Synchronizacja

Ostatnia modyfikacja: 11.03.2020

Wprowadzenie

- Współbieżny dostęp do współdzielonych danych może powodować niespójność danych.

- Utrzymanie spójności danych wymaga mechanizmów zapewniających uporządkowane wykonywanie współpracujących procesów.

- Przykład: problem producenta-konsumenta z ograniczonym buforem o pojemności *n*. Rozwiązanie (na następnym slajdzie) wykorzystuje współdzieloną zmienną *counter*, o wartości początkowej 0. Wartość zmiennej jest powiększana przy każdym dodaniu nowego obiektu do bufora, a zmniejszana przy usunięciu obiektu z bufora.

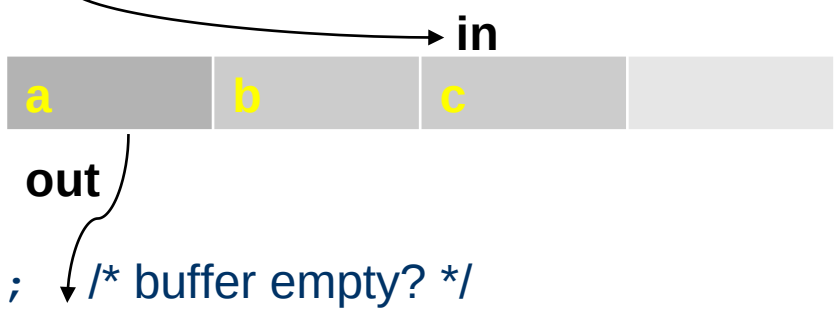
Producent-konsument (i bufor)

- **Dane współdzielone**

```
#define BUFFER_SIZE 4
typedef struct {      . . . } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0, counter = 0;
```

- **Producent**

```
while (true) {
        /* produce an item in next_produced */
        while (counter == BUFFER_SIZE) ;/* buffer full? */
        buffer [in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```



- **Konsument**

```
while (true) {
        while (counter == 0) ; /* buffer empty? */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next_consumed */
}
```

Producent-konsument - implementacja

- Dla poprawności działania powyższych algorytmów instrukcje:

  ```
  counter++;
  counter--;
  ```

  muszą być wykonywane **niepodzielnie** (ang. *atomically)*, tzn. wykonywanie tych instrukcji nie może być przerwane.

- Przykładowa realizacja operacji **counter++** oraz **counter-**

```
R1 = counter
R1 = R1 + 1
counter = R1
```

```
R2 = counter
R2 = R2 - 1
counter = R2
```

Producent-konsument - wyścigi

- Przykładowo niech początkowo **counter=3**, a procesor wykonuje kod procesów producenta i konsumenta w następującej kolejności:

producent:	**R1= counter**	(R1 = 3)
producent:	**R1 = R1 + 1**	(R1 = 4)
konsument:	**R2 = counter**	(R2 = 3)
konsument:	**R2 = R2 – 1**	(R2 = 2)
producent:	**counter = R1**	(counter = 4)
konsument:	**counter = R2**	(counter = 2)

- Zmienna **counter** może uzyskać wartość **4** lub **6** (zależnie od tego który z procesów zapisze do zmiennej ostatni), ale nie wartość poprawną (**5**).

- **Wyścigi** - sytuacja w której dwa lub więcej procesów zmieniają współbieżnie wspólną zmienną i gdy końcowa wartość zmiennej zależy od tego który z procesów zmodyfikuje zmienną ostatni.
- Dla zapobieżenia wyścigom procesy współbieżne muszą **synchronizować** swoje działanie.

Problem sekcji krytycznej

- **n** procesów współzawodniczy w dostępie do współdzielonej danej
- W każdym z procesów istnieje fragment kodu, nazywany sekcją krytyczną (ang. *critical section)*, w którym występuje dostęp do współdzielonych danych.
- Problem – zagwarantować, że w przypadku gdy jeden proces wykonuje kod sekcji krytycznej - żaden inny proces nie może znajdywać się w swojej sekcji krytycznej.
- Struktura procesu P_i

sekcja wejściowa

sekcja krytyczna

sekcja wyjściowa

reszta

Rozwiązanie problemu sekcji krytycznej

1. **Wzajemne wykluczanie** (**mutual exclusion**). Jeśli proces *Pi* jest w swojej sekcji krytycznej, to żaden inny proces nie działa w sekcji krytycznej.

2. **Postęp** (**progress**). Jeśli żaden proces nie działa w sekcji krytycznej, oraz istnieją procesy, które chcą wejść do sekcji krytycznych, to tylko procesy nie wykonujące swoich reszt mogą kandydować do wejścia do sekcji krytycznej i wybór ten nie może być odwlekany w nieskończoność.

3. **Ograniczone czekanie** (**bounded waiting**). Musi istnieć ograniczenie na liczbę wejść innych procesów do ich sekcji krytycznych po tym, gdy dany proces zgłosił chęć wejścia do swojej sekcji krytycznej i zanim uzyskał na to pozwolenie.
   - Zakładamy niezerową szybkość wykonywania procesów
   - Nie zakładamy relacji szybkości procesów.

Algorytm Petersona

- Tylko 2 procesy, P_0 oraz P_1
- Założenie: operacje maszynowe **`load`** i **`store`** są atomowe
- Dane współdzielone:

