ICS143B Project2 Main Memory Manager - Final Document

Anbang Xu(35086995)

1. Introduction

Design/implement a main memory manager for variable size partition, including allocating memory and deallocating memory. And using simulation to compare different allocation strategies such as first-fit and best-fit.

2. Data Structure

a. PackableMemory

+-----+
| size |
+-----+
| memory |
+-----+

The PackableMemory is a structure having an integer - '"size" and a byte array - "memory" for storage. It's used to pack/unpack an integer to/from a byte array. We can treat it as an integer array.

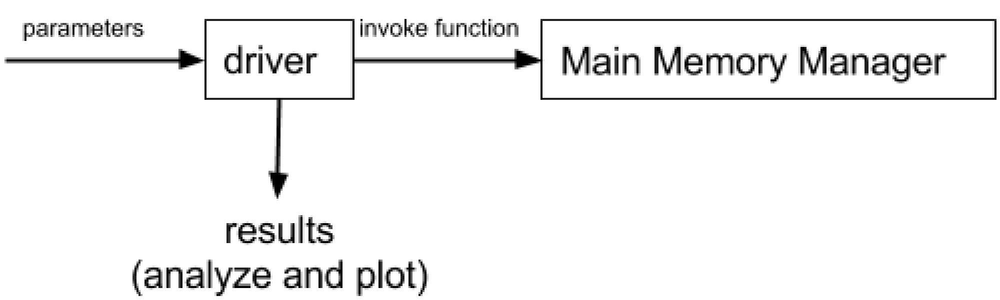
b. Main Memory Manager(MMManager)

+-----+
| size |
+-----+
| memory |
+-----+
| firstHole |
+-----+
| lastHole |
+-----+

The MMManager is a structure having size, memory, firstHole and lastHole. "size" indicates the total size of the initial word-addressable memory block. "memory" is a PackableMemory(an integer array) of size "size". "firstHole" is the index of first hole and "lastHole" is the index of last hole. If there is no hole, those indices are -1.

3. System Architecture

a. Overall Organization



(1). The user invokes driver, a. **generates streams of requests and releases** using parameters, b. repeatedly invokes request/release function, c. **gather statistics** in files for each request

(2). The driver repeats for **different parameters** and **different allocation strategies**

(3). Analyze, plot and describe results
What to vary?

- the total initial size
- distribution of request size
- steps of simulation
- allocation strategy selection

What to measure?

- average memory utilization - Occupied Block Size / Total Memory Block Size
- average search time - # holes examined / total # holes

b. Important function in MMManager:
(1). Initiate Memory Block
input: void, output: void
- Describe:

A. create a PackableMemory of input size
B. initiate the left tag and right tag
C. set previous and next reference
D. make firstHole/lastHole reference to start index of this memory block - 0

- What data structures may be changed? MMManager.size, MMManager.memory, MMManager.firstHole, MMManager.lastHole

(2). Allocate Memory
input: request size, output: start index of allocated block
- Describe:

A. choose the allocation strategy
B. find the suitable empty hole using chosen strategy
C. compute the start index of allocated block and initiate it
D. update the old hole, including size, tag and reference
E. return the start index of new allocated block

- What data structures may be changed? MMManager.size, MMManager.memory, MMManager.firstHole, MMManager.lastHole

(3). Release Memory
input: release size, output: void
- Describe:

A. check left, right tag if neighbor is occupied
B. based on checking result, update
    a. both are occupied
        - update tags
        - add the release hole to the end of empty holes
    b. left is occupied but right is not - merge with right hole
        - update tags
        - move right hole reference to the current hole
    c. right is occupied but left is not - merge with left hole
        - update tags
    d. both are empty holes - merge with left and right hole
        - update tags
        - remove right hole

- What data structures may be changed? MMManager.size, MMManager.memory, MMManager.firstHole, MMManager.lastHole

c. Important function in Driver:
(1). Generate Next Request Size
input: mean, deviation, maximum, output: randomized request size
- Describe:
    A. apply Gaussian distribution
    B. discard values outside of valid memory sizes
- What data structures may be changed? MMManager

(2). Record Memory Utilization
Input: void, Output: void
- Describe:
    A. add up block sizes, divide by total memory size at each iteration
    B. compute average from each release for all iterations

(3). Record Average Search Time
Input: void, Output: void
- Describe:
    A. compute (# holes examined) / (total # holes)
    B. compute average from each request for all iterations

