Sistemas Operativos

Universidad Complutense de Madrid

2020-2021

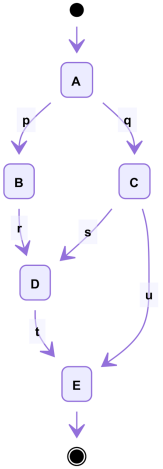
Comunicación y Sincronización

Soluciones de problemas

Juan Carlos Sáez

Examen Laboratorio Sept. 2020 - Entregable 1

- Crear programa multihilo
  - 5 Hilos (A, B, C, D y E) que se lancen simultáneamente
- Imponer sincronización necesaria para que cada hilo imprima en pantalla el siguiente mensaje según orden de preferencia impuesto por grafo
  - *"soy el hilo <identificador_hilo>"*
- Recursos para implementación:
  - Mutex
  - Variables condición
  - Otras variables compartidas



Examen Laboratorio Sept. 2020 - Entregable 1

```
/* Variables compartidas */
bool tA=false;
bool tB=false;
bool tC=false;
bool tD=false;
mutex_t m;
condvar cB; /* tA */
condvar cC; /* tA */
condvar cD; /* tB && tC */
condvar cE; /* tC && tD */

void A(void){
  printf("Soy el hilo A\n");
  lock(m);
  tA=true;
  cond_signal(cB);
  cond_signal(cC);
  unlock(m);
}
```

```
void B(void){
  lock(m);
  while (!tA)
    cond_wait(cB,m);
  unlock(m);
  printf("Soy el hilo B\n");
  lock(m);
  tB=true;
  cond_signal(cD);
  unlock(m);
}

void C(void){
  lock(m);
  while (!tA)
    cond_wait(cC,m);
  unlock(m);
  printf("Soy el hilo C\n");
  lock(m);
  tC=true;
  cond_signal(cD);
  cond_signal(cE);
  unlock(m);
}
```

```
void D(void){
  lock(m);
  while (!(tB && tC))
    cond_wait(cD,m);
  unlock(m);
  printf("Soy el hilo D\n");
  lock(m);
  tD=true;
  cond_signal(cE);
  unlock(m);
}

void E(void){
  lock(m);
  while (!(tC && tD))
    cond_wait(cE,m);
  unlock(m);
  printf("Soy el hilo E\n");
}
```

Problema 2

Dividir problema en dos subproblemas:

1. Problema productor-consumidor
   - 2 Productores: Hilo1 e Hilo2
   - 1 Consumidor: Hilo3
2. Los hilos 1 y 2 han de sincronizarse para insertar los números generados en orden en el buffer compartido entre hilos
   - Inserción de forma alternada: Hilo2 $\rightarrow$ 1, Hilo1 $\rightarrow$ 2, Hilo2 $\rightarrow$ 3, ...
   - El hilo 3 se limita a extraer los números del buffer e imprimirlos por pantalla en el orden de extracción

Problema 2

- Para resolver los subproblemas por separado supondremos que existen las siguientes funciones *thread-safe*, que luego implementaremos:
  - `produce(n)`: Inserta *n* en el buffer compartido de forma segura
  - `n=consume()`: Extrae el primer número del buffer y lo devuelve
  - Ambas operaciones tienen semántica bloqueante productor-consumidor

Código del Hilo 3 (común para ambos apartados del problema)

```
void thread3() {
    int i=0,n=0;
    for (i=0;i<2000;i++) {
        n=consume();
        printf("%d\n",n);
    }
}
```

Problema 2 - Sol. semáforos (I)

Productor-consumidor

```
int buf[N]; // shared ring buffer (max capacity: N items)
int ridx=0; // read position indicator
int widx=0; // write position indicator
semaphore producers; // To impose Mut. Exclusion when accessing widx (producers.c=1 initially)
semaphore gaps; // initial value=N
semaphore items; // initial value=0
```

```
void produce(int n) {
  wait(gaps);
  wait(producers);
  buf[widx]=n;
  widx=(widx+1)%N;
  signal(producers);
  signal(items);
}
```

```
int consume() {
  int n;
  wait(items);
  n=buf[ridx];
  ridx=(ridx+1)%N;
  signal(gaps);
  return n;
}
```

Problema 2 - Sol. semáforos (II)

