Tomáš Plachetka

Thread-safe knižnice

Zásobník (konečných) integerov

Stack 1

void push(int element);	uloží jedno číslo (element) na vrch zásobníka
int pop();	vyberie jedno číslo z vrchu zásobníka
bool empty();	vráti TRUE keď zásobník je prázdny, inak FALSE

Stack 2

void push(int element);	uloží jedno číslo (element) na vrch zásobníka
void pop();	vyberie jedno číslo z vrchu zásobníka
int top();	prečíta číslo z vrchu zásobníka (no nevyberie ho); vráti NULL keď zásobník je prázdny (NULL je hodnota, ktorú obsahuje každý typ, v tomto prípade int)

Zdá sa, že je len vecou vkusu, ktorú z týchto dvoch štruktúr použijeme (a ako ju prípadne rozšírime o ďalšie funkcie);
t.j. akým štýlom budeme písať programy používajúce zásobník

Príklady použitia Stack 1 a Stack 2

Príklad použitia Stack 1

```
int i, n;
for (i = 0; i < 1000; i = i + 1)
{
  ... read n...
  if (n < 100)
  {
    push(n);
  }
}
while (! empty())
{
  n = pop();
  ... compute f(n) ...
}
```

Príklad použitia Stack 2

```
int i, n;
for (i = 0; i < 1000; i = i + 1)
{
  ... read n ...
  if (n < 100)
  {
    push(n);
  }
}
n = top();
while (n != NULL)
{
  pop();
  ... compute f(n) ...
  n = top();
}
```

Ďalej budeme uvažovať Stack 1

Implementácia zásobníka

Reprezentácia zásobníka: pole veľkosti 1000000 + hĺbka zásobníka *depth*

```
int stack[1000000];
int depth = 0; /* Stack je na začiatku prázdny */

void push(int element)
{ /* Nestaráme sa o pretečenie, to nie je teraz dôležité */
  stack[depth] = element;
  depth = depth + 1;
}

int pop()
{ /* Predpokladáme, že depth > 0. O toto sa stará používateľ */
  depth--;
  return stack[depth];
}

bool empty()
{
  return depth == 0;
}
```

Threads

Malá sada funkcií (knižnica), ktoré nemusia byť vyjadriteľné v jazyku C, ale dajú sa v C používať

```
int thr_create(void f());
int my_thread_id();
```

…

Volanie thr_create(void f(void)) vyrobí nový tok riadenia (thread), ktorý začína volaním funkcie f(), tým volanie thr_create() skončí a vráti identifikátor nového threadu

Nový a pôvodný tok riadenia zdieľajú všetky globálne premenné programu.

Vykonávajú sa súbežne, pričom **rýchlosť vykonávania môže byť rôzna a program nemá možnosť ju ovplyvniť**. O tom, ktorý tok vykoná nasledujúcu inštrukciu, rozhoduje externý scheduler (dokonca smie vykonať inštrukcie z viacerých threadov naraz). **Scheduler garantuje len to, že neprestane vykonávať inštrukcie, kým všetky thready neskončia**

Rozdiel oproti PRAM modelom (obzvlášť nedeterministickým): nedeterminizmus v PRAM „nám pomáha“, ale tu **„hráme proti nedeterminizmu“**

Použitie zásobníka v programe s 2 riadeniami (threadmi)

```
input()
{
  int i, n;
  for (i = 0; i < 1000; i = i + 1)
  {
    ... read n...
    if (n < 100)
    {
      push(n);
    }
  }
}
```

```
compute()
{
  int n;
  while (! empty())
  {
    n = pop();
    ... compute f(n) ...
  }
}
```

input() a compute() sa vykonávajú súbežne, každý vlastným tempom (ktoré používateľ nepozná a nevie ho ovplyvniť). Majú vlastné lokálne premenné (i, n), ale **zdieľajú zásobník**

Problém: compute() skončí akonáhle empty() vráti prvýkrát TRUE. Napríklad hneď, ak je rýchlejší než input(). **Lenže compute() má skončiť až keď bol celý vstup prečítaný a spracovaný**

