Programowanie współbieżne

Ćwiczenia 12 – Przestrzenie krotek cz. 2

Zadanie 1: Przychodnia lekarska

W przychodni lekarskiej pracuje $L > 0$ lekarzy, z których każdy ma jedną z $0 < S \leq L$ specjalności, przy czym może być wielu lekarzy tej samej specjalności a dla danej specjalności istnieje co najmniej jeden lekarz ją posiadający. Do przychodni lekarskiej zapisanych jest P pacjentów. Lekarze są identyfikowani unikalnymi liczbami z przedziału $0 \ldots L-1$, pacjenci – liczbami z przedziału $0 \ldots P-1$, zaś specjalności – liczbami z przedziału $0 \ldots S-1$.

Każdy pacjent w pętli nieskończonej załatwia **void** wlasneZdroweSprawy(**int** pid), po czym zgłasza się do lekarza. Pacjent może życzyć sobie wizyty u konkretnego lekarza lub dowolnego lekarza konkretnej specjalności. Ponadto pacjenci mogą być uprzywilejowani bądź zwykli. Pacjent oczekuje, aż lekarz go przyjmie, po czym odbywa się **void** wizyta(**int** lid), gdzie lid jest identyfikatorem lekarza przyjmującego pacjenta.

Każdy lekarz natomiast w pętli nieskończonej **void** pijeKawe(**int** lid) a następnie przyjmuje pacjenta. W pierwszej kolejności próbuje przyjąć pacjenta uprzywilejowanego, jeśli tacy oczekują na wizytę. Jeśli nie ma czekających pacjentów uprzywilejowanych, to lekarz przyjmuje dowolnego pacjenta (tj. zwykłego lub uprzywilejowanego) lub czeka, jeśli nie ma żadnych pacjentów, których mógłby przyjąć. Przyjęcie pacjenta to **void** badanie(**int** pid), gdzie pid jest identyfikatorem badanego pacjenta.

W danej chwili lekarz może badać co najwyżej jednego pacjenta. Lekarz nie może czekać, jeśli oczekuje pacjent, którego mógłby przyjąć. Pacjenci uprzywilejowani mogą zagłodzić zwykłych.

Celem zadania jest zaimplementowanie procesów lekarza i pacjenta. Do wyboru lekarza przez pacjenta należy użyć funkcji **void** wybor(**bool** $*$ czyLekarz, **int** $*$ id, **bool** $*$ uprzyw). Funkcja informuje, czy pacjent wybrał lekarza czy specjalizację (czyLekarz), jaki jest identyfikator tego/tej lekarza/specjalizacji (id) i czy pacjent czekając na wybraną wizytę jest uprzywilejowany (uprzyw).

```
process lekarz(int lid, int sid) {   /* identyfikatory lekarza i specjalności */
  int     pid;
  do {
    pijeKawe(lid);
    if (! tsTryFetch("ZGLOSZENIE ?d %d %d %d", &pid, true, lid, sid))
      tsFetch("ZGLOSZENIE ?d ?d %d %d", &pid, NULL, lid, sid);
    tsPut("ZAPROSZENIE %d %d", pid, lid);
    badanie(pid);
  } while (1);
}

process pacjent(int pid) {   /* identyfikator pacjenta */
  int     lsid;
  bool    czyLekarz, uprzyw;
  do {
    wlasneZdroweSprawy(pid);
    wybor(&czyLekarz, &lsid, &uprzyw);
    if (czyLekarz)
      /* "dziura" na pozycji identyfikatora specjalności */
      tsPut("ZGLOSZENIE %d %d %d ?d", pid, uprzyw, lsid);
    else
      /* "dziura" na pozycji identyfikatora lekarza */
      tsPut("ZGLOSZENIE %d %d ?d %d", pid, uprzyw, lsid);
    tsFetch("ZAPROSZENIE %d ?d", pid, &lsid);
    wizyta(lsid);
```

```
  } while (1);
}
```

W powyższym rozwiązaniu należy zwrócić uwagę na „dziurawe" krotki. Dla przypomnienia, krotka z „dziurą" na danej pozycji pasuje na tej pozycji do dowolnego wzorca.

