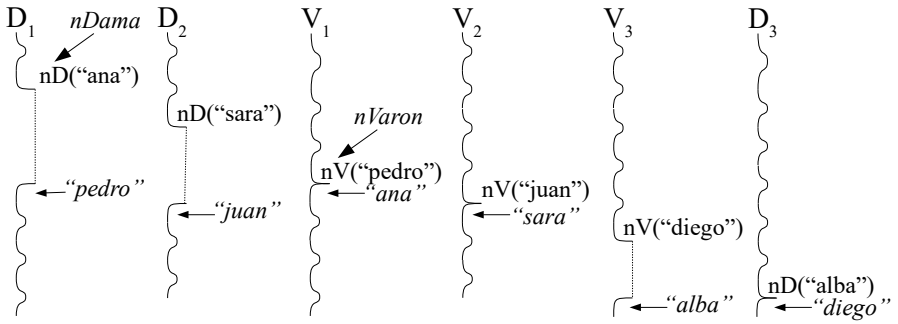
CC4302 Sistemas Operativos – Control – Semestre Primavera 2022 – Prof.: Luis Mateu

Pregunta 1

Este es el mismo problema de la tarea 3: Los nano threads representan damas y varones que buscan una pareja para bailar en una discoteca invocando *nDama* y *nVaron*. La función *nDama* recibe como parámetro el nombre de la dama. Si hay varones esperando pareja *nDama* retorna de inmediato el nombre del varón que llegó primero. Es decir que la asignación de parejas se hace por orden de llegada. Si no hay varones en espera *nDama* espera por una invocación de *nVaron*. Análogamente *nVaron* retorna al instante el nombre de la primera dama que todavía busca su pareja, o espera una invocación de *nDama*. El siguiente es un diagrama de tareas que muestra un ejemplo de ejecución.



El diagrama muestra que la pareja de ana (D_1) es pedro (V_1) y que por lo tanto *nDama* en D_1 retorna "pedro" y *nVaron* en V_1 retorna "ana".

Parte a.- (4 puntos) Programe las funciones *nDama* y *nVaron* como herramientas de sincronización nativas de nThreads, es decir usando operaciones como *START_CRITICAL*, *setReady*, *suspend*, *schedule*, *nth_putBack*, etc. Inicialice las variables globales que necesite en la función *nth_iniDisco*. Ud. ***no*** puede implementar la API solicitada en términos de otras herramientas de sincronización pre-existentes en nThreads (como semáforos, mutex, condiciones o mensajes). Los encabezados de las funciones pedidas son:

*char *nDama(char *nombre);*
*char *nVaron(char *nombre);*

Solución:

```
// Esto no da puntaje:
// Se necesitan los campos nombre y pareja en el
// descriptor de thread
NthQueue *damas, *varones;

void nth_iniDisco() {
  damas= nth_makeQueue();
  varones= nth_makeQueue;
}
```

```
char *nDama(char *nombre) {
  // 0,5 puntos por START y END_...
  START_CRITICAL
  char *pareja;
  // 0,2 puntos:
  nSelf()->nombre= nombre;
  // 0,2 puntos:
  if(nth_emptyQueue(varones)){
    nThread *w= // 0,2 puntos
      nth_getFront(varones);
    pareja= w->nombre; // 0,2
    w->pareja= nombre; // 0,2
    setReady(w); // 0,4
    schedule();  // 0,4
  }
  else {
    suspend(WAIT_DAMA); // 0,4
    schedule(); // 0,4
    // 0,2 puntos:
    pareja= nSelf()->pareja;
    // con pareja= variable
    //              global;
    // habria datarace
  }
  END_CRITICAL
  return pareja; // 0,2
  // o nSelf()->pareja
}
```