Riešenie problému s predčasným ukončením

compute() má skončiť až keď input() prečítal celý vstup

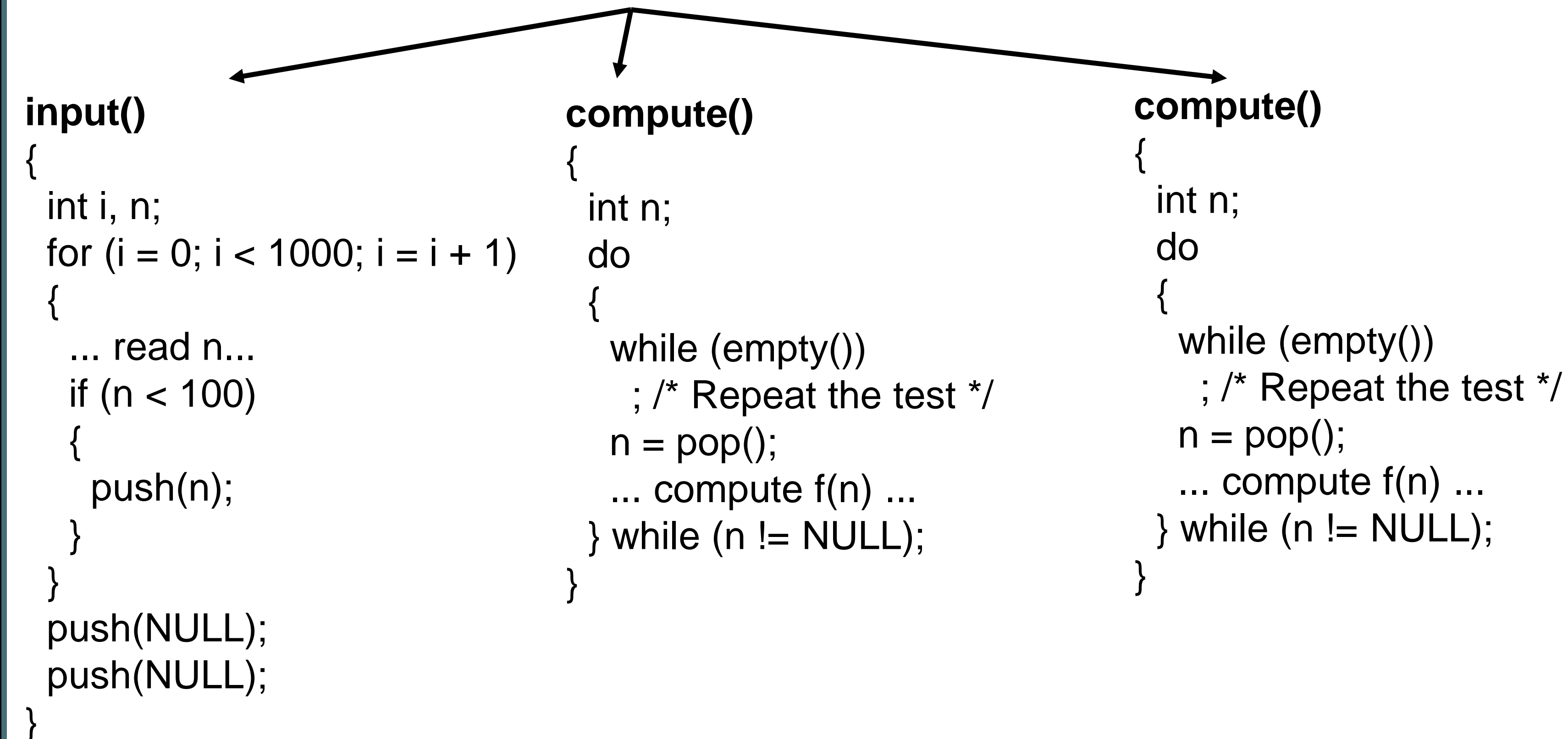
Riešenie: input() po prečítaní vstupu vloží do zásobníka nejakú špeciálnu hodnotu, napr. NULL. compute() skončí až keď vyberie NULL

```
input()
{
  int i, n;
  for (i = 0; i < 1000; i = i + 1)
  {
    ... read n...
    if (n < 100)
    {
      push(n);
    }
  }
  push(NULL);
}
```

```
compute()
{
  int n;
  do
  {
    while (empty())
      ; /* Repeat the test */
    /* The stack is now nonempty */
    n = pop();
    ... compute f(n) ...
  } while (n != NULL);
}
```

