Zad. 1

Lemat.

Dla (0 <= i <= n-1) jest co najwyżej n-i wątków na poziomie i.

Dowód: indukcja po poziomach.

1. i = 0: na poziomie 0 jest n – 0 wątków
2. Załóżmy, że na poziomie znajduje się co najwyżej ‘n – i’ wątków. Pokażemy, że na poziomie ‘i + 1’ znajduje się co najwyżej n – (i + 1) wątków.
   Załóżmy nie wprost, że na poziomie ‘i + 1’ znajduje się ‘n – i’ wątków.
   Niech A będzie ostatnim wątkiem, który ustawia victim[i+1]. Wtedy dla każdego innego wątku B na poziomie ‘i + 1’ zachodzi:
   write_b(victim[i + 1]) -> write_a(victim[i + 1]),
   zatem z kodu:
   write_b(level[B] = i + 1) -> write_b(victim[i + 1]) -> write_a(victim[i + 1]) -> read_a(level[B])

Ponieważ wątek B znajduje się na poziomie ‘i + 1’, to za każdym razem, gdy wątek A odczytuje wartość pola level[B], otrzymuje wartość ‘>= i + 1’. Z tego wynika, że wątek A nie mógł opuścić swojej pętli oczekiwania, co jest sprzeczne z naszym założeniem.

Zad. 2 ???

Dowód przez odwróconą indukcję.

1. Podstawa: Poziom ‘n – 1’ zawiera co najwyżej 1 wątek. Zagłodzenie na tym poziomie nie występuje.
2. Krok: Załóżmy, że każdy wątek, który dociera do poziomu ‘j+1’ lub wyżej w końcu dotrze do sekcji krytycznej. Załóżmy nie wprost, że pewien wątek A utknął na poziomie ‘j’. Wiemy z założenia indukcyjnego, że na wyższych poziomach nie ma żadnego wątku. Jak tylko A ustawi ‘level[A] = j’ to każdy wątek z poziomu ‘j – 1’ nie może wejść na poziom ‘j’. To oznacza, że żaden wątek nie wejdzie na poziom ‘j’ z niższych poziomów. Jeżeli A jest jedynym wątkiem, który utknął na poziomie ‘j’ to oznacza, że w końcu wejdzie na poziom ‘j+1’. W przeciwnym przypadku wiemy, że ‘victim[j]’ może przyjąć tylko jedną wartość, a zatem, któryś z zablokowanych wątków przejdzie o poziomu ‘j+1’.

Lemat: Dla każdego poziomu i (i < n) i wątku A:

- wątek przejdzie "kiedyś" na poziom (i+1).
- wątek przejdzie "kiedyś" przez wszystkie poziomy: (i+1)…(n-1)

Dowód ("odwrócona" indukcja):

1. Podstawa: poziom ‘n - 1’. Niech identyfikator wątku = A. Z poprzedniego zadania wiemy, że na poziomie ‘n-1’ jest tylko jeden wątek, czyli A.
2. Krok: poziom ‘k’ (gdzie k < n-1). Niech identyfikator wątku = A. Dwa przypadki:
   a) victim[k] != A -> OK.
   b) victim[k] == A.
   Z pkt 2. założenia indukcyjnego wynika, że "kiedyś" poziomy (k+1)…(n-1) zostaną opróżnione z wątków, które się tam znajdują lub zostanie ustawiona wartość victim[k] != A. Wynika stąd, że wątek A opuści "kiedyś" pętle while i przejdze na następny poziom. Stąd warunek 1. jest spełniony. Spełnienie warunku 2. wynika z założenia indukcyjnego.

Zad. 3

Rozpatrzmy wykonanie metody lock() dla wątków A, B i C

1. Write_A(level[A]=i) -> Write_A(victim[i]=A) -> …
2. Write_B(level[B]=i) -> Write_B(victim[i]=B) -> …
3. Write_C(level[C]=i) -> Write_C(victim[i]=C) -> …

W tej sytuacji wątek C jest ofiarą i będzie musiał przepuścić pozostałe wątki na kolejne piętra. Może dojść do takiej sytuacji, że po wyjściu z sekcji krytycznej wątki A i B ponownie będą chciały ponownie do niej wejść. Bez straty ogólności możemy założyć, że wykonają one pierwszy poziom metody lock() następująco: 1 -> 2,

Teraz wątek B jest ofiarą i musi przepuścić pozostałe wątki. Jeżeli wątek C, który wcześniej był ofiarą zostanie uśpiony to może dojść do sytacji w której w tym czasie watek A ponownie zacznie wykonywać metodę lock() po wyjściu z sekcji krytycznej i wykonując 1 przepuści wątek B, który był ofiarą i wątek C nie zdąży się wybudzić.

