CC4302
Sistemas Operativos
Profesor: Luis Mateu

- Semáforos
- Cena de filósofos resuelta con semáforos para garantizar la exclusión mutua
- Productor/consumidor resuelto con semáforos
- Productor/consumidor resuelto con patrón request usando semáforos en vez de mutex/condiciones
- Lectores/escritores resuelto con patrón request/semáforos

Semáforos

Representa un dispensador de fichas

Operaciones:

- Inicializar un semáforo con *val* fichas:

  *void **sem_init**(sem_t *sem, int pshared, unsigned val);*
- Extraer una ficha: *void **sem_wait**(sem_t *sem);*
- Depositar una ficha: *void **sem_post**(sem_t *sem);*
- Destruir el semáforo: *void **sem_destroy**(sem_t *sem);*
- *Principio fundamental: si no hay fichas disponibles, sem_wait espera hasta que otro thread deposite una ficha con sem_post*
- *Si hay varios threads en espera de una ficha, no se especifica un orden para otorgar las fichas*
- El uso más frecuente es para garantizar la exclusión mutua: en tal caso se inicializa con una sola ficha
- También se puede usar para suspender un thread hasta que se cumpla una condición: en tal caso se inicializa con 0 fichas
- Hay pocos usos en que parte con varias fichas

Cena de filósofos resuelta con semáforos para garantizar la exclusión mutua

```
sem_t pals[5];
…
for (int i= 0; i<5; i++)
   sem_init(&pals[i], 0, 1);  // Debe partir con exactamente con 1 ficha

void filosofo(int i) {
   for (;;) {
      int j= min(i, (i+1)%5);
      int k= max(i, (i+1)%5);
      sem_wait(&pals[j]);  // Garantiza exclusión mutua para palito j: lock
      sem_wait(&pals[k]); // Garantiza exclusión mutua para palito k: lock

      comer(i, (i+1)%5);

      sem_post(&pals[j]);  // unlock
      sem_post(&pals[k]); // unlock
      pensar();
   }
}
```

- *pals[i]* garantiza la exclusión mutua de los filósofos al usar el palito *i*

Productor/consumidor resuelto con semáforos

```
typedef struct {
  int size;
  void **array;
  int in, out;
  sem_t empty, full;
} Buffer;

Buffer *nuevoBuffer(int size) {
  Buffer *buf=
                malloc(sizeof(Buffer));
  buf->size= size;
  buf->array=
       malloc(size*sizeof(void*));
  buf->in= buf->out= 0;
  sem_init(&buf->empty, 0, size);
  sem_init(&buf->full, 0, 0);
  return buf;
}
```

```
void put(Buffer *buf, void *item) {
  sem_wait(&buf->empty); // lock
  buf->array[buf->in]= item;
  buf->in= (buf->in+1) % buf->size;
  sem_post(&buf->full); // unlock
}

void *get(Buffer *buf) {
  sem_wait(&buf->full); // lock
  void *item= buf->array[buf->out];
  buf->out= (buf->out+1) % buf->size;
  sem_post(&buf->empty); // unlock
  return item;
}
```

- Cada ficha en empty representa una casilla vacía
- Cada ficha en full representa una casilla llena
- Solución clásica
- No funciona con múltiples consumidores/productores
- Difícil tomar como molde para otros problemas

Productor/consumidor resuelto con patrón request usando semáforos en vez de mutex/condiciones

```
typedef struct {
  int size;
  void **array;
  int in, out, cnt;
  sem_t mutex;
  Queue *prodq, *consq;
} Buffer;

Buffer *nuevoBuffer(int size) {
  Buffer *buf= malloc(sizeof(Buffer));
  buf->size= size;
  buf->array= malloc(size*sizeof(void*));
  buf->in= buf->out= buf->cnt= 0;
  sem_init(&buf->mutex, 0, 1);
  buf->prodq= makeQueue();
  buf->consq= makeQueue();
  return buf;
}
```

```
void putBuf (Buffer *buf, void *item) {
  sem_wait(&buf->mutex); // lock
  if (buf->cnt==buf->size) {
     sem_t w; // ¡Como condición!
     sem_init(&w, 0, 0);
     put(buf->prodq, &w);
     sem_post(&buf->mutex); // unlock
     sem_wait(&w); // cond wait
  }
  buf->array[buf->in]= item;
  buf->in= (buf->in+1) % buf->size;
  buf->cnt++;
  if (emptyQueue(buf->consq))
     sem_post(&buf->mutex); // unlock
  else {
     sem_t *pw= get(buf->consq);
     sem_post(&buf->mutex); // unlock
     sem_post(pw); // cond signal
  }
}

void *getBuf(Buffer *buf) {
   … ejercicio …
}
```

Lectores/escritores resuelto con patrón request/semáforos (1era parte)

```
typedef enum { READER, WRITER } Kind;
sem_t mutex; // sem_init(&mutex, 0, 1);
int readers= 0, writing= 0;
Queue *q; // = makeQueue()
```

```
void enterWrite() {
   sem_wait(&mutex); // lock
   if (readers==0 && !writing) {
      writing = 1;
      sem_post(&mutex); // unlock
   }
   else
      enqueue(WRITER);
}
void exitWrite() {
   sem_wait(&mutex); // lock
   writing = 0;
   wakeup();
   sem_post(&mutex); // unlock
}
```

```
void enterRead() {
   sem_wait(&mutex); // lock
   if (!writing && emptyQueue(q)){
      readers++;
      sem_post(&mutex); // unlock
   }
   else
      enqueue(READER);
}
void exitRead() {
   sem_wait(&mutex); // lock
   readers--;
   if (readers==0)
      wakeup();
   sem_post(&mutex); // unlock
}
```

Lectores/escritores resuelto con patrón request/semáforos (2da parte)

```
typedef struct {
   Kind    kind;
   sem_t  w;
} Request;

void enqueue(Kind kind) {
   // ¡Patrón request!
   Request req= {kind};
   sem_init(&req.w, 0, 0);
   put(q, &req);
   sem_post(&mutex); // unlock
   sem_wait(&req.w);
}
```

```
void wakeup() {
   Req *pr= peek(q); // ~~get(q)~~
   if (pr==NULL)
      return;

   if (pr->kind==WRITER) {
      writing= 1; // entra un escritor
      sem_post(&pr->w); // signal
      get(q);  // Sale de la cola
   }
   else { // ¡ pr->kind==READER !
      do { // entran lectores consecutivos
         readers++;
         sem_post(&pr->w); // signal
         get(q);  // Sale de la cola
         pr= peek(q);
         // Si es escritor no sale de la cola
      } while ( pr!=NULL &&
                  pr->kind==READER );
   }
}
```

Conclusiones

- Los semáforos son más difíciles de usar que los mutex con condiciones
- Pero cualquier solución con mutex y condiciones se puede traducir a semáforos
- No son más eficientes
- Podrían ser menos eficientes
- Algunos ambientes de programación no ofrecen mutex/condiciones, pero sí ofrecen semáforos, por ejemplo el núcleo de Linux
- Tanto mutex/condiciones y semáforos se implementan en el bajo nivel mediante *spin-locks*
- Un *spin-lock* es un semáforo que puede almacenar una sola ficha e implementado usando *busy-waiting*
- La traducción de mutex/condiciones a semáforos, también sirve para para traducir a spin-locks

Ejercicio propuesto

- Resuelva los lectores/escritores con patrón request/semáforos por orden de llegada, pero cuando le toca a un lector, entran todos los lectores en espera, sin importar que hayan llegado después de un escritor
- En la clase pasada se resolvió el mismo problema con mutex y múltiples condiciones (patrón request)