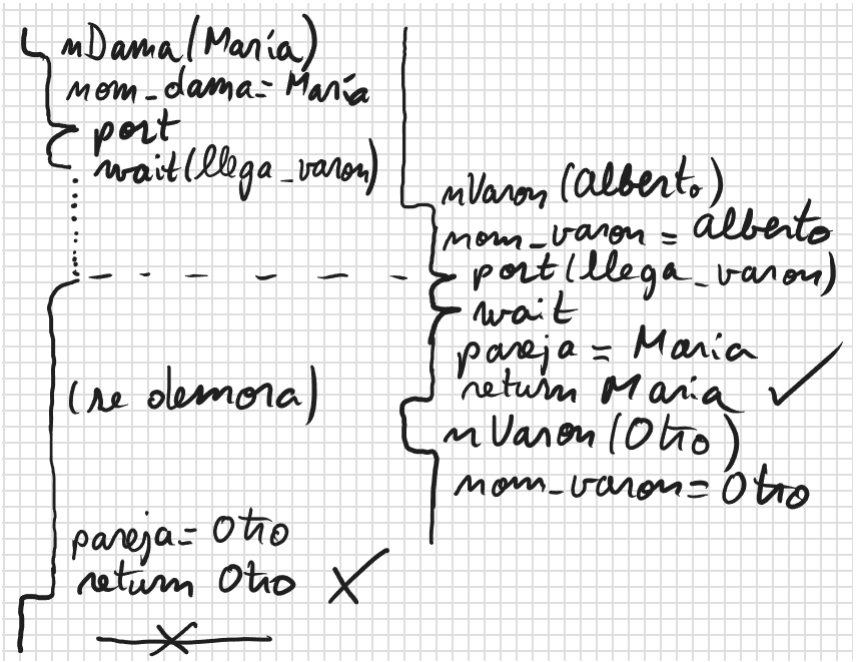
```
// 0,5 puntos por la otra función
// ¡revisen la que esté mejor!
char *nVaron(char *nombre){
  ...
  // 0,2 puntos:
  if(nth_emptyQueue(damas)){
    nThread *w=
      nth_getFront(damas);
    ...
  }
  else {
    // 0,3 puntos:
    suspend(WAIT_VARON);
    ...
  }
  ...
}
```

Parte b.- (1 punto) La implementación de arriba a la derecha es una solución **incorrecta** de la *parte a* basada en los semáforos de pthreads. En esta implementación **no se requiere** que las parejas se armen por orden de llegada. Haga un diagrama de threads que muestre un ejemplo de ejecución en donde Alberto dice que baila con María pero María dice que baila con otra persona.

```
sem_t mtx_damas;
sem_t llega_dama;
sem_t mtx_varones;
sem_t llega_varon;
char *nom_dama, *nom_varon;

char *nDama(char *nom) {
  sem_wait(&mtx_damas);
    nom_dama= nom;
    sem_post(&llega_dama);
    sem_wait(&llega_varon);
    char *pareja=nom_varon;
  sem_post(&mtx_damas);
  return pareja;
}
```

```
void nth_iniDisco() {
  sem_init(&mtx_damas,0,1);
  sem_init(&mtx_varones,0,1);
  sem_init(&llega_dama,0,0);
  sem_init(&llega_varon,0,0);
}
char *nVaron(char *nom) {
  sem_wait(&mtx_varones);
    nom_varon= nom;
    sem_post(&llega_varon);
    sem_wait(&llega_dama);
    char *pareja=nom_dama;
  sem_post(&mtx_varones);
  return pareja;
}
```

Solución:



Parte c.- (1 punto) Corrija esta implementación agregando un par de nuevos semáforos, por ejemplo *sale_dama* y *sale_varon*. **No necesita** armar las parejas por orden de llegada.