Zadanie 2: Porównywanie liczb

W przestrzeni krotek znajduje się $L > 0$ liczb całkowitych, to jest krotek w formacie `"LICZBA %d"`. Do dyspozycji jest $1 < N \leq L$ procesów, każdy z unikalnym identyfikatorem z przedziału $0 \ldots N-1$. Każdy z procesów ma za zadanie wybrać jedną z liczb (przy czym dwa procesy nie mogą wybrać tej samej liczby), porównać wybraną liczbę ze wszystkimi innymi $L-1$ liczbami, obliczając ile spośród nich jest większych od wybranej liczby, a następnie umieścić wynik obliczeń w przestrzeni krotek i zakończyć działanie. Celem zadania jest napisanie treści takiego procesu. Można założyć, że po zakończeniu obliczeń krotki z liczbami nie będą już potrzebne.

```
process proc(int id) {
  int   moja, tmp, wynik;
  tsFetch("LICZBA ?d", &moja);
  tsPut("TMP %d %d", id, moja);
  for (int i = id + N; i < L; i += N) {
    tsFetch("LICZBA ?d", &tmp);
    tsPut("TMP %d %d", i, tmp);
  }
  wynik = 0;
  for (int i = 0; i < L; ++i) {
    tsRead("TMP %d ?d", i, &tmp);
    if (tmp > moja)
      ++wynik;
  }
  tsPut("WYNIK %d %d", moja, wynik);
}
```

Kluczowym pomysłem w powyższym rozwiązaniu jest zorganizowanie krotek z liczbami tak, aby można było efektywnie po nich iterować. Odbywa się to poprzez numerowanie krotek. Takie numerowanie można zaimplementować na wiele sposobów. Zaprezentowane rozwiązanie stara się zbalansować pracę poszczególnych procesów, zakładając iż wszystkie one działają z podobną prędkością.

Zadanie 3: Zliczanie krotek

W przestrzeni krotek znajduje się pewna liczba krotek w formacie `"KROTKA %d"`, gdzie `"%d"` może mieć dowolną całkowitą wartość. Celem zadania jest napisanie współbieżnego programu liczącego te krotki. Do dyspozycji jest $N > 0$ procesów, identyfikowanych unikalnymi liczbami z przedziału $0 \ldots N-1$. W rezultacie swojego działania każdy proces powinien wypisać liczbę krotek `"KROTKA %d"` w przestrzeni i zakończyć działanie. Należy przyjąć, że po zakończeniu ostatniego procesu wszystkie krotki `"KROTKA %d"` powinny nadal znajdować się w przestrzeni krotek. Natomiast jakiekolwiek inne krotki wytworzone przez procesy mogą pozostać w przestrzeni jedynie jeśli ich liczba jest stała.

```
process zliczacz(int id) {
  int   n = 0;
  int   v, c;
  /* zainicjalizuj obliczanie */
  if (id == 0) {
    tsPut("WYNIK d %d", 0, 0);
```

```
  }
  /* policz lokalnie tyle krotek ile możesz */
  while (tsTryFetch("KROTKA ?d", &v)) {
    ++n;
    tsPut("TMP %d", v);
  }
  /* dodaj swój wynik do wyników reszty */
  tsFetch("WYNIK ?d ?d", &c, &v);
  tsPut("WYNIK %d %d", c + 1, v + n);
  tsRead("WYNIK %d ?d", N, &n);
  /* przywróć oryginalne krotki */
  while (tsTryFetch("TMP ?d", &v))
    tsPut("KROTKA %d", v);
  printf("WYNIK PROCESU %d = %d\n", id, n);
}
```

W powyższym rozwiązaniu, w liczeniu biorą udział wszystkie procesy (w miarę możliwości). Najpierw wyciągają one z przestrzeni krotek wszystkie krotki. Każdy proces liczy lokalnie, ile krotek udało mu się wyciągnąć, otrzymując lokalny wynik częściowy. Następnie te wyniki częściowe są sumowane dla wszystkich procesów, co daje sumaryczną liczbę krotek w przestrzeni. Należy także zwrócić uwagę na instrukcję **tsRead**. Wstrzymuje ona wszystkie procesy do momentu, gdy wszystkie wyniki częściowe zostały dodane do wyniku globalnego.

