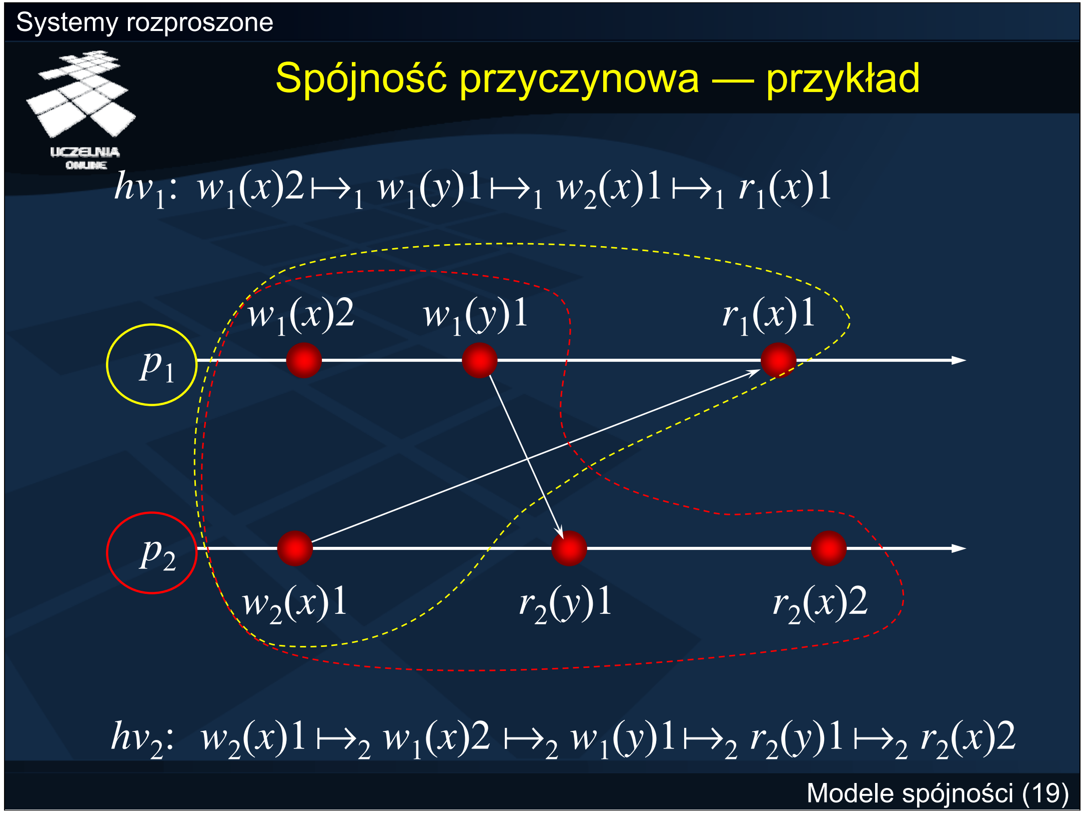
Spójność atomowa w praktyce nie jest realizowana. Jej znaczenie wiąże się głównie z wykorzystaniem do formalnej weryfikacji algorytmów.



Pamięć spójna przyczynowo musi zachowywać porządek przyczynowy operacji. Jeżeli więc operacja $o2$ zależy przyczynowo od $o1$, to we wszystkich lokalnych obrazach przetwarzania operacje te powinny występować właśnie w tej kolejności. W praktyce można to wyrazić następująco:

Zapisy potencjalnie powiązane przyczynowo muszą być widziane przez wszystkie procesy w takim samym porządku. Zapisy współbieżne mogą być na różnych maszynach oglądane w różnej kolejności.

Implementacja spójności przyczynowej musi przechowywać informacje o zależnościach przyczynowych pomiędzy operacjami. Można do tego celu wykorzystać wektorowe znaczniki czasu.



Na slajdzie pokazano przykładowe przetwarzanie w systemie zachowującym przyczynowe uporządkowanie operacji. Każdy z procesów w swoim obrazie przetwarzania musi umieścić wszystkie lokalne operacje oraz wszystkie operacje zapisu, zachowując przy tym warunek legalności. Pierwsze zapisy do zmiennej x realizowane przez oba procesy p_1 i p_2 są realizowane współbieżnie, nie są od siebie zależne przyczynowo. W efekcie ich wyniki mogą być obserwowalne na węzłach w różnej kolejności. Tak też się stało w tym przypadku. Proces p_1 odczytuje z x wartość 1, co oznacza, że zapis tej wartości musiał wystąpić po zapisie $w(x)2$ (inaczej naruszony byłby warunek legalności – odczytywana byłaby wartość nadpisana). W procesie p_2 następuje odczyt z x wartości 2, co oznacza, że odpowiedni zapis musiał trafić po $w(x)1$. W efekcie procesy p_1 i p_2 widzą oba zapisy do x w różnej kolejności. Jest to jak najbardziej dopuszczalne, ponieważ zapisy te były realizowane współbieżnie.

Warto zauważyć, że powyższa realizacja nie jest spójna sekwencyjnie. W modelu sekwencyjnym *wszystkie* zapisy powinny być postrzegane przez *wszystkie* procesy w tej samej kolejności.

Protokół dla spójności przyczynowej

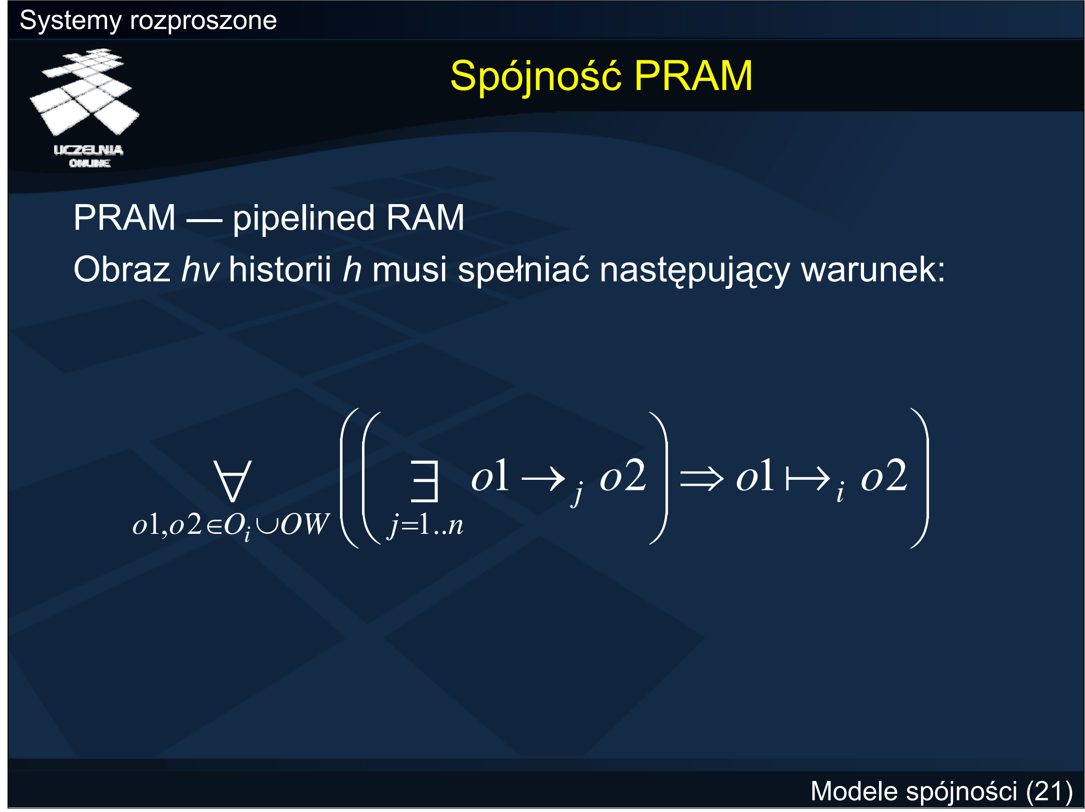
Algorytm dla procesu p_i

- **upon read(x)**
  return $M_i[x]$

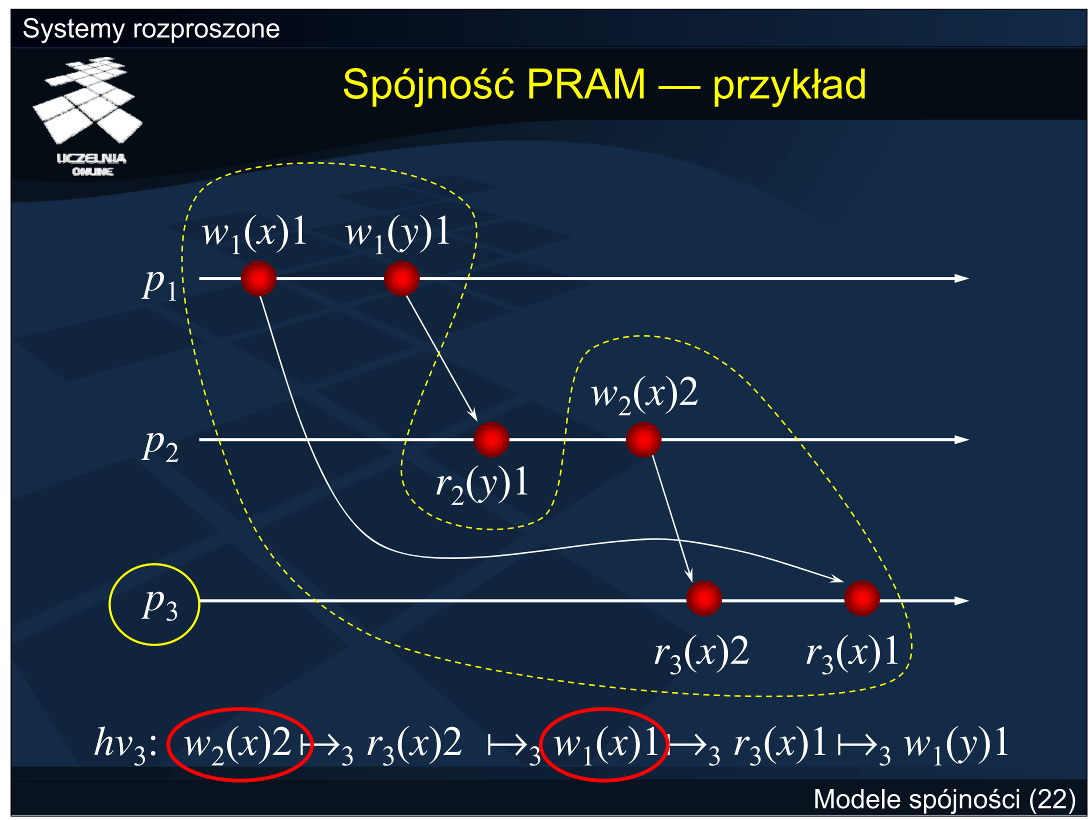
- **write(x, v)**
  $M_i[x] := v$
  causal_broadcast U(x, v)

- **upon receipt of U(x, v) from p_k**
  if $k \neq i$
  $M_i[x] := v$
  end if

Slajd przedstawia przykładowy protokół realizujący model spójności przyczynowej. Do jego implementacji wykorzystano rozgłaszanie zachowujące uporządkowanie przyczynowe (funkcja *causal_broadcast*), a więc cały ciężar realizacji protokołu spada w tym przypadku na warstwę komunikacji grupowej. Warto zwrócić uwagę, że zarówno operacje odczytu jak i zapisu nie wymagają w tym przypadku blokowania, synchronizującego przetwarzanie z operacjami na innych węzłach. Zarówno zapis jak i odczyt może być wykonywany z pełną szybkością. Przetwarzanie lokalne zmiennych zawsze generuje realizacje zachowujące legalność ponieważ zapisy od razu są odnotowywane w pamięci, co powoduje, że następujące po nich odczyty zwracają zapisaną wartość. Komunikaty rozgłoszeniowe są ignorowane przez węzły, które wysyłają odpowiednie aktualizacje.