Sincronización hilo1-hilo2

```
semáforo turnoImpar; // Inicializado a 1 (Empieza a insertar hilo2)
semáforo turnoPar; //Inicializado a 0
```

```
void thread1() {
  int i=0;
  int n=2;
  for (i=0;i<1000;i++,n+=2) {
    wait(turnoPar);
    produce(n);
    signal(turnoImpar);
  }
}
```

```
void hilo2() {
  int i=0;
  int n=1;
  for (i=0;i<1000;i++,n+=2) {
    wait(turnoImpar);
    produce(n);
    signal(turnoPar);
  }
}
```

Problema 2 - Sol. variables condición (I)

Productor-consumidor

```
int buf[N]; // shared ring buffer (max capacity: N items)
int ridx=0; // read position indicator
int widx=0; // write position indicator
int nr_items=0; // # of items in the shared buffer (0 at the beginning)
mutex m; // Mutual exclusion when accessing shared resources
condvar prod,cons; /* To block the producer
                      and the consumer respectively */
```

```
void produce(int n) {
  lock(m);
  while (nr_items==N)
    cond_wait(prod,m);
  buf[widx]=n;
  widx=(widx+1)%N;
  cond_signal(cons);
  unlock(m);
}
```

```
int consume() {
  int n;
  lock(m);
  while (nr_items==0)
    cond_wait(cons,m);
  n=buf[ridx];
  ridx=(ridx+1)%N;
  cond_signal(prod);
  unlock(m);
  return n;
}
```

Problema 2 - Sol. variables condición (II)

Sincronización hilo1-hilo2

```
bool turnoPar=false; // Indica el turno de inserción (inicialmente impar)
mutex mutexTurno; // Ex. mutua acceso a variable turnoPar.
condvar turno; // Cola donde se bloquea un hilo hasta que sea su turno
```

```
void thread1() {
  int i=0;
  int n=2;
  for (i=0;i<1000;i++,n+=2) {
    lock(mutexTurno);
//Bloqueo mientras no sea mi turno
    while (!turnoPar)
        cond_wait(turno,mutexTurno);
    //Insertar en el buffer
    produce(n);
    //Ceder turno al otro hilo
    turnoPar=false;
    cond_signal(turno);
    unlock(mutexTurno);
  }
}
```

```
void thread2() {
  int i=0;
  int n=1;
  for (i=0;i<1000;i++,n+=2) {
    lock(mutexTurno);
//Bloqueo mientras no sea mi turno
    while (turnoPar)
        cond_wait(turno,mutexTurno);
    //Insertar en el buffer
    produce(n);
    //Ceder turno al otro hilo
    turnoPar=true;
    cond_signal(turno);
    unlock(mutexTurno);
  }
}
```

Problema 3 (I)

- Implementación de un semáforo mediante un monitor
  - Usando un mutex y una variable condición

Especificación del monitor

```
typedef struct {
   mutex m;    /* mutex del monitor */
   condvar c;  /* Para hilos bloqueados en el semáforo */
   int s;      /* Contador interno del semáforo */
   int count;  /* contador >=0 para controlar bloqueos en wait() */
}sem_t;

//Inicialización/Destrucción del semáforo
void init_sem (sem_t* sem, unsigned int val);
void destroy_sem (sem_t* sem);
//Procedimientos del monitor
void wait(sem_t* sem);
void signal(sem_t* sem);
```

Problema 3 (II)

```
void wait(sem_t* sem) {
// Lock al entrar al monitor
  lock(&sem->m);
  sem->s--;
  /* Bloqueo si cont <=0 */
  while (sem->count<=0)
    cond_wait(&sem->c,&sem->m);
  sem->count--;
// unlock al salir del monitor
  unlock(&sem->m);
}

void init_sem (sem_t* sem,
               unsigned int val) {
  condvar_init(&sem->c);
  mutex_init(&sem->m);
  sem->count=sem->s=val;
}
```

```
void signal(sem_t* sem) {
// Lock al entrar al monitor
  lock(&sem->m);
  sem->s++;
  sem->count++;
  /* Bloqueo si cont <=0 */
  cond_signal(&sem->c);
// unlock al salir del monitor
  unlock(&sem->m);
}

void destroy_sem (sem_t* sem){
  condvar_destroy(&sem->c);
  mutex_destroy(&sem->m);
}
```

Problema 4.a - Monitor (Mutex y Var. cond.)