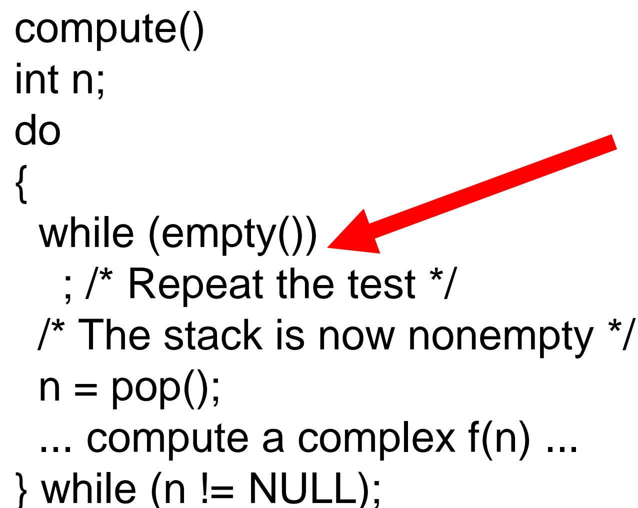
input(), compute(), compute()

Ešte jedno vylepšenie: keď input() po ukončení čítania vstupu vloží do zásobníka *x* hodnôt NULL, môžeme spustiť *x* inštancií compute(). Nech x = 2



Avšak toto nefunguje. Uvažujme takýto beh (toto sa môže stať, lebo v prvom aj druhom compute() súčasné volania empty() vrátia FALSE a stane sa 2x pop):
push(1)$_{1}$; empty()$_{2}$; empty()$_{3}$; pop()$_{2}$; pop()$_{3}$

```
input()
int i, n;
for (i = 0; i < 1000; i = i + 1)
{
  ... read n...
  if (n < 1000)
  {
    push(n);
  }
}
push(NULL);
```



```
compute()
int n;
do
{
  while (empty())
    ; /* Repeat the test */
  /* The stack is now nonempty */
  n = pop();
  ... compute a complex f(n) ...
} while (n != NULL);
```

Ani toto nefunguje. while(empty()) sa môže zacykliť aj keď je zásobník neprázdny

Dôvod: kompilátor nevie o tom, že tento program nie je sekvenčný. Je oprávnený optimalizovať každú funkciu lokálne, t.j. nahradiť časť sekvenčného kódu ľubovoľným ekvivalentným kódom. Napríklad, **"while (empty()) ;"** môže preložiť rozvinutím empty() ako **"while (depth == 0) ;"**. Keďže premenná depth sa v tele cyklu nemení, môže to ekvivalentne preložiť aj takto:

```
if (depth == 0)
{
  while (TRUE)
         ;
}
```

Vzájomné vylúčenie: Dekkerov a Petersenov algoritmus

Problém spočíva v tom, že viaceré thready pristupujú s pamäti (k premenným) zdieľaného zásobníka súčasne, resp. nevhodnou postupnosťou inštrukcií. Treba zabezpečiť, aby so zásobníkom manipulovali oddelene. Kým jeden thread manipuluje, ostatné thready musia čakať

Riešenia garantujúce vzájomné vylúčenie:
Dekkerov algoritmus pre vzájomné vylúčenie 2 threadov [~1960, unpublished paper referenced by Dijkstra in his manuscripts Over de sequentialiteit van procesbeschrijvingen,1962 and Cooperating sequential processes, 1963]

Petersonov algoritmus pre vzájomné vylúčenie 2 threadov [Petersen: Myths About the Mutual Exclusion Problem, Information Processing Letters, 1981]