Spójność PRAM (ang. *pipelind RAM*) jest modelem słabszym od spójności przyczynowej. Dla zapewnienia spójności PRAM wystarczy w obrazach historii każdego procesu zachować lokalny porządek wykonywania operacji każdego innego procesu. Nazwa *pipelined* oczywiście nie jest tu przypadkowa – chodzi o potok zleceń płynących od danego procesu. Potok ten musi zachowywać oryginalne uporządkowanie.



Na slajdzie pokazano przykładowe przetwarzanie w systemie zachowującym spójność PRAM. Zaznaczone operacje są uwzględniane w obrazie historii procesu p_3. Zapisy $w(x)1$ i $w(x)2$ realizowane są przez różne procesy, w związku z czym mogą być wykon ywane w procesie p_3 w dowolnej kolejności. W przypadku przetwarzania z rysunku zapis $w(x)2$ dociera do p_3 przed zapisem zapisem $w(x)1$. Z punktu widzenia modelu spójności PRAM ważne jest jednak to, że zapisy $w(x)1$ i $w(y)1$ są widziane w kolejności, w której były wykonywane przez p_1.

Warto zauważyć, że zapisy $w(x)1$ i $w(x)2$ są od siebie zależne przyczynowo. Łączy je odczyt zmiennej y zapisywanej przez p_1. W obrazie historii w procesie p_3 operacje te występują jednak w odwrotnej kolejności. W związku z tym przedstawione przetwarzanie nie zachowuje spójności przyczynowej.

Protokół dla spójności PRAM

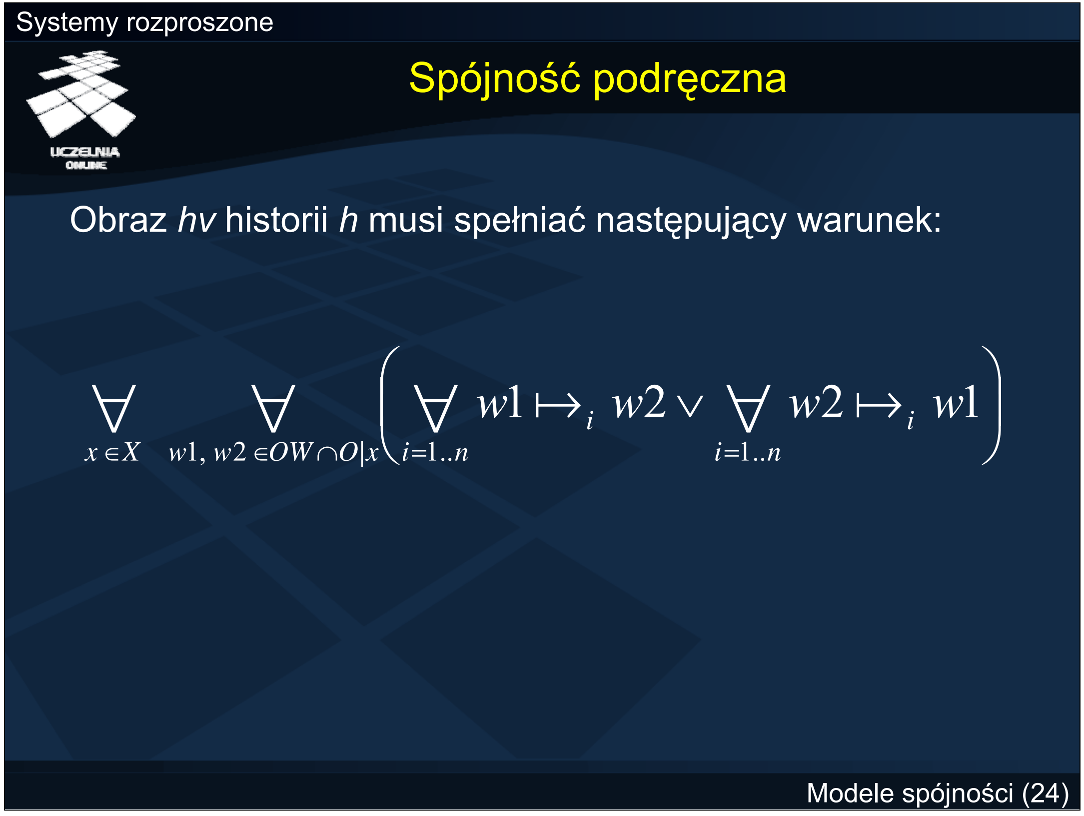
Algorytm dla procesu p_i

- **upon read(x)**
  return $M_i[x]$

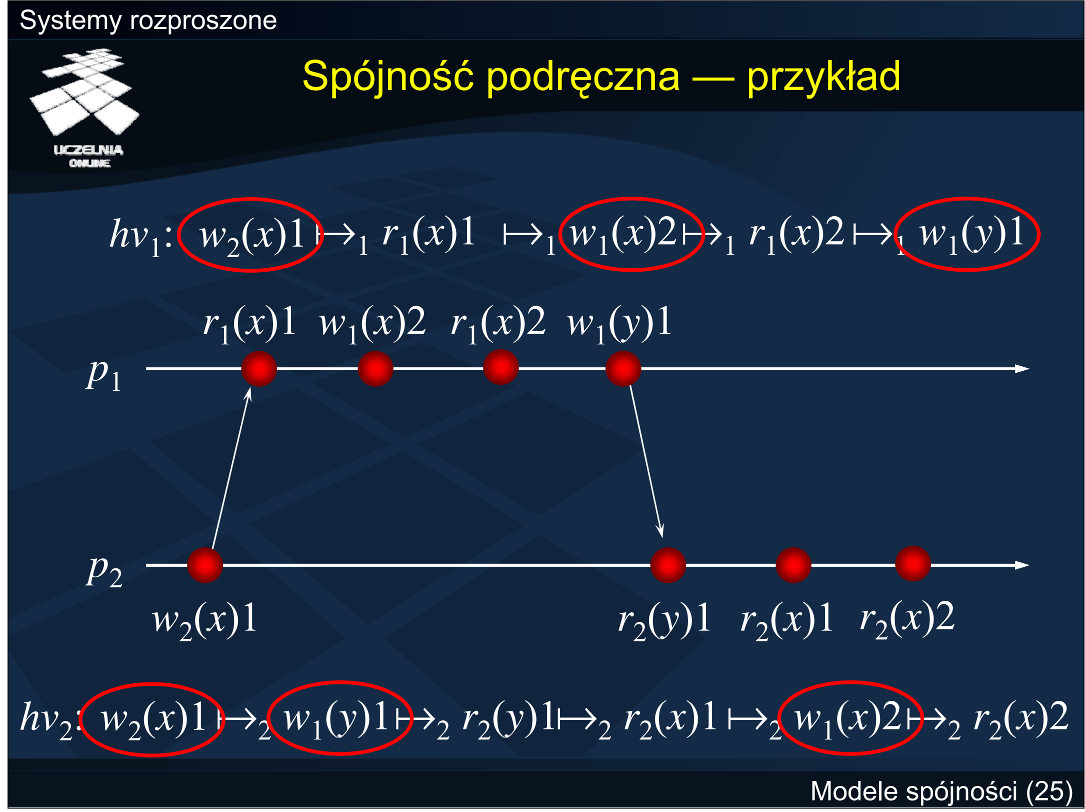
- **upon write(x, v)**
  $M_i[x] := v$
  FIFO_broadcast U(x, v)