(4). Select release Block
input: allocatedBlocks, output: index of release block in allocatedBlocks
- Describe:
    A. get the size k of allocated blocks
    B. choose a random number p between 1 and k
    C. release block recorded at position p of allocatedBlocks
- What data structures may be changed? MMManager

4. Test Cases
(1). Initiate Memory Block and Request Memory
Input:
```
init(100);
int n = 10;
for (int i = 0; i < n; i++)
    request(10);
```
Result:
```
Init 100 in [0,400]
Request 10 in [352,400], and the block start index is: 356
Request 10 in [304,352], and the block start index is: 308
Request 10 in [256,304], and the block start index is: 260
Request 10 in [208,256], and the block start index is: 212
Request 10 in [160,208], and the block start index is: 164
Request 10 in [112,160], and the block start index is: 116
Request 10 in [64,112], and the block start index is: 68
```

Request 10 in [16,64], and the block start index is: 20
Request 10, but insufficient memory
Request 10, but insufficient memory

(2). Release Memory

A. both left and right are occupied:

Input:

init(100);
request(10);
request(10);
request(10);
request(10);
release(B);

Result:

Init 100 in [0,400]
Request 10 in [352,400], and the block start index is: 356
Request 10 in [304,352], and the block start index is: 308
Request 10 in [256,304], and the block start index is: 260
Request 10 in [208,256], and the block start index is: 212
Release 10 in [304,352]

B. left is occupied but right is not:

Input:

init(100);
request(10);
request(10);
request(10);
request(10);
release(A);
release(B);

Result:

Init 100 in [0,400]
Request 10 in [352,400], and the block start index is: 356
Request 10 in [304,352], and the block start index is: 308
Request 10 in [256,304], and the block start index is: 260
Request 10 in [208,256], and the block start index is: 212
Release 10 in [352,400]
Release 10 in [304,400]

C. right is occupied but left is not:

Input:

init(100);
request(10);
request(10);
request(10);
request(10);
release(C);
release(B);

Result:

Init 100 in [0,400]

Request 10 in [352,400], and the block start index is: 356
Request 10 in [304,352], and the block start index is: 308
Request 10 in [256,304], and the block start index is: 260
Request 10 in [208,256], and the block start index is: 212
Release 10 in [256,304]
Release 10 in [256,352]

D. both are empty holes

Input:

init(100);
request(10);
request(10);
request(10);
request(10);
release(A);
release(C);
release(B);

Result:

Init 100 in [0,400]
Request 10 in [352,400], and the block start index is: 356
Request 10 in [304,352], and the block start index is: 308
Request 10 in [256,304], and the block start index is: 260
Request 10 in [208,256], and the block start index is: 212
Release 10 in [352,400]
Release 10 in [256,304]
Release 10 in [256,400]