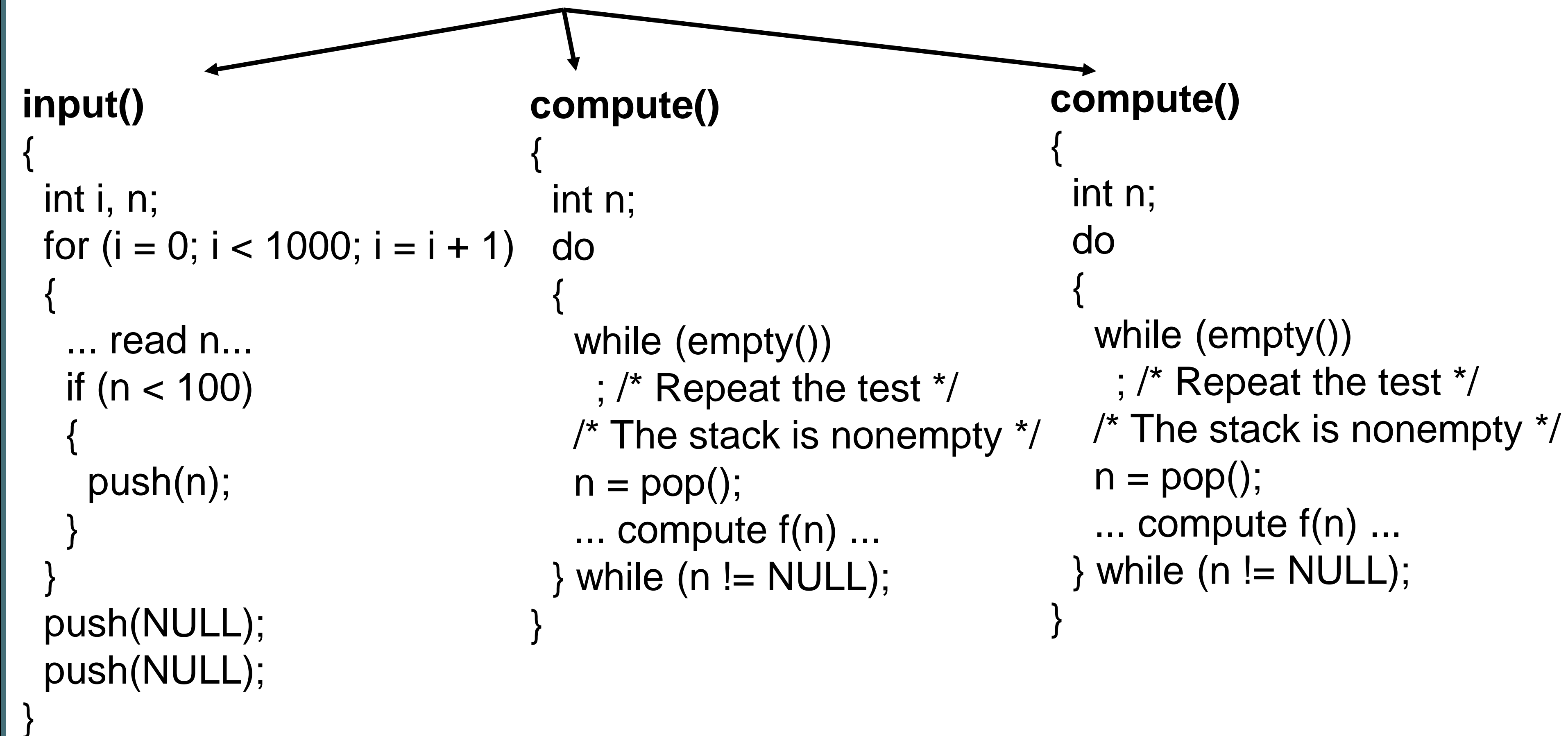
Oba tieto algoritmy sa dajú zovšeobecniť pre ľubovoľný počet threadov (bakery algorithm, ticket algorithm)

Kritické časti programu sú v našom prípade telá funkcií push(), pop() a empty()

Implementácia thread-safe zásobníka

```
int stack[1000000]; int depth = 0; /* Stack je na začiatku prázdny */
/* Deklarácie Petersonových premenných, definície Petersonových funkcií enter(), leave() */
void push(int element)
{
  enter(my_thread_id);
  stack[depth] = element; /* Nestaráme sa o pretečenie, to nie je teraz dôležité */
  depth = depth + 1;
  leave(my_thread_id);
}

int pop()
{
  enter(my_thread_id);
  depth--; /* Predpokladáme, že depth >= 0. O toto sa stará používateľ */
  return stack[depth];
  leave(my_thread_id);
}

bool empty()
{
  enter(my_thread_id);
  return depth == 0;
  leave(my_thread_id);
}
```

Thread-safe zásobník: 1 input(), 2 compute()



