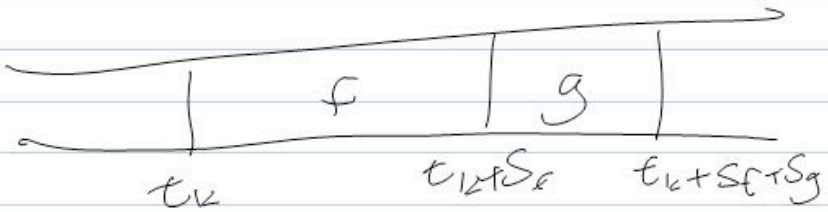
```
//----------Problem 1----------------------------------------------------
typedef void (*function_t)(void);
void dispatch(function_t* funcs, void* args, int ct) {
    int i = 0;
    while ( i<ct) {
         if (funcs[i](args[i]) == NULL)
            break;
         else
              i++;
    }
}

//----------Problem 2----------------------------------------------------
// data structures
void** stack;           // An array representing our stack
                        //  implementation.  Assume it's been
                        //  initialized.
int sp;                 // Assume this is set to the index of the array
                        //  that is the top of the stack
                        // Starts indexing at sp = 1 so if sp == 0 the
                        //  stack is empty.
int size;               // Maximum size of the stack (doesn't change)
pthread_mutex lock;
condition not_full, not_empty;

// returns the value of the element popped off the stack
void *pop() {
  void* element;

  acquire(lock);
  if ( sp > 0 ) {
    *element = stack[sp--];
  } else  {
    element = NULL;
  }
  release(lock);
  return element;
}

void push(void *element) {
  acquire(lock);
  if (sp == size || !element) return;  // stack is full || element's
got nada
  stack[++sp] = *element;
  release(lock);
}


//----------Problem 3----------------------------------------------------
/*  There are two acceptable solutions to problem 3 and they are both
below.  Commonly, people dropped the use of the locks altogether which
isn't correct.  Also, they'd only get the lock after the wait rather
than before it (and they must pass the locked lock as a parameter to
the wait). Another common problem was code that always waited without
testing a condition first. Some people confused Mesa and Hoare
semantics, some used ambiguous labels for condition variables (such as
using one cond variable signifying both not_empty and not_full: it
works if you write it very carefully but only one person did this
```

```
successfully.  Others used full and empty instead of not_full or
not_empty and that's not a great idea either).  There was also a lot of
confusion about using void* types.
*/

// data structures
void** stack;
int sp;
int size;
pthread_mutex lock;
condition not_full, not_empty;

// returns the value of the element popped off the stack
void *pop() {
  void* element;
  acquire(lock);
  if ( sp == 0 )       // sp == 0 means the stack is empty
    wait (not_empty, lock);

  *element = stack[sp--];
  signal(not_full);

  release(lock);
  return element;
}

void push(void *element) {
  if (!element) return;

  acquire(lock);
  if (sp == size)
    wait(not_full, lock);

  stack[++sp] = *element;
  signal(not_empty);
  release(lock);
}


/*
 *  PROBLEM 3 WITH IMPLICIT MUTEXES
 */


Monitor stack{
  // data structures
  void** stack;
  int sp;
  pthread_mutex lock;
  condition not_full, not_empty;

  // returns the value of the element popped off the stack
  void *pop() {
    // ENTER MONITOR
    void* element;
    if ( sp == 0 )       // sp == 0 means the stack is empty
      wait (not_empty);
```

```
    *element = stack[sp--];
    signal(not_full);

    // EXIT MONITOR
    return element;
  }

  void push(void *element) {

    if (!element) return;

    // ENTER MONITOR
    if (sp == size)
      wait(not_full);

    stack[++sp] = *element;
    signal(not_empty);
    // EXIT MONITOR
  }
}

//----------Problem 4-------------------------------------------------
```



Proof:

- For any scheduling algorithm that is not "shortest job first", there will be a job, S_f, that is longer than S_g.
- The total contribution to average response time of f and g is
  $2t_k + 2s_f + s_g$
- If f and g are interchanged (as per SJF), the total contribution to average response time of f and g is
  $2t_k + 2s_g + s_f$
- Since $s_g < s_f$, the latter situation (SJF) has shorter average response time