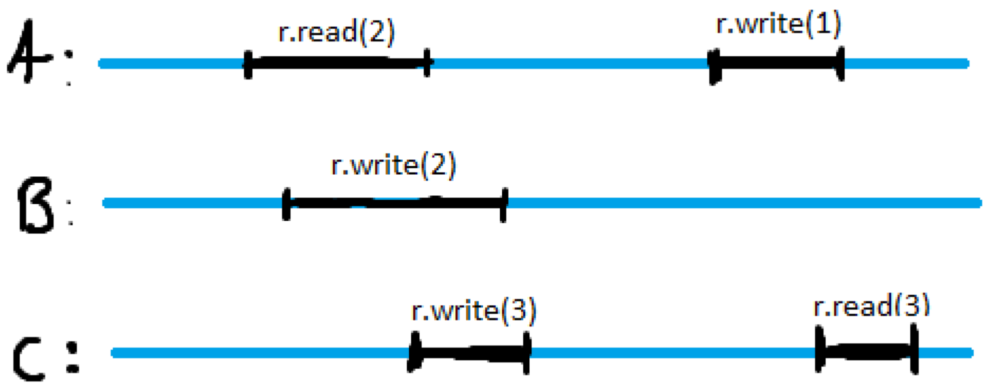
Mimo tego, że algorytm nie jest r -ograniczony to spełnia on własność niezagłodzenia, ponieważ gdy wątek C zostanie wybudzony nie będzie już ofiarą i będzie mógł wejść na wyższy poziom.

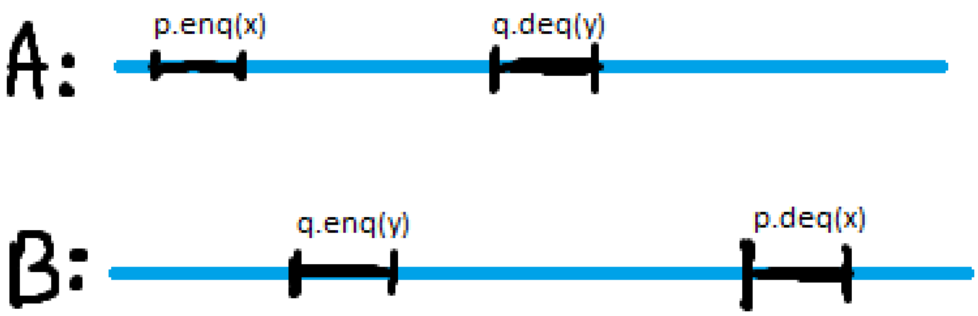
Zad. 4

1. Zakładamy nie wprost, że D_A -> D_B i CS_A -> CS_B, więc
   flag[A] = true - > victim = A -> flag[B] = true -> victim = B.
   Skoro B dostał się jako pierwszy do sekcji krytycznej to musiał odczytać 'victim = A' - sprzeczność.
2. W takim razie mamy:
   flag[A] = true -> flag[B] = true -> victim = B -> victim = A, czyli B może bez przeszkód wejść do CS przed A, mimo D_A -> D_B . Nie ma własności FCFS.
3. a) Sam odczyt nie starcza, bo nie rozróżniamy D_A -> D_B od D_B -> D_A
   b) Sam zapis do pliku nie wystarcza, ponieważ załóżmy D_A -> D_B w takim razie CS_A -> CS_B, dostajemy jakiś zapis zmiennych. Zauważmy, że w przypadku gdy D_B -> D_A to stan zmiennych jest taki sam - brak odczytów. A więc taki sam stan programu powinien raz wykonać CS_A -> CS_B, a raz CS_B -> CS_A.
   c) Sam zapis do komórki nie starczy, bo nie rozróżniamy D_A -> D_B od samego D_B.
4. Z powyższej analizy wynika, że zmodyfikowana definicja sekcji wejściowej (ograniczenie jej do jednej instrukcji) uniemożliwia uzyskanie jakiejkolwiek kolejności, co oznacza, że warunek 0-ograniczonego czekania jest niewykonalny. Dlatego taka definicja jest niepraktyczna i bezsensowna z punktu widzenia implementacji wzajemnego wykluczania.