5. Pseudo Code

```
a. Driver.main(){
        for(i = 0; i < sim_step; i++){
                do{
                        get size n of next request;
                        mmmanager.request(n);
                } while(request successful);
        record memory utilization;
        select block p to be released;
        mmmanager.release(p);
}

b. Driver.generateNextRequestSize(int a, int d, int totalSize){
        int size = (int) getGaussian(a, d);
         while (size < 2 || size > totalSize) {
                 size = (int) getGaussian(a, d);
         }
         return size;
}

c. Driver.selectReleasedBlock(ArrayList<Integer> allocatedBlocks){
```

```
        int k = allocatedBlocks.size();
        int p = random.nextInt(k);
        mmmanager.release(allocatedBlocks.get(p));
}

d. MMManager.init(int size){
        // 1. create a memory block with a specific size
         totalByteSize = INTEGER_SIZE * totalSize;
         memoryBlock = new PackableMemory(totalByteSize);

         // 2. create and init the hole, then return the block start index
         createAndInitHole(0, false, totalSize
                         - DIFF_BETWEEN_HOLESIZE_AND_BLOCKSIZE, -1, -1);
}

e. MMManager.createAndInitHole(int holeStartIdx, boolean occupied,
                int blockSize, int prev, int next) {
         int tag = occupied ? blockSize : -blockSize;

         // 1. set the tag
         // left tag
         memoryBlock.pack(tag, holeStartIdx);
         // right tag
         int rightTagOffset = INTEGER_SIZE * (TAG_SIZE + blockSize);
         memoryBlock.pack(tag, holeStartIdx + rightTagOffset);

         // 2. set reference
         // prev
         memoryBlock.pack(prev, holeStartIdx + INTEGER_SIZE * TAG_SIZE);
         // next
         memoryBlock.pack(next, holeStartIdx + INTEGER_SIZE
                         * (TAG_SIZE + PREV_INDEX_SIZE));

         // 3. set firstHole/lastHole
         firstHole = 0;
         lastHole = 0;
}

f. MMManager.request(int size) {
         // 1. run algorithm to find suitable memory block
         int holeStartIdx = bestFit(size);
         if (holeStartIdx == -1) {
                 System.out.println("Request " + size + ", but insufficient memory");
                 return -1;
         }

         // keep track of prev/next hole for future update
         int prevHole = getPrevHole(holeStartIdx);
```

```
        int nextHole = getNextHole(holeStartIdx);

        // if has sufficient memory, create a memory with input size
        // 2. compute the startIdx of new hole
        int newHoleEndIdx = getHoleEndFromHoleStart(holeStartIdx);
        int newHoleStartIdx = newHoleEndIdx - INTEGER_SIZE
                        * (getHoleSizeFromBlockSize(size));

        // 3. init the new block - blockSize
        memoryBlock.pack(size, newHoleStartIdx);
        memoryBlock.pack(size, newHoleEndIdx - INTEGER_SIZE * TAG_SIZE);

        // 4. update the old hole
        int remainHoleSize = (newHoleStartIdx - holeStartIdx) / 4;
        int remainBlockSize = remainHoleSize
                        - DIFF_BETWEEN_HOLESIZE_AND_BLOCKSIZE;
        // 4.1. if hole become too small
        if (remainBlockSize < 2) {
                memoryBlock.pack(size + remainHoleSize, holeStartIdx);
                memoryBlock.pack(size + remainHoleSize, newHoleEndIdx
                                - INTEGER_SIZE * TAG_SIZE);
                newHoleStartIdx = holeStartIdx;

                // remove hole
                if (prevHole != -1)
                        setNextHole(prevHole, nextHole);
                if (nextHole != -1)
                        setPrevHole(nextHole, prevHole);
                if (holeStartIdx == firstHole) {
                        firstHole = nextHole;
                }
                if (holeStartIdx == lastHole) {
                        lastHole = prevHole;
                        setNextHole(lastHole, -1);
                }
        } else { // 4.2. otherwise
                memoryBlock.pack(-remainBlockSize, holeStartIdx);
                memoryBlock.pack(-remainBlockSize, newHoleStartIdx - INTEGER_SIZE
                                * TAG_SIZE);
        }

        // return start index of new block
        return newHoleStartIdx + INTEGER_SIZE * TAG_SIZE;
}

g. MMManager.release(int blockIdx) {
        // 1. compute left, right if occupied
        int curHoleStart = blockIdx - INTEGER_SIZE * TAG_SIZE;
```

```
int curHoleEnd = getHoleEndFromHoleStart(curHoleStart);
int blockSize = getBlockSize(curHoleStart);
int left;
if (curHoleStart - INTEGER_SIZE * TAG_SIZE < 0)
	left = 1;
else
	left = memoryBlock.unpack(curHoleStart - INTEGER_SIZE * TAG_SIZE);
int right;
if (curHoleEnd + INTEGER_SIZE * TAG_SIZE > totalByteSize)
	right = 1;
else
	right = memoryBlock.unpack(curHoleEnd);

// 2. check
if (left >= 0 && right >= 0) {
	// 2.1. both are occupied
	// (1). update tag
	memoryBlock.pack(-blockSize, curHoleStart);
	memoryBlock.pack(-blockSize, curHoleEnd - INTEGER_SIZE * TAG_SIZE);
	// (2). add it to the lastHole
	if (lastHole == -1) {
		setPrevHole(curHoleStart, -1);
		setNextHole(curHoleStart, -1);
		lastHole = curHoleStart;
		firstHole = lastHole;
	} else {
		setPrevHole(curHoleStart, lastHole);
		setNextHole(lastHole, curHoleStart);
		lastHole = curHoleStart;
	}
	setNextHole(lastHole, -1);

	System.out.println("Release " + blockSize + " in [" + curHoleStart
			+ "," + curHoleEnd + "]");
	return curHoleStart;
} else if (left >= 0 && right < 0) {
	// 2.2. left is occupid but right is not - merge with right hole
	// rightHoleStart == curHoleEnd
	// (1). update tag - size
	int newBlockSize = blockSize + Math.abs(right) + 2 * TAG_SIZE;
	memoryBlock.pack(-newBlockSize, curHoleStart);
	int rightHoleEnd = getHoleEndFromHoleStart(curHoleEnd);
	memoryBlock.pack(-newBlockSize, rightHoleEnd - INTEGER_SIZE
			* TAG_SIZE);
	// (2). move right hole reference to cur
	// update reference from cur perspective
	setPrevHole(curHoleStart, getPrevHole(curHoleEnd));
	setNextHole(curHoleStart, getNextHole(curHoleEnd));
```

```
            // update reference from prev/next perspective
            setNextHole(getPrevHole(curHoleEnd), curHoleStart);
            setPrevHole(getNextHole(curHoleEnd), curHoleStart);

            if (curHoleEnd == firstHole)
                firstHole = curHoleStart;
            if (curHoleEnd == lastHole)
                lastHole = curHoleStart;

            System.out.println("Release " + blockSize + " in [" + curHoleStart
                        + "," + rightHoleEnd + "]");
            return curHoleStart;
        } else if (left < 0 && right >= 0) {
            // 2.3. left is not occupid but right is - merge with left hole
            // leftHoleEnd == curHoleStart
            // (1). update tag - size
            int newBlockSize = blockSize + Math.abs(left) + 2 * TAG_SIZE;
            int leftHoleStart = getHoleStartFromHoleEnd(curHoleStart);
            memoryBlock.pack(-newBlockSize, leftHoleStart);
            memoryBlock.pack(-newBlockSize, curHoleEnd - INTEGER_SIZE
                        * TAG_SIZE);

            System.out.println("Release " + blockSize + " in [" + leftHoleStart
                        + "," + curHoleEnd + "]");
            return leftHoleStart;
        } else {
            // 2.4. both are not occupied - merge with left and right holes
            // (1). update tag - size
            int newBlockSize = blockSize + Math.abs(left) + Math.abs(right) + 4
                        * TAG_SIZE;
            int leftHoleStart = getHoleStartFromHoleEnd(curHoleStart);
            int rightHoleEnd = getHoleEndFromHoleStart(curHoleEnd);
            memoryBlock.pack(-newBlockSize, leftHoleStart);
            memoryBlock.pack(-newBlockSize, rightHoleEnd - INTEGER_SIZE
                        * TAG_SIZE);
            // (2). remove right hole
            removeHole(curHoleEnd);

            System.out.println("Release " + blockSize + " in [" + leftHoleStart
                        + "," + rightHoleEnd + "]");
            return leftHoleStart;
        }
}

h. MMManager.firstFit(int requestSize) {
        numHoleExamined = 0;
        int curHole = firstHole;
        while (curHole >= 0) {
```

```
            numHoleExamined++;
            if (requestSize < getBlockSize(curHole))
                return curHole;
            curHole = getNextHole(curHole);
        }
        return -1;
}
```

```
i. MMManager.bestFit(int requestSize) {
        numHoleExamined = 0;
        int curHole = firstHole;
        int minDiff = Integer.MAX_VALUE;
        int returnHole = -1;

        while (curHole >= 0) {
            int blockSize = getBlockSize(curHole);
            numHoleExamined++;
            if (blockSize >= requestSize && (blockSize - requestSize) < minDiff) {
                minDiff = blockSize - requestSize;
                returnHole = curHole;
            }
            curHole = getNextHole(curHole);
        }
        return returnHole >= 0 ? returnHole : -1;
  }
```

