Sistemas Operativos y Distribuidos

Anexo práctico 5

Casos de estudio

GNU/Linux

La realización del anexo GNU/Linux es obligatorio. Es importante la comprensión de estos contenidos dado que serán necesarios para la resolución de los próximos anexos.

A.L.5.1. ¿Qué solución implementa Linux para manejar los interbloqueos? ¿Es la misma que la utilizada en Microsoft Windows? ¿Por qué?

A.L.5.2. Determinar el problema resultante al ejecutar el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>
pthread_mutex_t mutexA ;
pthread_mutex_t mutexB ;
int cont = 0;

void *routineA (void *t){
    long my_id = (long)t;
    while(true){
        printf("[%ld] Acceso\n", my_id);
        pthread_mutex_lock(&mutexA);
        pthread_mutex_lock(&mutexB);
        printf("[%ld] Procesamiento sobre recurso compartido: %d\n", my_id, cont++);
        pthread_mutex_unlock(&mutexB);
        pthread_mutex_unlock(&mutexA);
    }
    pthread_exit (EXIT_SUCCESS);
}
void *routineB (void *t){
    long my_id = (long)t;
    while(true){
        printf("[%ld] Acceso\n", my_id);
        pthread_mutex_lock(&mutexB);
        pthread_mutex_lock(&mutexA);
        printf("[%ld] Procesamiento sobre recurso compartido: %d\n", my_id, cont++);
        pthread_mutex_unlock(&mutexA);
        pthread_mutex_unlock(&mutexB);
    }
    pthread_exit (EXIT_SUCCESS);
}
int main (){
    long t1=1, t2=2;
    pthread_t th1 , th2;
    pthread_mutex_init (&mutexA , NULL);
    pthread_mutex_init (&mutexB , NULL);
    pthread_create (&th1 , NULL , routineA , (void *)t1);
    pthread_create (&th2 , NULL , routineB , (void *)t2);
    pthread_join (th1 , NULL);
    pthread_join (th2 , NULL);
    pthread_mutex_destroy (&mutexA);
    pthread_mutex_destroy (&mutexB);

return EXIT_SUCCESS;
}
```

A.L.5.3. Proponer una solución al problema planteado en el ejercicio **A.L.5.2** utilizando la primitiva *pthread_mutex_trylock()*.

A.L.5.4. Dada la siguiente solución al problema de los filósofos cenando:

```
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <semaphore.h>  /* Semaphore */
#define N 5
sem_t tenedor[N];

void *filosofo( void *filo ){
    int f;
    f = *((int *) filo);
    while(1){
        printf("Voy a pensar: %d\n", f);
        printf("Espero por primer Tenedor: %d\n", f);
        sem_wait(&tenedor[f]);
        printf("Espero por segundo Tenedor: %d\n", f);
        sem_wait(&tenedor[(f+1)%N]);
        printf("Voy a comer:  %d\n", f);
        sem_post(&tenedor[f]);
        sem_post(&tenedor[(f+1)%N]);
    }
    pthread_exit(EXIT_SUCCESS);
}

int main(){
    int i[N] = {0,1,2,3,4}, j;
    pthread_t f[N];
    for(j=0; j<N;j++){ sem_init(&tenedor[j], 0, 1); }
    for(j=0; j<N;j++){ pthread_create (&f[j], NULL, filosofo, (void *) &i[j]); }
    for(j=0; j<N;j++){ pthread_join(f[j], NULL); }
    for(j=0; j<N;j++){ sem_destroy(&tenedor[j]); }
    exit(EXIT_SUCCESS);
}
```

a) Ejecutar y explicar por qué se produce interbloqueo.
b) ¿Cuánto tiempo tarda en caer el proceso en interbloqueo?
c) Dibujar el grafo de asignación de recursos del interbloqueo.
d) Una solución que no contiene interbloqueos ni inanición utiliza un semáforo binario que engloba a todas las instrucciones luego de que el filósofo deja de pensar.
   i) Implementar esta solución.
   ii) ¿Cuál es el problema con esta solución?
e) Otra posible solución al problema de los filósofos plantea que todo filósofo hambriento obtenga en primera instancia el palillo que está a su izquierda. Si el palillo que está a su derecha también está disponible entonces el filósofo lo toma y comienza a comer. En caso contrario, el filósofo deja el palillo izquierdo y repite el ciclo. ¿Cuál es el problema asociado a la solución planteada?