Lista 12

tags: SO

[3] OSTEP

Zadanie 1

Zadanie 1. Zapoznaj się z poniższym programem. Rozważamy wartości przech «myid», «strtab», «vargp», «cnt», «argc» i «argv[0]». Określ czy są one **współ** będą źródłem **wyścigów** (ang. *data race*).

```
1  __thread long myid;
2  static char **strtab;
3
4  void *thread(void *vargp) {
5   myid = *(long *)vargp;
6   static int cnt = 0;
7   printf("[%ld]: %s (cnt=%d)\n", myid, strtab[myid], ++cnt);
8   return NULL;
9  }
10
11 int main(int argc, char *argv[]) {
12    ...
13    strtab = argv;
14    while (argc > 0) {
15      myid = --argc;
16      pthread_create(&tid, NULL, thread, (void *)&myid);
17    }
18    ...
19 }
```

::: info **zmienna współdzielona** – zmienna, do której odnoszą się conajmniej dwa wątki. **__thread** – każdy wątek dostaje osobną instancję obiektu, w ten sposób zapobiegamy potencjalnym wyścigom, gdy zmienna nie jest zamierzona jako obiekt współdzielony. **wyścig** – sytuacja, w której wynik programu zależy od przeplotu wątków. :::

Zmienna	Współdzielenie	Wyścigi
myid	✓	Tak, ponieważ modyfikujemy tę wartość i przekazujemy do niej referencję wątkom
strtab	✓	Nie, ponieważ wartość nie jest modyfikowana
vargp		
cnt	✓	Tak
argc		
argv[0]	✓?	Nie

Zadanie 2

Zadanie 2. Podaj definicję **sekcji krytycznej** [2, 6.2]. Następnie wymień i uzasadn spełniać rozwiązanie problemu sekcji krytycznej. Czemu w programach przestrze implementacji nie możemy używać **wyłączania przerwań** (ang. *interrupt disable*)? (*Amdahla* powiedz czemu programistom powinno zależeć na tym, by sekcje kryty najkrótsze – określa się to również mianem **blokowania drobnoziarnistego** (ang. *fi*

::: info **sekcja krytyczna** – fragment kodu, którego wykonania nie przeplatają się między wątkami.

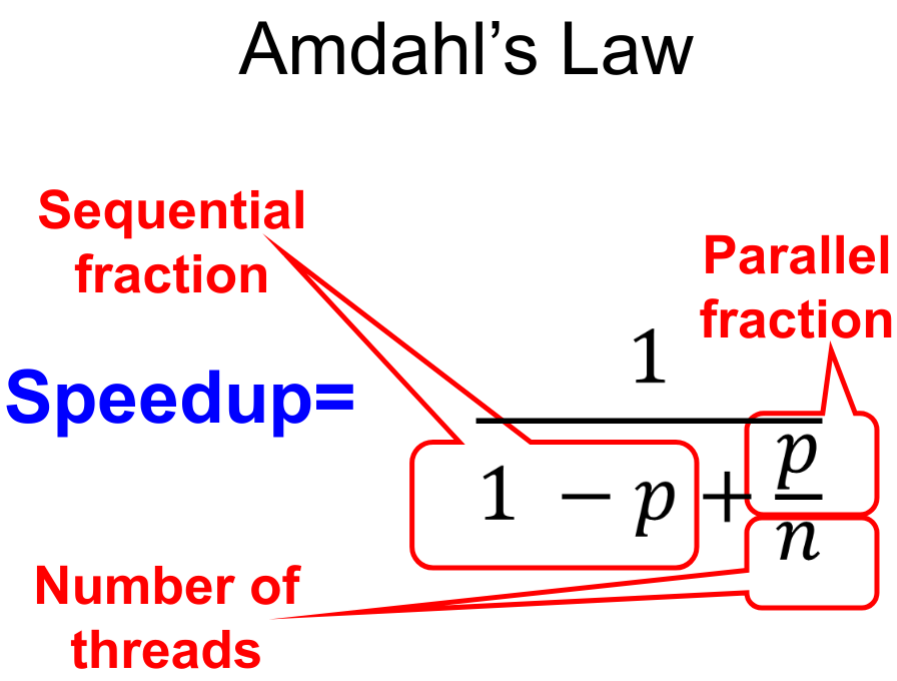
wyłączania przerwań – wyłączanie usługi przerwań na czas realizacji sekcji krytycznej. Niestety, przywrócenie obsługi przerwań mogłoby nigdy nie nastąpić (np. w wyniku błędu kodu sekcji krytycznej). :::