```
int znacznik[2], numer;
```

- Proces P_i (i=0 lub 1)

```
    ...
znacznik [i] = true;
numer = j;  // Uwaga: j == !i, tzn. j=1 dla P0 i j=0 dla P1
while (znacznik [j] && numer == j) ;
/* sekcja krytyczna */
...
znacznik[i] = false;
/* reszta */
    ...
```

- Algorytm spełnia wszystkie **3** warunki poprawności - rozwiązując problem sekcji krytycznej dla dwóch procesów.

Sprzętowe wspomaganie synchronizacji

- Systemy zazwyczaj posiadają środki do sprzętowego wspomagania obsługi sekcji krytycznej.
- Wszystkie omawiane dalej rozwiązania wykorzystują **blokowanie (locking)**
  - Obszary krytyczne są zabezpieczone przez blokady dostępu
- Systemy z jednym procesorem –mogą wyłączyć obsługę przerwań
  - Aktualnie wykonywany kod nie zostanie wywłaszczony (running without preemption)
  - Podejście zazwyczaj nieefektywne w systemach wieloprocesorowych
    - Systemy wykorzystujące wyłączanie przerwań nie są dobrze skalowalne

Współczesne procesory są wyposażone w specjalne instrukcje

- **Atomowe** = nieprzerywalne
  - Testowanie słowa procesora i jednoczesne ustawienie wartości innego słowa
  - Zamiana wartość dwóch słów w pamięci

Rozwiązanie problemu sekcji krytycznej za pomocą blokady

Ogólny schemat rozwiązania problemu sekcji krytycznej z blokadą (**lock**), którą może zająć tylko jeden (z dwóch) procesów. Podstawowe operacje

- Zajmij (**acquire**)
- Zwolnij (**release**)

```
. . .
acquire lock
        sekcja krytyczna
release lock
        reszta
. . .
```

- Dla większej liczby procesów powyższy schemat zapewnia jedynie **wzajemne wykluczanie**

Sprzętowe wspomaganie synchronizacji (c.d.)

- Nierozdzielna zamiana wartości dwóch zmiennych

```
void Zamień(boolean &a, boolean &b) {
    boolean temp = a; a = b; b = temp;
}
```

Wykorzystanie do ***wzajemnego wykluczania***:

- Dana dzielone: `boolean zamek=false;`
- Proces P_i

```
boolean klucz= true; /* dana lokalna */
do{
  Zamień(zamek,klucz) ;
    } while(klucz) ;
    /* sekcja krytyczna */
  . . .
  zamek = false;
    /* reszta */
```

Sprzętowe wspomaganie synchronizacji

- Wyłączanie przerwań na czas wykonywania sekcji krytycznej
- **Testuj i zmieniaj nierozdzielnie** zawartość słowa maszynowego.

