CC4302
Sistemas Operativos
Profesor: Luis Mateu

- Equivalencia entre semáforos y mutex/condiciones
- Mutex y condiciones implementados a partir de semáforos
- 3 patrones de paralelización con threads: dividir en subintervalos, divide y conquista en paralelo y productor/consumidor
- Búsqueda de un factor con patrón productor/consumidor
- Paralelismo del tipo AND y del tipo OR

Equivalencia entre semáforos y mutex/condiciones

- Todo problema resuelto con semáforos se puede resolver con mutex y condiciones
- Para demostrarlo basta implementar semáforos a partir de mutex y condiciones (ejercicio propuesto)
- En la solución con semáforos basta reemplazar las llamadas a las funciones de los semáforos por las funciones que implementan los semáforos a partir de mutex y condiciones
- Todo problema resuelto con mutex y condiciones se puede resolver con semáforos
- Para demostrarlo basta implementar los mutex y condiciones a partir de semáforos (resuelto en esta clase)
- En la solución con mutex y condiciones basta reemplazar las llamadas a las funciones de los mutex y condiciones por las funciones que los implementan a partir de semáforos

Mutex y condiciones a partir de semáforos: la API

```
typedef struct mutex Mutex;

typedef struct cond Cond;

void mutex_init(Mutex *m);
void cond_init(Cond *c);
void mutex_destroy(Mutex *m);
void cond_destroy(Cond *c);
void mutex_lock(Mutex *m);
void mutex_unlock(Mutex *m);
void cond_wait(Cond *c, Mutex *m);
void cond_signal(Cond *c);
void cond_broadcast(Cond *c);
```

Mutex y condiciones a partir de semáforos: implementación

```
#include "mutex.h"

struct mutex {
  sem_t mutex_sem;
  Queue *mq;
};

struct cond {
  Mutex *m;
  Queue *wq;
};

void mutex_init(Mutex *m) {
  sem_init(&m->mutex_sem, 0, 1);
  m->mq= makeQueue();
}

void cond_init(Cond *c) {
  c->wq= makeQueue();
}

void mutex_destroy(Mutex *m) {
  sem_destroy(&m->mutex_sem);
  destroyQueue(m->mq);
}

void cond_destroy(Cond *c) {
  destroyQueue(c->wq);
}
```

```
void mutex_lock(Mutex *m) {
  sem_wait(&m->mutex_sem);
}

void mutex_unlock(Mutex *m) {
  if (emptyQueue(m->mq))
    sem_post(&m->mutex_sem);
  else {
    sem_t *pw_sem= get(m->mq);
    sem_post(pw_sem);
  }
}

void cond_wait(Cond *c, Mutex *m) {
  c->m= m;
  sem_t w_sem;
  sem_init(&w_sem, 0, 0);
  put(c->wq, &w_sem);
  mutex_unlock(m);
  sem_wait(&w_sem);
}

void cond_signal(Cond *c) {
  if (!emptyQueue(c->wq)) {
    sem_t *pw_sem= get(c->wq);
    put(c->m->mq, pw_sem);
  }
}
// while para cond_broadcast
```

Observaciones:

- Como en el núcleo de Linux no hay mutex y condiciones pero sí se dispone de semáforos, podrá usar una implementación similar a esta en la tarea de módulos de Linux para lograr la sincronización
- Para las condiciones existe la operación:

```
pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex,
        struct timespec *abstime);
```

- Si no se ha recibido un signal/broadcast dentro de *abstime*, la operación termina
- Ejemplo de uso:

```
int inicio = … hora actual …;
while ( ! …  condición … ) {
  if ( … hora actual … - inicio > tolerancia)
    break;
  … llenar abstime …
  pthread_cond_timedwait(&cond, &mutex, &abstime);
}
```

- Esto no se puede implementar con semáforos porque no hay un *sem_timedwait*.