j. Help functions in MMManager:

```
public void removeHole(int curHole) {
        int prevHole = getPrevHole(curHole);
        int nextHole = getNextHole(curHole);
        if (prevHole != -1)
            setNextHole(prevHole, nextHole);
        if (nextHole != -1)
            setPrevHole(nextHole, prevHole);
        if (curHole == firstHole) {
            firstHole = nextHole;
        }
        if (curHole == lastHole) {
            lastHole = prevHole;
            setNextHole(lastHole, -1);
        }
  }

  public int getBlockSize(int startIdx) {
        return Math.abs(memoryBlock.unpack(startIdx));
  }

  public int getHoleStartFromHoleEnd(int endIdx) {
```

```
        int blockSize = Math.abs(memoryBlock.unpack(endIdx - INTEGER_SIZE
                        * TAG_SIZE));
        return endIdx - INTEGER_SIZE
                        * (DIFF_BETWEEN_HOLESIZE_AND_BLOCKSIZE + blockSize);
    }

    public int getHoleEndFromHoleStart(int startIdx) {
        int blockSize = getBlockSize(startIdx);
        return startIdx + INTEGER_SIZE
                        * (blockSize + DIFF_BETWEEN_HOLESIZE_AND_BLOCKSIZE);
    }

    public int getHoleSizeFromBlockSize(int blockSize) {
        return blockSize + DIFF_BETWEEN_HOLESIZE_AND_BLOCKSIZE;
    }

    public int getPrevHole(int curHole) {
        return memoryBlock.unpack(curHole + INTEGER_SIZE * TAG_SIZE);
    }

    public int getNextHole(int curHole) {
        return memoryBlock.unpack(curHole + INTEGER_SIZE
                        * (TAG_SIZE + PREV_INDEX_SIZE));
    }

    public void setPrevHole(int curHole, int prevHole) {
        if (curHole < 0)
            return;
        int curPreReferIdx = curHole + INTEGER_SIZE * TAG_SIZE;
        memoryBlock.pack(prevHole, curPreReferIdx);
    }

    public void setNextHole(int curHole, int nextHole) {
        if (curHole < 0)
            return;
        int curNextReferIdx = curHole + INTEGER_SIZE
                        * (TAG_SIZE + PREV_INDEX_SIZE);
        memoryBlock.pack(nextHole, curNextReferIdx);
    }

    public int getBlockStartIdx(int startIdx) {
        return startIdx + INTEGER_SIZE
                        * (TAG_SIZE + PREV_INDEX_SIZE + NEXT_INDEX_SIZE);
    }
```

