Programowanie współbieżne

Monitory, Zmienne warunkowe, Zamki, Regiony, Liczniki zdarzeń i Sekwensery

Prowadzący: dr inż. Jarosław Rulka
jaroslaw.rulka@wat.edu.pl

Zagadnienia

1. Monitory
   - zmienne warunkowe (ang. *condition variables*)
2. Zamki (ang. Locks)
3. Regiony
4. Liczniki zdarzeń, sekwensery (ang. *Event Counts*, *Sequencers*)

Wady semaforów

- Jako jeden z pierwszych mechanizmów synchronizacji jest generalnie mechanizmem niskiego poziomu.
- Rozproszone użycie – semafor jako zmienna współdzielona wykorzystywana w wielu miejscach.
- Słaba czytelność programów współbieżnych.
- Duża podatność na błędy, trudno wykazać poprawność programu.
  - zmiana kolejności wykonania operacji semaforowych lub brak jednej z nich może prowadzić do zawieszenia procesów.

Monitor – definicja

- Koncepcja mechanizmu monitora została zaproponowana przez Hoare'a.
- Monitor jest strukturalnym mechanizmem synchronizacji, którego definicja obejmuje:
  - hermetycznie zamkniętą definicję zmiennych (pól),
  - definicję wejść, czyli procedur umożliwiających **wyłączne** wykonanie operacji na polach monitora.
- Wewnątrz monitora może być **aktywny** co najwyżej jeden proces.

Monitor – definicja cd.

- Monitor może wykorzystywać mechanizm zmiennych warunkowych, na których można wykonywać dwie operacje:
  - wait – proces wywołujący operację zostaje bezwarunkowo zawieszony z jednoczesnym opuszczeniem i zwolnieniem monitora;
  - signal (signalAll) – uśpiony proces może zostać obudzony przez wysłanie sygnału związanego z daną zmienną warunkową przez inny proces w monitorze.
- Zmienne warunkowe, jako wewnętrzne zmienne monitora, dostępne są tylko w ramach wejść.
  - usypianie oraz budzenie procesu jest częścią implementacji jakiegoś wejścia.

