Programowanie współbieżne

Ćwiczenia 6 – monitory cz. 2

Zadanie 1: Bar

Napisz monitor *Bar* synchronizujący pracę barmana obsługującego klientów przy kolistym barze z N stołkami. Każdy klient realizuje następujący algorytm:

```
process Klient() {
  int i;
  while (true) {
    i = Bar.CzekamNaStołekIPiwo();
    // piję piwo na stołku i−tym
    Bar.ZwalniamStołek(i);
  }
}
```

Barman nalewa piwo nowo przybyłym klientom w kolejności wyznaczonej przez stołki przez nich zajmowane (chodzi „w kółko”):

```
process Barman() {
  int i=0;
  while (true) {
    i = Bar.KtoNastępny(i);
    // podejście do stołka i−tego i nalanie piwa
    Bar.Gotowe(i);
  }
}
```

Rozwiązanie

```
monitor Bar {
  enum StanStołka {wolny, nieobsłużony, obsłużony};
  /* stan stołka: nieobsłużony − siedzący na nim klient czeka na piwo
                  obsłużony    − siedzący na nim klient pije piwo */

  condition naKlienta;     /* barman czeka, jeżeli nie ma nic do roboty */
  condition naWejście;     /* klienci czekają na wejście do baru */
  condition naPiwo[N];     /* klient czeka na nalanie piwa */

  int iluWBarze = 0;       /* ilu klientów jest w barze */
  iluCzeka: int;           /* ilu klientów czeka na nalanie piwa */

  StanStołka stołek[N] = (wolny ,..., wolny);

  int CzekamNaStołekIPiwo(){
    int i;

    if (iluWBarze == N) wait(naWejście); /* nie ma wolnych stołków */
    iluWBarze++;
    iluCzeka++;
    i = 0;
    while (stołek[i] <> wolny) i = (i + 1) % N; /* klient szuka wolnego stołka */
    stołek[i] = nieobsłużony;        /* i−ty stołek był wolny, klient go zajmuje */
```

```
    signal(naKlienta);              /* zwalnia barmana, jeżeli ten czeka */
    wait(naPiwo[i]);                /* czeka na stołku i-tym na piwo */
    return i;
  }

  void ZwalniamStołek(int i){
    iluWBarze--;
    stołek[i] = wolny;
    signal(naWejście);
  }

  int KtoNastępny(int ostatnio){
    int i;
    if (iluCzeka == 0) wait(naKlienta);  /* nie ma nieobsłużonych klientów */
    i = (ostatnio + 1) % N;  /* barman szuka klienta począwszy od następnego
                                miejsca po tym, które ostatnio obsługiwał */
    while (stołek[i] <> nieobsłużony) i = (i + 1) % N;
    return i;
  }

  void Gotowe(int i){
    stołek[i] = obsłużony;
    iluCzeka--;
    signal(naPiwo[i]);
  }
}
```

Uwagi

Stan stołka musi być określony przez trzy wartości. Jeśli trzymamy informację tylko o tym, czy stołek jest wolny, czy zajęty, a rozróżniamy stołki obsłużone od nieobsłużonych sprawdzając czy kolejki z nimi związane są niepuste, to mamy rozwiązanie niepoprawne. Klient w CzekamNaStołekIPiwo() wykonuje *wait*(*naPiwo*[*i*]) po *signal*(*naKlienta*). Zgodnie z semantyką operacji signal po wykonaniu *signal*(*naKlienta*) zaczyna działać barman, który sprawdzając kolejkę *naPiwo*[*i*] stwierdza, że jest ona pusta, ponieważ klient nie wykonał jeszcze *wait*(*naPiwo*[*i*]). Ten sam problem występuje, jeżeli przyjmiemy założenie, że nalanie piwa jest wewnętrzną czynnością w monitorze, czyli procedury *KtoNastępny*() i *Gotowe*() złączymy w jedną.

Barman szukając nie obsłużonego klienta nie może zaczynać zawsze od tego samego miejsca, ponieważ prowadzi to do zagłodzenia.

Dlaczego nie skorzystaliśmy z wygodniejszych struktur danych – takich jak lista zamiast tablicy? Ponieważ barman szukając klienta nie mógłby wziąć pierwszego z listy oczekujących – powinien obsługiwać wszystkich czekających klientów, obok których przechodzi, niezależnie od tego w jakiej kolejności przyszli. Ten problem można rozwiązać porządkując listę według numerów stołków, przy których siedzą klienci, jednak wtedy pojawia się nowa trudność – od którego miejsca należy rozpocząć, żeby nie spowodować zagłodzenia? Jak widać rozwiązanie korzystające z tablicy jest prostsze.