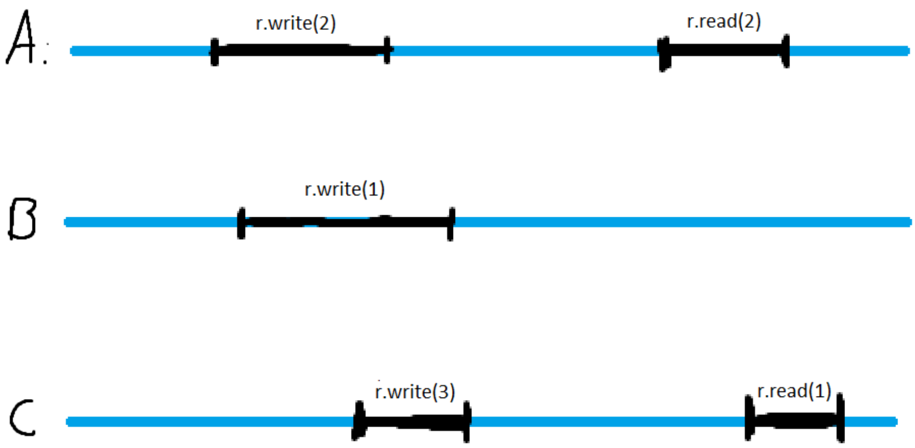
Zad. 5

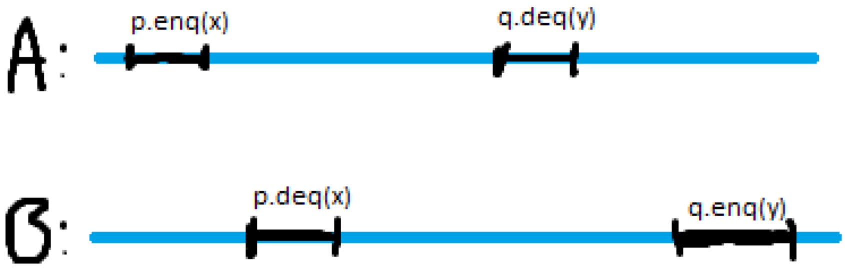
Spójne:





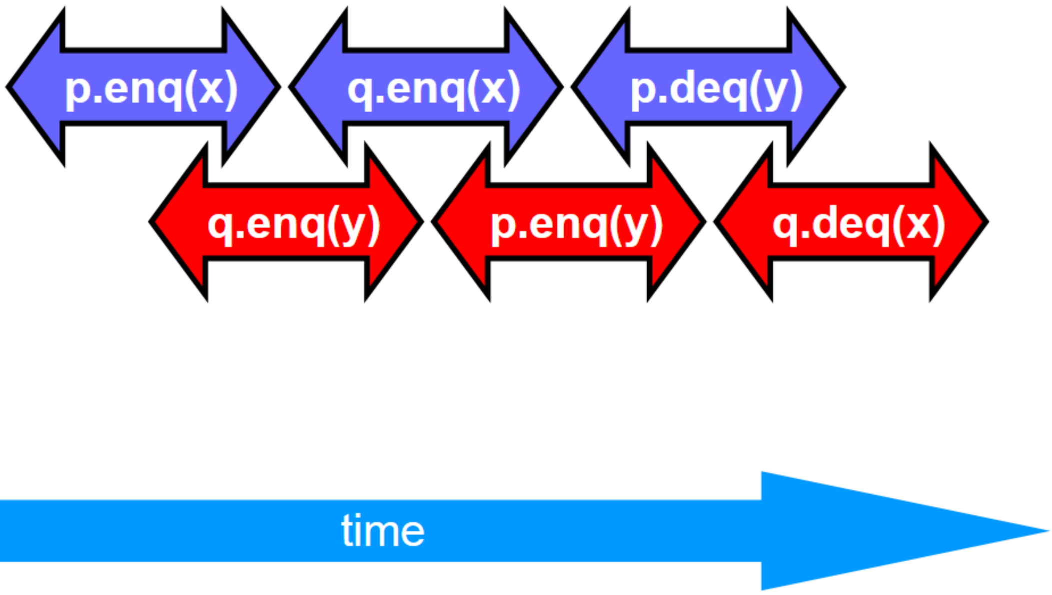
Niespójne:

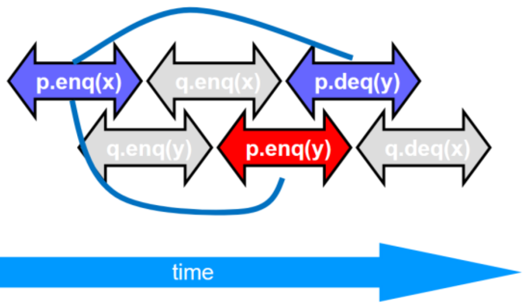


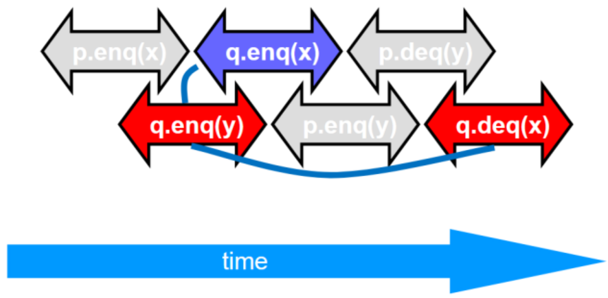


Zad. 6

Własność kompozycji mówi o tym, że jeżeli weźmiemy kilka sekwencyjnie spójnych obiektów to cała ich historia kompozycji będzie sekwencyjnie spójna.







