Programowanie współbieżne

Ćwiczenia 11 – Przestrzenie krotek cz. 1

Zadanie 1: Obliczanie całki oznaczonej

Rozważmy iteracyjne obliczanie całki oznaczonej na przedziale $[a, b]$ metodą trapezów. Krok iteracji polega na obliczeniu pola trapezu wyznaczonego przez punkty a, b, $f(a)$, $f(b)$ i porównaniu go z sumą pól trapezów zbudowanych na przedziałach $[a, c]$ i $[c, b]$, gdzie c jest środkiem przedziału $[a, b]$. Jeśli różnica jest większa od zadanej wielkości ϵ, obliczamy rekurencyjnie całki na przedziałach $[a, c]$ i $[c, b]$, z dokładnością $\frac{\epsilon}{2}$ każdą, i sumujemy je. Obliczenie tych całek powinno odbywać się współbieżnie. Cały proces powinien być realizowany przez pewną liczbę nierozróżnialnych wykonawców pobierających zlecenia z przestrzeni krotek i tam umieszczających zarówno kolejne zlecenia jak też wyniki. Inicjatorem obliczeń i odbiorcą ostatecznych wyników jest odrębny proces zleceniodawcy, który w tym celu wywołuje funkcję **double** calka(**double** a, **double** b, **double** epsilon). Celem zadania jest więc napisanie treści w. w. funkcji oraz procesu wykonawcy. Wykonawca powinien działać w pętli nieskończonej.

```
double calka(double a, double b, double epsilon) { /* wołana przez proces zleceniodawcy */
  double fa = f(a), fb = f(b);
  double x, calka, xp, xk, w;
  tsPut("ZLECENIE %f %f %f %f %f %f", a, b, fa, fb, (fa + fb) * (b - a) / 2, epsilon);
  x = a;
  calka = 0.0;
  do {
    tsFetch("WYNIK ?f ?f ?f", &xp, &xk, &w);
    calka += w;
    x += xk - xp;
  } while (x != b);
  return calka;
}

process wykonawca() {
  double a, b, c, fa, fb, fc, pab, pac, pcb, epsilon;
  do {
    tsFetch("ZLECENIE ?f ?f ?f ?f ?f ?f", &a, &b, &fa, &fb, &pab, &epsilon);
    c = (a + b) / 2;
    fc = f(c);
    pac = (fa + fc) * (c - a) / 2;
    pcb = (fc + fb) * (b - c) / 2;
    if (fabs(pab - (pac + pcb)) > epsilon) {
      tsPut("ZLECENIE %f %f %f %f %f %f", a, c, fa, fc, pac, epsilon / 2);
      tsPut("ZLECENIE %f %f %f %f %f %f", c, b, fc, fb, pcb, epsilon / 2);
    } else {
      tsPut("WYNIK %f %f %f", a, b, pab);
    }
  } while (1);
}
```

W tym rozwiązaniu w funkcji calka pojawia się jedna dość subtelna kwestia. Ze względu na zaokrąglenia wyników operacji na liczbach rzeczywistych, może się okazać, że po zsumowaniu wszystkich odcinków „nie trafimy” w punkt b. Aby uniknąć takiej sytuacji, możemy odbierać wyniki zgodnie z kolejnością podprzedziałów. Odpowiedni fragment w. w. funkcji przyjmie wtedy następującą postać.

```
xp = a;
calka = 0.0;
do {
  tsFetch("WYNIK %f ?f ?f", xp, &xk, &w);
  calka += w;
  xp = xk;
} while (x != b);
```

Zadanie 2: Sortowanie bąbelkowe

W przestrzeni krotek znajduje się ciąg N liczb. Każda liczba to krotka w formacie "LICZBA %d %d", gdzie pierwszy element to unikalny indeks tej liczby w ciągu (z przedziału $0 \ldots N-1$) a drugi to wartość tej liczby. Celem zadania jest napisanie współbieżnego algorytmu sortowania bąbelkowego tego ciągu.

W wersji sekwencyjnej algorytm działa w przebiegach. W każdym przebiegu, $N-1$ razy wykonuje się operację porównywania i ewentualnej zamiany elementów – pierwszego z drugim, drugiego z trzecim, trzeciego z czwartym, itd. Po dojściu do końca ciągu, przebieg się kończy i rozpoczynany jest następny, ale jedynie jeśli w dopiero co zakończonym dokonano co najmniej jednej zamiany.