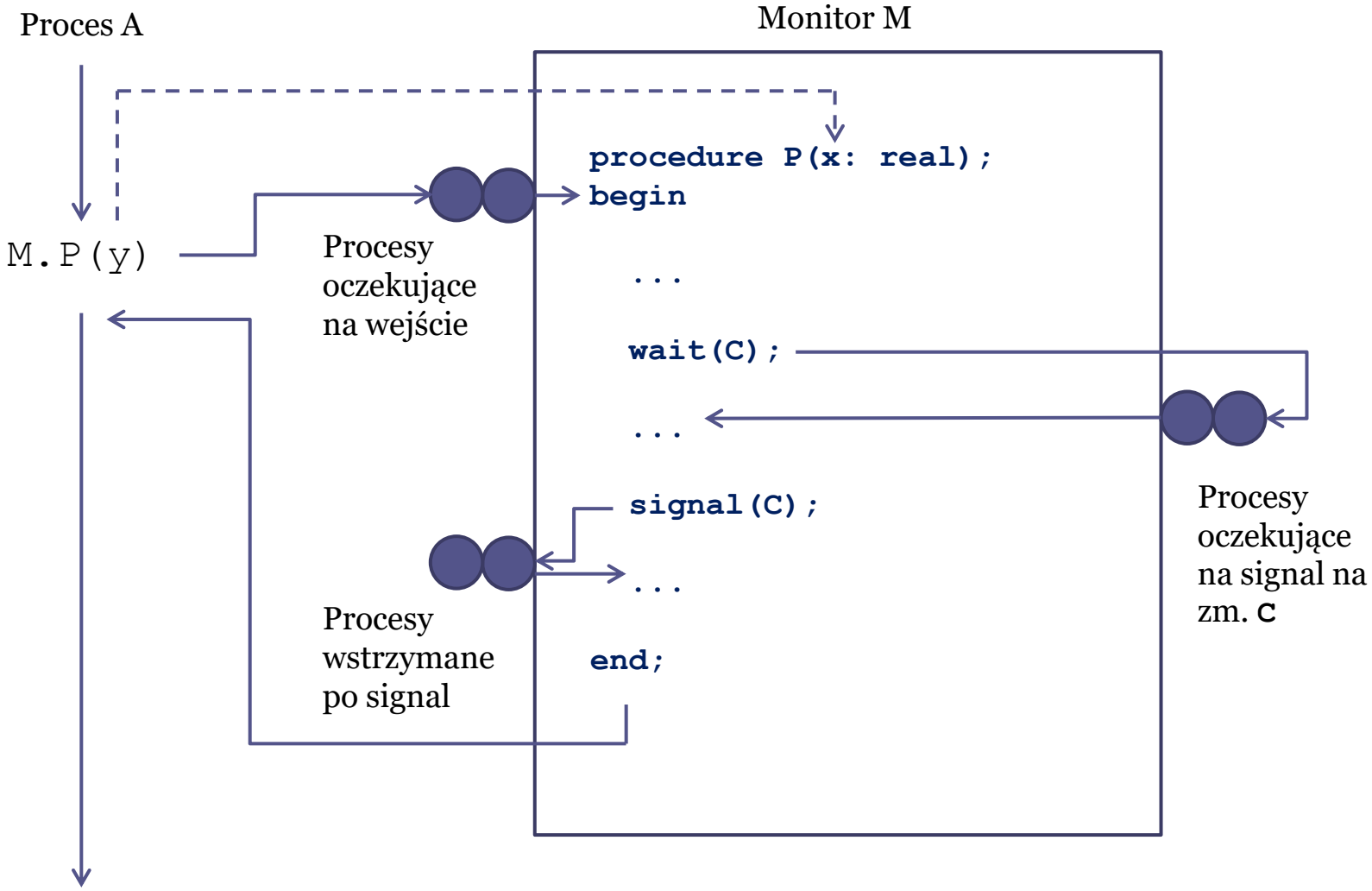
Ogólny schemat definicji monitora

```
1.  type Nazwa_Monitora = monitor
2.     deklaracje zmiennych
3.     deklaracje zmiennych warunkowych

4.     procedure entry proc_1(…);
5.     begin
6.        ...
7.     end;
8.     ...
9.     procedure entry proc_n(...);
10.    begin
11.       ...
12.    end;

13. begin
14.    kod inicjalizujący
15. end.
```

Przejście procesu przez monitor



Monitory – semantyka Hoare'a i Mesa

- Przypuśćmy że proces P wykonał operację `wait` i został zawieszony. Po jakimś czasie proces Q wykonuje operacje `signal` odblokowując P.
- **Problem**: Który proces dalej kontynuuje pracę: P czy Q?
  - Zgodnie z zasadą działania monitora tylko jeden proces może kontynuować pracę.
- Semantyka Hoare'a:
  - Proces odblokowany (P) kontynuuje jako pierwszy.
  - Może ułatwiać pisanie poprawnych programów. W przypadku semantyki Mesa nie mamy gwarancji, że warunek, na jaki czekał P jest nadal spełniony (P powinien raz jeszcze sprawdzić warunek).
- Semantyka Mesa:
  - Proces który wywołał operacje `signal` kontynuuje pierwszy.
  - P może wznowić działanie, gdy Q opuści monitor.
  - Wydaje się być zgodna z logiką, po co wstrzymywać proces, który zgłosił zdarzenie.
- **Porada**: Aby uniknąć problemów z semantyką najlepiej przyjąć, że operacja `signal` jest zawsze ostatnią operacją procedury monitora.