Założenia rozwiązania sekcji krytycznej

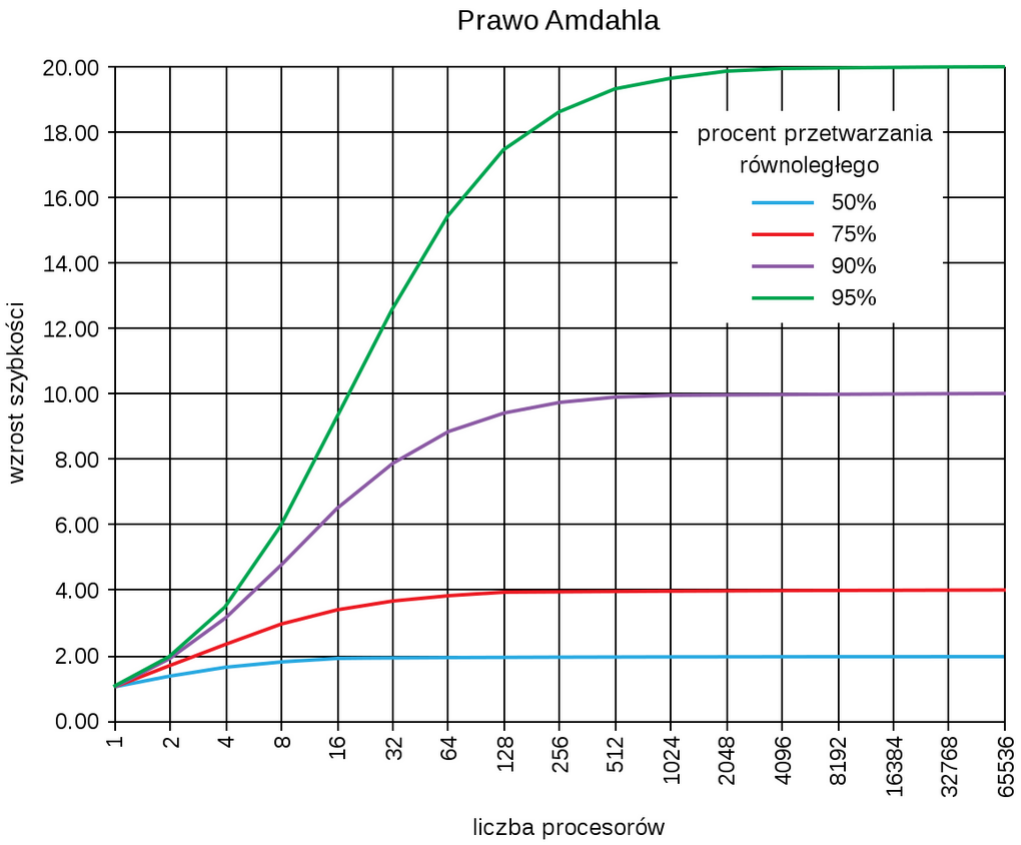
A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Prawo Amdahla i blokowanie drobnoziarniste



Wiadome jest, że program działa najszybciej, gdy możliwie największa jego część wykonuje się współbierzenie. W takim razie redukując liczbę i długość sekcji krytycznych (które wykonywane są sekwencyjnie) zwiększamy współbierzność i przyspieszamy program.



Zadanie 3

Zadanie 3. Podaj w pseudokodzie semantykę **instrukcji atomowej** `compare-and-` zaimplementuj **blokadę wirującą** (ang. *spin lock*) [3, 28.7]. Niech typ «`spin_t`» bę Podaj ciało procedur «`void lock(spin_t *)`» i «`void unlock(spin_t *)`». C nie jest **sprawiedliwa** (ang. *fair*) [3, 28.8]? Uruchamiamy n identycznych wątków wchodzi do sekcji krytycznej, po czym zostaje wywłaszczony przez jądro. Ile czasu za jednokrotne przejście przez sekcję krytyczną – algorytm planisty to **round-robin**, kv