- **upon receipt of U(x, v) from p_k**
  if $k \neq i$
  $M_i[x] := v$
  end if

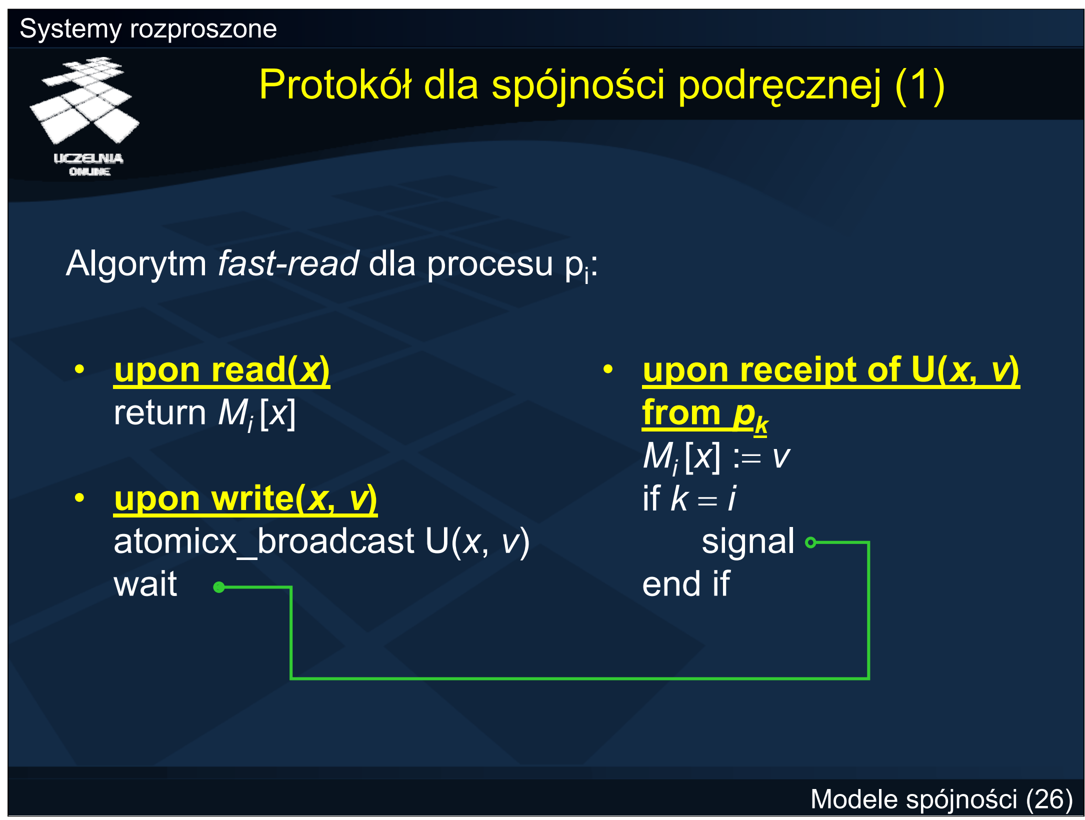
Protokół spójności dla modelu PRAM jest bardzo podobny do protokołu spójności dla spójności przyczynowej. Jedna różnica polega na zastąpieniu rozgłaszania zachowującego porządek przyczynowy (*causal_broadcast*) rozgłaszaniem zachowującym kolejność wysyłania komunikatów rozgłoszeniowych danego procesu (ang. *FIFO broadacst*). Oczywiście realizacja funkcji *FIFO_broadcast* jest znacznie „tańsza" niż realizacja funkcji *causal_broadcast*, ale też model PRAM oferuje mniej niż spójność przyczynowa. W tym przypadku również operacje zapisu i odczytu nie są blokowane, co oczywiście pozytywnie wpływa na efektywność.



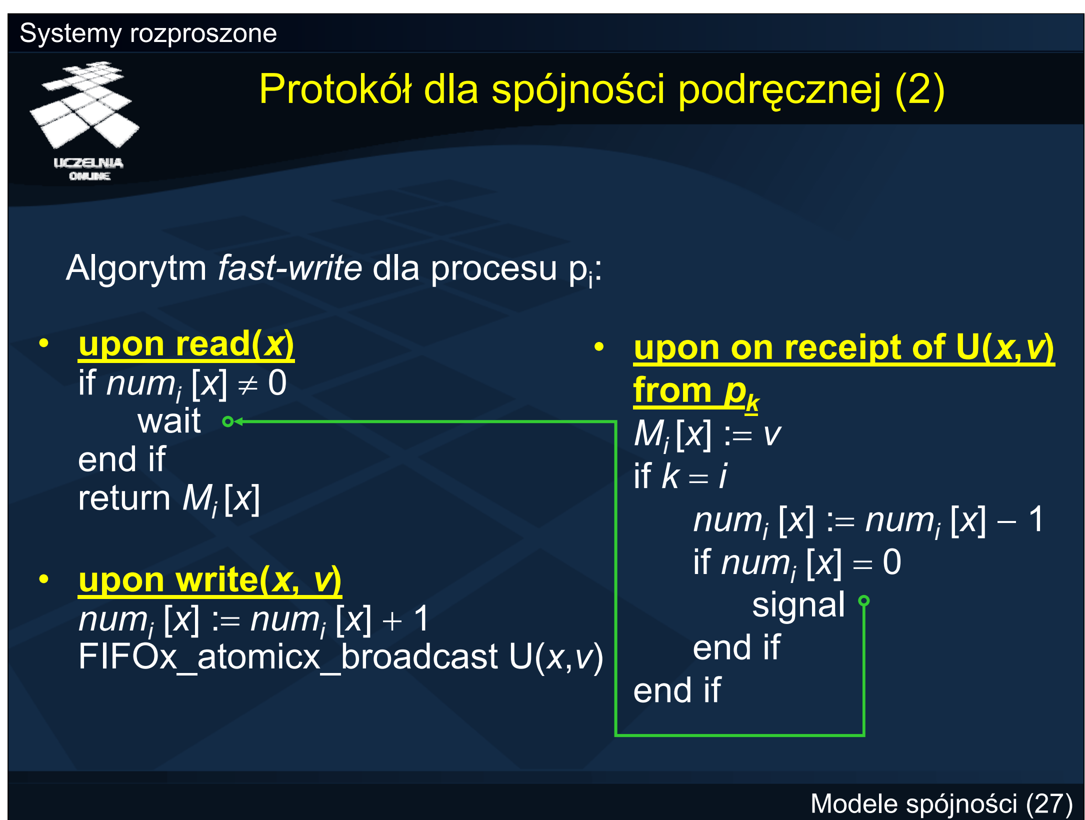
Spójność podręczna (ang. *cache consistency*) jest modelem, w którym zachowany jest globalny porządek operacji zapisu do poszczególnych zmiennych. Część wspólna zbiorów *OW* i *O*|*x* reprezentuje wszystkie zapisy w systemie dotyczące zmiennej *x*. Obrazy historii przetwarzania we wszystkich procesach powinny umieszczać te zapisy w tej samej kolejności. Model ten nie wymaga, aby zachowane było lokalne uporządkowanie operacji o ile dotyczą one różnych zmiennych. Jest to więc model, który jest całkowicie niezależny do spójności PRAM i spójności przyczynowej.



Slajd przedstawia przykładowe przetwarzanie w systemie zachowującym spójność podręczną. Jak ławo zauważyć operacje zapisu wykonywane na zmiennej *x* są widziane przez oba procesy w tej samej kolejności. Zmienna *y* nie wnosi tu nic nowego, ponieważ jest tylko jeden zapis. Warto jednak zwrócić uwagę na uporządkowanie wszystkich zapisów w obrazach historii przetwarzania. Widać z niego, że nie jest zachowywany porządek lokalny. Proces p_2 widzi zapis do zmiennej *y* przez zapisem *w*(*x*)2, odwrotnie niż proces p_1. Tym bardziej więc nie jest zachowywany porządek przyczynowy, co widać w sekwencji operacji $w_1(x)2 \rightarrow w_1(y)1 \rightarrow r_2(y)1 \rightarrow r_2(x)1$.

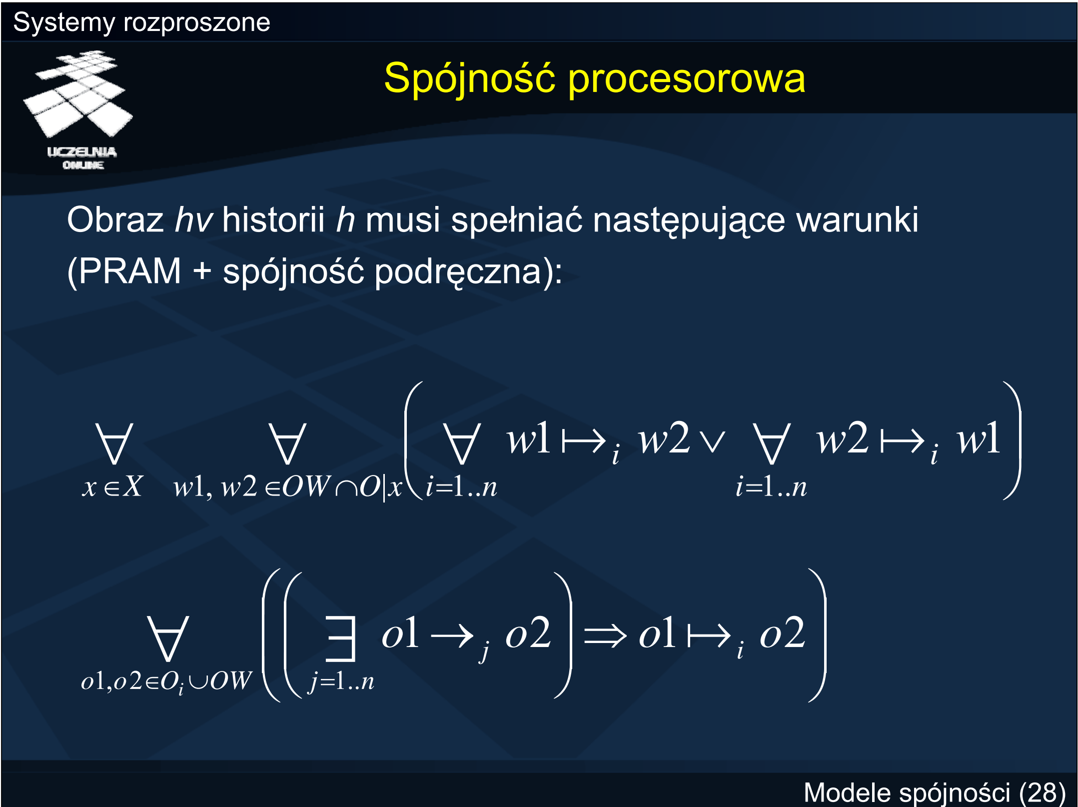


Protokół spójności *fast-read* dla spójności podręcznej przypomina analogiczny protokół dla spójności sekwencyjnej. W tym przypadku następuje również blokowanie podczas realizacji zapisu, ponieważ zapisy te muszą być uporządkowane względem innych zapisów do tych samych zmiennych. W przeciwieństwie do poprzedniego protokołu nie występuje tu jednak pełna wersja rozgłaszania niepodzielnego, porządkującego globalnie wszystkie komunikaty, ale jego wersja uproszczona, porządkująca komunikaty aktualizujące wybraną zmienną. W literaturze rozgłoszenie takie nie jest jakoś specjalnie nazywane, stąd w zapisie protokołu użyto nazwy *atomicx_broadcast*, dla odróżnienia od *atomic_broadcast*. Oczywiście realizacja *atomicx_broadcast* jest mniej kosztowna od pełnego niepodzielnego rozgłaszania, stąd protokół *fast-write* dla spójności podręcznej powinien charakteryzować się większą wydajnością od analogicznego protokołu dla spójności sekwencyjnej. Rozgłoszenie *atomicx_broadcast* może być koordynowane współbieżnie przez różne procesy, o ile dotyczy różnych zmiennych. Jego implementacja może więc być bardziej rozproszona, a więc lepiej wykorzystująca dostępne zasoby.

