CC4302
Sistemas Operativos
Profesor: Luis Mateu

- Problema: orden de atención indefinido
- Atención por orden de llegada
- Lectores/escritores sin hambruna: por orden de llegada
- Patrón request para atender en orden específico

Problema: orden de atención indefinido

- Los threads suspendidos con *pthread_mutex_lock* o *pthread_cond_wait* se despiertan en cualquier orden
- Primera solución para atender por orden de llegada: threads sacan un número cuando se colocan en espera, como los clientes de una farmacia o isapre.
- Segunda solución: cada thread se coloca en espera en su propia condición y las condiciones se agregan a y extraen de una cola FIFO.
- Ejemplo: Múltiples threads comparten un recurso único que debe ser ocupado en exclusión mutua. Para coordinarse, los threads invocan la función *ocupar()* para solicitar el recurso e invocan *desocupar()* para liberarlo.

Atención en orden no especificado

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;

void ocupar() {
    pthread_mutex_lock(&m);
}

void desocupar() {
    pthread_mutex_unlock(&m);
}
```

Los mutex se otorgan en un orden no especificado

Atención en orden no especificado

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  c = PTHREAD_COND_INITIALIZER;
int busy = 0;

void ocupar() {
    pthread_mutex_lock(&m);
    while (busy)
        pthread_cond_wait(&c, &m);
    busy= 1;
    pthread_mutex_unlock(&m);
}

void desocupar() {
    pthread_mutex_lock(&m);
    busy= 0;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```

Cuando un thread va a esperar bastante tiempo, es más eficiente que espere con wait que con lock

Las llamadas a pthread_cond_wait se despiertan en un orden no especificado

Atención por orden de llegada: sacar número como en las farmacias/isapres

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  c = PTHREAD_COND_INITIALIZER;
int ticket_dist = 0, display = 0; // ticket distrib. & display

void ocupar()  {
    pthread_mutex_lock(&m);
    int my_num= ticket_dist++;
    while (my_num!=display)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

void desocupar()  {
    pthread_mutex_lock(&m);
    display++;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```

¡Importante!

¡Sí atiende por orden de llegada!

Lectores/ecritores sin hambruna

- La solución del problema de los lectores/escritores vista en la clase pasada tiene un defecto: *hambruna*
- 2 lectores se pueden concertar para entrar alternadamente de manera que siempre haya un lector presente, haciendo esperar al escritor para siempre
- Una manera de evitar la hambruna es otorgar los permisos de lectura/escritura por orden de llegada
- ¡Hay que sacar número!

Lectores/escritores por orden de llegada

```
pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  c = PTHREAD_COND_INITIALIZER;
int ticket_dist = 0, display = 0; // ticket distributor and display
int readers= 0; // no se necesita writing
```

```
void enterWrite() {
   pthread_mutex_lock(&m);
   int my_num= ticket_dist++;
   while (my_num!=display ||
            readers>0)
      pthread_cond_wait(&c, &m);
   pthread_mutex_unlock(&m);
}
void exitWrite() {
   pthread_mutex_lock(&m);
   display++;
   pthread_cond_broadcast(&c);
   pthread_mutex_unlock(&m);
}
```

¡Sí atiende por orden de llegada!

```
void enterRead() {
   pthread_mutex_lock(&m);
   int my_num= ticket_dist++;
   while (my_num!=display)
      pthread_cond_wait(&c, &m);
   readers++;
   display++;
   pthread_cond_broadcast(&c);
   pthread_mutex_unlock(&m);
}
void exitRead() {
   pthread_mutex_lock(&m);
   readers--;
   if (readers==0)
      pthread_cond_broadcast(&c);
   pthread_mutex_unlock(&m);
}
```

Problema: numero explosivo de cambios de contexto

- Volviendo al problema del recurso compartido, considere que hay *n* threads en espera
- Al liberar el recurso se deben despertar los *n* threads, pero solo uno adquiere el recurso
- Se producen *n-1 cambios de contexto* inútiles
- Para el caso de los lectores/escritores, considere que hay *n* lectores en espera
- Cuando el escritor sale despierta a los *n* threads
- Algunos threads no tendrán el siguiente número y se volverán a dormir
- Cuando por fin se despierta el thread con el siguiente número, vuelve a despertar a todos los lectores
- En el peor caso se producen $O(n^2)$ cambios de contexto inútiles

Patrón *request* para atender en un orden específico: caso orden de llegada

```
typedef struct {
   int ready;
   pthread_cond_t  w;
} Request;

pthread_mutex_t  m = PTHREAD_MUTEX_INITIALIZER;
int busy = 0;
Queue *q;

void ocupar()  {
   pthread_mutex_lock(&m);
   if (!busy)
      busy = 1;
   else {
      Request req= { 0,
         PTHREAD_COND_INITIALIZER };
      put(q, &req);
      while (! req.ready)
         pthread_cond_wait(&req.w, &m);
   }
   pthread_mutex_unlock(&m);
}
```

```
void desocupar()  {
   pthread_mutex_lock(&m);
   if (emptyQueue(q))
      busy = 0;
   else {
      Request *preq= get(q);
      preq->ready = 1;
      pthread_cond_signal(&preq->w);
   }
   pthread_mutex_unlock(&m);
}
```

Aplicaciones del patrón request

- Para atender por orden de llegada
- Pero la cola no tiene por qué ser FIFO
- Puede ser un cola de prioridades para atender las solicitudes más importantes primero
- En el caso general se puede usar cualquier tipo de datos que permita satisfacer los requerimientos en el orden de atención
- Una mejor solución del problema de los lectores/escritores es atender por orden de llegada, pero con la siguiente excepción: *cuando se va un escritor y le toca entrar a un lector, entran todos los lectores en espera*