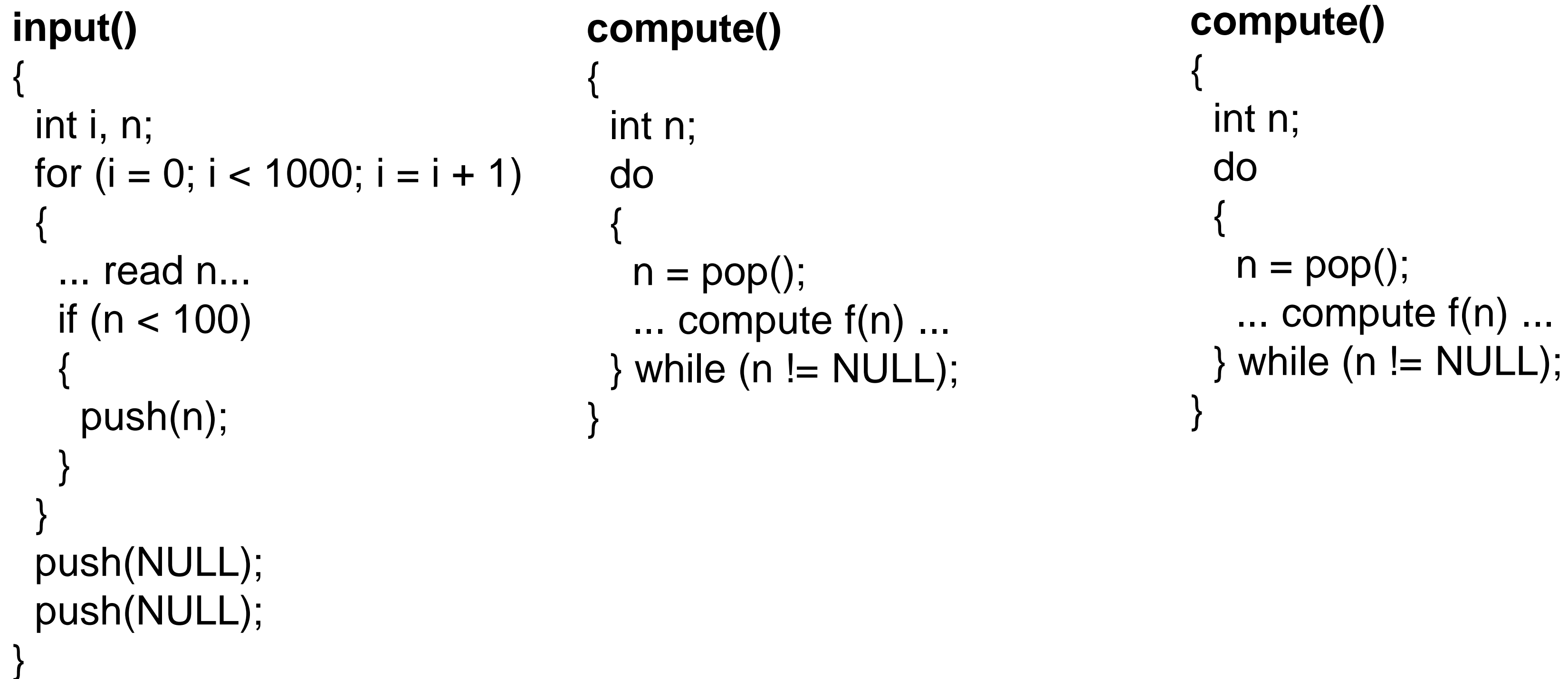
Nič sme nevyriešili, problém ostal, stále kvôli behu
push(1)$_1$; empty()$_2$; empty()$_3$; pop()$_2$; pop()$_3$

Skutočná príčina: Funkcia **empty() vracia neaktuálnu informáciu**

Thread-safe zásobník bez empty()

```
int stack[1000000]; int depth = 0; /* Stack je na začiatku prázdny */
/* Deklarácie Petersonových premenných, definície Petersonových funkcií enter(), leave() */

void push(int element)
{
  peterson_enter(my_thread_id);
  stack[depth] = element; /* Nestaráme sa o pretečenie, to nie je teraz dôležité */
  depth = depth + 1;
  peterson_leave(my_thread_id);
}

int pop()
{
  peterson_enter(my_thread_id);
  while (depth == 0) /* Nepredpokladáme, že depth >= 0, túto situáciu riešime v zásobníku */
  {
    peterson_leave(my_thread_id);
    peterson_enter(my_thread_id);
  }
  depth--;
  return stack[depth];
  peterson_leave(my_thread_id);
}
```

Thread-safe zásobník bez empty()



Toto nielenže funguje, ale zároveň zjednodušuje implementáciu aj použitie zásobníka.