Especificación del monitor

```
int nr_readers=0; // # lectores que están dentro de su SC o que quieren entrar
int nr_writers=0; // # de escritores que están dentro de su SC
mutex mtx;
condvar wrcond;
condvar rdcond;

/* Procedimientos de monitor */
void ReaderEnter();
void ReaderExit();
void WriterEnter();
void WriterExit();
```

```
void Reader() {
    while(true) {
        ReaderEnter();
        <Reader critical section>
        ReaderExit();
    }
}
```

```
void Writer() {
    while(true) {
        WriterEnter();
        <Writer critical section>
        WriterExit();
    }
}
```

Problema 4.a - Monitor (Mutex y Var. cond.)

```
void ReaderEnter() {
    lock(mtx);
    nr_readers++;
    while (nr_writers >0)
      cond_wait(rdcond,mtx);
    unlock(mtx);
}

void ReaderExit() {
    lock(mtx);
    nr_readers--;
    /* Wake up one writer if
      there are no readers */
    if (nr_readers==0)
      cond_signal(wrcond);
    unlock(mtx);
}
```

```
void WriterEnter() {
    lock(mtx);
    while (nr_writers >0 || nr_readers > 0)
      cond_wait(wrcond,mtx);
    nr_writers++;
    unlock(mtx);
}

void WriterExit() {
    lock(mtx);
    nr_writers--;
    /* Wake up one writer if
       there are no readers.
       Otherwise, wake up all readers */
    if (nr_readers==0)
      cond_signal(wrcond);
    else
      cond_broadcast(rdcond);
    unlock(mtx);
}
```

Problema 4.b - Monitor (Mutex y Var. cond.)

Modificaciones para dar prioridad a los escritores

- Modificar la semántica de las variables `n_escritores` y `n_lectores`
  - `nr_readers`: Num. lectores que están dentro de su SC
  - `nr_writers`: Num. escritores que están dentro de su SC o que quieren entrar
  - Se cambia el orden en el que se incrementan (antes o después del while()).
- Añadimos variable compartida booleana `writer_in_cs`
  - Se inicializa a `false`. Se hace cierta cuando escritor dentro de su SC.

Problema 4.b - Monitor (Mutex y Var. cond.)

```
void ReaderEnter() {
    lock(mtx);
    while (nr_writers >0)
      cond_wait(rdcond,mtx);
    nr_readers++;
    unlock(mtx);
}

void ReaderExit() {
    lock(mtx);
    nr_readers--;
    if (nr_readers==0)
      cond_signal(wrcond);
    unlock(mtx);
}
```

```
void WriterEnter() {
    lock(mtx);
    nr_writers++;
    while (writer_in_cs || nr_readers > 0)
      cond_wait(wrcond,mtx);
    writer_in_cs=true;
    unlock(mtx);
}

void WriterExit() {
    lock(mtx);
    nr_writers--;
    writer_in_cs=false;
    if (nr_writers>0)
      cond_signal(wrcond);
    else
      cond_broadcast(rdcond);
    unlock(mtx);
}
```

Problema de los filósofos comensales

Dos tipos de soluciones al problema:

1 Solución con mutexes o semáforos

- Cada filósofo coge un palillo y luego otro
- Cada palillo representado mediante un mutex o semáforo inic. a 1
  - Coger palillo: `lock(m)` o `wait(sem)`
  - Soltar palillo: `unlock(m)` o `signal(sem)`
- Problema: posibles interbloqueos
  - Importa el orden a la hora de coger los palillos

2 Solución con monitor

- Cada filósofo *i* coge/deja los dos palillos a la vez usando sendos procedimientos de monitor:

```
cogerPalillosMonitor(i);
comer();
dejarPalillosMonitor(i);
```

- Ésta es la aproximación propuesta en el problema 5

Problema 5 (Filósofos) - Sol. con monitor

Se proporcionan dos soluciones:

- **Sol. 1**: Alternativa sencilla
  - Filósofos y palillos se numeran con IDs del 0 al 4
  - Posible inanición: Si filósofos k y $k+2$ se "compinchan" para comer y pensar alternativamente, el filósofo $k+1$ se muere de hambre
    - Los palillos que necesita el filósofo $k+1$ para comer nunca están a la vez en la mesa
- **Sol. 2**: Solución más sofisticada
  - Evita inanición en el caso de que filósofos intenten "compincharse" para intentar matar a otro de hambre

