Wprowadzenie do programowania w języku C
grupa RKr, wtorek 16:15-18:00
lista nr 5 *(na pracownię 12.11.2019) (wersja 4)*

Zadanie 1. [10p na pracowni lub 5p po pracowni**]**

*uwaga: Jeśli zrobisz **[A]**, to **[B]** będzie oczywiste. Jeśli zrobisz **[C]**, to **[D]** będzie oczywiste.*
Wszystkie makra muszą się rozwijać do wartości/wyrażeń, aby można było stosować złożenie.
*Makra mogą być type-free, ale w **[A]** i **[B]** warto założyć ustalony logiczny rozmiar słowa bitowego, np. 8 lub 4.*
Preferowane są rozwiązania bazujące na operacjach bitowych, ale zwykła arytmetyka też jest OK.
Makra muszą być używalne we instrukcjach i deklaracjach, więc nie wolno używać funkcji bibliotecznych.

[2.5p | **A]** Napisz makro **ROUND_DN_toPOT**(x), które zaokrągli wartość x do najbliższej potęgi dwójki z dołu.

[2.5p | **B]** Napisz makro **ROUND_UP_toPOT**(x), które zaokrągli wartość x do najbliższej potęgi dwójki z góry.

wskazówka: Wykorzystaj wiodący (najstarszy) zapalony bit zaokrąglanej liczby.
Przydatny będzie operator trójargumentowy „_?_ :_” , wykorzystaj go wiele razy i ładnie sformatuj.
Nazwy makr to skróty od “round down to power-of-two” i “round up to power-of-two”.
Jeśli liczba wejściowa jest potęgą dwójki, to makro nie powinno zmieniać tej wartości.

[2.5p | **C]** Napisz makro **ROUND_DN_toMPOT**(x, k), które zaokrągli wartość x z dołu do najbliższej wielokrotności potęgi dwójki określonej przez token k będący wykładnikiem tej potęgi, tj. 2^k.
Innymi słowy zaokrąglamy do największej liczby postaci $a*(2^k)$, dla której zachodzi ($a*(2^k) \le x$).

[2.5p | **D]** Napisz makro **ROUND_UP_toMPOT**(x, k), które zaokrągli wartość x z góry do najbliższej wielokrotności potęgi dwójki określonej przez token k będący wykładnikiem tej potęgi, tj. 2^k.
Innymi słowy zaokrąglamy do najmniejszej liczby postaci $b*(2^k)$, dla której zachodzi ($x \le b*(2^k)$).

wskazówka: Zastanów się jak wygląda reprezentacja bitowa liczby, która jest wielokrotnością potęgi dwójki.
Nazwy makr to skróty od “round down to multiple power-of-two” i “round up to multiple power-of-two”.
Jeśli liczba wejściowa jest wielokrotnością potęgi dwójki, to makro nie powinno zmieniać tej wartości.

Zadanie 2. [10p]
Napisz funkcję **RefBox allocatePOTMemAligned**(**int** expBbSize, **int** log2unit, **int** log2align), która dokonuje dynamicznej alokacji pamięci w taki sposób, że do użytku mamy liczbę bajtów, która jest najmniejszą potęgą dwójki zawierającą expBbSize, a wirtualny adres początkowy będzie wyrównany względem log2align, tj. wartość adresu ma być wielokrotnością liczby $(2^{log2align})$. Parametr log2align oznacza logarytm dwójkowy wyrównania pamięci. Dzięki wyrównaniu uzyskasz do dyspozycji kilka/naście bitów w adresie (wskaźniku), wykorzystaj je do zapamiętania logarytmu dwójkowego z finalnego rozmiaru pamięci mierzonego w jednostce, której wielkość jest równa $(2^{log2unit})$. Będzie Ci potrzebna poniższa definicja.

```
typedef struct { void* haxRef; void* natPtr; } RefBox;
```

Użyj tej funkcji do alokacji pamięci dla bufora cyklicznego (*ang. ring buffer, circular buffer*), a w implementacji bufora skorzystaj z rozmiaru alokacji zapamiętanego we wskaźniku.
Zaimplementuj następujące operacje (gdzie **Tval** to wybrany przez Ciebie typ komórki w buforze, np. **int**): tworzenie bufora „**RefBox create**(**int** expNumCells)”, niszczenie bufora „**void destroy**(**RefBox**)”, wstawianie „**void put**(**RingBuffer***, **Tval** [, **int** log2align])”, pobranie „**Tval get**(**RingBuffer***[, **int** log2align])”.

wskazówka: Alokację dynamicznej pamięci trzeba zrobić zwykłą funkcją malloc(.), a dealokację poprzez free(.).
Wykorzystaj niektóre z makr z zadania nr 1.