Dokonca to zjednodušuje aj interface zásobníka: ušetrili sme funkciu empty() aj jej implementáciu

Používateľovi Stack 1 nevadí absencia funkcie empty(), lebo si môže mimo zásobníka počítať počet volaní push() a pop()

Lenže vyrobili sme iný problém. Časová zložitosť je ∞, lebo pop() v najhoršom prípade robí neohraničene veľa krokov

Neohraničená časová zložitosť pop()

```
int pop()
{
  enter(my_thread_id);
  while (depth == 0)
  {
    leave(my_thread_id);
    enter(my_thread_id);
  }
  depth--;
  return stack[depth];
  leave(my_thread_id);
}
```

Pripomeňme si: program musí fungovať pre akýkoľvek schedule (hráme **proti** nedeterminizmu).

Lenže keď je zásobník prázdny a scheduler vykonáva vyššie označené 2 riadky vždy tesne za sebou, tak zásobník ostane navždy prázdny a horeuvedený while nikdy neskončí. A to aj v prípade, keď súčasne iný thread vykonáva push(), lebo ten sa pred uložením do zásobníka cyklí vo svojom enter()

Dijkstrove riešenie: semafory

Semafor [Dijkstra, manuscript Over de sequentialiteit van procesbeschrijvingen,1962] je dátová štruktúra, ktorá ponúka nasledujúce **3 funkcie. Dôležité je, že každá z nich sa vykonáva v jednom atomickom kroku**:

```
sem_init(semaphore s, int value);
sem_wait(semaphore s);          /* P(s) */
sem_signal(semaphore s);        /* V(s) */
```

```
int sem_value;

sem_init(int value)
{
  sem_value = value;
}

sem_signal(semaphore s)
{
  sem_value = sem_value + 1;
  if (sem_value > 0)
    ... resume execution of some thread
    waiting on this semaphore (if any)...
}
```

```
sem_wait(semaphore s)
{
  sem_value = sem_value - 1;
  if (sem_value < 0)
  {
    ...suspend execution of the current thread,
    wait on semaphore s...
  }
}
```

Threads so semaformi

Malá sada funkcií (knižnica), ktoré nemusia byť vyjadriteľné v jazyku C, ale dajú sa v C používať

```
int thr_create(void f(void));
int my_thread_id();
sem_init(semaphore s, int value);
sem_wait(semaphore s);    /* P(s) */
sem_signal(semaphore s); /* V(s) */
...
```

Implementácia thread-safe zásobníka so semaformi

```
int stack[1000000]; int depth = 0; /* Stack je na začiatku prázdny */
semaphore s_mutex, s_depth, s_max;
sem_init(s_mutex, 1);
sem_init(s_empty, 0);
sem_init(s_full, 1000000);

void push(int element)
{
  sem_wait(s_full); /* Staráme sa dokonca aj o to, aby zásobník nepretiekol, hoci to nie je teraz dôležité */
  sem_wait(s_mutex);
  stack[depth] = element;
  depth = depth + 1;
  sem_signal(s_empty);
  sem_signal(s_mutex);
}

int pop()
{
  int element;
  sem_wait(s_empty); /* Keď je zásobník prázdny, čakáme na semafore kým nie je čo vyberať */
  sem_wait(s_mutex);
  depth--;
  element = stack[depth];
  sem_signal(s_full);
  sem_signal(s_mutex);
}
```

Thread-safe zásobník bez empty()

```
input()
{
  int i, n;
  for (i = 0; i < 1000; i = i + 1)
  {
    ... read n...
    if (n < 100)
    {
      push(n);
    }
  }
  push(NULL);
  push(NULL);
}
```

```
compute()
{
  int n;
  do
  {
    n = pop();
    ... compute f(n) ...
  } while (n != NULL);
}
```

```
compute()
{
  int n;
  do
  {
    n = pop();
    ... compute f(n) ...
  } while (n != NULL);
}
```

Problémy sú vyriešené. Použitie zásobníka sa nezmenilo, zmenila sa len jeho implementácia (dokonca sa opäť zjednodušila). Stačí použiť semafory
Časová zložitosť (celkový počet krokov) horeuvedeného programu je v najhoršom prípade konštantná

Súčasné počítače (procesory aj operačné systémy) ich ponúkajú a efektívne implementujú. To znamená, že thread, ktorý čaká na semafore v sem_wait(), skutočne nerobí žiadne kroky (nezaťažuje procesor, nekonzumuje elektrický prúd, ...)