Patrones de paralelización con threads

- *Descomponer en subintervalo*: en un ciclo en donde no hay dependencia entre iteraciones, se puede descomponer el intervalo del ciclo en NT subintervalos y cada thread se ocupa de un subintervalo.
  - Ejemplos: multiplicación de matrices, búsqueda de un factor
- *Divide y conquista en paralelo*: en una función con 2 llamadas recursivas, la primera llamada se ejecuta en un nuevo thread y la segunda en el thread original.
- Generar demasiados threads es ineficiente
- Cuando se hayan completado ***nt*** threads se continúa secuencialmente con la recursividad
- Ejemplo: quicksort paralelo, exploración de todas las combinaciones
- *Productor/consumidor*: un solo productor genera unidades de trabajo y las deposita en un buffer, NT consumidores iteran extrayendo unidades de trabajo y las ejecutan. ¡Nuevo!

Ejemplo: Búsqueda de un factor de un entero x

- Desventaja de la búsqueda de un factor vista en clase auxiliar: si un thread encuentra tempranamente un factor, de todas formas hay que esperar que terminen todos los otros threads
- Algún thread podrían tardar porque no hay un factor en el rango en donde les tocó buscar
- Solución: un thread productor de trabajos (jobs) con intervalos de tamaño 1000000 en donde buscar
- Múltiples threads consumidores de jobs iteran extrayendo intervalos y buscando un factor en ese intervalo
- Cuando un thread encuentra un factor, el productor envía trabajos nulos, para notificar a los consumidores que deben terminar

Ejemplo: Búsqueda de un factor de un entero x con patrón productor/consumidor

```
#include <pthread.h>
#include <math.h>
#include "buffer.h "

// busca un factor del número
// entero x en el rango [i, j]

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

typedef struct {
  ulonglong x;
  uint i, j, *pres;
  // pthread_mutex_t *pm;
} Job;
```

```
void *threadFun(void* ptr) {
  Buffer *buf = ptr;
  for (;;) {
    Job *pjob= bufGet(buf);
    if (pjob==NULL)
      return NULL;
    if (*pjob->pres!=0) {
      free(pjob);
      return NULL;
    }
    ulonglong x= pjob->x;
    uint i= pjob->i;
    uint j= pjob->j;
    uint res= buscarFactor(x, i, j);
    if (res!=0) {
      // pthread_mutex_lock(pjob->pm);
      *pjob->pres= res;
      // pthread_mutex_unlock(pjob->pm);
    }
    free(pjob);
  }
}
```

Ejemplo: Búsqueda de un factor de un entero x con patrón productor/consumidor

```
// busca un factor del número
// entero x utilizando nt cores
uint buscarFactorParalelo(
        ulonglong x, int nt) {
 pthread_t pid[nt];
 uint j= sqrt(x);
 j++;
 uint delta= 1000000;
 // pthread_mutex_t m=
 //       PTHREAD_MUTEX_INITIALIZER;
 uint res= 0;
 Buffer *buf= makeBuffer(nt);
 for (int k= 0; k<nt; k++) {
  pthread_create(&pid[k],
       NULL, threadFun, buf);
 }
```

```
 for (int i=2; i<=j; i+=delta) {
  // pthread_mutex_lock(pjob->pm);
  int end= res!=0;
  // pthread_mutex_unlock(pjob->pm);
  if (end)
   break;
  Job job= { x, i, i+delta-1, &res
                /*, &m */ };
  Job *pjob= malloc(sizeof(Job));
  *pjob= job;
  bufPut(buf, pjob);
 }

 for (int k=0; k<nt; k++)
  bufPut(buf, NULL);
 for (int k=0; k<nt; k++)
  pthread_join(pid[k], NULL);

 destroyBuffer(buf);
 return res;
}
```

Paralelismo del tipo AND y del tipo OR

- El típico paralelismo que habíamos visto hasta el momento es del tipo *AND*: se necesita los resultados de todos los threads
- La búsqueda de un factor en paralelo es del tipo *OR*: si uno de los threads encuentra un resultado, no se necesitan los resultados de los demás threads
- El problema es detenerlos
- Encontrar en paralelo ***una*** combinación de valores en donde una función booleana se hace verdadera (SAT), también es paralelismo del tipo OR
- El patrón productor/consumidor permite limitar el trabajo inútil
- *Alternativa*: usar una variable booleana compartida que indica cuando se encontró un factor, pero eso significa un pequeño sobrecosto en consultar esa variable por cada división hecha
- Esa variable se lee casi siempre, por lo que los cores la pueden mantener en sus propios cachés, *¡excepto si por mala suerte queda en la misma línea de una variable que los threads sí modifican!*