CS460 NOTES on Memory Management

Management of Real Memory

Assume our Wanix

Presently, we only have 8 procs, P1 to P8, each has a Umode image at 7MB + pid*1MB, i.e. P1 at 8MB, P2 at 9 MB, etc.
This is just for simplicity.
In a real system, when we kfork(filename) new process to run filename, we should allocate a Umode image memory by the filename size:

```
       |Code|Data|Heap        |stack     |
```

When Wanix kernel starts, we assume the first 8MB are already in use.
Free memory begins from 8MB to 128 MB

When we create a process with a Umode image, we must load its Umode image file into memory. Depending on the image file size, we can dynamically allocate a piece memory from the FREE area, and load the image file into allocated area. When the process dies, we free its Umode memory area and return it to the FREE area. Similarly, when a proc EXEC to a new file, if the new file needs a memory area size > existing Umode area size, we may allocate a NEW free area to load the new image and releases the old image area, etc. As the procs come and go, the memory area would look like the following:

```
                 8M                                                  128M
   ---------------|-----------------------------------------------------|---
   |      kernel  |   |  U    |   | U  |          |  U     |         |
   ------------------------------------------------------------------------
                  a1          a2       a3                  a4
                  s1          s2       s3                  s4
```

where each white area represents a HOLE and each U area represents an area in USE. As can be seen, they are of different sizes. To keep track of where the HOLEs are (for allocation), we may use a freeMemory pointer (in Wanix kernel) to point at a list of freeBlock data structures, as shown below.

```
       freeMem
                  -----------------     -----    -----         -----
          .--------> |nextPtr    .----|--->| .-|--->| .-|-------->|-. |
                     |beginAddress a1 |    |a2 |    |a3 |         |a4 |
                     |size         s1 |    |s2 |    |s3 |         |s4 |
                     ------------------    -----    -----         -----
```

The freeMem list represents the CURRENT FREE memory areas still available.

When a proc needs a free memory area (to fork a new proc or to exec a new file), it calls

```
segment *malloc(size);
```

which returns a piece of free memory of the requested size pointed by the return value. It returns 0 if no free memory can be allocated. When a proc ends (or release an old Umode image), it calls

```
mfree(beginAddress, size);
```

which releases the area back to the freeMem list. So, how to implement the malloc()/mfree() functions?

```
2.   u32 *malloc(request_size)
     {
         scan freeMem list for a HOLE whose size is >= request_ize):
         if find such a HOLE at, say [ai, si]:
            if (si == request_size){ // exactly fit
               delete the hole from freeMem list
               return ai;
            }
            // si > request_size
            allocate request_size from HIGH end of the hole by
            changing [ai,si] to [ai, si-request_size], and
            return   ai + si-request_size;
         }
         else  // no hole big enough
            return 0;
     }
```

Since malloc() tries to allocate from the FIRST hole big enough for the request_size, it is called the First_Fit Algorithm, which is the most often implemented algorithm in practice. Alternatively, there are also Best-Fit and Worst-Fit algorithms, which are never used in practice.

Corresponding to malloc(), the algorithm mfree(address, size) works as follows: When releasing a piece of memory, 3 possible cases may arise, as shown in the diagrams

```
case 1 :   ----------------------------------
           | xxxx | yyy |to be freed | zzzz |    no adjacent hole
           ----------------------------------

case 2 :   ----------------------------------
           | xxxx |     |to be freed |yyyyyy|    has a hole on the left
           ----------------------------------
           ----------------------------------          OR
           | xxxx |yyyy |to be freed |      |    has a hole on the right
           ----------------------------------

case 3 :   ----------------------------------
           | xxxx |     |to be freed |      |    has holes on BOTH sides
           ----------------------------------
```

In the freeMem list, there should be NO adjacent holes, each hole is of maximal size.

For case 1, we create a new hole to represent the released area.

For case 2, we absorb the released area into its adjacent hole, making a bigger hole.
For case 3, we consolidate 3 holes into ONE hole, which means actually delete a hole from the freeMem list.

3. malloc()/mfree() in Umode area:

The C lib provides malloc() and mfree() for dynamic memory alloc/free in User space, which is the the HEAP area of a user mode image. They work exactly the same as malloc()/mfree() outlined above.