::: info **instrukcja atomowa** – instrukcja niepodzielna, czyli też bezpieczna wątkowo (nie może się zdarzyć, że w trakcie jej wykonywania inny wątek przejmie sterowanie). **blokada wirująca** -- zablokowany w ten sposób wątek, wykonuje nieskończoną pętlę, w której sprawdza dostępność wyczekiwanego zasoby. Mówi się wtedy, że wątek aktywnie czeka na zasób. **sprawiedliwość** -- mówimy, że pewien model synchronizacji jest sprawiedliwy, jeśli każdy wątek ostatecznie otrzymuje dostęp do żądanych zasobów. :::

Compare-and-swap

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

Blokada wirująca

```
// 0 -> free, 1 -> busy
typedef int spin_t;

void lock(spin_t *lock) {
  while(compare_and_swap(lock, 0, 1) == 1); // spin
}

void unlock(spin_t *lock) {
  *lock = 0;
}
```

Czemu blokada nie jest sprawiedliwa?

The next axis is **fairness**. How fair is a spin lock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Simple spin locks (as discussed thus far) are not fair and may lead to starvation.

Uruchamiamy n wątków. Kolejno każdy z nich wchodzi do skcji krytycznej i zostaje wywłaszczony. Ile czasu zajmie wszystkim wątkom jednokrotne przejście przez sekcję krytyczną?

Założenia

- wykorzystujemy *round robin* jako algorytm planisty (dyspozytora)
- kwant czasu wynosi $1ms$
- blokady są wirujące
- ???

Round robin

Wątki są ułożone na liście cyklicznej. Algorytm konsekutywnie je uruchamia, przy czym każdy wątek otrzymuje sterowanie na pewien ustalony kwant czasu, po którego upłynięciu zostaje on wywłaszczony.

Rozwiązanie

Niech t oznacza czas wykonania ścieżki krytycznej. Rozpatrzmy i-ty wątek T_i. Wiemy, że $i-1$ wątków przeszło już sekcję krytyczną i zostało wywłaszczone oraz $n-i$ wątków wciąż czeka na swoją kolej.

Wykonując algorytm *round robin* na każdy kwant czasu spędzony w T_i przypada $n-i$ kwantów czasu dla pozostałych wątków (które pętlą się bezcelowo w swoim aktywnym czekaniu). A zatem czas przejścia ścieżki krytycznej przez T_i to $$ t_i = (t-1) \cdot (n-i+1) + 1 $$

Sumując po wszystkich t_i mamy $$ \sum_{i=1}^n t_i = n + (t-1)\sum_{i=1}^n(n-i+1) = n + (t-1)\sum_{i=1}^ni = n+ \frac{n(n+1)(t-1)}{2} $$ Morał z tej historii jest taki, że czas jest kwadratowy względem ilości wątków

Zadanie 4

Zadanie 4. Wiemy, że **aktywne czekanie** (ang. *busy waiting*) nie jest właściwym na zwolnienie blokady. Czemu oddanie czasu procesora funkcją «`yield`» [3, 28.13] ni problemów, które mieliśmy z blokadami wirującymi? Zreferuj implementację **blokad** w [3, 28.14]. Czemu jest ona niepoprawna bez użycia funkcji «`setpark`»?

yield pozwala wątkom na zrzeczenie się sterowania i pójście spać. Jako alternatywa do aktywnego czekania pozwala to nam zaoszczędzić na czasie procesora pomijając bezcelowe pętlenie.

Czemu oddanie sterowania funkcją *yield* nie rozwiązuje wszystkich problemów?

Czas jest nadal kwadratowy (zmienia się tylko stała). W żaden sposób nie zmieni się również problem głodzenia wątków.