Solución:

```
...
sem_t sale_dama;
sem_t sale_varon;


char *nDama(char *nom) {
  ...
    // 0.6 puntos
    sem_post(&sale_dama);
    sem_wait(&sale_varon);
  sem_post(&mtx_damas);
  return pareja;
}
```

```
void nth_iniDisco() {
  ...
  // 0.2 puntos
  sem_init(&sale_dama,0,0);
  sem_init(&sale_varon,0,0);
}
char *nVaron(char *nom) {
  ...
    // 0.2 puntos
    sem_post(&sale_varon);
    sem_wait(&sale_dama);
  sem_post(&mtx_varones);
  return pareja;
}
```

Pregunta 2

I. (4,5 puntos) Los semáforos de pthreads no garantizan el otorgamiento de las fichas por orden de llegada. Defina el tipo *Sem* y programe las siguientes funciones que implementan semáforos que sí garantizan que las fichas se otorgan por orden de llegada:

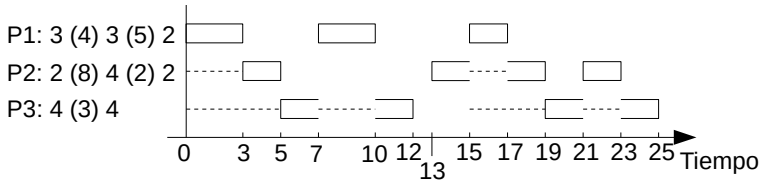
*void iniSem(Sem *s);*
*void waitSem(Sem *s);*
*void postSem(Sem *s);*

Su implementación debe basarse en los mutex y condiciones de pthreads. Debe usar el patrón *request* para evitar cambios de contexto inútiles y otorgar las fichas por orden de llegada.

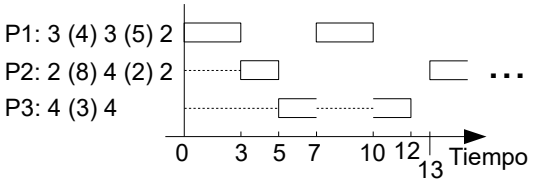
Solución:

```
// 0,3 puntos: definir struct
typedef struct {
  int cnt;
  pthread_mutex_t mtx;
  Queue *q;
} Sem;

// 0,5 puntos: definir Request
typedef struct {
  int  ready;
  pthread_cond_t cond;
} Request;

void semPost(Sem *s) {
  // 0,5 puntos: uso de lock y unlock
  lock(&s->mtx);
  // 0,3 puntos: caso cola vacía
  if (emptyQueue(s->q))
    s->cnt++;
  else {
    // 0,5 puntos:
    Request *pr=get(s->q);
    pr->ready= 1; // 0,2
    signal(&pr->cond);// 0,5
  }
  unlock(&s->mtx);
}
```

```
void semWait(Sem *s) {
  lock(&s->mtx);
  if (s->cnt>0) // 0,3: hay fichas
    s->cnt--;
  else {
    Request r= {0, // 0,4
      PTHREAD_COND_INITIALIZER};
    put(s->q,&r); // 0,5
    // 0,5: esperar
    while (r.ready==0)
      wait(&s->cond,&s->mtx);
  }
  unlock(&s->mtx);
}

void iniSem(Sem *s) {
  s->cnt= 0;
  pthread_mutex_init(
          &s->mtx, NULL);
  s->q= makeQueue();
}
```



II. (1,5 puntos) El diagrama de la derecha muestra el scheduling de 3 procesos. La estrategia de scheduling es en base a prioridades fijas y distintas. Junto a cada proceso se indica la duración de las ráfagas de CPU y entre paréntesis la duración de los estados de espera. Responda: **a.-** Explique si se trata de scheduling *preemptive* o *non-preemptive* y por qué; **b.-** Complete el diagrama.



Solución:

a.- En el instante 7 se aprecia que **el scheduler suspende la ejecución de la ráfaga de P3** para transferir la CPU a P1 que acaba de pasar a estado READY. Como esta ráfaga no se ejecuta de principio a fin, **el scheduling es *preemptive***. (0,7 puntos)

b.- (0,8 puntos)