Sumowanie wyników częściowych w zaprezentowanym rozwiązaniu odbywa się w sekcji krytycznej – proces najpierw wyciąga krotkę z wynikiem globalnym z przestrzeni, dodaje do wyniku globalnego swój wynik lokalny, a następnie umieszcza zwiększony wynik globalny ponownie w przestrzeni. Kolejne procesy modyfikują więc wynik globalny sekwencyjnie. Innymi słowy, czas obliczania wyniku globalnego mając dane wyniki lokalne jest proporcjonalny do liczby procesów, co może być nieefektywne. Poniższe rozwiązanie to poprawia.

```
process zliczacz2(int id) {
  int   n = 0;
  int   v;
  /* policz lokalnie tyle liczb ile możesz */
  while (tsTryFetch("KROTKA ?d", &v)) {
    ++n;
    tsPut("TMP %d", v);
  }
  /* globalna redukcja sumy lokalnych wyników */
  for (int skok = 1; id + skok < N; skok *= 2) {
    if (id % (2 * skok) != 0) break;
    tsFetch("WYNIK %d ?d", id + skok, &v);
    n += v;
  }
  tsPut("WYNIK %d %d", id, n);
  /* wczytaj globalny wynik */
  tsRead("WYNIK %d ?d", 0, &n);
  /* przywróć oryginalne liczby */
  while (tsTryFetch("TMP ?d", &v))
    tsPut("KROTKA %d", v);
  printf("WYNIK PROCESU %d = %d\n", id, n);
}
```

W ulepszonym rozwiązaniu sumowanie wyników częściowych odbywa się w czasie proporcjonalnym do logarytmu z liczby procesów. Ostateczny wynik globalny produkowany jest przez proces o indeksie 0, przez co zmienia się także nieznacznie instrukcja wstrzymująca **tsRead**. Warto także

zauważyć, iż w ulepszonym rozwiązaniu sumaryczna liczba instrukcji na krotkach `"WYNIK %d %d %d"` jest mniejsza jedynie o 1 od sumarycznej liczby instrukcji na tych krotkach w rozwiązaniu poprzednim. Efektywność ulepszonego rozwiązania bierze się stąd, iż używa ono wielu takich krotek, przez co dostęp do nich może odbywać się współbieżnie.

Zadanie 4: Odśmiecanie krotek (jeśli starczy czasu)

Celem zadania jest wykorzystanie poprzedniego rozwiązania do zaimplementowania efektywnej, samo-odśmiecającej się bariery wykonywanej przez $N > 0$ procesów indeksowanych od 0 do $N-1$.

Genezą problemu jest fakt, iż niektóre programy mogą pozostawiać w przestrzeni krotki, które już nigdy nie zostaną użyte – śmieci. Dla przypomnienia, bariera to operacja wykonywana przez wszystkie N procesów w ten sposób, iż żaden proces nie może zakończyć jej wykonywania o ile wszystkie procesy nie rozpoczęły jej wykonywania. Mając daną instrukcję bariery, odśmiecanie nieużywanych krotek może być zaimplementowane następująco – po zakończeniu obliczeń generujących takie krotki procesy wykonują instrukcję bariery, po której następuje odśmiecanie wygenerowanych krotek przy wykorzystaniu przez każdy proces instrukcji **tsTryFetch**, dopóki zwraca ona **true** dla danego wzorca, i tak dalej dla kolejnych wygenerowanych wcześniej wzorców krotek. Mamy wtedy gwarancję, że odśmiecane krotki nie będą już potrzebne.

