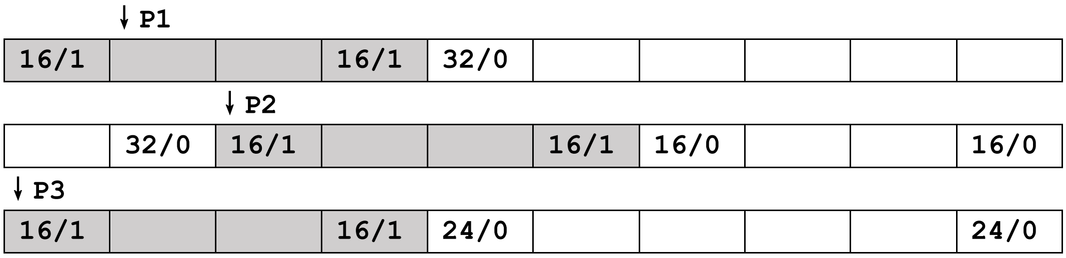
Problem 1: Memory Allocation (14 points)

There are some **unallocated memory** and **heap**, as shown below. Heap is organized as a sequence of **contiguous** allocated and free blocks. **Allocated** blocks are shaded, and **free** blocks are blank (each block represents 1 word = 4 bytes). **Headers** and **footers** are labeled with the number of bytes and allocated bit. The allocator maintains **double-word** alignment. You are given the execution sequence of memory allocation operations (i.e., **`malloc()`** or **`free()`**) from 1 to 6.




```
1. P4 = malloc(4)
2. free(P1)
3. P5 = malloc(9)
4. P6 = malloc(5)
5. free(P3)
6. P7 = malloc(2)
```

Please answer the questions below. Assume that **immediate coalescing** strategy and **splitting free blocks** are employed.

1. Assume **first-fit** algorithm is used to find free blocks. Please draw the **final** status of memory and mark with block size in headers and footers after the operation sequence is executed (5').
2. Assume **best-fit** algorithm is used to find free blocks. Please draw the **final** status of memory and mark with block size in headers and footers after the operation sequence is executed (5').
3. Please calculate the total bytes of **internal fragments** (Note: **DO NOT** consider **`P1`**, **`P2`** and **`P3`** for internal fragments) (4').

Problem 2: Lock (13 points)

```
typedef struct __node_t {
  __node_t* next;
  tid_t tid;
} node_t;

typedef struct __lock_t {
      node_t* head;
} lock_t;

void lock_init(lock_t* lock) {
      lock->head = NULL;
}
/* each caller should alloc and hold a node by itself */
void lock(lock_t* lock, node_t* node) {
      if (lock == NULL || node == NULL) {
          /* output error message... */
          exit(-1);
      }
      node->next = NULL;
      node->tid = gettid();
      node_t* old = test_and_set(lock->head, node);
      if (old == NULL) return;
      old->next = node;
      park();
}

void unlock(lock_t* lock, node_t* node) {
      if (lock == NULL || node == NULL) {
          /* output error message... */
          exit(-1);
      }
      if (node->next == NULL) {
          if (compare_and_swap(_[1]_, _[2]_, _[3]_)) {
              return;
          }
          while(node->next == NULL)
              continue;
      }
      unpark(node->next->tid);
}
```

Above code is an implementation of a kind of lock called MCS. Each `lock_t` structure represents a lock. Each thread who wants a lock will maintain a **`node_t`** structure. Nodes are connected through their arriving order. The head of lock will always point to the newest node. **`Test_and_set`** and **`compare_and_swap`** operations are **atomic**. **DO NOT** consider the CPU may reorder the instructions.

1. **`compare_and_swap(a, b, c)`** will set a's value to c and **return 1** if and only if a's old value equals b, otherwise **return 0 and do nothing**. Fill the blanks in the code. **Hint:** consider the situation that another thread acquires the same lock between line 32 and 33. Your code is used to keep the lock running correctly under this situation. (3')

2. Why do we need line **36** and **37**? (2')

3. There are two problems in this implementation. Figure them out and try to fix them **if possible**. **Hint:** Consider about **correctness** and **security**. (4')

4. Analyze the lock about its **fairness** and **performance**. (4')