CC4302 Sistemas Operativos – Control 1 – Semestre Primavera 2025 – Profs.: Mateu, Arenas

Pregunta 1

La función *buscarValor* del cuadro de la derecha entrega la dirección del nodo que contiene el valor *val* en el árbol binario *a*. Como los valores no están ordenados, si no está en el subárbol izquierdo, también hay que buscar en el subárbol derecho. **Programe la función *buscarValorPar***, que entrega el mismo resultado pero realizando la búsqueda **en paralelo** con *p* threads.

Puede invocar *buscarValor* desde *buscarValorPar*. Si un thread encuentra el valor, espere pacientemente a que los demás threads completen la búsqueda.

```
typedef struct nodo {
  int val;
  struct nodo *izq, *der;
} Nodo;
Nodo *buscarValor(
          Nodo *a, int val) {
  if (a==NULL)
    return NULL;
  if (a→val==val)
    return a;
  Nodo *resIzq=
    buscarValor(a→izq, val);
  if (resIzq≠NULL)
    return resIzq;
  return
    buscarValor(a→der, val);
}
Nodo *buscarValorPar(Nodo *a,
             int val, int p);
```

Pregunta 2

Un puente puede soportar PESOMAX como peso máximo. Los vehículos deben solicitar permiso para cruzar el puente invocando la función e*ntrar(p)*, en donde *p* es el peso del vehículo. Cuando esta función retorna, el vehículo puede entrar al puente. El vehículo notificará su salida llamando a la función *salir(p)*. El peso *p* nunca superará a PESOMAX. Los permisos para entrar al puente se otorgan por orden de llegada, lo que significa que se autoriza a un vehículo a entrar al puente si y solo si (i) su peso más la de los vehículos que ya están cruzando el puente no excede PESOMAX, y (ii) no hay un vehículo en espera que haya solicitado entrar al puente antes que él. Cuando un vehículo sale se permite que entren al puente tantos vehículos como sea posible sin violar (i) y (ii). **Programe** las funciones iniPuente, entrar y salir con los encabezados del cuadro de abajo.

Restricción: Debe resolver este problema de sincronización usando un solo mutex y **una sola condición**. Por lo tanto necesitará que los vehículos pidan número.

```
void iniPuente(void);
void entrar(double p);
void salir(double p);
```

Pregunta 3

Parte a.- El cuadro de más abajo implementa ***incorrectamente*** las funciones *pedir* y *devolver* palitos de la cena de filósofos. Usa el patrón *request* para otorgar los palitos por orden de llegada.

Recuerdo: En este problema 5 filósofos numerados de 0 a 4 cenan juntos en una mesa redonda en donde hay 5 palitos, numerados de 0 a 4. Cada filósofo alterna entre comer y pensar. Para poder comer el filósofo *id* necesita los palitos *id* y *(id+1)%5*. No necesita ningún palito para pensar. Los filósofos deben comer en paralelo cuando sea posible pero no deben comer con el mismo palito simultáneamente, y tampoco pueden sufrir hambruna.

El siguiente código funciona correctamente el 99.9% de las veces. Confeccione un diagrama de threads que muestre que 2 filósofos pueden llegar a comer con el mismo palito. Necesitará 3 filósofos en su diagrama.

```
typedef struct {
  int rdy, id;
  pthread_cond_t w;
} Req;
int ocup[5]= {0,0,0,0,0};
Queue *q; // = makeQueue();
pthread_mutex_t m=
          PTHREAD_MUTEX_INITIALIZER;
void pedir(int id) {
  lock(&m);
  if ( ocup[id] ||
       ocup[(id+1)%5] ) {
    Req req={0, id,
          PTHREAD_COND_INITIALIZER};
    put(q, &req);
    while (!req.rdy)
      wait(&req.w, &m);
  }
  ocup[id]= 1;
  ocup[(id+1)%5]= 1;
  unlock(&m);
}
```

```
void devolver(int id) {
  lock(&m);
  ocup[id]= 0;
  ocup[(id+1)%5]= 0;
  // Ahora 2 filósofos podrían comer en ||
  while (!emptyQueue(q)) {
    Req *preq= peek(q);
      // No extrae preq de la cola
    if (ocup[preq→id] ||
        ocup[(preq→id+1)%5]) {
      break;
    }
    get(q);// Ahora sí se extrae preq
    preq→rdy= 1;
    signal(&preq→w);
  }
  unlock(&m);
}
void filosofo(int id) {
  for (;;) {
    pedir(id);
    comer(id, (id+1)%5);
    devolver(id);
    pensar();
} }
```

Parte b.- Haga cambios menores en esta implementación que corrijan el defecto. Puede responder esta parte sin responder *a*.