Wzajemne wykluczanie

```
1.  type MONITOR_ZASOBU = monitor;
2.    zasób: TypZasobu;
3.    procedure entry Dostęp();
4.    begin
5.      {chronione działania na zasobie}
6.    end; {DOSTĘP}
7.  begin
8.  end; {MONITOR_ZASOBU}

9.  mon_zasobu: MONITOR_ZASOBU;

10. process Proc(i: Integer);
11. begin
12.   loop
13.     własne_sprawy;
14.     mon_zasobu.Dostęp();
15.   end loop;
16. end; {Proc}
```

Implementacja semafora ogólnego

```
1.  type SEMAFOR = monitor;
2.    licznik: Integer;
3.    kolejka: Condition;
4.
5.    procedure entry P();
6.    begin
7.      DEC(licznik);
8.      if licznik < 0 then
9.        wait(kolejka);
10.     end if;
11.   end; {P}
```

```
12.   procedure entry V();
13.   begin
14.     INC(licznik);
15.     signal(kolejka)
16.   end; {V}

17. begin
18.   licznik := 0
19. end; {SEMAFOR}
```

Producenci i konsumenci – ograniczony bufor cykliczny (1/4)

```
1. const N: Integer := wielkość_bufora;
2. type Buffer = monitor
3.   pula: array [0..N - 1] of ElemType;
4.   wej: Integer;
5.   wyj: Integer;
6.   pełny: Condition;
7.   pusty: Condition;
8.   licz: Integer;
```

Producenci i konsumenci – ograniczony bufor cykliczny (2/4)

```
9.   procedure entry wstaw(elem: in ElemType);
10.  begin
11.    if licz = N then
12.      pełny.wait();
13.    end if;
14.    pula[wej] := elem;
15.    wej := (wej + 1) mod N;
16.    licz := licz + 1;
17.    pusty.signal();
18.  end;
```

Producenci i konsumenci – ograniczony bufor cykliczny (3/4)

```
19.   procedure entry pobierz(elem: out ElemType);
20.   begin
21.     if licz = 0 then
22.       pusty.wait();
23.     end if;
24.     elem := pula[wyj];
25.     wyj := (wyj + 1) mod N;
26.     licz := licz - 1;
27.     pełny.signal();
28.   end;
29.begin
30.   wej := 0;
31.   wyj := 0;
32.   licz := 0;
33.end;
```

Producenci i konsumenci – ograniczony bufor cykliczny (4/4)

```
34. buf: Buffer;

35. procedure Producent;
36.    elem: ELemType;
37. begin
38.    loop
39.       produkuj(elem);
40.       buf.wstaw(elem);
41.    end loop;
42. end;
```

```
43. procedure Konsument;
44.    elem: ELemType;
45. begin
46.    loop
47.       buf.pobierz(elem);
48.       konsumuj(elem);
49.    end loop;
50. end;
```

Czytelnicy i pisarze (1/5)

```
1.  czytelnia: CZYTELNIA;

2.  procedure Czytelnik();
3.  begin
4.    loop
5.      czytelnia.POCZ_CZYTANIA();
6.      czytanie();
7.      czytelnia.KON_CZYTANIA();
8.      sprawy_własne();
9.    end loop;
10. end;
```

```
10. procedure Pisarz();
11. begin
12.   loop
13.     czytelnia.POCZ_PISANIA();
14.     pisanie();
15.     czytelnia.KON_PISANIA();
16.     sprawy_własne();
17.   end loop;
18. end;
```

Czytelnicy i pisarze (2/5)

```
18. type CZYTELNIA = monitor
19.    licz_czyt: Integer; // liczba czytelników w czytelni
20.    licz_pisz: Integer; // liczba pisarzy w czytelni
21.    czytelnicy: Condition;
22.    pisarze: Condition;
23.    czyt_pocz: Integer; // liczba czytelników w poczekalni
24.    pis_pocz: Integer; // liczba pisarzy w poczekalni
```

Czytelnicy i pisarze (3/5)

```
25.  procedure entry POCZ_CZYTANIA();
26.  begin
27.    if licz_pisz + pis_pocz > 0 then
28.      czyt_pocz := czyt_pocz + 1;
29.      wait(czytelnicy);
30.      czyt_pocz := czyt_pocz - 1;
31.    end if;
32.    licz_czyt := licz_czyt + 1;
33.  end; {POCZ_CZYTANIA}

34.  procedure entry KON_CZYTANIA();
35.  begin
36.    licz_czyt := licz_czyt - 1;
37.    if licz_czyt = 0 then
38.      signal(pisarze);
39.    end if;
40.  end; {KON_CZYTANIA}
```