```
boolean Testuj_i_Ustal(boolean &cel) {
    boolean rv = cel;
    cel = true;
    return rv;
}
```

Wykorzystanie do realizacji ***wzajemnego wykluczania***:

- Zmienna dzielona: `boolean zamek = false;`
- Proces P_i

```
    . . .
while (Testuj_i_Ustal(zamek)) ;
  . . ./* sekcja krytyczna */

zamek = false;/* zwolnienie blokady */
    . . . /*     reszta */
```

Wzajemne wykluczanie z ograniczonym czekaniem

- Współdzielone: `boolean czekanie[n], zamek;`
- Kod procesu nr ***i*** (spośród ***n***)

```
/* sekcja wejściowa */
    czekanie[i] = true; klucz = true;
    while ( czekanie[i] && klucz )
          klucz=Testuj_i_Ustal(zamek);
    czekanie[i] = false;
     /* sekcja krytyczna */
          ...
    /* sekcja wyjściowa */
    j = (i+1) % n;
    while ( j != i && ! czekanie[j] ) j= (j+1) % n;
    if ( j == i ) zamek = false;
    else czekanie[j] = false;
      /* reszta */
          ...
```

Semafor

- Mechanizm synchronizacji nie wymagający aktywnego czekania.
- Semafor *S* zawiera licznik - liczbę całkowitą, którą można modyfikować jedynie za pomocą dwóch **nierozdzielnych operacji** (oryginalnie nazwanych **P** i **V**):

```
wait (S):  while (S.value≤ 0) do ;
           S .value--;

signal (S): S .value++;
```

Przykład: sekcja krytyczna dla *n* procesów

- Zmienne dzielone
  - `semaphore mutex;`
  - początkowo `mutex.value = 1`
- Proces P_i

```
        ...

      wait(mutex);

    /* sekcja krytyczna  */
        ...

      signal(mutex);

      /* reszta */
    ...
```

Implementacja semaforów

- Trzeba zagwarantować, że żadna para procesów nie może jednocześnie wykonać **`wait()`** i **`signal()`** na tym samym semaforze.
- Tak więc wykonywanie funkcji **`wait()`** and **`signal()`** musi odbywać się pod ochroną sekcji krytycznej
  - Można w tej sekcji krytycznej korzystać z odpytywania (**busy waiting**) jeżeli chroniony kod jest krótki, bo wówczas mała jest strata cykli procesora
- Aplikacje mogą spędzać w swoich sekcjach krytycznych dużo czasu, więc odpytywanie nie jest wówczas wskazane.

Implementacja semafora bez aktywnego czekania

- Związemy z semaforem rekord (strukturę)

```
typedef struct {
    int value;
    struct process *L; /* lista procesów oczekujących */
} semaphore;
```

- Załóżmy dwie proste operacje:
  - **block** zawiesza proces, który ją wywołał.
  - **wakeup(P)** wznawia wykonanie zawieszonego procesu **P**.

Implementacja semafora - c.d.

- Operacje semaforowe zdefiniowane są następująco:

```
wait(semaphore *S):
    S->value--;
    if (S->value < 0) {
            /* dodaj ten proces do listy S->L i uśpij */
            block();
    }
signal(semaphore *S):
    S->value++;
    if (S->value <= 0) {
    /* usuń  process P z listy S->L i obudź go */
      wakeup(P);
    }
```

Semafor - uniwersalny mechanizm synchronizacji

- Zadanie: Wykonaj ***B*** w P_j dopiero po tym, jak się wykona *A* w P_i
- Rozwiązanie:
  - Zainicjuj wartość semafora **flag** przez 0
  - Kod:

P_i	P_j
⋮	⋮
A	**wait(flag)**
signal(flag)	*B*

Blokada i głodzenie procesów

- **Blokada** – dwa lub więcej procesów czekają bezterminowo na zdarzenie, które może być jedynie wytworzone przez jeden z oczekujących procesów.
- Niech *S* oraz *Q* będą dwoma semaforami zainicjowanymi przez 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
⋮	⋮
`signal(S);`	`signal(Q);`
`signal(Q)`	`signal(S);`

- **Głodzenie** – czas oczekiwania procesu na semafor jest bardzo długi lub nieograniczony wskutek niewłaściwego szeregowania procesów oczekujących (np. szeregowanie LIFO, bądź statyczne priorytetowe).
- **Inwersja priorytetów** – niskopriorytetowy proces, który zajął semafor potrzebny wykonującemu się procesowi o wyższym priorytecie, powoduje wstrzymanie tego drugiego (jak gdyby zyskiwał na „ważności”).
  - Rozwiązanie: **dziedziczenie priorytetów** (**priority inheritance**)

Semafory zliczające

- **Semafor zliczający** *(wielowartościowy)* – zmienna całkowita może przybierać wartości z (jednostronnie) nieograniczonego przedziału wartości.
- **Semafor binarny** – zmienna może przybierać wartości 0 oraz 1; może być prostszy do implementacji.
- Semafor zliczający **S** można implementować za pomocą dwóch binarnych.

Struktury danych:

```
binary-semaphore S1, S2;
int C;
```

Inicjalizacja: `S1.value = 1;  S2.value = 0;` `C =` początkowa wartość ***S***

Wait(S){

```
    wait(S1);
    C--;
    if (C < 0) {
        signal(S1);
        wait(S2);
    }
    signal(S1);
}
```

Signal(S){

```
    wait(S1);
    C ++;
    if (C <= 0) signal(S2);
    else signal(S1);
}
```

Klasyczne problemy synchronizacji

- Problem ograniczonego buforowania
- Problem czytelników i pisarzy
- Problem obiadujących filozofów

Problem ograniczonego buforowania

- Dane dzielone

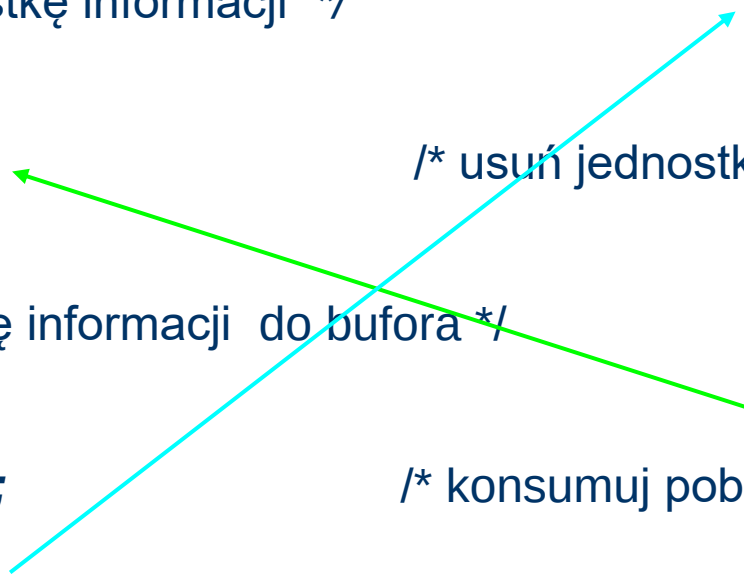
semaphore pełny, pusty, mutex;
SemInit(pełny,0); SemInit(pusty,n); SemInit(mutex,1);

- Proces producenta

```
do {
/* wytwórz jednostkę informacji  */
   …
   wait(pusty);
   wait(mutex);
/* dodaj jednostkę informacji  do bufora */
   …
   signal(mutex);
   signal(pełny);
} while (1);
```

Proces konsumenta