Zadanie 3. [10p] Dostępne w serwisie SKOS.

Dodatkowe informacje do zadania nr 2.

Funkcja malloc(.) przydziela Ci tyle pamięci w bajtach ile podasz jej przez parametr formalny (wejściowy). Więc pierwsza rzecz jaką należy zrobić to zaokrąglić **expBbSize** (expected byte-based size) do najbliższej potęgi dwójki w górę, nazwijmy to zaokrąglenie jako **POTBbSize** (power-of-two byte-based size). Chcemy ten rozmiar przechowywać w formie logarytmu dwójkowego, niestety liczba bitów potrzebna do reprezentacji logarytmu dwójkowego z POTBbSize może być większa niż log2align. Znaczy to, że jeśli chcielibyśmy zapisać **log2POTBbSze** na liczbie bitów równej log2align, to rozmiar alokacji ograniczony jest do $(2^{log2align})$ bajtów, co jest dość ubogie, np. jeśli nasze wyrównanie to 8 bajtów (log2align = 3), to maksymalny rozmiar alokacji również wynosi 8 bajtów (maksymalny log2POTBbSize wynosi log2align). Aby to obejść będziemy wyrażali nasz rozmiar w większych jednostkach, nazwiemy go **POTUbSize** (power-of-two unit-based size), a dwójkowy logarytm z tej wielkości będziemy nazywać **log2POTUbSize**. Teraz maksymalny rozmiar reprezentowalnej alokacji to (1 << log2POTUbSize)·(1 << log2unit), lub inaczej $(2^{log2POTUbSize})\cdot(2^{log2unit}) = (2^{log2POTUbSize + log2unit})$.

Kolejny problem jaki należy rozwiązać to fakt, że adres pamięci (wartość wskaźnika) może być ogólnie mówiąc i upraszczając dowolną wartością z wirtualnej przestrzeni adresowej programu, nie mamy wiedzy o tym co robi alokator systemu operacyjnego, na którym działa nasz program. Oznacza to, że nie możemy założyć niczego o wartościach bitów tego adresu, ten adres nazwijmy **nativeAddress**. Możemy jednak oszukać system operacyjny prosząc go o przydzielenie trochę większej ilości pamięci, ten nadmiar nazwiemy **waste**. Czyli cała alokacja ma mieć rozmiar (POTBbSize + waste). Parametr waste musi być na tyle duży, abyśmy mogli przesuwać wskaźnik reprezentujący początek naszego obszaru pamięci w staki sposób aby wyzerować dolne bity (dokładnie **log2align** bitów). Ten przesunięty adres nazwiemy **baseAddress**. Jeśli waste jest dostatecznie duży, to (baseAddress + POTBbSize) nie powinno przekroczyć (nativeAddress + POTBbSize + waste). Z drugiej strony chcemy aby waste był możliwie najmniejszy.

Typ strukturalny **RefBox** zawiera **haxRef**, którego wartość to nasz baseAddress zmontowany z log2POTUbSize na dolnych bitach, ten typ strukturalny zawiera też **natPtr** (native pointer), którego wartość numeryczna to nativeAddress. Pole natPtr jest potrzebne tylko do dealokacji poprzez funkcję free(.), a wszelkie odczyty z używanego przez nas obszaru pamięci należy robić poprzez **haxPtr** zerując dolne bity w **haxRef**. Należy znaleźć takie przesunięcie (**offset**), że baseAddress = nativeAddress + offset.

Bufor cykliczny to jedna z bardzo prostych i jednocześnie ważnych struktur danych, łatwo znaleźć niezbędne informacje w sieci, np. na Wikipedii. Dobra implementacja takiego bufora NIE zlicza ile bufor posiada elementów. Taki bufor potrzebuje zaledwie kilku informacji: adres początkowy bufora (u nas to będzie baseAddress zawarty w haxRef), pojemność bufora **capacity** (u nas to będzie wielkość wyliczona na podstawie log2POTUbSize zawartego w haxRef i rozmiaru typu reprezentującego komórkę naszego bufora), a także dwa indeksy read-index oraz write-index. Każdy z tych indeksów, po każdej operacji inkrementacji, powinien być przycinany do pojemności bufora poprzez operację modulo (_ % capacity). Możemy też założyć, że bufor nie będzie pokrywał całej dostępnej pamięci, ani nawet jej połowy, więc na jeden indeks możemy przeznaczyć liczbę bitów mniejszą/równą połowie liczby bitów wskaźnika. To oznacza, że deskryptor dla całego bufora zmieści się w rozmiarze około dwóch wskaźników (lub mniejszym).

```
typedef struct { void* haxRef; short readId; short writeId } RingBuffer;
```