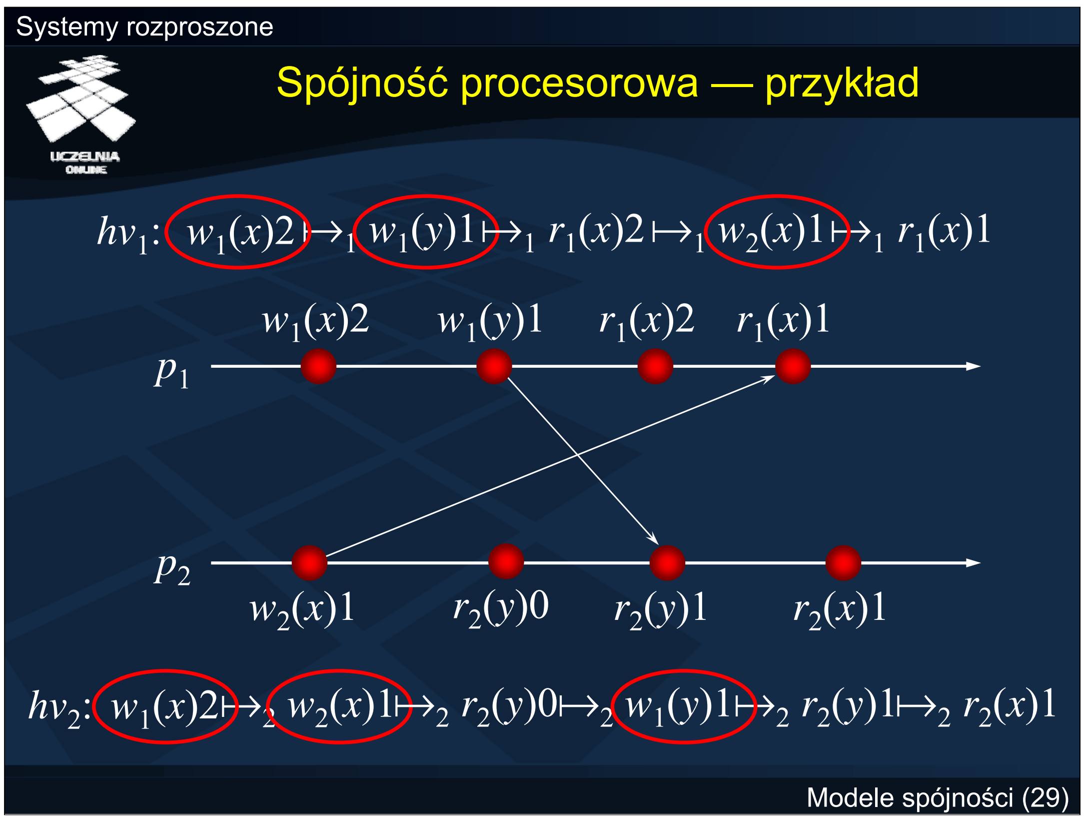


Przez analogię można również skonstruować protokół *fast-write* dla spójności podręcznej, który strukturalnie jest identyczny z protokołem *fast-write* dla spójności sekwencyjnej. W tym przypadku również blokowanie zostaje przeniesione do operacji odczytu. Blokowaniu podlegają jednak jedynie operacje odczytu, które odwołują się do zmiennych, dla których nie dokończono jeszcze realizacji ostatnich operacji zapisu. W celu stwierdzenia, które operacje wymagają synchronizacji, protokół przechowuje tablicę liczników *num*, w której *i*-ta pozycja reprezentuje liczbę aktualnie realizowanych zapisów na *i*-tej zmiennej. Wyzerowanie licznika powoduje zniesienie blokowania oczekującego zapisu.

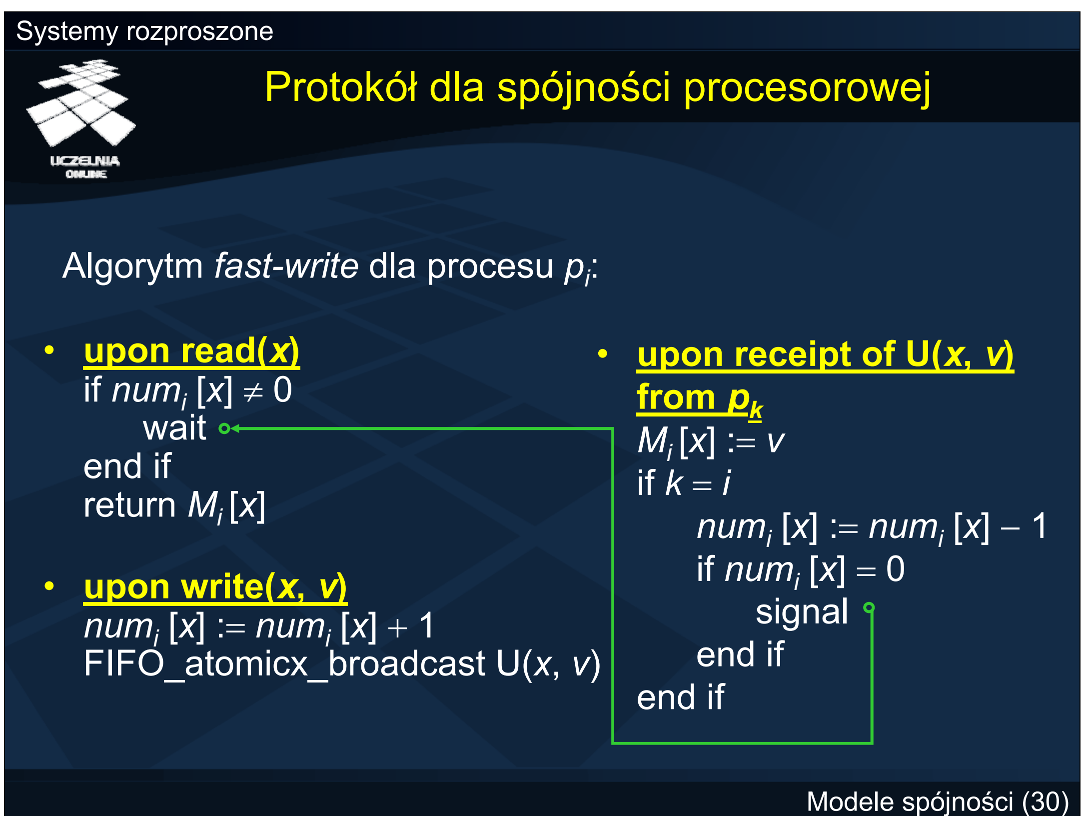
W protokole *fast-write* zapisy nie są blokowane. W związku z tym jest możliwość zlecenia wielu zapisów, które będą realizowane w tle. W szczególności możliwe jest zlecenie wielu zapisów do tej samej zmiennej. Zapisy do tych samych zmiennych powinny być jednak porządkowane, dlatego do rozgłaszania zleceń zapisu użyto rozgłaszania uporządkowanego FIFO, ale nie w pełnej wersji, a jedynie w odniesieniu do poszczególnych zmiennych. Podobnie jak poprzednio, implementacja rozgłaszania porządkującego zapisy do pojedynczych zmiennych jest prostsza do zrealizowania od rozgłaszania porządkującego wszystkie komunikaty.



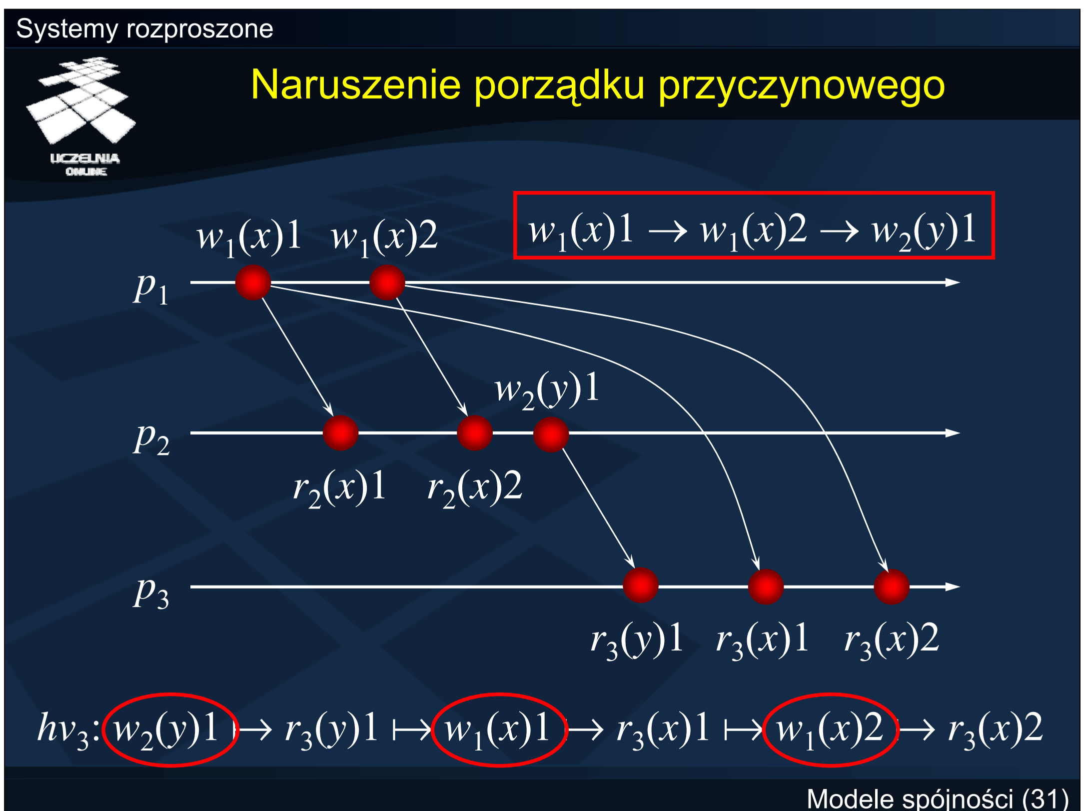
Spójność procesorowa nie wprowadza kompletnie nowej jakości. Jest to po prostu połączenie spójności PRAM i podręcznej. Definicja modelu spójności zawiera warunki, które były wymieniane w definicjach modeli PRAM i spójności podręcznej. W skrócie pamięć jest spójna procesorowo, jeżeli lokalny porządek zlecania operacji w poszczególnych procesach jest zachowany w obrazach historii przetwarzania wszystkich procesów (PRAM), oraz gdy zapisy do poszczególnych zmiennych są globalnie uporządkowane (spójność podręczna).