```
do {
   wait(pełny);
   wait(mutex);
/* usuń jednostkę informacji z bufora */
   …
   signal(mutex);
   signal(pusty);
/* konsumuj pobraną jednostkę informacji  */
   …
} while (1);
```



Problem czytelników i pisarzy

- Dane są wspólne dla wielu wykonujących się współbieżnie procesów
  - **Czytelnicy** – jedynie odczytują dane, nie wykonują żadnych modyfikacji danych
  - **Pisarze** – odczytują i modyfikują dane
- Problem
  - Należy umożliwić czytelnikom współbieżny odczyt danych – o ile danych nie modyfikuje pisarz
  - Tylko jeden pisarz może w danej chwili modyfikować dane.
- Można rozważać kilka wariantów sformułowania problemu czytelników i pisarzy:
  - **Pierwszy wariant** –żaden czytelnik nie jest powstrzymywany przed rozpoczęciem czytania o ile aktualnie pisarz nie ma prawa wykorzystywania danych
  - **Drugi wariant** – pisarz rozpoczyna czytanie jak najszybciej.
  - W obydwóch wariantach istnieje możliwość głodzenia -> sformułowano dalsze warianty …
  - W niektórych systemach rozwiązywanie problemu czytelników i pisarzy jest wspierane przez jądro (**reader-writer locks**)

Problem czytelników i pisarzy

- Dane dzielone: **semaphore mutex, pisanie; int czyt;**
  początkowo: **SemInit(mutex,1); SemInit(pisanie,1); czyt = 0**
- Proces pisarza

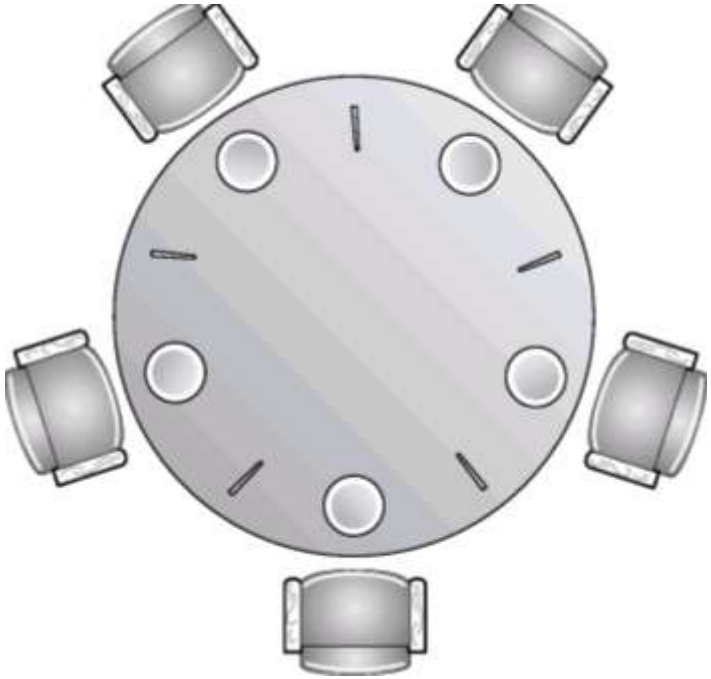
```
wait(pisanie);
 …
/* tu następuje pisanie */

…
signal(pisanie);
```

Proces czytelnika

```
 wait(mutex);
 czyt++;
if (czyt == 1)
      wait(pisanie);
signal(mutex);
 /* tu następuje czytanie */
 …
 wait(mutex);
   czyt--;
   if (czyt == 0)
    signal(pisanie);
 signal(mutex):
```

Problem obiadujących filozofów



- Dane dzielone

  semaphore pałeczka[5]; /* początkowo =1 */

Problem obiadujących filozofów (c.d.)

- Filozof *i*:

```
do {
        wait(pałeczka[i]);
        wait(pałeczka[(i+1) % 5])
        …
        /* spożywaj posiłek */
        …
        signal(pałeczka[i]);
        signal(pałeczka[(i+1) % 5]);
        …
           /* filozofuj */
        …
      } while (1);
```

Problem obiadujących filozofów (c.d.)

- Przeciwdziałanie blokadzie
  - Dopuść jedynie 4 filozofów do stołu
  - Pozwól podnieść pałeczki jedynie wtedy, gdy jest dostępna jedna z lewej i jedna z drugiej strony (podnoszenie pałeczek musi dokonywać się w sekcji krytycznej).
  - Wykorzystaj rozwiązanie asymetryczne – każdy nieparzysty filozof podnosi najpierw pałeczkę ze swej lewej strony, a potem z prawej. Dla parzystych filozofów kolejność jest odwrotna.
- Rozwiązania, które zabezpieczają przed blokadą, nie muszą zabezpieczać przed głodzeniem.

Niepoprawne użycie semaforów

- Niepoprawne użycie semaforów:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Pominięcie wait (mutex) czy signal (mutex) (lub obydwóch)
- Semafory nie chronią przed blokadą czy głodzeniem procesów.

Monitory (*)

- Monitor jest konstrukcją synchronizującą wysokiego poziomu, pozwalającą na bezpieczne dzielenie dostępu do abstrakcyjnych typów danych przez wiele współbieżnych procesów.