Czytelnicy i pisarze (4/5)

```
41.  procedure entry POCZ_PISANIA();
42.  begin
43.    if licz_czyt + licz_pisz > 0 then
44.      pis_pocz := pis_pocz + 1;
45.      wait(pisarze);
46.      pis_pocz := pis_pocz - 1;
47.    end if;
48.    licz_pisz := 1;
49.  end; {POCZ_PISANIA}

50.  procedure entry KON_PISANIA();
51.  begin
52.    licz_pisz := 0;
53.    if (czyt_pocz > 0) then
54.      signalAll(czytelnicy);
55.    else
56.      signal(pisarze);
57.    end if;
58.  end; {KON_PISANIA}
```

Czytelnicy i pisarze (5/5)

```
59.  begin
60.    licz_czyt := 0;
61.    licz_pisz := 0;
62.    czyt_pocz := 0;
63.    pis_pocz := 0;
64. end; {CZYTELNIA}
```

Pięciu filozofów (1/4)

```
1.  widelce: WIDELCE;

2.  procedure Filozof(nr: Integer);
3.  begin
4.    loop
5.      własne_sprawy();
6.      widelce.WEŹ(nr);
7.      jedzenie();
8.      widelce.ODŁÓŻ(nr);
9.    end loop;
10. end;
```

Pięciu filozofów (2/4)

```
11. type WIDELCE = monitor;
12.   WIDELEC: array[0..4] of Condition;
13.   zajęty: array[0..4] of Boolean;
14.   LOKAJ: Condition;
15.   jest: Integer;
```

Pięciu filozofów (3/4)

```
16.  procedure entry WEŹ(i: Integer);
17.  begin
18.    if jest = 4 then
19.      wait(LOKAJ);
20.    end if;
21.    jest := jest + 1;
22.    if zajęty[i] then
23.      wait(WIDELEC[i]);
24.    end if;
25.    zajęty[i] := true;
26.    if zajęty[(i + 1) mod 5] then
27.      wait(WIDELEC[(i + 1) mod 5]);
28.    end if;
29.    zajęty[(i +1) mod 5] := true;
30.  end; {BIORĘ}
```

Pięciu filozofów (4/4)

```
31.   procedure entry ODŁÓŻ(i: Integer);
32.   begin
33.     zajęty[i] := false;
34.     signal(WIDELEC[i]);
35.     zajęty [(i + 1) mod 5] := false;
36.     signal(WIDELEC[(i + 1) mod 5]);
37.     jest := jest - 1;
38.     signal(LOKAJ);
39.   end; {ODKŁADAM}

40. begin
41.   for i := 0 to 4 do
42.     zajęty [i] := false;
43.   end for;
44.   jest := 0
45. end; {WIDELCE}
```

Pięciu filozofów rozwiązanie z możliwością zagłodzenia

```
1.  type WIDELCE = monitor
2.    wolne: array[0..4] of Integer;
3.    filozof: array[0..4] of Condition;

4.    procedure entry WEŹ(i: Integer);
5.    begin
6.      if wolne[i] < 2 then
7.        wait(filozof[i]);
8.      end if;
9.      DEC(wolne[(i + 4) mod 5]);
        DEC(wolne[(i + 1) mod 5]);
10.   end; {WEŹ}
```

```
11.   procedure entry ODŁÓŻ(i:
          Integer);
12.   begin
13.     INC(wolne[(i + 4) mod 5]);
14.     INC(wolne[(i + 1) mod 5]);
15.     if wolne[(i + 4) mod 5] = 2 then
16.       signal(filozof[(i + 4) mod 5]);
17.     end if;
18.     if wolne [(i + 1) mod 5] = 2 then
19.       signal(filozof[(i + 1) mod 5]);
20.     end if;
21.   end; {ODŁÓŻ}
22. begin
23.   for i := 0 to 4 do
24.     wolne[i] := 2;
25.   end for;
26. end; {WIDELCE}
```