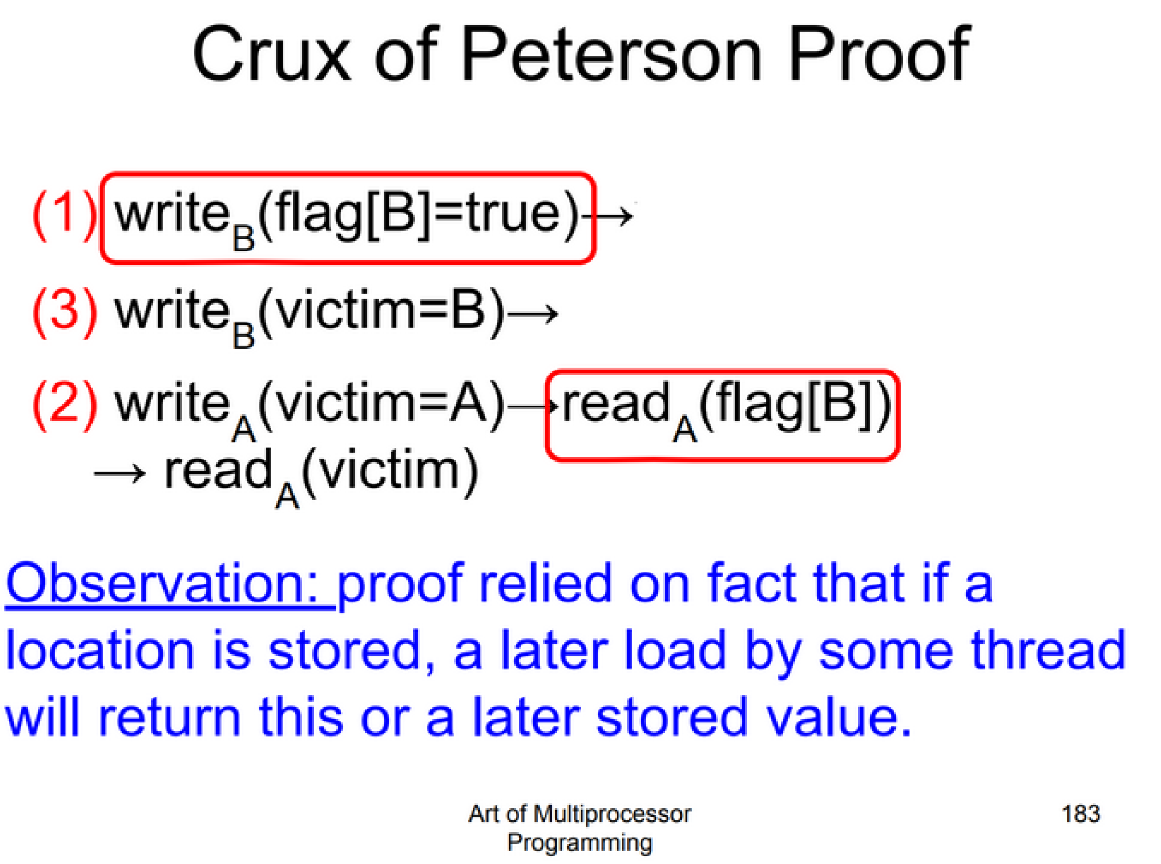
Jeżeli ‘p’ jest kolejką to ‘y’ musiałby być włożony do kolejki przed ‘x’ (B: p.enq(y) - > A: p.enq(x))
Dla kolejki ‘q’ – analogicznie ‘x’ musiałby być przed ‘y’ (A: q.enq(x) - > B: q.enq(y))

Jednak na podstawie programu widzimy, że powstaje nam cykl:

A: p.enq(x) - > A: q.enq(x), B: q.enq(y) - > B: p.enq(y)

Zatem jest to przykład, w którym obiekty p i q są sekwencyjnie spójne, lecz cała historia już nie.

Zad. 8



Jeżeli kod będzie wykonywany na procesorze o modelu pamięci słabszym niż sekwencyjna spójność może dojść do sytuacji:

read_A(flag[B] = false) -> read_B(flag[A] = false) -> write_A(flag[A] = true) -> write_B(flag[B] = true)

Przez co oba wątki wejdą do sekcji krytycznej.