Problema 5 (Filósofos) - Sol. 1 con monitor

Especificación del monitor

```
typedef enum {LIBRE,OCUPADO} estado;
estado palillos[5]={LIBRE,LIBRE,LIBRE,LIBRE,LIBRE};
mutex mtx;
condvar condv[5]; /* Una VC por cada filósofo */;
/* Procedimientos de monitor */
void cogerPalillosMonitor(int i);
void dejarPalillosMonitor(int i);
```

```
void cogerPalillosMonitor(int i) {
    lock(mtx);
    while(palillo[i]==OCUPADO
    || palillo[(i+1)%5]==OCUPADO) {
        cond_wait(vcond[i],mutex);
    }
    palillo[i]=OCUPADO;
    palillo[(i+1)%5]=OCUPADO;
    unlock(mtx);
}
```

```
void dejarPalillosMonitor(int i) {
  lock(mtx);
  palillo[i]=LIBRE;
  palillo[(i+1)%5]=LIBRE;
  cond_signal(condv[(i+1)%5]);
  cond_signal(condv[(i-1)%5]);
  unlock(mtx);
}
```

Problema 5 (Filósofos) - Sol. 2 con monitor

```
typedef enum {PENSANDO,HAMBRIENTO,COMIENDO} estado_filosofo;
estado_filosofo estado[5]={PENSANDO,..,...,...PENSANDO};
mutex mtx;
condvar espera[5]; //Una variable condición por cada filósofo
```

```
void cogerPalillosMonitor(int i) {
    lock(mtx);
    estado[i]=HAMBRIENTO;
    actualiza_estado(i);
    while (estado[i]!= COMIENDO)
        cond_wait(espera[i], mtx);
    unlock(mtx);
}
```

```
void dejarPalillosMonitor(int i) {
    lock(mtx);
    estado[i]=PENSANDO;
    actualiza_estado((i-1)%5);
    actualiza_estado((i+1)%5);
    unlock(mtx);
}
```

```
void actualiza_estado (int i){
  if( estado[(i-1)%5] != COMIENDO && estado[i]== HAMBRIENTO &&
  estado[(i+1)%5] != COMIENDO ){
    estado[i]=COMIENDO;
    cond_signal(espera[i]);
  }
}
```

Problema 6 (I)

Especificación del monitor

```
typedef enum {TABACO=0,CERILLAS,PAPEL} ingrediente;
bool en_mesa[3]={false,false,false}; /* Inicialmente no hay de nada en la mesa */
mutex mtx;
condvar fumadores,agente;

/* Procedimientos de monitor */
void ConsigueIngredientes(ingrediente ing);
void DejaIngredientes(ingrediente ing1,ingrediente ing2);
```

```
void Fumador(ingrediente ing) {
   while(true) {
      ConsigueIngredientes(ing);
      fumar();
   }
}
```

```
void Agente() {
   ingrediente ing1,ing2;
   while(true) {
      (ing1,ing2)=ingrAleatorios();
      DejaIngredientes(ing1,ing2);
   }
}
```

Problema 6 (II)

```
void ConsigueIngredientes(ingrediente i)
{
    ingrediente id_ingr[2];

    lock(mtx);
    id_ingr[0]=(i+1)%3;
    id_ingr[1]=(i+2)%3;
    /* Bloqueo mientras no haya
     * los ingredientes necesarios
     * en la mesa
     */
    while ( ! en_mesa[id_ingr[0]] ||
        ! en_mesa[id_ingr[1]]) {
        cond_wait(fumadores,mtx);
    }

   /* El fumador coge de la mesa estos dos
        ingredientes */
   en_mesa[id_ingr[0]]=false;
   en_mesa[id_ingr[1]]=false;

    cond_signal(agente);
    unlock(mtx);
}
```

```
void DejaIngredientes(ingrediente i1,
       ingrediente i2){
    lock(mtx);
    while (en_mesa[0] ||
             en_mesa[1] ||
             en_mesa[2] )
        cond_wait(agente,mtx);

   /* El agente deja en la mesa los dos ingredientes */
    en_mesa[i1]=true;
    en_mesa[i2]=true;

    cond_broadcast(fumadores);
    unlock(mtx);
}
```