Śpiący fryzjerzy (1/4)

```
1.  zakład: ZAKŁAD_FRYZJERSKI;

2.  procedure Klient(nr: Integer);
3.    wynik: Boolean;
4.  begin
5.    loop
6.      własne_sprawy();
7.      wynik := false;
8.      while (wynik = false) do
9.        zakład.ŻĄDANIE_USŁUGI(wynik);
10.     end while;
11.     strzyżenie();
12.   end loop;
13. end;
```

```
14. procedure Fryzjer(nr: Integer);
15. begin
16.   loop
17.     zakład.ROZPOCZĘCIE_USŁUGI();
18.     strzyżenie();
19.     zakład.ZAKOŃCZENIE_USŁUGI();
20.   end loop;
21. end;
```

Śpiący fryzjerzy (2/4)

```
22. type ZAKŁAD_FRYZJERSKI = monitor;
23.    const poj_pocz : Integer := pojemność_poczekalni;
24.    l_czek : Integer := 0; // liczba klientów w poczekalni
25.    wolne_fot : Integer := liczba_foteli;
26.    klient : Condition;
27.    fryzjer : Condition;
28.    fotel : Condition;
```

Śpiący fryzjerzy (3/4)

```
29.  procedure entry ŻĄDANIE_USŁUGI(wynik: out Boolean);
30.  begin
31.    if l_czek < poj_poczek then
32.      l_czek := l_czek + 1;
33.      signal(klient);
34.      wait(fryzjer);
35.      wynik := true;
36.    else
37.      wynik := false;
38.    end if;
39.  end; {ŻĄDANIE_USŁUGI}
```

Śpiący fryzjerzy (4/4)

```
40.   procedure entry ROZPOCZĘCIE_USŁUGI();
41.   begin
42.     if (l_czek = 0) then
43.       wait(klient);
44.     end if;
45.     l_czek := l_czek - 1;
46.     if (wolne_fot = 0) then
47.       wait(fotel);
48.     end if;
49.     wolne_fot := wolne_fot - 1;
50.     signal(fryzjer);
51.   end; {ROZPOCZĘCIE_USŁUGI}

52.   procedure entry ZAKOŃCZENIE_USŁUGI();
53.   begin
54.     wolne_fot := wolne_fot + 1;
55.     signal(fotel);
56.   end; {ZAKOŃCZENIE_USŁUGI}
57. end; {ZAKŁAD_FRYZJERSKI}
```

Różnice między semaforami a zmiennymi warunkowymi

Semafory	Zmienne warunkowe
1. Mogą być wykorzystywane wszędzie w programie, ale nie powinny być stosowane w monitorze.	1. Mogą być wykorzystywane tylko w monitorze.
2. `Opuść()` – nie zawsze blokuje wątek wywołujący: np. gdy wartość licznika semafora jest większa od zera.	2. `Wait()` – zawsze blokuje wątek wywołujący.
3. `Podnieś()` – zwalnia jeden z zablokowanych wątków (jeśli są zablokowane) lub zwiększa wartość licznika semafora.	3. `Signal()` – albo zwalnia jeden zablokowany wątek (o ile są zablokowane wątki), albo wykonuje się bez żadnego efektu (gdy nie ma zablokowanych wątków).
4. Jeśli `Podnieś()` zwalnia zablokowany wątek, to oba wątki kontynuują swoje działania.	4. Jeśli `Signal()` zwolni zablokowany wątek, to wątek wywołujący opuszcza/przekazuje monitor (Hoare type) w oczekiwaniu na wznowienie, które nastąpi gdy odblokowany wątek wywoła `wait()` lub zakończy procedurę monitora.

