@unca.edu

Matt Lineback <mlinebac@unca.edu>

CSCI 331: Advice on memory HW: allocate and deallocate cases

1 message

Adam Whitley <awhitley@unca.edu> Sat, Apr 29, 2017 at 5:49 PM
Bcc: mlinebac@unca.edu

Hey Operating Systems students,

I think it might be warranted to send this email of advice about the memory manager homework. The deallocate and allocate methods involve a lot of linked list manipulation and several special cases, so I'll give you my suggestion on how to split up your logical cases. Please **make a new email rather than replying to this one**, or I could end up with a giant conversation tree rooted at this email, with several independent conversations balled into one "gmail conversation."

Allocate.

Much like lab 6, there's error checking for invalid pid, invalid u, and a pid that's already been allocated an address space.

You're going to be using a loop to iterate through the linked list one node at a time to find a free section at least big enough for the allocation amount u. In each iteration, you'll need the current node in the list from list.get(i).

You can skip over any iteration where the current node is an already-allocated address space rather than a free section, by checking the pid of the current node and using continue;

If the current node is a free section of the **exact** size u that was requested, you merely need to change the owner of the current node from being a free section to being owned by the requested pid.

If the current node is a free section **bigger** than the size u that was requested, then you need to "adjust" that free section to start u units later and be u units smaller. Also, you'll need to make a new linked list MemorySection node to represent the new address space, and insert it just **before** that free section node, using list.add with a newly constructed MemorySection object.

If your loop ended, you couldn't allocate anything because not enough room.

Deallocate.

Much like lab 6, there's error checking for an invalid value for parameter u.

You're going to be using a loop to iterate through the list to find where the address space is that you need to deallocate. In each iteration, you will need to use the current item in the list by obtaining list.get(i).

Check to see if the current node is owned by a different pid than the method parameter. If so, continue to the next iteration. Once that "if it's the wrong pid then continue" case is over with, then for the rest of the loop body, you can assume it IS the correct pid, and you've found the one you should deallocate.

Check the special case where you're deallocating the first node in the list, where i is 0. In this situation, the current node is the first one in the list, and there are 3 sub cases:
  1) this is the first and last node, and the memory map consists of just 1 node. If so, then turn the entire memory map from one address space into one giant free section by changing the owner pid.
  2) if a free section comes right after this node. In this case, you need to remove the node for this address space and grow the free section -- increase its size and make it start earlier so it "absorbs" the units of this address space we're deallocating.
  3) if an address space comes right after this node. In this case, turn the pid into a free section much like case 1 above

Next, check the special case where you're deallocating the last node in the list, where i is the length of the list minus one. In this situation, the current node is the last one in the list. and there are 3 sub cases. This is a lot like a mirror image of

the earlier special case where you're deallocating the first node in the list, if you were to visualize it. The three subcases are:
 1) this is also the first node in the memory map. This case is ALREADY DONE above so it won't logically occur here. It was the earlier "case 1" above that already catches this situation.
 2) There is a free section to the left of the current node
 3) The node to the left of the current node is an address space

Once you've dealt with those special cases, you know the only cases left are those from memory manager slide 14 (link to the slide). In this situation the left and right neighbor nodes both exist. In this situation, you will need handles to the current node list.get(i) as well its left neighbor list.get(i-1) and its right neighbor list.get(i+1). Do the four cases as depicted in the diagram. I used local variables named x, a, b, to match the picture, for clarity. The way you tell these four cases apart is by examining the pid of a and b (the neighbor nodes). On slide 14, the four cases were named (a) (b) (c) and (d). I won't lead you through all of them, but we can look at one of the four cases for example. Case (a) is when both neighbors are address spaces, not free sections. You can identify that case by checking if the left neighbor pid and the right neighbor pid are both nonzero, meaning not free sections.

If your loop ended, then the requested pid had no address space.

Regards,
Adam Whitley, Ph.D.

Lecturer of Computer Science
University of North Carolina at Asheville
Rhoades/Robinson Hall, Room 222
One University Heights, CPO #2320
Asheville, NC 28804
(828)-250-3920