Problema 7.a

Versión no *thread-safe*

```
void Salvaje() {
    while(true) {
      .... Do something ...
      getServingFromPot();
      eat();
    }
}

void Cocinero() {
    while(true) {
      putServingsInPot(M);
    }
}
```

Version *thread-safe*

```
void Salvaje() {
    while(true) {
      .... Do something ...
      getServingSafe();
      eat();
    }
}

void Cocinero() {
    while(true) {
      prepareServingsSafe(M);
    }
}
```

- Reemplazamos `getServingFromPot()` y `putServingsInPot()` con variantes seguras de estas funciones
  - `getServingSafe()` invoca `getServingFromPot()` de forma segura
  - `prepareServingsSafe()` invoca `putServingsInPot()` de forma segura

Problema 7.a (mutex y variables condición)

Idea general de la solución

- El cocinero permanecerá bloqueado en `prepareServingsSafe()` mientras haya raciones en el caldero
- Si hay raciones en el caldero → al menos un salvaje podrá comer
  - se servirá una ración y retornará inmediatamente de `getServingSafe()`
- Si no quedan raciones en el caldero:
  1. El salvaje hambriento **despertará al cocinero** y **se bloqueará**
  2. Cuando el cocinero se despierte, rellenará el caldero con M raciones, **despertará a todos los salvajes** bloqueados y **volverá a dormirse**
  3. Tarde o temprano, el salvaje hambriento se despertará y comerá

Problema 7.a (mutex y variables condición)

Variables compartidas

```
int servings=0; /* # de raciones en el caldero (sup. vacío inicialmente) */
mutex mtx;    /* Para garantizar exclusión mutua al acceder a servings */
condvar cook, savages; /* Para bloquear al cocinero y a los salvajes cuando sea necesario */
```

```
void getServingSafe() {
    lock(mtx);

    while (servings == 0){
      cond_signal(cook);
      cond_wait(savages,mtx);
    }

    getServingFromPot();
    servings--;
    unlock(mtx);
}
```

```
void prepareServingsSafe(int m) {
    lock(mtx);

    while (servings > 0)
      cond_wait(cook,mtx);

    putServingsInPot(m);
    servings=m;
    cond_broadcast(savages);
    unlock(mtx);
}
```

Problema 7.b (semáforos)

- Cualquier solución que emplea variables condición y mutexes puede reescribirse usando solo semáforos
  - Recuerda que un semáforo puede usarse para emular un mutex (sem.c=1 inicialmente)
  - … y como cola de espera de propósito general (sem.c=0 inicialmente)

Variables condición puede emularse mediante dos semáforos y un contador

1. Semáforo `sem_mtx` actúa como "mutex" asociado a la "variable condición"
   - Contador del semáforo se inicializa a 1
   - Semáforo compartido entre var. condición del programa concurrente
2. Semáforo `sem_queue` que sirve de cola de espera
   - Contador del semáforo se inicializa a 0
   - Cuando hilo invoca `wait(sem_queue)` $\rightarrow$ se bloquea en la cola
   - Para despertar un hilo bloqueado en la "cola" : `signal(sem_queue)`
3. Contador `nr_waiting` lleva la cuenta del número de hilos esperando en la cola
   - Se inicializa a 0

Variables condición con semáforos

- `cond_wait()`

Variables condición

```
lock(mutex);
..
while(condition==false)
  cond_wait(vcond,mutex);
..
unlock(mutex);
```

Implementación con semáforos

```
/* "Lock" the mutex */
wait(sem_mtx);
..
while(condition==false) {
  nr_waiting++;
  signal(sem_mtx); // "Unlock" the mutex
  /* Goes to sleep in the queue */
  wait(sem_queue);
  wait(sem_mtx); // "Lock" the mutex
}
..
/* "Unlock" the mutex */
signal(sem_mtx);
```

Variables condición con semáforos

- `cond_signal()`

Variables condición

```
lock(mutex);
..
cond_signal(vcond);
..
unlock(mutex);
```

