Hanh Do Phung
University ID: 14252074
Email: hdd29@drexel.edu

CS 543: ASSIGNMENT 3
Implementing slab allocation memory management policy

SOFTWARE ARCHITECTURE:

1. Introduction

The purpose of this assignment is to implement an extension to the existing Inferno's memory management policy, by storing pointers to free blocks in a combination of a sorted binary tree (a heap) by block sizes and doubly linked circular lists. The extension, which is called slab allocation, will only handle part of the free blocks of size 1 quanta to 4096.

2. Software architecture

The way slab allocation works is to maintain a number of free blocks smaller than 4096 that can be access in constant time in its data structure. Whenever there is an allocation request made by a use application, via malloc(), the slab allocation technique will round up the request to its nearest power of two, k, then allocate in constant time a block of size 2^k.

There might be many different ways to implement the goal extension. But the clearest and probably cleanest way I can think of is to have an array of pointers that each point to a singly linked list of free blocks of certain sizes. Since the quanta used in Inferno is 32, which is 2^5, the array will start with its pointer at index 0 pointing to the list of blocks of size 32, index 1 point at list of 64-sized blocks etc. to the last index, 7.

To implement this array of pointer, I added another member, Bhdr * slab[8], to the Pool struct in alloc.c. The fwd pointer in Bhdr struct in pool.h is reused as the link pointer between block headers in each of the list within the slab. Note that the slab and the tree structure in the existing code are mutually exclusive, meaning there are no blocks that can be both in the slab and in the tree. To make this distinction, I set the prev pointer in Bhdr to nil whenever the block is added to the slab, in contrast to this prev pointer will never be nil when block is in the tree. Specific changes in the existing code is listed below:

1. pooladd(): this function now checks whenever a block being asked to be added to pool should be maintained by the slab or the tree, then add it to the appropriate place. The condition for being added to the slab is to have a size less than 4096 and more than 1 quanta, and is a power of 2 (check by a function called checkPowerOf2. The index in the slab array is calculated as log2(block_size) – 5. (as the beginning idx 0 is already 2^5). The bhdr will then be the inserted to head of that list for simplicity, and its prev pointer is set to nil.

2. dopoolalloc() : this is called to allocate a block to a request. If the request was for a block of size < 4096, the function round that size to the next power of 2 using a newly added function nextPowerOf2. Then, it will peak into the slab to check if there is a block of that size available at the head of the appropriate list and allocate from there by setting the magic number to A and returning D2B(block). If not available in slab, it will go to the tree and find a block based on the existing code.

3. poolfree() : this function is called whenever a block is free. It also checks if a block should be in the slab or not. If it is not supposed to be in the slab, the function will again check if the adjacent blocks to it are in the slab or not (by asking whether prev pointer is nil). If either or both the adjacent blocks are free and not in the slab, it will coalesce the block with its adjacent(s).
4. 3 added helper functions: nextPowerOf2, checkPowerOf2, and printSlab.

The slab should start out empty and will get more blocks added to it when applications free memory (allocated from tree before) which satisfies the conditions to be in a slab.

TESTING and EVALUATION

The scheduler was mostly debugged using a function called printSlab(Bhdr ** slab) which prints the sizes of blocks in the slab in their respective buckets in the form of:
[idx] 2^k : block1size → block2size → …
...
…
[8] 4096: ...

The scheduler was tested on functionality by running the emu, with multiple commands enter at the prompt ; I also have tested the GUI by calling wm/wm to display some windows which are memory hungry. All seems to work and the print function printed out the expected results (of blocks being added then removed from slab)