Zreferuj implementację *blokad usypiających*

```
int TestAndSet(int *old_ptr, int new){
    /* Ustawia strażnika i zwraca
     * jego poprzedni stan dla testu */
    int old = *old_ptr;
    *old_ptr = new;
    return old;
}
```

```
typedef struct __lock_t {
  /* Blokada jest założona jeśli flaga jest ustawiona na 1
   * Widzimy, że mamy tu również kolejkę wątków i strażnika,
   * który chroni strukturę przed modyfikacją na czas
   * zakładania i zdejmowania blokady */
  int flag;
  int guard;
  queue_t *q;
} lock_t;


void lock_init(lock_t *m) {
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}

void lock(lock_t *m) {
  /* Pętl się dopóki strażnik pełni wartę */
  while (TestAndSet(&m->guard, 1) == 1);


  if (m->flag == 0) {
    /* Jeśli blokada nie jest założona to ją załóż i zwolnij strażnika */
    m->flag = 1;
    m->guard = 0;
  }

  else {
    /* W przeciwnym przypadku ustaw się w kolejce i idź spać */
    queue_add(m->q, gettid());    // dodaj wątek do kolejki
    setpark();                    // zasygnalizuj senność
    m->guard = 0;                 // zwolnij strażnika
    park();                       // zaśnij
  }
}

void unlock(lock_t *m) {
  /* Pętl się dopóki strażnik pełni wartę */
  while (TestAndSet(&m->guard, 1) == 1);

  if (queue_empty(m->q))
    /* Zwolnij blokadę, jeśli kolejka jest pusta */
    m->flag = 0;
  else
    /* Wybudź wątek z kolejkii zwolnij strażnika */
    unpark(queue_remove(m->q));
  m->guard = 0;
}
```

Zauważmy, że blokada jest przekazywana niejawnie do oczekujących w kolejce wątków (tj. flaga nie zmienia stanu w trakcie przekazywania kontroli nad blokadą -- wybudzony wątek jest już po warunku sprawdzającym flagę)

Zalety blokad usypiających

- Kolejka zapobiega zagładzaniu wątków
- Spędzamy mniej czasu na akytwnym czekaniu (czas sekcji krytycznej odnosi się teraz do mechanizmu blokady, a nie kodu użytkownika)

Czemu setpark?

setpark

```
By calling this routine, a thread can indicate it is about to park.
If it then happens to be interrupted and another thread calls unpark
before park is actually called, the subsequent park returns immediately
instead of sleeping.
```

```
queue_add(m->q, gettid());    // dodaj wątek do kolejki
// setpark();                 // zasygnalizuj senność
m->guard = 0;                 // zwolnij strażnika
park();
```

Może się zdarzyć, że między instruckją 3 i 4 wątek jest już w kolejce, ale nie jest uśpiony. Wystarczy teraz obudzić taki zbudzony wątek, który pierwsze co zrobi po odzyskaniu kontroli to pójdzie spać, ale nie ma już nikogo kto mógłby go z powrotem wybudzić.

Zadanie 5

Zadanie 5. Podaj cztery warunki konieczne do zaistnienia zakleszczenia. Na podst
w jaki sposób można **przeciwdziałać zakleszczeniom** (ang. *deadlock prevention*
stosowane w jądrze *Linux* i *Mimiker*, buduje graf skierowany, w którym wierzchoł
Jak `lockdep` wykrywa, że może wystąpić zakleszczenie? Z jakimi scenariuszami sob

Podpowiedź: Narzędzie `lockdep` jest przystępnie wyjaśnione w rozdziale 3.4 pracy licencjackiej *„Dy currency in Operating Systems”*[1] Jakuba Urbańczyka.

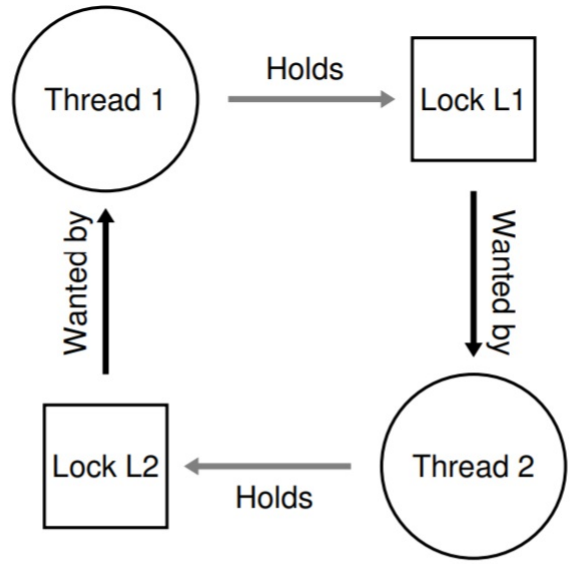


Figure 32.7: **The Deadlock Dependency Graph**

Warunki konieczne dla zakleszczenia (warunki Coffmana)

Conditions for Deadlock

Four conditions need to hold for a deadlock to occur [C+71]:

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur. Thus, we first explore techniques to *prevent* deadlock; each of these strategies seeks to prevent one of the above conditions from arising and thus is one approach to handling the deadlock problem.