Implementación con semáforos

```
/* "Lock" the mutex */
wait(sem_mtx);
..
if (nr_waiting>0) {
  /* Wakes up a blocked thread */
  signal(sem_queue);
  nr_waiting--;
}
..
/* "Unlock" the mutex */
signal(sem_mtx);
```

Variables condición con semáforos

- `cond_broadcast()`

Variables condición

```
lock(mutex);
..
cond_broadcast(vcond);
..
unlock(mutex);
```

Implementación con semáforos

```
/* "Lock" the mutex */
wait(sem_mtx);
..
while (nr_waiting>0) {
  /* Wakes up a blocked thread */
  signal(sem_queue);
  nr_waiting--;
}
..
/* "Unlock" the mutex */
signal(sem_mtx);
```

Problema 7.b (semáforos)

Variables compartidas (mutex y var. condición)

```
int servings=0;
mutex mtx;
condvar cook, savages;
```

Variables compartidas (semáforos)

```
int servings=0;
semaphore sem_mtx;     /* contador del semáforo se inicializa a 1 */
semaphore cook_queue;  /* contador del semáforo se inicializa a 0 */
semaphore sav_queue;   /* contador del semáforo se inicializa a 0 */
int cook_waiting = 0;  /* # de hilos bloqueados en cook_queue (0 o 1) */
int nr_sav_waiting = 0; /* # de salvajes bloqueados */
```

Problema 7.b (semáforos)

Con variables condición

```
void getServingSafe() {
    lock(mtx);

    while (servings == 0){
      cond_signal(cook);
      cond_wait(savages,mtx);
    }

    getServingFromPot();
    servings--;
    unlock(mtx);
}
```

Con semáforos

```
void getServingSafe() {
    wait(sem_mtx); /* lock */

    while (servings == 0) {
      /* cond_signal */
      if (cook_waiting>0){
        signal(cook_queue);
        cook_waiting--;
      }
      /* cond_wait */
      sav_waiting++;
      signal(sem_mtx);
      wait(sav_queue);
      wait(sem_mtx);
    }

    getServingFromPot();
    servings--;
    signal(sem_mtx); /* unlock */
}
```

Problema 7.b (semáforos)

Con variables condición

```
void prepareServingsSafe(int m) {
    lock(mtx);

    while (servings > 0)
      cond_wait(cook,mtx);

    putServingsInPot(m);
    servings=m;
    cond_broadcast(savages);
    unlock(mtx);
}
```

Con semáforos

```
void prepareServingsSafe(int m) {
    wait(sem_mtx); /* lock */

    while (servings > 0){
      /* cond_wait */
      cook_waiting++;
      signal(sem_mtx);
      wait(cook_queue);
      wait(sem_mtx);
    }

    putServingsInPot(m);
    servings=m;
    /* cond_broadcast */
    while (sav_waiting>0){
      signal(sav_queue);
      sav_waiting--;
    }
    signal(sem_mtx); /* unlock */
}
```

Problema 8

Idea general de la solución

- Cuando un ciente llegue a la gasolinera debe coger turno
  - Usaremos una variable `nextAvailableTurn`, que se incrementará cada vez que llegue un cliente
- Como hay dos surtidores (*pumps*), los clientes con los dos turnos de menor valor podran usar los surtidores
  - Usaremos la variable `smallestTurn` para anotar el primer turno disponible (cliente que será atendido primero)
  - Por tanto, los clientes cuyo turno sea menor que `smallestTurn + 2`, podrán usar un surtidor

Comportamiento del cliente

```
void customer(int price) {
   int pump;
   pump=getUnusedPump();
   /* Cliente usa el surtidor */
   PumpFuel(pump,price);
   /* Cliente deja de usar el surtidor */
   releasePump(pump);
}
```

Problema 8 (Solución #1)

Variables compartidas

```
#define NR_PUMPS 2
mutex mtx;    /* Para garantizar exclusion mutua al acceder a var. compartidas */
condvar cv; /* Para bloquear a los clientes */
bool pumpInUse[NR_PUMPS]={FALSE,FALSE}; /* Los surtidores están libres inicialmente */
int nextAvailableTurn=0;
int smallestTurn=0;
```

```
int getUnusedPump(void) {
  int my_turn,i=0;
  lock(mtx);
  my_turn=nextAvailableTurn++;
  /* Esperar a que haya surtidor libre */
 while (my_turn >= smallestTurn+NR_PUMPS)
    cond_wait(cv,mtx);

  /* Escoger surtidor libre */
  while(pumpInUse[i])
    i++;

  pumpInUse[i]=TRUE;
  unlock(mtx);
  return i;
}
```

```
void releasePump(int pump){
  int i;

  lock(mtx);
  smallestTurn++;
  pumpInUse[pump]=FALSE;
  /* Despertar a todos los clientes
   * para que puedan comprobar
   * si ha llegado su turno
     */
   cond_broadcast(cv,mtx);
   unlock(mtx);
}
```