```
type monitor-name = monitor
        deklaracje zmiennych
        procedure entry P1 :(…);
                begin … end;
        procedure entry P2(…);
                begin … end;
                   ⋮
        procedure entry Pn (…);
                begin…end;
        begin
                kod inicjalizacji
        end
```

- Dalsze informacje – w podręczniku

Synchronizacja - implementacje

- Systemy zgodne z POSIX
- Linux
- MS Windows

Systemy zgodne z POSIX

- Interfejs UNIX System V – **zestawy semaforów**
- Nowszy interfejs **semaforów posixowych**
  - **semafory nazwane**
  - **semafory nienazwane**
- **Muteksy**
- **Zmienne warunku** (*condition variables)*
- **Bariera**
- **Blokady czytelników-pisarzy** (Read-Write locks)

Linux - synchronizacja

- Linux:
  - Jądro do wersji 2.6 wyłączało przerwania dla krótkich sekcji krytycznych
  - Jądra od wersji 2.6 są w pełni wywłaszczalne
- Linux dostarcza:
  - **semaforów**
  - **atomowych operacji na liczbach całkowitych**
  - **blokad aktywnych** (*spinlocks)*
  - **blokad czytelników-pisarzy**
- W systemach z pojedynczym procesorem blokady aktywne są zastąpione przez włączanie/wyłączanie wywłaszczania jądra.

Synchronizacja w MS Windows

- W systemach jednoprocesorowych wykorzystuje się maskowanie przerwań do zabezpieczenia dostępu do zasobów globalnych.
- Wykorzystywane są **blokady aktywne** (*spinlocks)* w systemach wieloprocesorowych.
- Różnorodne obiekty synchronizacji są dostępne dla programisty:
  - **zdarzenia** (**events**) – przyjmują jeden z dwóch stanów: aktywny i wyłączony. Służą do synchronizacji jednostronnej – jeden wątek zmienia stan obiektu, podczas gdy drugi wątek oczekuje na stan aktywny. Są dwa rodzaje obiektów **zdarzenie:** wyłączany automatycznie (przepuszczenie tylko jednego wątku przez funkcję oczekującą na stan aktywny) i wyłączany ręcznie (uaktywnienie obiektu powoduje odblokowanie wszystkich oczekujących wątków).
  - **muteksy** (**zamki**) realizują blokadę; mogą być wielokrotnie zajęte (a później tylokrotnie zwolnione) przez jeden i ten sam wątek
  - **semafory** wielowartościowe
  - **sekcje krytyczne** - specjalne tanie muteksy działające w obrębie procesu

Transakcje niepodzielne

- Model systemu
- Odtwarzanie na podstawie dziennika
- Punkty kontrolne
- Współbieżne transakcje niepodzielne

Transakcje niepodzielne – model systemu

- Transakcja, to fragment programu, w którym występuje ciąg dostępów do obiektów (odczytu, zapisu), zakończony operacją **zatwierdzania** bądź **zaniechania**.

- Operacja zatwierdzania oznacza pomyślne zakończenie transakcji (*commited transaction*), a zaniechania - rezygnację z zakończenia operacji wskutek błędu logicznego bądź awarii.

- Podstawową kwestią rozważaną w przetwarzaniu transakcji jest zachowanie ich niepodzielności pomimo ewentualnych awarii systemu komputerowego.

- Dla zapewnienia niepodzielności wykonywania transakcji transakcję zaniechaną (*aborted transaction)* należy **wycofać** (*roll back*), tzn. przywrócić dane zmienione przez wykonane już operacje transakcji do stanu, jaki miały bezpośrednio przed rozpoczęciem wykonywania transakcji. Skutków operacji zatwierdzonej nie można cofnąć.

Transakcje niepodzielne – model systemu

Typy środków magazynowania danych, z których korzystają transakcje:

- Pamięć ulotna (ang. *volatile storage*), to pamięć w której informacje na ogół nie są w stanie przetrwać awarii systemu. Przykład: pamięć operacyjna,
- Pamięć nieulotna (ang. *nonvolatile storage*), to pamięć w której informacje na ogół są w stanie przetrwać awarii systemu, ale jest to pamięć istotnie wolniejsza od ulotnej; ponadto nie jest to pamięć niezawodna. Przykład: pamięć dyskowa.