Zadanie 2: Ładowanie cegieł

Grupa N (N > 0) robotników ma załadować stertę cegieł na samochód. Każdy z nich musi wiedzieć od kogo ma odbierać cegły i komu podawać, czyli musi znać numery obu swoich sąsiadów (należy przyjąć, że sterta cegieł ma numer 0, a samochód – N+1). Pracę można rozpocząć dopiero wtedy, gdy zgłoszą się wszyscy robotnicy. Po zakończeniu załadunku robotnicy przychodzą po wypłatę. Każdy podaje swoją stawkę za pracę, na podstawie której wylicza się stawkę średnią, którą otrzymują wszyscy. Każdy robotnik działa według schematu:

```
process Robotnik (int id) {
  int mojaStawka = ...;
  int lewy, prawy, wypłata;

  while (true) {
    Załadunek.ChcęPracować(id, &lewy, &prawy);
    podawajCegły(lewy, prawy);
    Załadunek.Wypłata(mojaStawka, &wypłata);
    odpoczynek(wypłata);
  }
}
```

Należy napisać monitor *Załadunek*. Dodatkowe wymaganie: nie wolno deklarować żadnych struktur rozmiaru N lub większego.

Rozwiązanie

Aby poznać numer prawego sąsiada nie używając dodatkowych struktur danych należy odwrócić kolejność procesów wykorzystując do tego operację signal. Przypominamy, że signal wstrzymuje działanie procesu do momentu, gdy zwalniany przez niego proces opuści monitor. Procesy wstrzymane przy wykonaniu operacji signal zwalniane są w kolejności odwrotnej do tej w jakiej ją wykonywały.

```
monitor Załadunek {
  int ilu=0, ostatni=0, stawka=0;
  condition naPozostałych;

  void chcęPracować(int id, &lewy, &prawy) {

    ilu++;
    lewy = ostatni;
    ostatni = id;
    if (ilu < N) {
      wait(naPozostałych);
      signal(naPozostałych);
      prawy = ostatni;
      ostatni = id;
    } else {
      prawy = N + 1;
      ilu = 0;
      signal(naPozostałych);
    }
  }

  void Wypłata(int mojaStawka, &wypłata) {

    ilu++;
    if (ilu == 1) ostatni = 0;
    stawka = stawka + mojaStawka;
    if (ilu < N) wait(naPozostałych)
    else stawka = stawka / N;
    wypłata = stawka;
    signal(naPozostałych);
  }
}
```

Przeanalizujmy wykonanie procesów na przykładzie 4 robotników, którzy zgłaszają się w kolejności: 2, 3, 4, 1.

2	4	3	1
lewy=0			
ostatni=2			
wait			
	lewy=2		
	ostatni=4		
	wait		
		lewy=4	
		ostatni=3	
		wait	
			lewy=3
			ostatni=1
			prawy=N+1
			signal
signal			
	signal		
		signal (pusty)	
		prawy=1	
		ostatni=3	
	prawy=3		
	ostatni=4		
prawy=4			
ostatni=2			

Nie potrzeba osobnych kolejek do wstrzymywania procesów czekających na pracę i na wypłatę, ponieważ nie ma niebezpieczeństwa, że procesy z tych dwóch grup się przemieszają.

Rozwiązanie niepoprawne

Zwróćmy uwagę, że następujące rozwiązanie nie jest poprawne:

```
void chcęPracować(int id, &lewy, &prawy) {
  ilu++;
  lewy = ostatni;
  ostatni = id;
  if (ilu < N) wait(naPozostałych)
  else ostatni = N+1;
  prawy = ostatni;
  ostatni = id;
  ilu--;
  if not empty(naPozostałych) signal(naPozostałych);
}
```

Każdy z procesów musi poznać swoich sąsiadów: lewego, czyli proces, który zgłosił się bezpośrednio przed nim i prawego, czyli proces, który zgłosił się bezpośrednio po nim - ten właśnie warunek nie jest spełniony. Przeanalizujmy to na naszym przykładzie.

2	4	3	1
lewy=0			
ostatni=2			
wait			
	lewy=2		
	ostatni=4		
	wait		
		lewy=4	
		ostatni=3	
		wait	
			lewy=3
			ostatni=1
			ostatni=N+1
			prawy=N+1
			ostatni=1
			signal
prawy=1 (błąd)			
ostatni=2			
signal			
	prawy=2 (błąd)		
	ostatni=4		
	signal		
		prawy=4 (błąd)	
		ostatni=3	
		signal (pusty)	