Zapobieganie zakleszczeniom

Mutual exclusion

Unikamy blokad dla zasobów tylko do odczytu, stosujemy procedury atomowe.

Hold-and-wait

- Wątki mogą założyć blokady naraz i w sposób atomowy
- *Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.*

No preemption

Natrafiające na blokadę wątki wyzbywają się blokad wszystkich swoich zasobów i próbują nabyć je na nowo. (taka naduprzejmość może spowodować powstania livelocka).

Circular wait

Możemy nadać porządek wszystkim zasobom: $R_1 < R_2 < ... < R_n$. Proces chcący nabyć zasób R_i musi najpierw zwolnić wszystkie zasoby R_j takie, że $i >= j$.

Szkic dowodu Łuki grafu skierowanego to odpowiednio relacje posiadania i nabywania zasobów.

Załóżmy nie wprost, że w takim grafie wystąpił cykl $C = (R_i, P_i, R_{i+1}, P_{i+1}, ... R_j, P_j, R_i)$ Od razu widzimy, że dla każdego i, j zachodzi $R_i \not \equiv R_j$ (gdyby tak nie było to by znaczyło, że dwa procesy mają blokadę na tym samym zasobie). Z ustalonego wcześniej porządku wiemy również, że dla każdego i prawdą jest, że $R_i < R_{i+1}$. Otrzymujemy wtedy sprzeczność $R_i < ... < R_j$ z $R_j < R_i$.

Inne metody

- dyspozytor (np. algorytm bankiera)
- aktywnie wykrywamy i rozwiązujemy zakleszczenia

Lockdep

Definiuje się *klasy blokad* będące grupą blokad przestrzegającą tych samych zasad. Algorytm obserwuje zależności pomiędzy klasami i przy pewnych założeniach wnioskuje czy schematy zakładania blokad w programie mogą doprowadzić do deadlocka. W tym celu *lockdep* trzyma dwie listy dla każdej klasy L.

- lista *before* – przechowuje wszystkie klasy, których blokady były założone w momencie zakładania blokad L.
- lista *after* – przechowuje wszystkie klasy, których blokady były zakładane, podczas gdy blokada na L była założona.

Te dwie listy unikalnie wyznaczają kolejność nabywania blokad przez inne klasy względem L. Gdy blokada L jest pozyskana validator stwierdza czy występuje ryzyko zakleszczenia sprawdzając czy któraś z obecnie trzymanych blokad nie znajduje się na liście *after* klasy L. Jako że *after* lista dyktuje, które blokady mogą być zakładane tylko po nabyciu L, więc jeśli któraś z obecnych blokad jest na tej liście to znaczy, że ryzykujemy zakleszczenie.

Z jakimi scenariuszami sobie nie radzi?

Lockdep dokonuje osądu pesymistycznie, tj. może otrzegać przed zakleszczeniem, gdy w rzeczywistości nie może do niego dojść (ale już nie na odwrót, oczywiście). W przykładzie ze źrodła widzimy, że do zakleszczenia nie może dojść, ponieważ żaden wierzchołek nie może być jednocześnie korzeniem i nim nie być. Ten stan rzeczy zaburza jednak dyktowany przez listy porządek nabywania blokad.

```
struct node_data {
  ...
  mutex_t lock;
};
struct tree_node {
  ...
  struct node_data *data;
  mutex_t lock;
};

void lock_tree_node_and_storage(struct tree_node *A) {
  if (is_root(A)) {
    mutex_lock(A->lock);
    mutex_lock(A->data->lock);
  } else {
    mutex_lock(A->data->lock);
    mutex_lock(A->lock);
  }
}
```

Zadanie 6

Zadanie 6. Poniżej znajduje się propozycja[2] programowego rozwiązania problemu **wz**
(ang. *mutual exclusion*) dla dwóch procesów. Znajdź kontrprzykład, w którym to ro

```
1  shared boolean blocked [2] = { false, false };
2  shared int turn = 0;
3
4  void P (int id) {
5    while (true) {
6      blocked[id] = true;
7      while (turn != id) {
8        while (blocked[1 - id])
9          continue;
10       turn = id;
11     }
12     /* put code to execute in critical section here */
13     blocked[id] = false;
14   }
15 }
16
17 void main() { parbegin (P(0), P(1)); }
```