- Pamięć trwała (ang. *stable storage*) przechowuje niezawodnie informacje „dowolnie długo”. Pamięć trwała może być zrealizowana jedynie w przybliżeniu, przy zastosowaniu zwielokrotniania informacji w wielu jednostkach pamięci nieulotnej (możliwie niezależnych jeśli chodzi o awarie).

Przy realizacji transakcji chodzi więc o zapewnienie ich niepodzielności, pomimo możliwej utraty danych przechowywanych w pamięci ulotnej.

Odtwarzanie na podstawie dziennika

- Niepodzielność transakcji może być zrealizowana przy pomocy zapisu do rejestru transakcji w *pamięci trwałej* rekordów zawierających opis kolejnych operacji transakcji: nazwę modyfikowanego obiektu, stan przed i po operacji.
- Pierwszą operację transakcji T_i poprzedza specjalny rekord rozpoczęcia

  <T_i starts>
- Gdy dochodzi do zatwierdzenia transakcji T_i - w rejestrze zapisuje się specjalny rekord zatwierdzenia

  <T_i commits>
- Każdy wpis do rejestru musi (idealnie) dotrzeć do pamięci trwałej – zanim nastąpi zmiana danych w trybie transakcji.

Odtwarzanie na podstawie dziennika - c.d.

- Rejestr umożliwia rekonstrukcję stanu obiektów do wartości przed rozpoczęciem transakcji, jeśli w czasie przeprowadzania transakcji wystąpi awaria, powodująca utracenie danych w pamięci ulotnej, ale nie powodująca zaginięcia danych rejestru.
  - Jeśli w rejestrze jest rekord rozpoczęcia transakcji T_i, a nie ma rekordu zatwierdzenia – procedura wycofaj, ***undo*(T_i),** odtwarza stare wartości obiektów (które zostały zmienione przez wykonane operacje transakcji)
  - Jeśli w rekordzie są zarówno rekord rozpoczęcia jak i zakończenia – procedura przywróć, ***redo* (T_i),** nadaje najnowsze wartości wszystkim danym zmienionym przez zakończoną transakcję.
- Aby zagwarantować poprawność nawet przy awarii w czasie odtwarzania - procedury *wycofaj* i *przywróć* powinny być **idempotentne** (skutek wielokrotnego użycia taki, jak jednokrotnego).

Punkty kontrolne

- Po awarii systemu trzeba w zasadzie przejrzeć cały rejestr, aby określić które transakcje mają być przywrócone, a które wycofane. Rejestr z czasem może stać się duży, a jego przeglądanie czasochłonny
- Dla zmniejszenia kosztów rekonstrukcji danych system rejestruje z wyprzedzeniem operacje zapisu, a ponadto organizuje *punkty kontrolne*, w których zapisuje:
  - rekordy i zmienione dane pozostające w pamięci ulotnej (RAM) są kopiowane do pamięci trwałej (gdzie dane nie giną „nigdy” wskutek awarii)
  - w rejestrze transakcji (przechowywanym w pamięci trwałej) zapisuje się specjalny rekord punktu kontrolnego.
- Procedura rekonstrukcji:
  - rejestr jest przeszukiwany wstecz, aby znaleźć ostatni punkt kontrolny
  - szukając dalej wstecz odnajdywana jest pierwsza transakcja T, która nie została zatwierdzona
  - operacje przywróć i wycofaj należy wykonać w odniesieniu do T oraz wszystkich transakcji, które rozpoczęły się po niej.

Transakcje współbieżne

- Wykonanie współbieżne musi być równoważne wykonaniu sekwencyjnemu – **szeregowalność** (*serializability*)
- Wykonywanie wszystkich transakcji w sekcji krytycznej skutkuje nieefektywnością
- Dla gwarantowania poprawności wykonywania współbieżnego wykorzystuje sią algorytmy kontroli szeregowalności, np. **protokół blokowania dwufazowego**.
- Przykład planu szeregowego i równoważnego planu współbieżnego

T0	T1
Czytaj(A)	
Pisz(A)	
Czytaj(B)	
Pisz(B)	
	Czytaj(A)
	Pisz(A)
	Czytaj(B)
	Pisz(B)

T0	T1
Czytaj(A)	
Pisz(A)	
	Czytaj(A)
	Pisz(A)
Czytaj(B)	
Pisz(B)	
	Czytaj(B)
	Pisz(B)