W wersji współbieżnej proces sortowania powinien być realizowany przez pewną liczbę nierozróżnialnych wykonawców. W efekcie dla dowolnych krotek "LICZBA i v" i "LICZBA j w", powinien być spełniony warunek: jeśli $i < j$, to $v \leq w$. Dodatkowo, po zakończeniu sortowania, wszystkie procesy wykonawców powinny się poprawnie zakończyć. Nie musimy natomiast przejmować się sprzątaniem dodatkowych krotek wytworzonych podczas działania algorytmu.

```
process wykonawca() {
  bool   koniec;
  int    i, j, v, w;
  while (! tsTryRead("KONIEC")) {
    koniec = true;
    i = 0;
    tsFetch("LICZBA %d ?d", i, &v);
    for (j = 1; j < N; ++j) {
      tsFetch("LICZBA %d ?d", j, &w);
      if (v > w) {
        int tmp = v;
        v = w;
        w = tmp;
        koniec = false;
      }
      tsPut("LICZBA %d %d", i, v);
      i = j;
      v = w;
    }
    tsPut("LICZBA %d %d", i, v);
    if (koniec) {
      tsPut("KONIEC");
    }
  }
}
```

Powyższe rozwiązanie nie pozwala wykonawcom na „wyprzedzanie się” wzajemnie. Zakończenie sortowania wymuszane jest przez pojawienie się krotki "KONIEC". Niestety jej wstawienie do przestrzeni dokonywane jest dopiero po sprawdzeniu całego ciągu. W tym czasie wielu innych wykonawców mogło już niepotrzebnie zacząć kolejny przebieg. Nie będziemy starali się tego wyeliminować, ponieważ sortowanie bąbelkowe generalnie nie jest najefektywniejszą metodą sortowania.

Zadanie 3: Problem hetmanów

Problem hetmanów polega na rozmieszczeniu N hetmanów na szachownicy $N \times N$ tak, aby żaden hetman nie mógł zbić innego zgodnie z zasadami gry w szachy. Celem zadania jest napisanie algorytmu współbieżnego znajdującego wszystkie rozwiązania tego problemu. Algorytm ma do dyspozycji pewną liczbę nierozróżnialnych procesów. Wszystkie wygenerowane przez nie odpowiedzi powinny być umieszczone w przestrzeni krotek. Pojedyncza odpowiedź powinna zawierać tablicę z ustawieniem wszystkich hetmanów. W rozwiązaniu można skorzystać z funkcji **bool** konflikt(**int** w1, **int** k1, **int** w2, **int** k2), która informuje, czy hetman z pola $(w1, k1)$ bije hetmana stojącego na $(w2, k2)$, gdzie $w1$ i $w2$ to numery wierszy a $k1$ i $k2$ to numery kolumn. Procesy realizujące algorytm nie muszą się kończyć.

W naszym rozwiązaniu procesy będą realizować zlecenia, które są opisane przez krotki w formacie `"ZLECENIE %d[%d] %d %d"` i odpowiadają częściowym rozwiązaniom problemu. Pierwszy element takiej krotki to n-elementowa tablica ($0 \leq n < N$) liczb całkowitych z zakresu $0 \ldots N-1$, której i-ty element ($0 \leq i < n$) to indeks wiersza hetmana ustawionego w i-tej kolumnie. Innymi słowy, krotka odpowiada częściowemu rozwiązaniu, w którym hetmani zostali już umieszczeni w n pierwszych kolumnach. Drugi i trzeci element krotki, natomiast, to indeks odpowiednio pierwszego i ostatniego wiersza w $i+1$-szej kolumnie, w którym można próbować umieścić hetmana, aby otrzymać rozwiązanie dla $n+1$ hetmanów. Zakładamy, że początkowo przestrzeń krotek zawiera pojedynczą krotkę, której pierwszym elementem jest 0-elementowa tablica, drugim elementem jest liczba 0, a trzecim – liczba $N-1$.

```
#define GRANULARNOSC 4 /* liczba równa co najmniej 1 */
process wykonawca() {
  int   t[N];     /* częściowe rozwiązanie */
  int   n;        /* liczba hetmanów w częściowym rozwiązaniu */
  int   pw, ow;   /* pierwszy i ostatni wiersz do sprawdzenia */
  do {
    tsFetch("ZLECENIE ?d[?d] ?d ?d", &n, t, &pw, &ow);
    if (ow − pw + 1 > GRANULARNOSC) {
      int sw = (pw + ow) / 2;
      tsPut("ZLECENIE %d[%d] %d %d", n, t, pw, sw);
      tsPut("ZLECENIE %d[%d] %d %d", n, t, sw + 1, ow);
    } else {
      for (int w = pw; w <= ow; ++w) {
        /* to sprawdzenie też można próbować zrównoleglić, */
        /* ale na potrzeby zajęć darujemy sobie to         */
        bool konf = false;
        for (int k = 0; k < n && !konf; ++k)
          konf = konflikt(w, n, k, t[k]);
        if (!konf) {
          t[n] = w;
          if (n < N − 1)  /* częściowe rozwiązanie */
            tsPut("ZLECENIE %d[%d] %d %d", n + 1, t, 0, N − 1);
          else            /* pełne rozwiązanie */
            tsPut("ROZWIAZANIE %d[%d]", N, t);
        } /* if ! konf */
      } /* for w */
    } /* if */
  } while (1);
} /* process */
```

Zastanów się, jak wykorzystując dodatkowy proces koordynatora sprawić, aby procesy realizujące algorytm (w tym koordynator) się skończyły.

