# CC41B – CC4302 – Sistemas Operativos
# Auxiliar N°1

Profesor: Luis Mateu
Auxiliar: Juan Manuel Barrios

20 de Agosto 2010

Durante el curso usaremos nSystem, el cual permite crear threads dentro de un solo proceso Unix. No usaremos otras librerías (como *pthreads*) porque al estudiar el *scheduling* de tareas veremos como está hecho nSystem y le haremos algunas modificaciones.

Una vez que lo bajen desde la página del profesor pueden compilarlo, para esto lean el README que está incluido. En resumen hay que entrar al directorio *src* y escribir `make`. Si ya está compilado se puede recompilar con `make clean` y luego `make`[1].

El siguiente es un programa de ejemplo en nSystem:

```
#include "nSystem.h"

int escritor(int num, int espera);

int nMain() {
    nTask tareas[3];
    int i;
    for (i = 0; i < 3; ++i)
        tareas[i] = nEmitTask(escritor, i, i * 200);
    for (i = 0; i < 3; ++i)
        nWaitTask(tareas[i]);
    nPrintf("Fin ejemplo\n");
}

int escritor(int num, int espera) {
    int i = 5;
    while (i > 0) {
        nPrintf("thread %d: %d\n", num, i);
        nSleep(espera);
        i--;
    }
}
```

[1] Para compilar nSystem en un Linux de 64 bits se debe agregar la opción `-m32` a las variables `CFLAGS` y `LFLAGS` del archivo Makefile. Además, dependiendo de la distribución, requerirá tener instalada la librería `glibc-devel` o `libc6-dev`.

Con `nEmitTask` se ejecuta el método `escritor` en un thread independiente retornando un descriptor del thread iniciado. Con `nWaitTask` se bloquea la ejecución de un thread a la espera de que la tarea termine, y cuando la tarea finaliza el valor de retorno de `nWaitTask` corresponde al valor retornado por el método iniciado con `nEmitTask`.

Notar que el ejemplo no tiene *main* si no que el método principal es *nMain*, esto es porque el *main* lo proveerá nSystem quien primero creará el entorno para usar threads y luego ejecutará nuestro *nMain*. Por tanto para compilar este archivo no funcionará con `gcc ejemplo.c` (arrojará el mensaje *undefined reference to 'main'*). Notar que varias de las funciones comúnmente usadas tienen una versión para nSystem como `nSleep`, `nPrintf`, `nOpen`, `nClose`, `nFprintf`, `nMalloc`, etc. para que sea usada por cada thread sin interferir con los otros threads activos.

**Pregunta:** ¿Que sucede si en el ejemplo usamos `sleep` en vez de `nSleep`?[2]

Para compilar el código primero definamos la variable de ambiente `NSYSTEM` (en bash `export NSYSTEM=/home/.../nSystem` o en tcsh `setenv NSYSTEM /home/.../nSystem`) y luego:

- Opción 1: crear un archivo .o y luego incluirlo con nSystem para generar el ejecutable:

```
gcc -c ejemplo.c -I$NSYSTEM/include
gcc ejemplo.o $NSYSTEM/lib/libnSys.a -o ejemplo
```

- Opción 2: usar alguno de los Makefile que están en los ejemplos de nSystem[3]:

```
make APP=aux1p1
```

Si abrimos un Makefile veremos que se compone de reglas (escritas antes de un dos puntos), y para poder aplicar cada regla se deben cumplir primero sus condiciones (escritas después de los dos puntos). Cada condición puede corresponder a otras reglas del Makefile o a nombres de archivos en el mismo directorio.

Si una condición corresponde a una regla entonces se debe ejecutar ésta primero (la cual también debe cumplir sus condiciones), y si corresponde a un archivo éste debe existir y su fecha de modificación debe ser más reciente que sus dependencias (en caso contrario debe aplicar la regla correspondiente para recompilarlo).

Una vez que se cumplen todas las condiciones de una regla se ejecutan las instrucciones bajo ella, notar que éstas instrucciones *deben* estar precedidas de un tabulador y no de espacios en blanco. En el código fuente de esta auxiliar se incluye un Makefile de ejemplo.

[2]Respuesta: se bloqueará el proceso Unix completo, es decir el proceso que contiene a nSystem y todos los threads, en vez de detenerse sólo el thread que invoca `sleep`.

[3]Para Linux de 64 bits se debe agregar la opción `-m32` a las variables `CFLAGS` y `LFLAGS` en el Makefile.