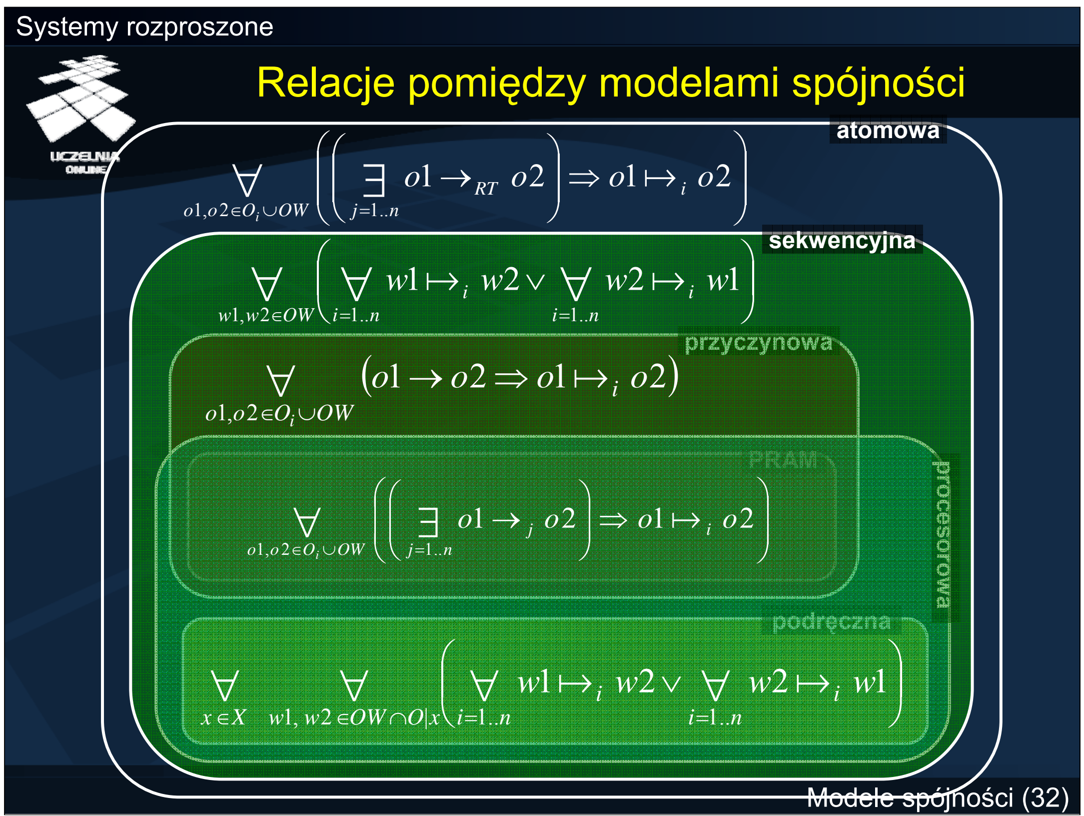
Rysunek przedstawia przykładowe wykonanie w systemie zachowującym spójność procesorową. Jak widać z rysunku w obrazach historii obu procesów zapisy do zmiennej *x* widziane są w tej samej kolejności. Uporządkowanie zleceń z poszczególnych procesów również jest zachowane. Nie oznacza to jednak, że istnieje jeden, globalny porządek wykonywania operacji zapisu (co oznaczałoby efektywnie, że spełnione są wymagania modelu sekwencyjnego). Zapisy *w*(*x*)1 i *w*(*y*)1 występują w obrazach historii procesów w odwrotnej kolejności.



Slajd przedstawia protokół spójności typu *fast-write* dla spójności procesorowej. Struktura protokołu jest identyczna z poprzednimi protokołami *fast-write*. Różnica polega na specyficznej metodzie rozgłaszania zleceń aktualizacji danych. Rozgłaszanie to, oznaczone jako *FIFO_atomicx_broadcast* porządkuje globalnie wszystkie zapisy pojedynczych procesów oraz porządkuje globalnie zapisy odnoszące się do tych samych zmiennych. Zapisy nie są blokowane. Aktualizacje realizowane są po odebraniu komunikatu aktualizacyjnego. Odczyty mogą być blokowane, gdy dotyczą zmiennej, której aktualizacja zlecona przez ten węzeł jest jeszcze realizowana.



Przykład na slajdzie pokazuje realizację w systemie zachowującym spójność procesorową, w której naruszona jest jednak spójność przyczynowa. Proces p_3 dokonuje na początku odczytu ze zmiennej *y*, której aktualizacja dociera do tego procesu jako pierwsza. Następnie docierają do tego procesu dwie aktualizacje zmiennej *x*, które są obserwowane kolejnymi odczytami. Okazuje się jednak, że zapis $w(y)1$ w procesie p_2 jest przyczynowo zależny od wcześniejszych zapisów do zmiennej *x* realizowanych przez proces p_1. Zależność ta nie jest jednak uwzględniana przy propagacji zapisów do procesu p_3. Wskazuje to jednoznacznie na rozróżnialność spójności przyczynowej i procesorowej.



Omówione modele spójności przy dostępie ogólnym są w pewnych zależnościach między sobą. Idąc od dołu, a więc od modeli najmniej restrykcyjnych mamy model PRAM, gwarantujący zachowanie porządku lokalnego wykonywanych operacji przez poszczególne procesy. Rozszerzając zachowywany porządek lokalny modelu PRAM o uporządkowanie przyczynowe dostajemy spójność przyczynową. Spójność podręczna wymaga uporządkowania, które jest całkowicie ortogonalne w stosunku do wymagań spójności PRAM czy przyczynowej. Łącząc własności spójności PRAM i spójności podręcznej dostajemy spójność procesorową. Jeżeli wszystkie zapisy w systemie są globalnie uporządkowane, a więc obserwowane przez wszystkie procesy w tej samej kolejności, to dostajemy model sekwencyjny, który jest silniejszy zarówno od spójności przyczynowej jak i procesorowej. Ostatecznie uwzględnienie w uporządkowaniu globalnym kolejności występowania zdarzeń w czasie rzeczywistym prowadzi do modelu atomowego, który jest z kolei silniejszy od modelu sekwencyjnego.

Dostęp synchronizowany – założenia

- W skład systemu DSM wchodzą
  - zbiór sekwencyjnych procesów $P = \{p_1, p_2, \ldots, p_n\}$
  - zbiór współdzielonych zmiennych $X = \{x_1, x_2, \ldots\}$
  - zbioru obiektów synchronizujących $S = \{s_1, s_2, \ldots\}$
- Obiekty synchronizujące
  - zamek (ang. *lock*), na którym wykonywane są operacje:
    - *acquire* — nabycie zamka
    - *release* — zwolnienie zamka
  - bariera (ang. *barrier*) z operacją
    - synchronizacja na barierze

Drugą grupę modeli spójności w ramach modeli nastawionych na dane tworzą modele przy dostępie synchronizowanym. Modele te próbują zaradzić niedogodnościom modeli przy dostępie swobodnym, w których albo protokół okazywał się być nieefektywny (spójność atomowa i sekwencyjna), albo oferowane własności uszeregowania zapisów były niewystarczające do działania aplikacji. Niekiedy może się okazać nawet, że modele słabsze (spójność przyczynowa, PRAM czy podręczna) są wystarczające do konkretnych algorytmów, ale wykazanie tej wystarczalności jest zadaniem bardzo trudnym. Powoduje to zrozumiałą niechęć programistów do stosowania słabszych modeli spójności.

Rozwiązaniem zaproponowanym przez twórców niektórych systemów z rozproszoną pamięcią dzieloną jest propozycja kompromisu pomiędzy wygodą programowania takich systemów a efektywnością ich pracy. Modele przy dostępie ogólnym nie wymagają żadnej zmiany kodu algorytmu – program powinien działać poprawnie bez modyfikacji, niestety nieefektywnie. Jeżeli zrezygnować z pełnej transparentności i wymusić na programiście oznakowanie kodu dodatkowymi instrukcjami podpowiadającymi systemowi jakie są oczekiwania co do spójności danych, to implementacja odpowiedniego protokołu spójności może się okazać dużo bardziej efektywna. Jeżeli tylko dodatkowe instrukcje i ich umiejscowienie będzie proste, to rozwiązanie takie ma szansę na popularyzację.

Kolejne slajdy prezentują modele spójności przy dostępie synchronizowanym. Model systemu pozostaje taki jak poprzednio. Działania wykonywane są więc przez zbiór n procesów rozlokowanych na n węzłach. Procesy wykonują operacje na zbiorze zmiennych współdzielonych, których repliki znajdują się na każdym węźle. Dodatkowo istnieje zbiór S obiektów synchronizujących. W zależności od modelu wykorzystywane są następujące obiekty synchronizujące: zamki i bariery. Na zamkach wykonywane są dwie operacje: *acquire* – zajęcie (nabycie) zamka i *release* – zwolnienie zamka. Dla bariery wykonuje się operację synchronizacji na barierze. Synchronizacja taka polega na wstrzymaniu przetwarzania procesów zgłaszających się do bariery do czasu zgłoszenia się wszystkich procesów.