Takmer-Resume

Resume: Implementovali sme zásobník, ktorý sa dá používať aj v programoch s viacerými threadmi. Toto zovšeobecnenie dokonca zjednodušilo interface zásobníka, prirodzene sme prišli na to, že funkciu empty() zásobník v skutočnosti nemusí ponúkať. Semaforová implementácia je efektívna, t.j. neovplyvňuje výrazne zložitosť programov, ktoré používajú zásobník (časovú ani pamäťovú)

Takto sa to učí snáď vo všetkých knihách o programovaní, operačných systémoch atď.

Pozrime sa ešte raz na efektivitu. **Paradoxne, zlý prípad pre semaforovú implementáciu zásobníka je sekvenčný program (program s 1 threadom).** Vráťme sa k nemu. Funkciu empty() nemáme. Nepotrebujeme ju, dokonca v programe nepotrebujeme priebežne aktualizovať hĺbku zásobníka spolu s každým volaním push() a pop(). Hodnoty n prečítané zo vstupu stačí len ukladať do zásobníka a následne vyberať

Obe funkcie push() a pop() majú konštantnú zložitosť. Lenže keď sa v analýze časovej zložitosti hrá o konštanty, tak **akékoľvek zvýšenie časovej zložitosti** funkcií push() a pop() **považujeme za zlé**. Lenže presne to sme urobili v ich implementácii pridaním semaforových operácií. Navyše, **pomiešali sme dva druhy konštánt**: "semaforová konštanta" nepochádza zo sekvenčného výpočtového modelu. Poďme to zase napraviť

Finálna implementácia Stack 1

Interná reprezentácia zásobníka: pole veľkosti *STACK_SIZE*

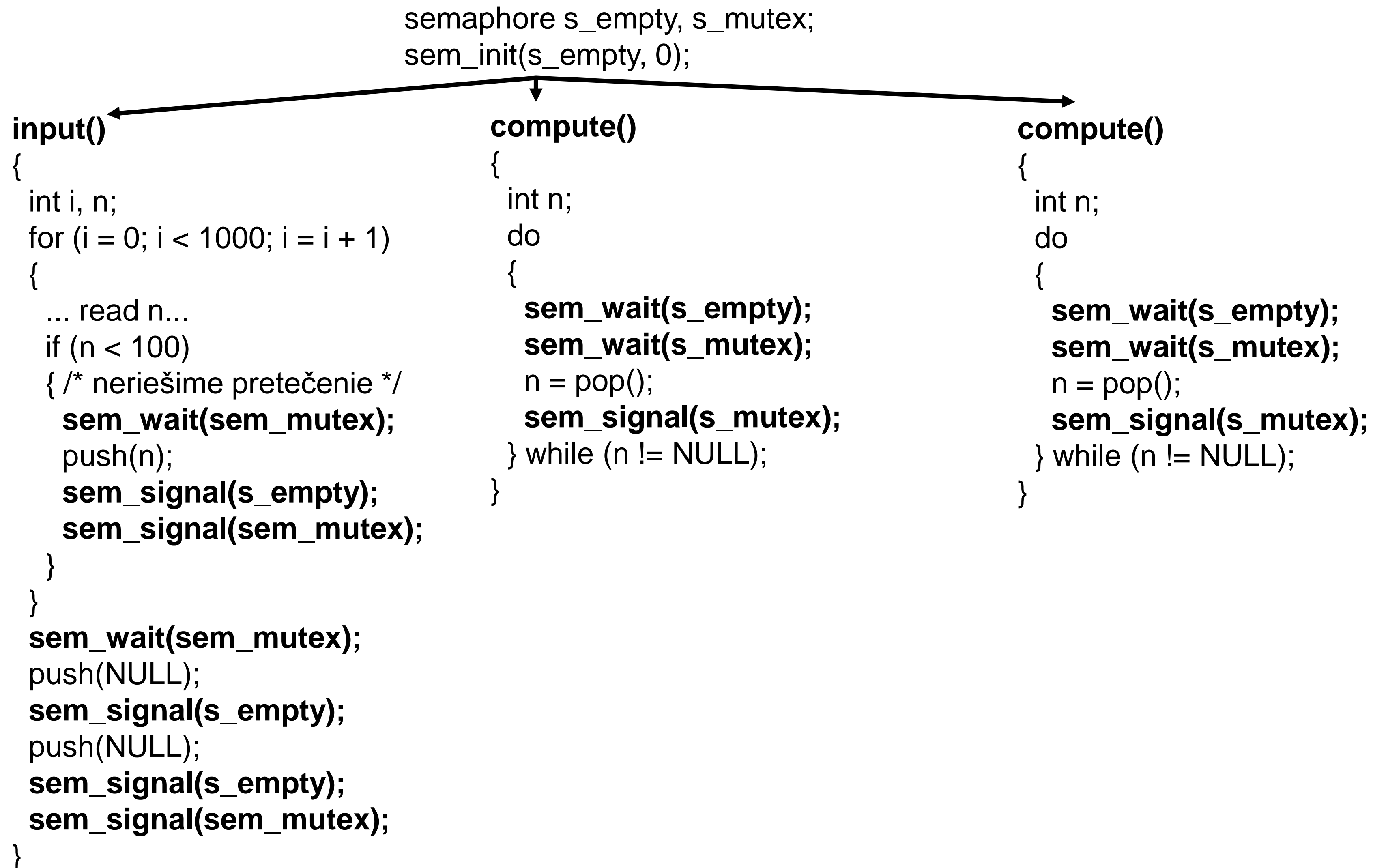
```
int stack[1000000];
int depth = 0; /* Stack je na začiatku prázdny */

void push(int element)
{
  stack[depth] = element; /* Nestaráme sa o pretečenie, o to sa stará používateľ */
  depth = depth + 1;
}

int pop()
{
  depth--; /* Predpokladáme, že depth >= 0. O toto sa stará používateľ */
  return stack[depth];
}
```