## Buscar en un árbol

Dada la siguiente estructura de datos de un árbol binario:

```
typedef struct Nodo {
    int valor;
    struct Nodo* izq;
    struct Nodo* der;
} Nodo;
```

Implementar los siguientes métodos que buscan un número dentro del árbol (los valores en el árbol no tienen ningún orden específico):

- `int buscarSeq1(Nodo *node, int num)` Busca `num` en el árbol haciendo un recorrido en profundidad.
- `int buscarSeq2(Nodo *node, int num)` Busca `num` en el árbol haciendo un recorrido simultáneo en cada rama creando tantos threads como nodos en el árbol.
- `int buscarSeq3(Nodo *node, int num)` Como el anterior, pero con la optimización que la ejecución de todas las tareas termina cuando alguna encuentra el elemento buscado.

## Solución

El primer caso es una búsqueda en profundidad:

```
int buscarSeq1(Nodo *node, int num) {
    if (node == NULL)
        return FALSE;
    else if (node->valor == num)
        return TRUE;
    else
        return buscarSeq1(node->izq, num) || buscarSeq1(node->der, num);
}
```

El segundo caso es crear una tarea por cada nodo:

```
int buscarSeq2(Nodo *node, int num) {
    if (node == NULL) {
        return FALSE;
    } else if (node->valor == num) {
        return TRUE;
    } else {
        nTask task1 = nEmitTask(buscarSeq2, node->izq, num);
        nTask task2 = nEmitTask(buscarSeq2, node->der, num);
        int r1 = nWaitTask(task1);
        int r2 = nWaitTask(task2);
        return r1 || r2;
    }
}
```

Notar que es un error hacer `return nWaitTask(task1) || nWaitTask(task2)` porque si `nWaitTask(task1)` retorna verdadero no ejecutará la segunda parte del OR y por tanto podría no esperar por `task2` y podría finalizar todo el programa estando `task2` aún en ejecución.

El tercer caso es:

```
int FOUND = FALSE;
int buscarSeq3(Nodo *node, int num) {
    if (FOUND) {
        return TRUE;
    } else if (node == NULL) {
        return FALSE;
    } else if (node->valor == num) {
        FOUND = TRUE;
        return TRUE;
    } else {
        nTask task1 = nEmitTask(buscarSeq3, node->izq, num);
        nTask task2 = nEmitTask(buscarSeq3, node->der, num);
        int r1 = nWaitTask(task1);
        int r2 = nWaitTask(task2);
        return r1 || r2;
    }
}
```

Notar que en este caso `FOUND` es una variable única compartida por todos los threads que eventualmente puede ser escrita de forma simultánea por varios threads. Sin embargo en este caso no implica un problema porque todos estarán escribiendo el mismo valor `TRUE` y la asignación de una constante es una operación atómica. Notar que este es uno de los pocos casos en que es posible leer y escribir una variable global sin tener problemas, sin embargo **la gran mayoría de las veces es necesario usar algún mecanismo de sincronización para leer y modificar una variable global** como usar semáforos, monitores, etc. lo que será parte de la próxima clase auxiliar.

# Ejemplo de I/O

El siguiente es un programa de ejemplo, que lee desde la entrada estándar un número $n$, y luego escribe $n$ archivos en paralelo con una cantidad variable de líneas.

```
#include <stdio.h>
#include <stdlib.h>
#include "nSystem.h"

#define MIN_LINEAS 100000
#define MAX_LINEAS 150000

int escritor(int num);

int nMain() {
    nSetTimeSlice(1);
```

```
    nSetNonBlockingStdio();
    nPrintf("Cantidad de archivos? ");
    char c[4];
    nRead(0, c, 3);
    int cant = atoi(c);
    if (cant <= 0)
        return;
    srand(time(0));
    nTask tareas[cant];
    int i;
    for (i = 0; i < cant; ++i) {
        tareas[i] = nEmitTask(escritor, i);
    }
    for (i = 0; i < cant; ++i) {
        nWaitTask(tareas[i]);
    }
}

int escritor(int num) {
    char archiv[30];
    sprintf(archiv, "archivo%d.txt", num);
    int fd = nOpen(archiv, O_CREAT | O_WRONLY | O_TRUNC, 0644);
    nFprintf(fd, "Archivo %d\n\n", num);
    int lineas = MIN_LINEAS + (random() % (MAX_LINEAS - MIN_LINEAS));
    nPrintf("Escribiendo archivo %s de %i lineas\n", archiv, lineas);
    int i;
    for (i = 1; i <= lineas; ++i) {
        nFprintf(fd, "linea %i: %i\n", i, rand());
    }
    nClose(fd);
    nPrintf("Archivo %s finalizado\n", archiv);
}
```

Como recomendación final, todas las tareas del curso serán en C por lo que no estará de más tener a mano el libro *"El lenguaje de programación C"* de Kernighan y Ritchie para resolver dudas sobre detalles del lenguaje.