Ciekawostka: Okazuje się, że nawet recenzenci renomowanego czasopisma „*Communications of t*

Mamy środowisko dwuwątkowe. Synchronizacja odbywałaby się tak, że wątek chcący nabyć zasób najpierw oświadcza swoją chęć nabycia flagą *blocked*. Jeśli tura przypada innemu wątkowi to czeka, aż tamten wątek zwolni swoją blokadę *blocked* po czym ustawia turę na swoją. Wątek, któremu przypada tura może natychmiastowo przejść do sekcji krytycznej, po której zakończeniu zrzeka się blokady chowając flagę *blocked*.

Kontrprzykład

```
[1]: <4, 9>
// wątek 1 wchodzi do pętli wewnętrznej,
// ale zostaje wywłaszczony przed ustawienie tury
[0]: <4, 12>
// wątek 0 wchodzi do sekcji krytycznej natychmiast
// ponieważ wciąż jest jego tura
[1]: <10, 12>
// wątek 1 również wchodzi do sekcji krytycznej
```

Zadanie 7

Zadanie 7. Algorytm Petersona[3] rozwiązuje programowo problem wzajemnego wykluc
wersję implementacji tego algorytmu dla dwóch procesów. Uzasadnij jego poprawno

```
1  shared boolean blocked [2] = { false, false };
2  shared int turn = 0;
3
4  void P (int id) {
5    while (true) {
6      blocked[id] = true;
7      turn = 1 - id;
8      while (blocked[1 - id] && turn == (1 - id))
9        continue;
10     /* put code to execute in critical section here */
11     blocked[id] = false;
12   }
13 }
14
15 void main() { parbegin (P(0), P(1)); }
```

Ciekawostka: Czasami ten algorytm stosuje się w praktyce dla architektur bez instrukcji atomowyc

```
 shared boolean blocked [2] = { false, false };
shared int turn = 0;

void P (int id) {
  while (true) {
    blocked[id] = true;
    turn = 1 - id;
    while (blocked[1 - id] && turn == (1 - id))
      continue;
    /* put code to execute in critical section here */
    blocked[id] = false;
  }
}

void main() { parbegin(P(0), P(1)); }
```

Dowód nie wprost. Załóżmy, że oba wątki znajdują się w sekcji krytycznej i przyjrzyjmy się ostatniemu przebiegowi algorytmu.

$$ write_0(blocked[0]=true) → write_0(turn=1) → read_0(blocked[1]) → read_0(turn)$$

$$ write_1(blocked[1]=true) → write_1(turn=0) → read_1(blocked[0]) → read_1(turn)$$

Załóżmy bez stray ogólności, że [0] był jako ostatni wykonał zapis do $turn$.

$$ write_1(turn=0) → write_0(turn=1)$$

Stąd wynika, że zaobserwowana przez [0] wartość jest taka, jak wartość ustawiona ostatnio stąd wiemy, że

$$ write_0(turn=1) → read_0(turn == 1) $$

Wątek [0] ostatecznie przeszedł jednak do sekcji krytycznej, czyli `blocked[1 - id] && turn == (1 - id)` obliczyło się do fałszu. Wiemy, że $turn == 1$, co implikuje $blocked[1] == false$.

$$ write_0(turn = 1) → read_0(blocked[1] == false) → read_0(turn == 1) $$

Dopasowując nowe fakty do ogólnego przebiegu wątku [0] wnioskujemy, że

$$ write_1(blocked[1] = true) → write_1(turn=0)→write_0(turn=1)→read_0(blocked[1]==false) $$

Zauważmy, że

$$ write_1(blocked[1] = true) → read_0(blocked[1] == false) $$

To jest sprzeczność, ponieważ wiemy, że żaden inny zapis do `blocked[1]` nie miał miejsca przed wejściem do ścieżek krytycznych.