Hotovo

Použitie finálneho zásobníka v programe s 3 threadmi

Lenže sme tam, kde sme boli na začiatku. Teraz sa zásobník nedá použiť ani pre program s 2 threadmi

Čoby sa nedal. Dá sa, dokonca trošku efektívnejšie než predtým. Takto:



Programovanie s threadmi je **konzervatívne rozšírenie sekvenčných programov** o možnosť vytvorenia ďalšieho toku riadenia a o semafory. Nesmie ovplyvniť zložitosť sekvenčných programov, ani spôsob optimalizácie sekvenčného programu v kompilátore

Môže však pozitívne ovplyvniť spôsob organizácie dátových štruktúr (zjednodušiť a zefektívniť enkapsuláciu dátových štruktúr) v sekvenčných programoch

Súčasné trendy v teoretickej informatike a IT

Programovanie s threadmi sa nechápe ako **konzervatívne rozšírenie sekvenčných programov (algoritmov)** o možnosť vytvorenia ďalšieho toku riadenia a o semafory. Ovplyvňuje zložitosť sekvenčných programov aj spôsob optimalizácie sekvenčného programu v kompilátore

Procesory aj operačné systémy semafory ponúkajú. Považujú sa za "drahé", ale napriek tomu sa synchronizácia masovo používa v (thread-safe) knižniciach.
Súčasný teoretický výskum sa masovo orientuje na tzv. lock-free schémy aplikované na rôzne dátové štruktúry (čo dátová štruktúra, to lock-free schéma resp. niekoľko schém).

"Lock-free schéma" znamená použitie všelijakých modifikácií Petersenovho algoritmu, ktoré využívajú atomické inštrukcie test-and-set, compare-exchange a pod., tiež dostupné v súčasných procesoroch a operačných systémoch. Avšak správne fungujú len pre istú podtriedu počítačových systémov, pričom rôzne lock-free schémy predpokladajú rôzne podtriedy

Spoločnou vlastnosťou lock-free schém je, že transformujú časovú zložitosť akéhokoľvek algoritmu na neohraničenú, prinajmenšom v najhoršom prípade. Avšak skok trebárs z O(1) na ∞ sa pri analýze časovej zložitosti za drahý nepovažuje (hoci napr. zmena z O(1) na O(log log n) áno). Možno preto, že najhoršie prípady tiež vyšli z módy, lebo "nikdy nenastávajú"