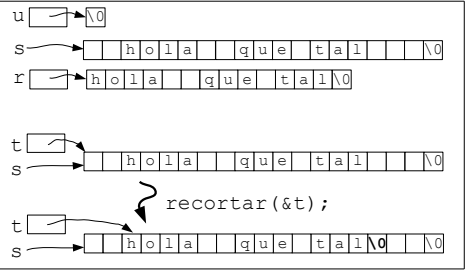
Pregunta 1

Parte a.- Programe las siguientes funciones:

```
char *recorte(char *str);
void recortar(char **pstr);
```

Ambas funciones eliminan los espacios en blanco superfluos al principio y al final de un string. Para ello la función *recorte* crea un nuevo string sin alterar el string que recibe como parámetro, mientras que la función *recortar* reutiliza la memoria ocupada por el string recibido como parámetro. Ejemplos de uso:

```
char *u= recorte("   "); // u es ""
char s[]= "  hola  que tal   ";
char *r= recorte(s); // s no cambia
// string r es "hola  que tal"
free(r); // libera el string r
char *t= s;   // t==s
recortar(&t);
// string t es "hola  que tal"
// string s ¡cambia!
```



Restricciones: No use el operador de subindicación de arreglos [ ] ni su equivalente **(p+i)*, use aritmética de punteros.

Ayuda para recorte: Pida memoria para el nuevo string con *malloc*. No es posible usar *recortar* en la función *recorte* porque esto haría imposible usar *free* para liberar más tarde la memoria pedida con *malloc*.

Ayuda para recortar: Avance **pstr* hasta encontrar el primer caracter que no sea un espacio en blanco. Busque donde colocar una nueva terminación para el string.

Parte b.- Un *left/right mutex* es un tipo de mutex que se puede otorgar completo o por mitades. La siguiente implementación es incorrecta e ineficiente, pero funciona el 99,9% de las veces:

```
enum { LEFT= 0, RIGHT= 1 };
int busy[2]= { 0, 0 }; // Ambas mitades libres
```

```
int halfLock() {
  while (busy[LEFT] && busy[RIGHT])
    ; // ¡busy waiting!
  int side= busy[LEFT] ? RIGHT
                       : LEFT;
  busy[side]= 1;
  return side;
}
void halfUnlock(int side) {
  busy[side]= 0;
}
```

```
void fullLock() {
  while ( busy[LEFT] ||
          busy[RIGHT])
    ; // ¡busy waiting!
  busy[LEFT]= busy[RIGHT]= 1;
}
void fullUnlock() {
  busy[LEFT]= busy[RIGHT]= 0;
}
```

Solo un thread puede obtener el mutex completo llamando a *fullLock*. Posteriormente, ese thread devuelve el mutex con *fullUnlock*. Hasta 2 threads pueden obtener cada uno una mitad distinta del mutex por medio de *halfLock*, que entrega LEFT si se otorgó la mitad izquierda o RIGHT si fue la mitad derecha. Más tarde cada thread devuelve su respectiva mitad con *halfUnlock*, especificando cuál fue la mitad que se le había otorgado.

Reprograme el código de más arriba de manera 100% correcta y eficiente. No importa que su solución sufra de hambruna.

Pregunta 2

Parte i.- La siguiente función busca un factor de *x*:

```
typedef unsigned long long ulonglong;
typedef unsigned int uint;
long long buscarFactor(ulonglong x, uint i, uint j) {
  for (int k= i; k<=j; k++) {
    if (x % k == 0)
      return k;
  }
  return 0;
}
```

Modifique esta función de manera que si el usuario ingresa *control-C* (señal *SIGINT*) mientras se ejecuta entonces se retorna -1. Use variables globales.

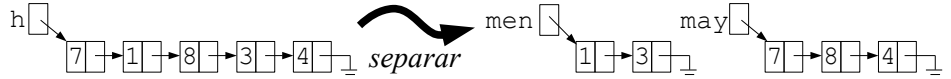
Parte ii.- Programe la función *separar* definida como:

```
typedef struct nodo {
  int x;
  struct nodo *prox;
} Nodo;
void separar(Nodo *lista, int z, Nodo **pmen, Nodo **pmay);
```

Esta función recibe en *lista* una lista simplemente enlazada desordenada en donde cada nodo almacena un entero. Ud. debe desarmar *lista* de tal forma que en **pmen* queden los nodos que almacenan enteros menores que *z* y en **pmay* queden los nodos que almacenan enteros mayores o iguales que *z* . En el siguiente ejemplo de uso, la lista *h* ha sido creada con 5 nodos que almacenan 7, 1, 8, 3 y 4. Luego se invoca *separar* como se indica en este código:

```
Nodo *h= …;
Nodo *men, *may;
separar(h, 4, &men, &may);
```

La siguiente figura muestra los cambios que produce la invocación de *separar* en la lista.



Restricciones: No puede usar *malloc*. Debe reutilizar los nodos que recibe en *lista*. Los nodos en **pmen* y **pmay* deben seguir el mismo orden en que aparecían en *lista*. Por claridad Ud. debe usar recursividad.