Implementacja monitora za pomocą semafora ogólnego

```
1.  type monitor;
2.    Dostęp: Semaphore;
3.    ZW: Semaphore;
4.    LZW: Integer;
5.
6.    procedure init();
7.    begin
8.      Dostęp(1);
9.      ZW(0);
10.     LZW := 0;
11.   end; {init}

12.   procedure entry P1();
13.   begin
14.     P(Dostęp);
15.     ...
16.     V(Dostęp);
17.   end; {P1}
```

```
18.   procedure signal(ZW, LZW);
19.   begin
20.     if LZW > 0 then
21.       V(ZW);
22.     end if;
23.   end; {signal}

24.   procedure wait(ZW, LZW);
25.   begin
26.     INC(LZW);
27.     V(Dostęp);
28.     P(ZW);
29.     DEC(LZW);
30.   end; {wait}

31. end; {monitor}
```

Zamki (Locks)

- Mechanizm wyłącznego dostępu do sekcji krytycznej.
- Obiekt zamka może być w posiadaniu tylko przez jeden wątek.
- Operacje:
  - **`lock()`** – zajęcie/zaryglowanie zamka (założenie blokady)
    - jeśli zamek jest zwolniony, to funkcja powoduje jego zajęcie i wątek kontynuuje swoje działanie,
    - jeśli zamek jest zajęty, to funkcja powoduje wstrzymanie wątku wywołującego i wstawienie do kolejki oczekujących na zwolnienie zamka.
  - **`unlock()`** – zwolnienie/odryglowanie zamka (zwolnienie blokady)
    - funkcja zdejmuje blokadę z obiektu zamka (zamek może być zajęty przez inny wątek),
    - w przypadku wstrzymywania wątków na obiekcie, pierwszy z nich jest wznawiany z jednoczesnym zajęciem zamka.
  - **`trylock()`** – próba zajęcia zamka w sposób nieblokujący wątku w przypadku niepowodzenia (niepowodzenie sygnalizowane jest zwracaną wartością false).

Zamki (Locks) – Producenci i konsumenci

```
1.  buf: array[0..N - 1] of Element;
2.  licznik: Integer := 0;
3.  wej, wyj: Integer := 0;
4.  zamek: Lock;
5.  procedure Producent()
6.    el: Element;
7.  begin
8.    loop
9.      produkuj(el);
10.     zamek.lock();
11.     while licznik = N do
12.       zamek.unlock();
13.       zamek.lock();
14.     end while;
15.     buf[wej] := el;
16.     wej := (wej + 1) mod N;
17.     INC(licznik);
18.     zamek.unlock();
19.   end loop;
20. end; {Producent}
```

```
21. procedure Konsument()
22.   el: Element;
23. begin
24.   loop
25.     zamek.lock();
26.     while licznik = 0 do
27.       zamek.unlock();
28.       zamek.lock();
29.     end while;
30.     el := buf[wyj];
31.     wyj := (wyj + 1) mod N;
32.     DEC(licznik);
33.     zamek.unlock();
34.     konsumuj(el);
35.   end loop;
36. end; {Konsument}
```

Regiony krytyczne

- **Region krytyczny** jest blokiem programu – oznaczonym jako *S*, wykonywanym przy wyłącznym dostępie do pewnej zmiennej współdzielonej, wskazanej w jego definicji – oznaczonej jako *v*.
- Wykonanie regionu krytycznego uzależnione jest od wartości wyrażenia logicznego – *B*, a przetwarzanie blokowane jest do momentu, aż wyrażenie będzie prawdziwe.

```
shared v: T;
region v when B do S;
```

Region krytyczny - producenci i konsumenci (1/3)

```
1. const N: Integer := rozmiar_bufora;
2. shared buf: record
3.    pula: array [0..N - 1] of ElemType;
4.    wej: Integer;
5.    wyj: Integer;
6.    licz: Integer;
7. end;
```

Region krytyczny - producenci i konsumenci (2/3)

```
8. procedure Producent
9.    elem: ElemType;
10.begin
11.   loop
12.      produkuj(elem);
13.      region buf when licz < N do
14.      begin
15.         pula[wej] := elem;
16.         wej := (wej + 1) mod N;
17.         licz := licz + 1;
18.      end;
19.   end loop;
20.end; {Producent}
```

