# CC4302 – Sistemas Operativos
# Pauta Auxiliar 5

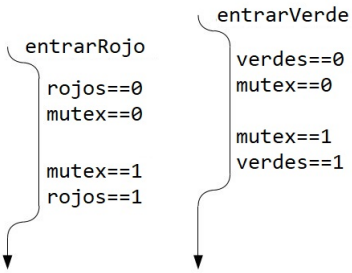
Profesor: Luis Mateu
Auxiliar: Diego Madariaga

22 de abril de 2020

## 1. P1 Control 1, 2013/2

### Parte 1

inicialmente: `rojos = verdes = mutex = 0`




Un hincha rojo entra al baño cuando ya había uno verde.

### Parte 2

Solución con tres semáforos y baño de capacidad ilimitada.

```
int rojos= 0, verdes= 0;
nSem mutex; /* = nMakeSem(1); */
nSem r; /* = nMakeSem(1); */
nSem v; /* = nMakeSem(1); */

void entraRojo() {
        nWaitSem(r);
        if (rojos == 0)
                nWaitSem(mutex);
        rojos++;
        nSignalSem(r);
}

void saleRojo() {
        nWaitSem(r);
        rojos--;
        if (rojos == 0)
                nSignalSem(mutex);
        nSignalSem(r);
}
```

```
void entraVerde() {
        nWaitSem(v);
        if (verdes == 0)
                nWaitSem(mutex);
        verdes++;
        nSignalSem(v);
}

void saleVerde() {
        nWaitSem(v);
        verdes--;
        if (verdes == 0)
                nSignalSem(mutex);
        nSignalSem(v);
}
```

## Parte 3

Solución con cuatro semáforos y baño de capacidad limitada a 4.

```
int rojos= 0, verdes= 0;
nSem mutex; /* = nMakeSem(1); */
nSem r; /* = nMakeSem(1); */
nSem v; /* = nMakeSem(1); */
nSem capacidad; /* nMakeSem(4) */

void entraRojo() {
        nWaitSem(r);
        if (rojos == 0)
                nWaitSem(mutex);
        rojos++;
        nSignalSem(r);
        nWaitSem(capacidad);
}

void saleRojo() {
        nWaitSem(r);
        rojos--;
        if (rojos == 0)
                nSignalSem(mutex);
        nSignalSem(r);
        nSignalSem(capacidad);
}

void entraVerde() {
        nWaitSem(v);
        if (verdes == 0)
                nWaitSem(mutex);
        verdes++;
        nSignalSem(v);
        nWaitSem(capacidad);
}

void saleVerde() {
        nWaitSem(v);
        verdes--;
        if (verdes == 0)
                nSignalSem(mutex);
        nSignalSem(v);
        nSignalSem(capacidad);
}
```

# 2. P1 Control 1, 2006/2

## Parte 1

- **1er. caso**: funciona correctamente. (1.25 puntos.)

  La función signal modifica sc y wait modifica wc. En el caso de sc en wait solo se consulta por su valor, no se modifica. Este es de los pocos casos (y en particular este ejemplo) donde cierta ejecución no provoca problemas al hacer solo lectura en sc por parte de wait.

- **2º caso**: no funciona en absoluto porque las variables son independientes. Las modificaciones que se hagan a sc en un proceso pesado, no seran vistas por el otro proceso. (0.5 puntos.)

  Si son procesos pesados, entonces cada proceso dispondrá de su propia área de datos dentro de la memoria principal, los procesos no podrán acceder a la memoria del otro proceso. Por lo anterior, cuando el P1 ejecute signal colocará un ticket, pero cuando P2 ejecute wait no verá el ticket puesto por P1 ya que para P2 sc seguirá siendo 0.

- **3er. caso**: No funciona porque varios threads que invocan signal pueden dejar inconsiste el valor de sc (o alternativamente, los que invocan wait, dejar inconsistente wc). El alumno debe hacer un diagrama de threads mostrando que dos threads que incrementan concurrentemente sc (o wc) pueden incrementarlo solo en 1. (1.25 puntos.)

  No se puede confiar en que la ejecución de sc++ o wc++ sean atómicas, al ser realizada la traducción a código de máquina se pueden obtener múltiples instrucciones que permitan que ocurran dataraces. Para que se entienda veamos la definición de signal de la siguiente forma:

```
void signal () {
        int aux = sc;
        sc = aux + 1;
}
```

  Lo anterior es para simular lo que podría pasar al ser traducido a código de máquina. Básicamente el problema es que dos threads pueden consultar por el valor de sc y guardarlo en aux, ambos ver 0 por ejemplo. Y luego ambos procesos actualizar el valor de sc con el valor que tienen en aux más uno. Como ambos threads vieron 0, el resultado final luego de la ejecución de 2 signal será sc con valor 1.

## Parte 2

### Recordatorio API mensajes

- `int nSend (nTask t, void *m)`: envío de mensaje $m$ a $t$. Espera hasta que se responda el mensaje con nReply.
- `void *nReceive (nTask *pt, int delay)`: espera a que se le envíe un mensaje y lo entrega (retorna). $*pt$ es la identificación del emisor, si $delay >= 0$ se espera como máximo $delay$ [ms], si se indica como $-1$ se esperará infinitamente.
- `int nReply (nTask t, int rc)`: responde el mensaje emitido por $t$, $t$ se desbloqueará retornando $rc$ (en la llamada de nSend).

### Implementación del semáforo de la parte 1 con mensajes

```
/* Declaracion de estructuras de datos: 0.3 puntos */
typedef enum /* posibles mensajes que se enviaran */
{
        SEMSIGNAL,
        SEMWAIT
```

```
} SemOp;
nTask semServer; /* = nEmitTask(semProc); en el nMain */

/* Envio de mensajes al servidor: 0.5 puntos */
void signal() {
        SemOp op = SEMSIGNAL;
        nSend(semServer, &op);
}
void wait() {
        SemOp op= SEMWAIT;
        nSend(semServer, &op);
}

/* Servidor utilizado para el semaforo */
int semProc() {
        int c = 0; /* contador de tickets */
        FifoQueue q = MakeFifoQueue(); /* cola para el semaforo */
        for(;;) { /* Ciclo y recepcion de mensajes: 0.5 */
                nTask t;
                SemOp *pop= (SemOp*) nReceive(&t, -1);
                if (*pop==SEMSIGNAL) { /* contabilizar los signal: 0.5 */
                        c++;
                        nReply (t, 0);
                }
                else /* wait: encolar: 0.5 */
                        /* aunque hayan tickets disponibles se encola, luego
                           se sacaran de la cola: TAREA: hacerlo sin encolar
                           a todos, o sea, si ya hay tickets no encolar */
                        PutObj(q, t);
                /* Desencolar a los que pueden continuar: 0.7 puntos */
                while (c>0 && !EmptyFifoQueue(q)) {
                        c--;
                        nReply((nTask)GetObj(q), 0);
                }
        }
}
```