Zadanie 8

Zadanie 8. Poniżej podano błędną implementację **semafora zliczającego** przy por
nych. Wartość «count» może być ujemna – wartość bezwzględna oznacza wtedy lic
Znajdź kontrprzykład i zaprezentuj wszystkie warunki niezbędne do jego odtworzeni

```
1  struct csem {
2    bsem mutex;
3    bsem delay;
4    int count;
5  };
6
7  void csem::csem(int v) {
8    mutex = 1;
9    delay = 0;
10   count = v;
11 }
```

```
13 void csem::P() {
14   P(mutex);
15   count--;
16   if (count < 0) {
17     V(mutex);
18     P(delay);
19   } else {
20     V(mutex);
21   }
22 }
```

```
23 void cs
24   P(mut
25   count
26   if (c
27     V(d
28   V(mut
29 }
```

::: info **semafor zliczający** – pozwala na dostęp dla wielu wątków jednocześnie. **semafor binarny** – szczególny przypadek semafora zliczającego dla jednego wątku (różni się od mutexa tym, że nie posiada właściela). :::

```
struct csem {
  bsem mutex;
  bsem delay;
  int count;
};

void csem::csem(int v) {
  mutex = 1;
  delay = 0;
  count = v;
}

void csem::P() { // take
  P(mutex);
  count--;
  if (count < 0) {
    V(mutex);
    P(delay);
  } else {
    V(mutex);
  }
}

void csem::V() { // give
  P(mutex);
  count++;
  if (count <= 0)
    V(delay);
  V(mutex);
}
```

Kontrprzykład

1. Semafor jest początkowo pusty.
2. Wątki A i B wykonują po kolei część procedury P() po czym są wywłaszczane. Osatecznie count będzie wynosił -2.

```
P(mutex);
count--;
if (count < 0) {
    V(mutex);
    // P(delay); <- tutaj normalnie byśmy zasnęli,
    // ale wątek zostaje wywłaszczony
```

3. Wątki C i D wykonują V(). Po wykonaniu obu procedur count będzie z powrotem równy 0.

```
P(mutex);
count++;
if (count <= 0)
  V(delay);
V(mutex);

// count <= 0 dla obu wątków,
// więc wykonujemy V(delay) dwukrotnie
// ale delay to semafor binarny,
// czyli potrafimy nadać mu stan 1, ale już nie 2
// tym sposobem gubimy jeden żeton
```

4. Kontrolę otrzymuje wątek A, który woła P(delay). Od teraz *delay* == 0.
5. Teraz wykonanie wznawia wątek B i również woła P(delay), ale *delay* == 0, więc zasypia. Wtedy *count* == 0, ale mamy śpiący wątek!

Zadanie 9

Zadanie 9. Przeanalizuj poniższy pseudokod wadliwego rozwiązania problemu p
Zakładamy, że kolejka «queue» przechowuje do n elementów. Wszystkie operacje
Startujemy po jednym wątku wykonującym kod procedury «producer» i «consume
usypia wołający wątek, a «wakeup» budzi wątek wykonujący daną procedurę. Wskaż
doprowadzi do (a) błędu wykonania w linii 6 i 13 (b) zakleszczenia w liniach 5 i 12.

```
1  def producer():
2    while True:
3      item = produce()
4      if queue.full():
5        sleep()
6      queue.push(item)
7      if not queue.empty():
8        wakeup(consumer)
```

```
9  def consumer():
10   while True:
11     if queue.empty():
12       sleep()
13     item = queue.pop()
14     if not queue.full()
15       wakeup(producer)
16     consume(item)
```

Wskazówka: Jedna z usterek na którą się natkniesz jest znana jako problem zagubionej pobudki (a

błąd wykonania

:::spoiler *producer*

```
queue = [#####]

[consumer] <9,14>:
    empty?
    pop => [#### ]
    not full?
        ~> wakeup

[producer] <1,8>
    ...
    push => [#####]
    ...

[producer] <1,5>
    full?
        sleep

[consumer] <15>:
    wakeup

[producer] <6>:
    push [*]
```

:::

:::spoiler *consumer*

```
queue = [     ]

[producer] <1,7>:
    full?
    push => [#    ]
    not empty?
        ~> wakeup

[consumer] <9, 16>:
    ...
    pop => [     ]
    ...

[consumer] <9, 12>:
    empty?
        sleep

[producer] <8>:
    wakeup

[consumer] <13>:
    pop [*]
```

:::

zakleszczenia

```
queue = [#####]

[producer] <1, 5>:
    full?
        sleep

[consumer] <9,16>:
    ...
    pop => [#### ]
    ...

...

[consumer] <9,16>:
    ...
    pop => [     ]
    ...

[consumer] <9,12>:
    empty?
        sleep
```