Region krytyczny - producenci i konsumenci (3/3)

```
21.procedure Konsument
22.   elem: ElemType;
23.begin
24.   loop
25.      region buf when licz > 0 do
26.      begin
27.         elem := pula[wyj];
28.         wyj := (wyj + 1) mod N;
29.         licz := licz - 1;
30.      end;
31.      konsumuj(elem);
32.   end loop;
33.end; {Konsument}
```

Liczniki zdarzeń (*Event Counts*)

- Mechanizm synchronizacji (1979) opierający się na specyficznym typie zmiennej
  - zmienna przyjmuje niemalejące wartości typu całkowitoliczbowego (inicjalizowana na zero);
  - zmienna zlicza wystąpienia zdarzeń.
- Dostępne są tylko trzy operacje:
  - **`advance(ec)`**
    - operacja atomowa;
    - sygnalizuje wystąpienie zdarzenia;
    - uaktualnia wartość licznika;
  - **`read(ec, v)`**
    - zwraca wartość licznika `v`;
    - zwracana wartość może być już nieaktualna (operacja nie jest atomowa);
  - **`await(ec, v)`**
    - czeka na osiągnięcie przez licznik co najmniej wartości `v`;
    - nie musi się zakończyć natychmiast po wykonaniu *v*-tej operacji `advance`.

Liczniki zdarzeń – producent i konsument

```
1.  const N: Integer := rozmiar_bufora;
2.  bufor: array [0..N - 1] of Elem;
3.  CEV: EventCount;
4.  PEV: EventCount;

5.  procedure producent()
6.    elem: Elem;
7.    wej: Integer := 0;
8.  begin
9.    loop
10.     elem := produkuj();
11.     await(CEV, (wej - N) + 1);
12.     bufor[wej mod N] = elem;
13.     wej = wej + 1;
14.     advance(PEV);
15.   end loop;
16. end; {producent}
```

```
18. procedure konsument()
19.   elem: Elem;
20.   wyj: Integer := 0;
21. begin
22.   loop
23.     await(PEV, wyj + 1);
24.     elem = bufor[wyj mod N];
25.     wyj = wyj + 1;
26.     advance(CEV);
27.     konsumuj(elem);
28.   end loop;
29. end; {konsument}
```

Sekwensery (*Sequencers)*

- Sekwensery są uzupełnieniem mechanizmu licznika zdarzeń;
- Zmienna tego typu przyjmuje niemalejące wartości całkowitoliczbowe zaczynając od zera;
- Dostępna jest tylko jedna atomowa operacja na zmiennej:
  - **`ticket(SQ, v)`** – zwraca aktualną wartość **`v`** zmiennej SQ z jednoczesną jej inkrementacją.
- Liczniki zdarzeń i sekwensery są mechanizmami synchronizacji niższego poziomu w stosunku do semaforów;

Liczniki zdarzeń i sekwensery – implementacja zamka

```
1.  type Lock = record
2.    EC: EventCount; // inicjalizowany na zero
3.    SQ: Sequencer; // inicjalizowany na zero
4.  end;

5.  procedure lock(L: Lock);
6.    t: Integer;
7.  begin
8.    ticket(L.SQ, t);
9.    await(L.EC, t);
10. end;

11. procedure unlock(L: Lock);
12. begin
13.   advance(L.EC);
14. end;
```

Liczniki zdarzeń i sekwensery – implementacja semafora ogólnego

```
1.  type Semaphore = record
2.    EC: EventCount; // inicjalizowany na zero
3.    SQ: Sequencer; // inicjalizowany na zero
4.    I: Integer; // wartość początkowa semafora
5.  end;

6.  procedure P(S: Semaphore);
7.    t: Integer;
8.  begin
9.    ticket(S.SQ, t);
10.   await(S.EC, t - S.I + 1);
11. end;

12. procedure V(S: Semaphore);
13. begin
14.   advance(S.EC);
15. end;
```