k. Apply graph plot and more details:

```
main(String[] args) {
        mmm.init(maxMemorySize);
```

```
        /**
         * a: [100, 600] 6 steps d: [40, 200] 5 steps
         */
        int a;
        int d = 40; // 0.02 * maxMemorySize
        for (; d <= 200; d += 40) {
                XYLineChart.means.clear();
                XYLineChart.stats.clear();
                XYLineChart.deviation = d;
                ArrayList<Result> results1 = new ArrayList<Result>();
                ArrayList<Result> results2 = new ArrayList<Result>();
                for (a = 100; a <= 600; a += 100) {
                        XYLineChart.means.add(a);
                        results1.add(runSimulator(a, d, Strategy.FIRST_FIT));
                        results2.add(runSimulator(a, d, Strategy.BEST_FIT));
                }
                XYLineChart.stats.put(Strategy.FIRST_FIT, results1);
                XYLineChart.stats.put(Strategy.BEST_FIT, results2);
                XYLineChart.createMemoryUtilChartPanel();
                XYLineChart.createSearchRatioChartPanel();
        }
    }

l. runSimulator(int a, int d, Strategy strategy) {
        mmm.reset(maxMemorySize);
        double utilization = 0;
        double searchRatio = 0;
        int count = 0;
        ArrayList<Integer> allocatedBlocks = new ArrayList<Integer>();
        // run simulator with strategy1
        for (int i = 0; i < simSteps; i++) {
                System.out.println("Step " + i + "(" + strategy + "):");
                // get size n of next request
                int size = 0;
                Integer allocatedBlock;
                while (true) {
                        size = generateNextSize(a, d, maxMemorySize);
                        int curTotalHoles = mmm.getTotalHoles();
                        allocatedBlock = mmm.request(size, strategy);
                        if (mmm.getTotalHoles() == 0)
                                searchRatio = (searchRatio * count + (double) 1)
                                                / (count + 1);
                        else
                                searchRatio = (searchRatio * count + (double) mmm.numHoleExamined
                                                / curTotalHoles)
                                                / (count + 1);
                        count++;
```

```
            if (allocatedBlock == -1)
                break;
            allocatedBlocks.add(allocatedBlock);
        }

        // record memory utilization
        int occupiedSize = mmm.getSizeOfOccupiedBlock(allocatedBlocks);
        utilization = (utilization * i + (double) occupiedSize
                / maxMemorySize)
                / (i + 1);

        // select block p to be released from 1 to k
        int k = allocatedBlocks.size();
        if (k == 0) {
            continue;
        }
        int p = random.nextInt(k);
        mmm.release(allocatedBlocks.get(p));
        mmm.printEmptyHole();

        allocatedBlocks.remove(p);
        mmm.printOccupiedBlock(allocatedBlocks);
        // System.out.println("Utilization: " + utilization);
        // System.out.println("SearchRatio: " + searchRatio);
        // System.out.println();
    }
    return new Result(utilization, searchRatio);
}
```