Spójność słaba

- Operacje dostępu
  - do danych współdzielonych
  - do zmiennych synchronizujących
- Definicja modelu
  1. Operacje na zm. synchronizujących są spójne atomowo
  2. Nie można wykonać (zakończyć) operacji dostępu do zmiennej synchronizującej przed globalnym zakończeniem wcześniejszych operacji dostępu do zmiennych współdzielonych
  3. Analogiczne ograniczenie w drugą stronę

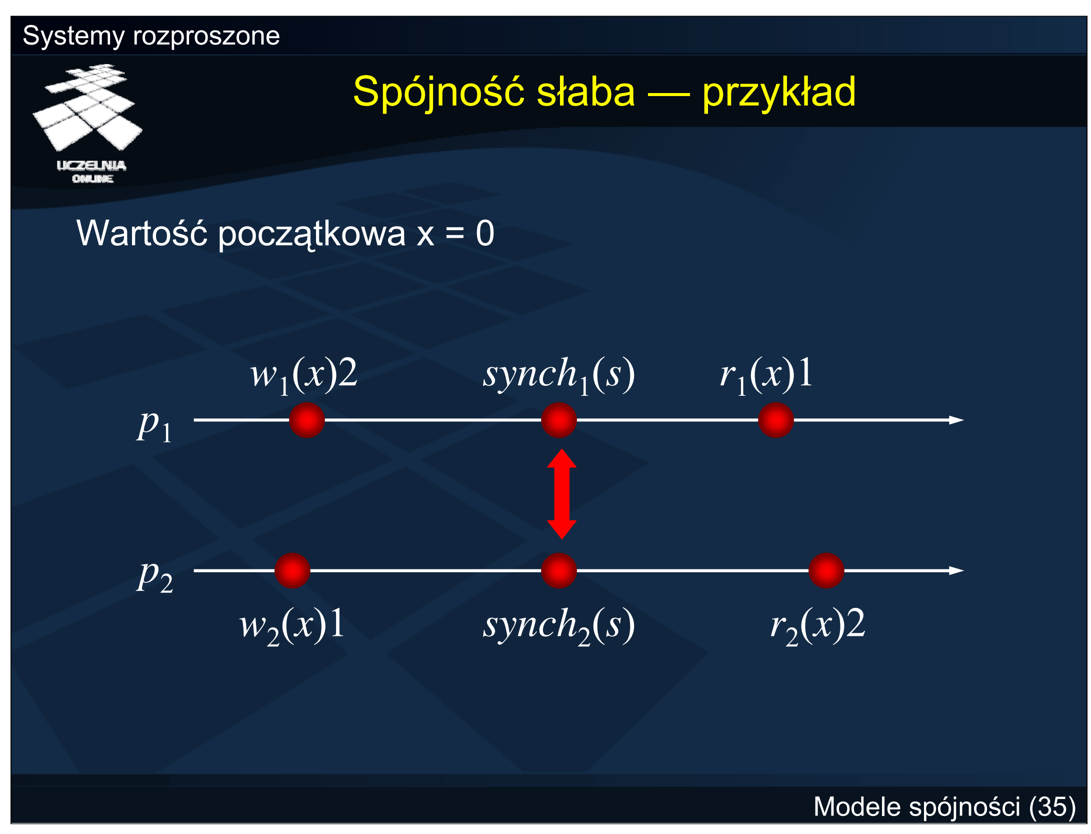
Pierwszym modelem spójności przy dostępie synchronizowanym była **spójność słaba** (ang. *weak consistency*). W modelu tym rozróżniono po raz pierwszy dwa typy operacji: operacje dostępu do danych współdzielonych i operacje na zmiennych synchronizujących. Podstawową motywacją dla wprowadzenia jawnej operacji synchronizującej było umożliwienie opóźnienia propagacji aktualizacji do innych węzłów do czasu aż to będzie faktycznie potrzebne. W modelu sekwencyjnym praktycznie każda, najmniejsza operacja zmusza system do podejmowania kroków zmierzających do uspójnienia danych. Niekiedy jednak logika algorytmu wskazuje, że dane będą odczytywane później, po zaktualizowaniu większej grupy zmiennych. W spójności słabej zapisy do zmiennych współdzielonych nie muszą być od razu propagowane. Można je zbuforować i przesłać łącznie podczas wykonywania operacji synchronizującej.

W spójności słabej wyróżnia się jeden typ operacji synchronizującej. Definicja spójności słabej mówi, że:

1) Operacje na zmiennych synchronizujących są spójne atomowo (w nowszym podejściu sekwencyjnie). Wykonując operację synchronizującą proces może więc uzyskać gwarancję, że jego wcześniejsze modyfikacje na pewno dotrą do pozostałych procesów.

2) Nie można wykonać (zakończyć) operacji dostępu do zmiennej synchronizującej przed globalnym zakończeniem wcześniejszych operacji dostępu do zmiennych współdzielonych. Chodzi o to, żeby operacja synchronizująca swoim działaniem objęła wszystkie wcześniej wykonane operacje danego procesu.

3) Nie można wykonać (zakończyć) operacji dostępu do zmiennej współdzielonej przed globalnym zakończeniem wcześniejszych operacji dostępu do zmiennych synchronizujących. Motywacją w tym przypadku jest uniemożliwienie np. odczytywania danych, jeżeli dane te nie zostały jeszcze zaktualizowane, ponieważ operacja synchronizująca cały czas jest jeszcze wykonywana.


Systemy rozproszone
Spójność słaba — przykład
UCZELNIA ONLINE
Wartość początkowa x = 0
$w_1(x)2$
$synch_1(s)$
$r_1(x)1$
p_1
p_2
$w_2(x)1$
$synch_2(s)$
$r_2(x)2$
Modele spójności (35)

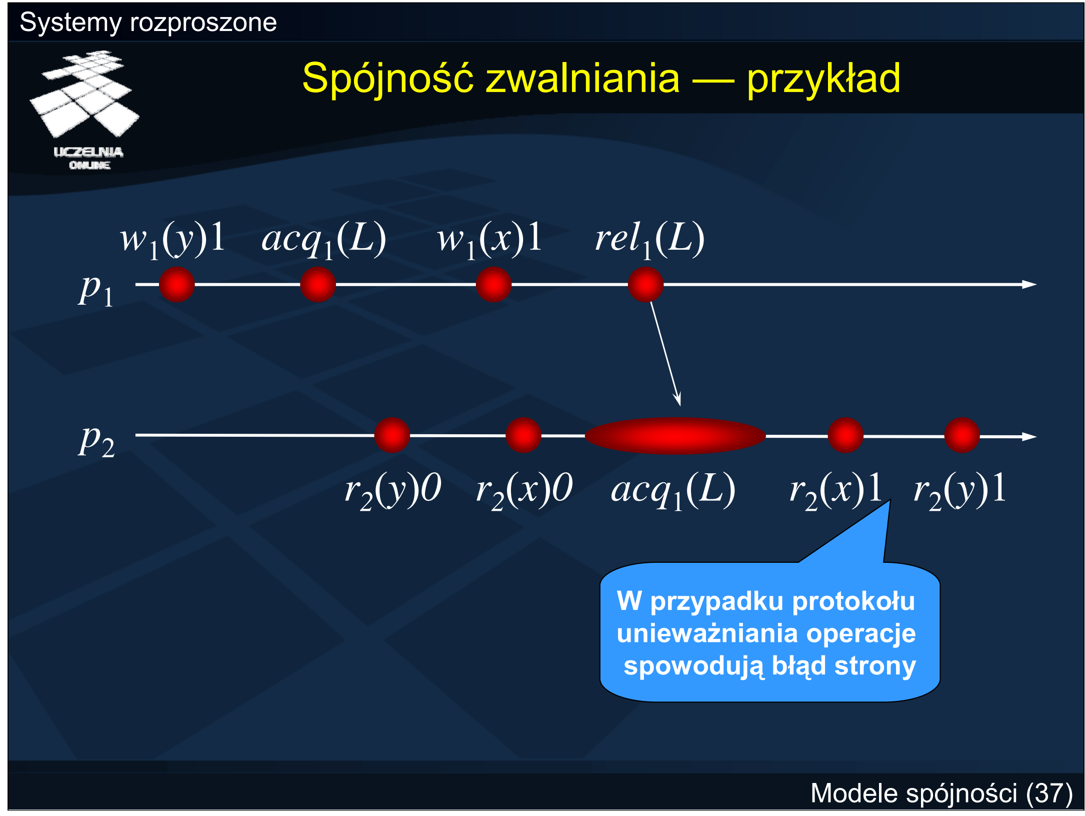
Spójność zwalniania

- Rodzaje synchronizacji
  - wzajemne wykluczanie (*acquire-release*)
  - synchronizacja na barierze
- Operacje synchronizujące są spójne procesorowo, a pozostałe operacje są spójne w sensie PRAM
- Aktualizacja/unieważnianie replik przy operacji bariery oraz przy
  - *release* w przypadku spójności *eager release*
  - *acquire* w przypadku spójności *lazy release*

Kolejny model spójności to spójność zwalniania (ang. *release consistency*). Model ten udoskonalił spójność słabą poprzez rozróżnienie dwóch oddzielnych operacji synchronizujących wykonywanych na zamkach. Operacje te w swojej logice działają podobnie do semaforów, umożliwiając realizację wzajemnego wykluczania. Dodatkowo podczas ich wykonywania następuje propagacja zmian do pozostałych serwerów. Operacja bariery służy do globalnej synchronizacji całej pamięci na wszystkich węzłach.

Rozróżniamy dwa istotnie różne protokoły spójności dla spójności zwalniania; protokoły, które były niekiedy mylnie nazywane modelami spójności. Otóż w protokole *eager release* węzeł, który wykonuje operację zwalniania zamka rozsyła jednocześnie informację aktualizującą do wszystkich pozostałych serwerów. Do czasu wykonania operacji *release* żadna komunikacja nie jest potrzebna. Sama informacja aktualizacyjna może mieć postać aktualizacji bezpośrednio danych lub może powodować unieważnienie replik.

Drugi protokół to *lazy release*, w którym stosuje się właśnie podejście *leniwe*. W przeciwieństwie do protokołu *eager* nie wysyła się aktualizacji do wszystkich, bo najprawdopodobniej wszyscy ich nie potrzebują. Aktualizacja danych jest tu opóźniona do czasu wykonania następnej operacji *acquire*. W tym momencie bowiem wiadomo, kto tych danych potrzebuje. Synchronizacja następuje więc pomiędzy dwoma konkretnymi serwerami: tym, który dokonywał ostatnich modyfikacji (posiada więc aktualną kopię) i tym, który potrzebuje te dane. Jest to rozwiązanie bardzo oszczędne komunikacyjnie, ponieważ angażuje tylko węzły, które muszą być w tej sytuacji angażowane. Niestety pewnym problemem staje się w tym podejściu kumulacja informacji o wcześniejszym przetwarzaniu. Każdy kolejny węzeł musi bowiem uczestniczyć w przenoszeniu informacji o modyfikacjach, zarówno tych, które on sam dokonał, jak i tych wykonanych przez wszystkie węzły, które uczestniczyły w przetwarzaniu.



Rysunek pokazuje przykładowe przetwarzanie w systemie stosującym spójność zwalniania. Odczyty realizowane przez proces p_2 przed operacją *acquire* zwracają wartości początkowe dla zmiennych *x* i *y*, gdyż protokół aktualizuje dane tylko podczas wykonywania operacji synchronizujących. Proces p_2 wykonuje operację *acquire* na zamku, który wcześniej zajmowany był przez proces p_1, co powoduje przesłanie aktualizacji z p_1 do p_2. Dzięki temu kolejne odczyty, zarówno zmiennej *x* jak i *y* odzwierciedlają modyfikacje dokonane przez proces p_1. Jeżeli stosowany jest protokół unieważniania, to odwołanie do strony, gdzie znajduje się zmienna *x* lub zmienna *y* spowoduje pobranie aktualnej wersji tej strony z węzła p_1.