Zadanie 4: Problem jedzących filozofów

Zakładamy, że mamy N filozofów. Filozofowie są rozróżnialni – każdy ma swój indeks z przedziału $0 \ldots N-1$. Przestrzeń krotek natomiast zawiera N krotek odpowiadającym widelcom w formacie "WIDELEC \%d", gdzie \%d przyjmuje wartości z przedziału $0 \ldots N-1$.

Rozważmy rozwiązanie z procesem lokaja, który będzie przyjmował zgłaszających się filozofów, decydując, kiedy pozwolić usiąść im przy stole.

```
enum {
  GLODNY,    /* filozof zgłaszający się do lokaja chce usiąść */
  NAJEDZONY  /* filozof zgłaszający się do lokaja chce wstać */
};

process lokaj() {
  int wolneMiejsca = N − 1;     /* dowlona wartość od 1 do N−1 */
  int kto;                      /* indeks zgłaszającego się filozofa */
  int co;                       /* czego chce zgłaszający się filozof */
  do {
    if (wolneMiejsca > 0) {
      tsFetch("ZGŁOSZENIE ?d ?d", &co, &kto);
    } else {
      co = NAJEDZONY;
      tsFetch("ZGŁOSZENIE %d ?d", co, &kto);
    }
    switch (co) {
    case GLODNY:
      −−wolneMiejsca;
      tsPut("ZGODA %d", kto);
      break;
    case NAJEDZONY:
      ++wolneMiejsca;
      break;
    }
  } while (1);
}

process filozof(int i) {        /* i − indeks filozofa 0... N−1 */
  do {
    myślenie();
    tsPut("ZGŁOSZENIE %d %d", GLODNY, i);
    tsFetch("ZGODA %d", i);
    tsFetch("WIDELEC %d", i);
    tsFetch("WIDELEC %d", (i + 1) % N);
    jedzenie ();
    tsPut("WIDELEC %d", i);
    tsPut("WIDELEC %d", (i + 1) % N);
    tsPut("ZGŁOSZENIE %d %d", NAJEDZONY, i);
  } while (1);
}
```

Które z operacji **tsPut** w procesie filozofa mogą mieć inną kolejność? Które z operacji **tsFetch** w procesie filozofa mogą mieć inną kolejność? Kiedy najwcześniej można wykonać **tsPut** na krotce informującej, że filozof jest NAJEDZONY?

Wadą tego rozwiązania jest to, że dostęp do stołu odbywa się sekwencyjnie – lokaj wpuszcza filozofów jednego po drugim. Spróbujmy zwiększyć współbieżność rozwiązania zastępując lokaja pulą biletów. Dokładniej, zakładamy że początkowo w przestrzeni krotek znajduje się M nierozróżnialnych biletów, gdzie $M \in \{1 \ldots N\}$, to jest krotek o formacie "BILET". Wtedy proces filozofa może wyglądać następująco.

```
process filozof(int i) {            /* i − indeks filozofa  0... N−1 */
  do {
    myślenie();
    tsFetch("BILET");
    tsFetch("WIDELEC %d", i);
    tsFetch("WIDELEC %d", (i + 1) % N);
    jedzenie();
    tsPut("WIDELEC %d", i);
    tsPut("WIDELEC %d", (i + 1) % N);
    tsPut("BILET");
  } while (1);
}
```

Krotki biletów zwiększają współbieżność, ale stanowią dodatkowy narzut pamięciowy. Możemy się ich pozbyć, zmieniając schemat, według którego filozofowie podnoszą widelce.

```
process filozof(int i) {            /* i − indeks filozofa  0... N−1 */
  do {
    myślenie();
    if (i % 2 == 0) {
      tsFetch("WIDELEC %d", i);
      tsFetch("WIDELEC %d", (i + 1) % N);
    } else {
      tsFetch("WIDELEC %d", (i + 1) % N);
      tsFetch("WIDELEC %d", i);
    }
    jedzenie();
    tsPut("WIDELEC %d", i);
    tsPut("WIDELEC %d", (i + 1) % N);
  } while (1);
}
```

Które z rozwiązań będzie najefektywniejsze przy parzystej liczbie filozofów, każdy wykonujący się z tą samą prędkością?