Bazą dla bariery może być poprzednie rozwiązanie, jak poniżej.

```
void bariera(int id) {
  for (int skok = 1; id + skok < N; skok *= 2) {
    if (id % (2 * skok) != 0) break;
    tsFetch("BARIERA %d", id + skok);
  }
  tsPut("BARIERA %d", id);
  tsRead("BARIERA %d", 0);
}
```

Rozwiązanie opiera się na hierarchicznym, logarytmicznym czekaniu aż wszystkie procesy dotrą do bariery. Jako ostatni dociera proces 0, stąd instrukcja wstrzymująca **tsRead**.

To rozwiązanie ma dwie wady. Po pierwsze, będzie ono działało jedynie raz. Przy drugim wywołaniu, krotka `"BARIERA 0"` będzie już w przestrzeni, więc instrukcja wstrzymująca się nie zablokuje. Po drugie, usunięcie tej krotki z przestrzeni nie jest takie proste. Dokładniej, musimy mieć gwarancję, że już żaden inny proces nie będzie jej potrzebował. To sugeruje, iż powinniśmy mieć kolejną barierę.

Poprawione rozwiązanie rozwiązuje te dwa problemy następująco. Po pierwszy, bariery mają swoje numery (epoki) – każde wywołanie funkcji bariery zwiększa numer epoki. Po drugie, kolejne wywołanie bariery usuwa (w odpowiednim momencie) krotki bariery z poprzedniej epoki.

```
void bariera2(int id) {
  static int epoka = 0;
  for (int skok = 1; id + skok < N; skok *= 2) {
    if (id % (2 * skok) != 0) break;
    tsFetch("BARIERA %d %d", epoka, id + skok);
  }
  tsPut("BARIERA %d %d", epoka, id);
  tsRead("BARIERA %d %d", epoka, 0);
  if (id == 0)
    tsTryFetch("BARIERA %d %d", epoka - 1, 0);
}
```

W tym rozwiązaniu, funkcja bariery może być wołana wielokrotnie. Po każdym jej zakończeniu przez wszystkie procesy, w przestrzeni krotek pozostanie dokładnie jedna krotka `"BARIERA %d %d"` – liczba nieodśmieconych krotek bariery będzie stała, niezależnie od tego, ile razy funkcja bariery była poprzednio wołana. Niemniej jednak takie rozwiązanie może być nadal niezadowalające.

Poniższe rozwiązanie eliminuje tę wadę, zwiększając dwukrotnie (ale nie asymptotycznie) złożoność operacji bariery.

```
void bariera3(int id) {
  for (int faza = 1; faza <= 2; ++faza) {
    for (int skok = 1; id + skok < N; skok *= 2) {
      if (id % (2 * skok) != 0) break;
      tsFetch("BARIERA %d %d", faza, id + skok);
    }
    if (faza == 1) {
      tsPut("BARIERA %d %d", faza, id);
      tsRead("BARIERA %d %d", faza, 0);
    } else {
      if (id != 0)
        tsPut("BARIERA %d %d", faza, id);
      else
        tsFetch("BARIERA %d %d", faza - 1, 0);
    }
  }
}
```

Rozwiązanie opiera się na dwóch fazach. Jeśli jakikolwiek proces rozpoczyna fazę drugą, to wszystkie procesy, włączając w to proces 0, wykonały już instrukcję **tsPut** z fazy pierwszej – zapewniona jest właściwość bariery. Jeśli natomiast proces 0 w fazie drugiej wykonuje ostatnią instrukcję **tsFetch** dla krotki wygenerowanej w fazie pierwszej, to wszystkie pozostałe procesy wykonały **tsPut** z fazy drugiej, zakończyły więc fazę pierwszą, a zatem usuwana przez proces 0 krotka z fazy pierwszej nie będzie już potrzebna żadnemu procesowi do **tsRead** z fazy pierwszej. Innymi słowy, po zakończeniu funkcji przez proces 0 w przestrzeni nie zostanie żadna krotka `"BARIERA %d %d"`.