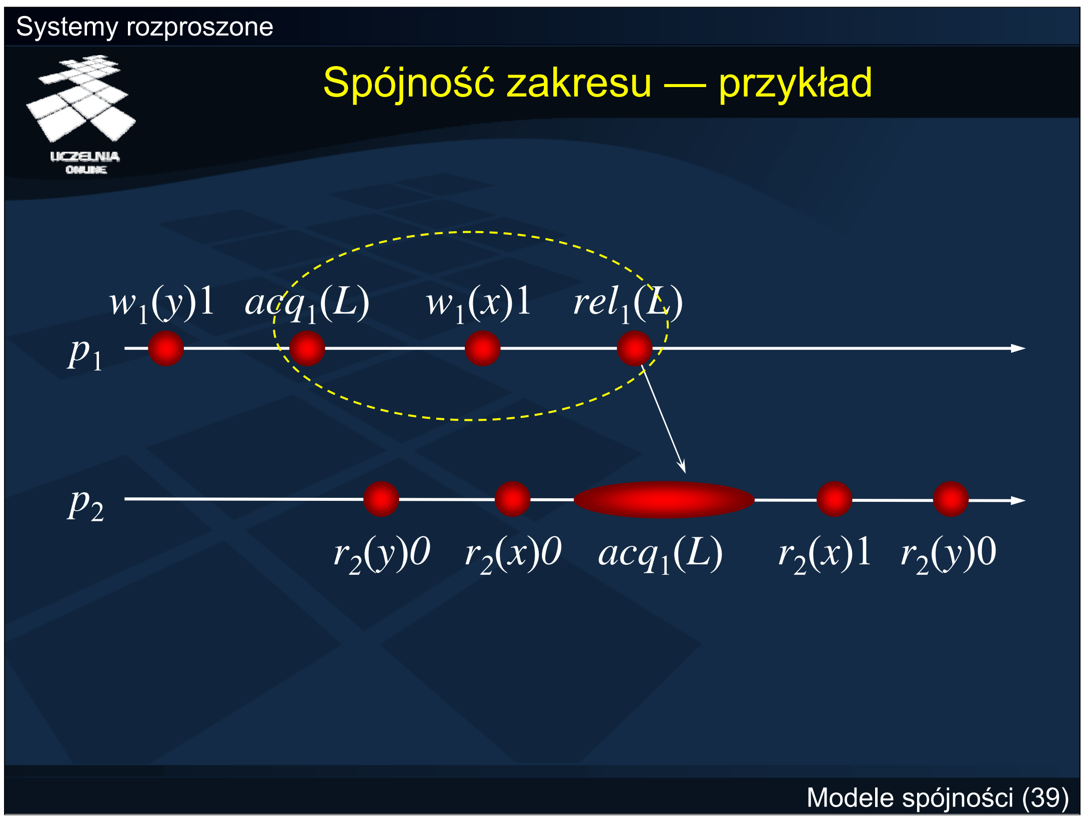
Podsumowując: węzeł, który zwalnia blokadę przekazuje węzłowi, który przejmuje blokadę informacje o *wszystkich* modyfikacjach, których węzeł docelowy nie zna. Do reprezentacji historii przetwarzania i wydzielania fragmentów historii nieznanych innym węzłom stosuje się wektorowe etykiety czasowe.

Operacje synchronizujące mogą być wykonywane współbieżnie na wielu serwerach, ponieważ istnieje wiele zamków.

Spójność zakresu

- Operacje *acquire* i *release* odpowiednio otwierają i zamykają zakres
- Operacja bariery zamyka zakres globalny i otwiera następny zakres globalny
- Istnieją jawne operacje otwarcia i zamknięcia zakresu: *open_scope* i *close_scope*
- Po otwarciu zakresu aktualizowane/unieważnianie są wszystkie repliki zmodyfikowane w czasie poprzedniego otwarcia zakresu (poprzedni zakres musi być zamknięty)

Spójność zakresu (ang. *scope consistency*) jest dalszą modyfikacją przetwarzania obserwowanego w spójności zwalniania. Wymiana informacji następuje tu tak samo jak w przypadku protokołu *lazy release*, a więc przy wywoływaniu operacji *acquire*. Zmienia się jednak zakres tej informacji. O ile spójność zwalniania wymagała przesłania *wszystkich* informacji nieznanych węzłowi docelowemu, o tyle w spójności zakresu przesyłane są tylko te modyfikacje, które zostały wykonane od ostatniej operacji *acquire* na żądanym zamku. Najprawdopodobniej bowiem to właśnie te dane są potrzebne zajmującemu blokadę procesowi, a nie te, które były modyfikowane wcześniej czy później. Oczywistą konsekwencją tego rozwiązania jest zmniejszenie rozmiarów przesyłanych komunikatów. Informacja aktualizacyjna przesyłana jest tylko do węzła, który jej potrzebuje i w takim zakresie, jakiego potrzebuje.



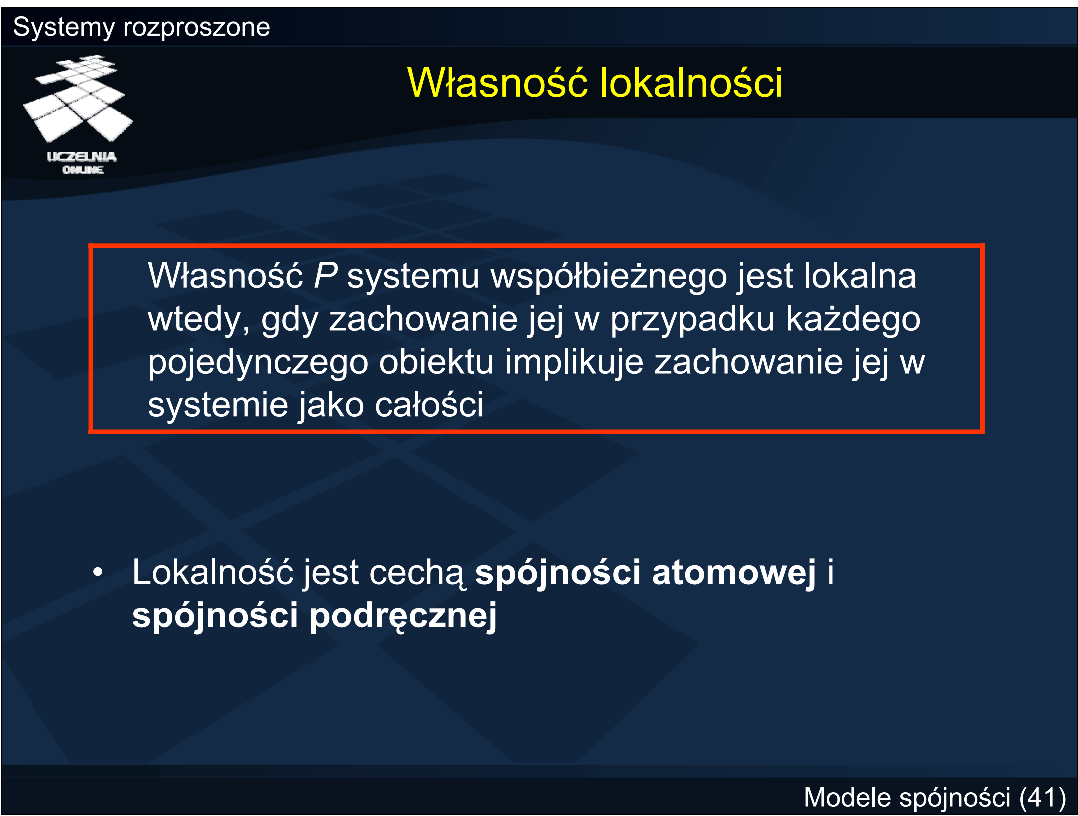
Rysunek przedstawia przykładowe przetwarzanie tożsame z tym, które było prezentowane w przypadku spójności zwalniania. W systemie stosującym spójność zakresu również nastąpi synchronizacja danych w momencie wykonywania operacji *acquire*. Jednakże proces p_2 nie zostanie poinformowany o modyfikacjach zmiennej *y*, ponieważ była ona modyfikowana przed zakresem wyznaczonym operacją *acquire*. Poprawny dostęp do zmiennej *y* wymagałby w tym przypadku: przesunięcia operacji *acquire* przed zapis do *y* lub użycie innej zmiennej synchronizującej i wskazanie nowego zakresu, do którego można by się odwołać w procesie p_2.

Spójność wejścia

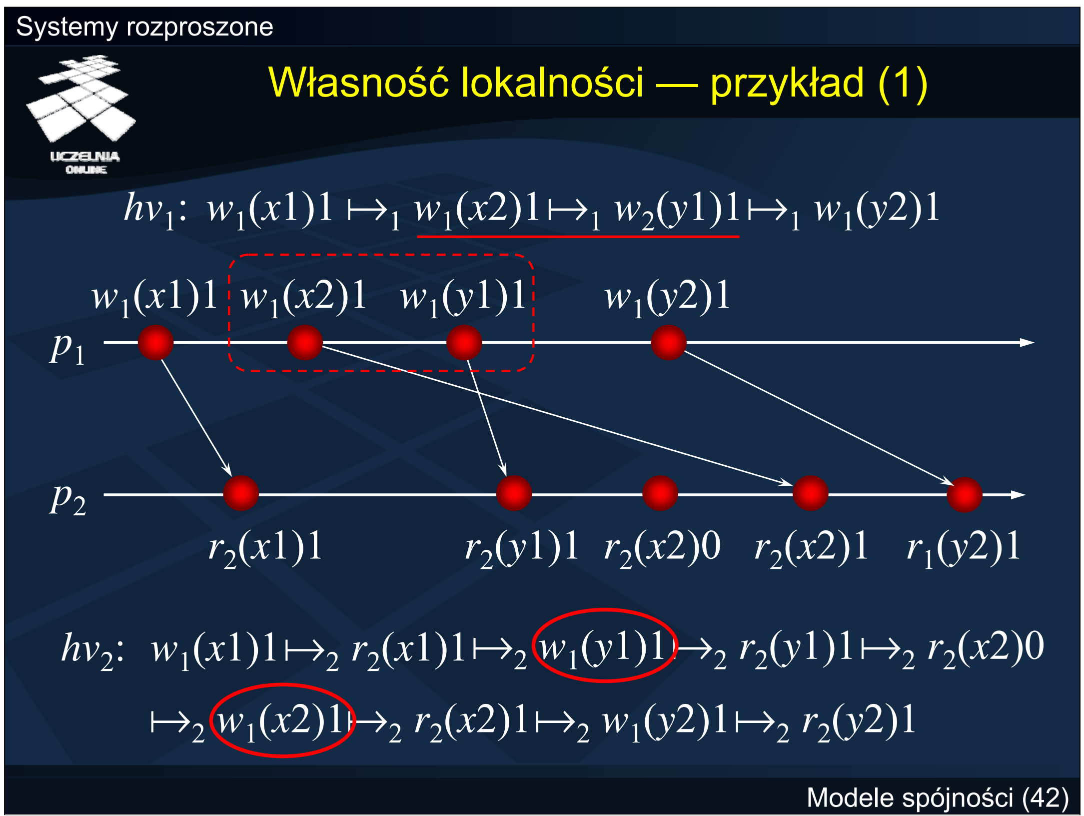
- Spójność wejścia zbliżona jest do spójności zakresu
- Są dwa rodzaje operacji acquire: *współdzielona* i *wyłączna* (odczyt i zapis)
- Z każdym globalnym obiektem związana jest zmienna synchronizująca, na której wykonywana jest odpowiednia operacja przed dostępem do zmiennej