Problema 8 (Solución #1)

La solución #1 tiene limitaciones

- Cuando los clientes bloqueados se despiertan con `cond_broadcast()`, todos compiten por el mutex dentro de `cond_wait()`
  - Al hacer esto, los turnos no se respetan → solo porque un cliente A llegue antes que otro cliente B, no significa que A adquiera el mutex antes que B cuando ambos clientes se despiertan al mismo tiempo con `cond_broadcast()`
- Problema: Supongamos que los surtidores están en uso. Sean A, B los clientes que deben acceder a los surtidores a continuación (cuando queden libres), considerando que A llegó antes que B.
  - Si los dos surtidores se liberan al mismo tiempo, la implementación no garantiza que A acceda a un surtidor antes que B ya que la condición (`my_turn < smallestTurn+NR_PUMPS`) es cierta para ambos clientes.
  - Por tanto, los turnos no se respetan estrictamente
  - El problema sería peor si hubiera más de 2 surtidores

Problema 8 (Solución #2)

- Es posible solucionar el problema de los turnos garantizando que los surtidores se asignen en orden a los clientes
- Necesitamos dos variables adicionales:
  1. `nr_unused_pumps`: Para llevar la cuenta del número de surtidores libres
  2. `currentTurn`: Indica el turno (estricto) del siguiente cliente al que le tocaría usar un surtidor
     - Esta variable reemplaza a la variable `smallestTurn` en la Solución #1

Variables compartidas

```
#define NR_PUMPS 2
mutex mtx;    /* Para garantizar exclusion mutua al acceder a var. compartidas */
condvar cv; /* Para bloquear a los clientes */
bool pumpInUse[NR_PUMPS]={FALSE,FALSE}; /* Los surtidores están libres inicialmente */
int nextAvailableTurn=0;
int nr_unused_pumps=NR_PUMPS;
int currentTurn=0;
```

Problema 8 (Solución #2)

```
int getUnusedPump(void) {
  int my_turn,i=0;
  lock(mtx);
  my_turn=nextAvailableTurn++;
  /* Esperar a que sea mi turno */
  while (my_turn != currentTurn ||
         nr_unused_pumps==0 )
    cond_wait(cv,mtx);

  /* Escoger surtidor libre */
  while(pumpInUse[i])
    i++;

  pumpInUse[i]=TRUE;
  nr_unused_pumps--;

  /* Incrementar turno actual */
  currentTurn++;
  /* y despertar clientes si es necesario */
  if (nr_unused_pumps>0)
    cond_broadcast(cv,mtx);

  unlock(mtx);
  return i;
}
```

```
void releasePump(int pump){
  int i;

  lock(mtx);
  pumpInUse[pump]=FALSE;
  nr_unused_pumps++;
  /* Despertar a todos los clientes
   * para que puedan comprobar
   * si ha llegado su turno
     */
   cond_broadcast(cv,mtx);
   unlock(mtx);
}
```

Problema 8 (Solución #2)

- La solución #2 puede ser bastante ineficiente cuando el número de clientes es elevado
  - `cond_broadcast()` despierta a todos los clientes que esperan, pero solo un cliente podría proseguir con la ejecución
    - Esto provoca que el SO haga muchos cambios de contexto
- La solución podría mejorarse usando una cola FIFO de variables condición
  - En caso de que un cliente deba bloquearse (no es su turno), el cliente crea una variable condición dinámicamente, encola la variable condición en la cola FIFO e invoca `cond_wait()` sobre ella
  - Cuando un surtidor se libere, se despertará al cliente cuyo turno sea el que corresponda → cliente bloqueado en la variable condición que está al comienzo de la cola FIFO
    - De este modo ya no sería necesario invocar `cond_broadcast()`