W spójności zwalniania i spójności zakresu zmienne synchronizujące są luźno związane ze zmiennymi współdzielonymi. To programista w swojej głowie ustala, że np. dostęp do zmiennej x chroniony będzie zamkiem L_1, a do zmiennej y zamkiem L_2. Inaczej jest w przypadku spójności wejścia, gdzie programista ma obowiązek jawnego powiązania każdego obiektu współdzielonego z jakąś zmienną synchronizującą. Dzięki takiemu rozwiązaniu następuje jeszcze bardziej precyzyjny transfer aktualizacji pomiędzy serwerami, ponieważ system wie na podstawie zlecanych operacji synchronizujących które obiekty ma aktualizować. Niestety wadą tego modelu jest konieczność definiowania wielu zmiennych synchronizacyjnych i konieczność ustalania powiązań między nimi.

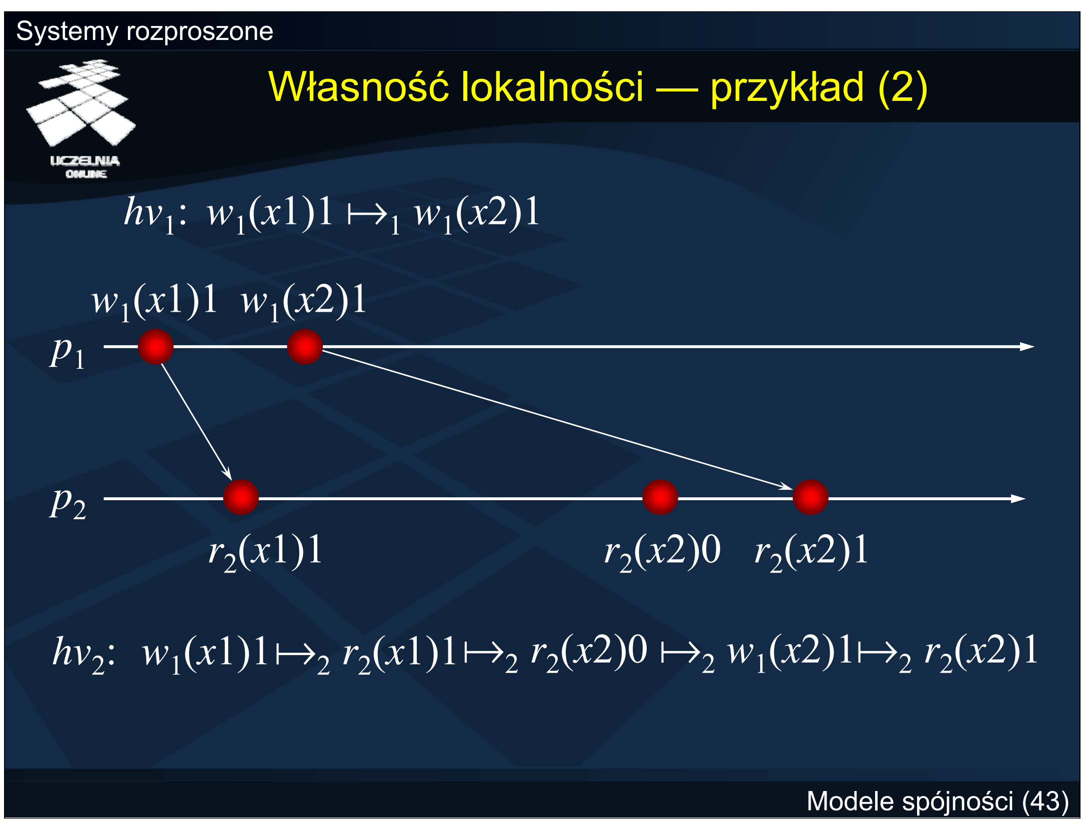
Dodatkową cechą spójności wejścia jest wyróżnienie dwóch typów operacji *acquire*: zajęcie zamka może być w trybie wyłącznym lub dzielonym. Umożliwia to realizację współbieżnego odczytu tych samych danych na wielu węzłach, co nie było możliwe w przypadku spójności zwalniania i zakresu.



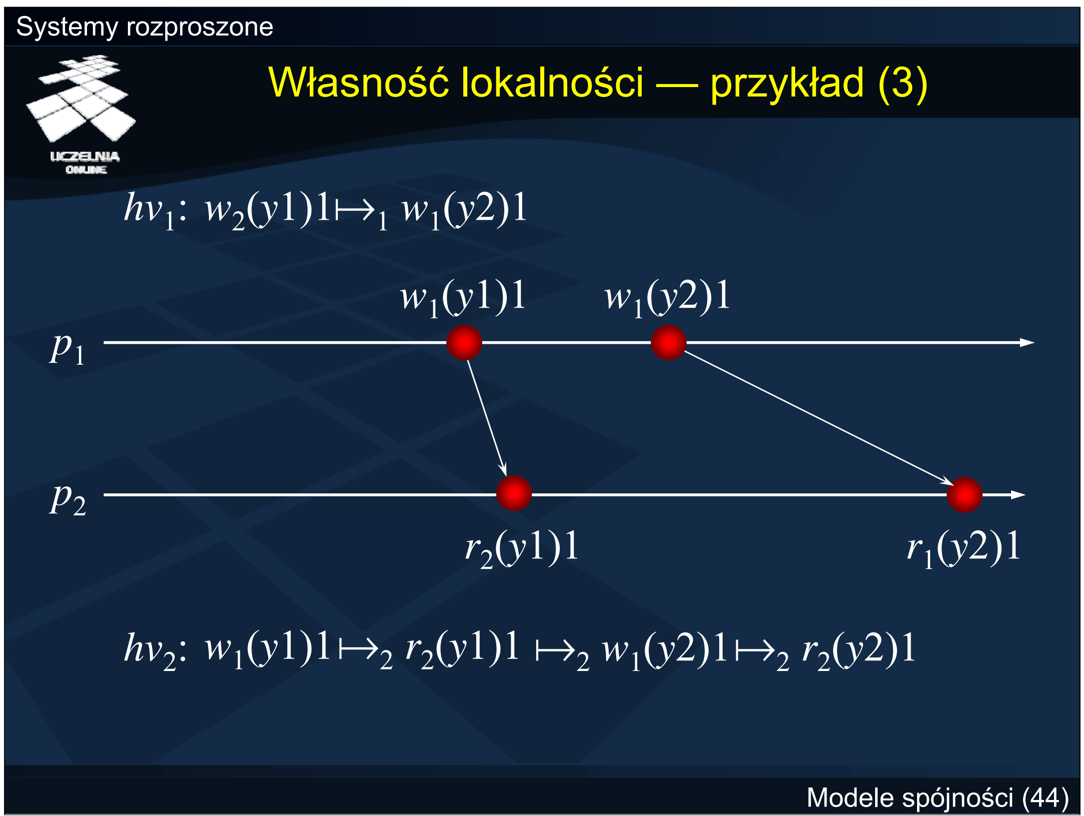
W analizie modeli spójności używa się niekiedy pojęcia **lokalności** (ang. *locality*), której definicja zawarta jest na slajdzie. Lokalność jest cechą modeli spójności atomowej i podręcznej. W przypadku spójności podręcznej jest to dość naturalne, bo definicja modelu odnosi się właśnie do pojedynczych obiektów. Jeżeli więc zapisy do każdego pojedynczego obiektu są globalnie uporządkowane, to cały system zachowuje spójność podręczną.



Przykład na slajdzie (prezentowany dalej na kolejnych dwóch slajdach) ma na celu zaprezentowanie, że spójność sekwencyjna nie ma własności lokalności. W przykładzie proces p_1 dokonuje zapisów do 4 zmiennych: $x1$, $x2$, $y1$ i $y2$. Proces p_2 jedynie czyta te zmienne. Jak łatwo zauważyć obraz historii przetwarzania w procesie p_2 zawiera inną kolejność operacji zapisu do zmiennych $x2$ i $y1$ niż obraz w procesie p_1. Nie jest zachowana więc spójność sekwencyjna. Pomimo, że... (nast. slajd)



Rysunek przedstawia fragment przetwarzania z poprzedniego slajdu dotyczący operacji na zmiennych *x*1 i *x*2. Jak widać z obrazów historii przetwarzania operacje na takim podzbiorze zmiennych są spójne sekwencyjnie. Ale... (nast. slajd)



Rysunek przedstawia inny fragment przetwarzania z wcześniejszego slajdu dotyczący operacji na zmiennych *y*1 i *y*2. Jak widać z obrazów historii przetwarzania operacje na takim podzbiorze zmiennych również są spójne sekwencyjnie. Niestety na podstawie analizy dla tych dwóch podzbiorów zmiennych nie możemy powiedzieć, że system jako całość zachowuje spójność sekwencyjną. Oznacza to, że spójność sekwencyjna nie ma własności lokalności.

Wracając do pełnego przykładu – spójność atomowa posiada własność lokalności, ponieważ uwzględniane są uporządkowania operacji w czasie rzeczywistym. W takim przypadku kolejność operacji *w*(*x*2)1 i *w*(*y*1)1 z procesu p_1 musiałaby być zachowana w obrazie historii w procesie p_2.