# CC4302 – Sistemas Operativos
# Pauta Auxiliar 7

Profesor: Luis Mateu
Auxiliar: Diego Madariaga

6 de mayo de 2020

## 1. Mensajería en nSystem

```
int nSend(nTask task, void *msg) {
        int rc;
        START_CRITICAL(); //deshabilita interrupciones
        nTask this_task= current_task;
        if (task->status==WAIT_SEND) {
                task->status= READY;
                /* En primer lugar en la cola */
                PushTask(ready_queue, task);
        }
        PutTask(task->send_queue, this_task);
        this_task->send.msg= msg;
        this_task->status= WAIT_REPLY;
        ResumeNextReadyTask(); //cambio de contexto
        rc= this_task->send.rc;
        END_CRITICAL();
        return rc;
}

void *nReceive(nTask *ptask, int timeout) {
        void *msg;
        nTask send_task;
        START_CRITICAL();
        nTask this_task= current_task;
        if (EmptyQueue(this_task->send_queue) && timeout != 0) {
                /* La tarea espera indefinidamente */
                this_task->status= WAIT_SEND;
                /* Se suspende indefinidamente hasta un nSend */
                ResumeNextReadyTask();
        }
        send_task= GetTask(this_task->send_queue);
        if (ptask!=NULL) *ptask= send_task;
        msg= send_task==NULL ? NULL : send_task->send.msg;
        END_CRITICAL();
        return msg;
}

void nReply(nTask task, int rc) {
        START_CRITICAL();
        PushTask(ready_queue, current_task);
        task->send.rc= rc;
        task->status= READY;
        PushTask(ready_queue, task);
        ResumeNextReadyTask();
        END_CRITICAL();
}
```

# 2. Mensajería en nSystem con timeout

Revisar la implementación en *nMsg.c*. En el ejercicio anterior se revisó la implementación sin timeouts. Ahora, para realizar una espera por cierta cantidad de tiempo determinada hay que revisar el método `nReceive`. Ahí se puede ver que para esperar por cierto tiempo se hace invocando al procedimiento: `ProgramTask(timeout)` el cual hace dormir a la tarea por la cantidad de milisegundos indicada.

Si durante la espera programada la tarea recibe un mensaje, esta despierta inmediatamente. Para ver como se implementa esto hay que revisar el procedimiento `nSend`, donde en caso que el receptor aún este en espera se invoca el procedimiento `CancelTask(task)`.

Los procedimientos `ProgramTask` y `CancelTask` están implementados en *nTime.c*. Las tareas con timeout se guardan en la cola de prioridades `wait_squeue` ordenados según el momento en el que deben despertar, y se programa una interrupción que se gatillará más tarde, donde se activarán las tareas que correspondan.

## 2.1. nMsg.c

```
#include "nSysimp.h"
#include "nSystem.h"

/*************************************************************
 * Epilogo
 *************************************************************/

static int pending_sends=0;
static int pending_receives=0;

void MsgEnd()
{
  if ( pending_sends!=0 || pending_receives!=0)
  {
    nFprintf(2, "\nNro. de tareas bloqueadas en un nSend: %d\n",
                        pending_sends);
    nFprintf(2, "Nro. de tareas bloqueadas en un nReceive: %d\n",
                        pending_receives);
} }

/*************************************************************
 * nSend, nReceive y nReply
 *************************************************************/

int nSend(nTask task, void *msg)
{
  int rc;

  START_CRITICAL();
  pending_sends++;
  { nTask this_task= current_task;

    if (task->status==WAIT_SEND || task->status==WAIT_SEND_TIMEOUT)
    {
      if (task->status==WAIT_SEND_TIMEOUT)
        CancelTask(task);
      task->status= READY;
      PushTask(ready_queue, task); /* En primer lugar en la cola */
    }
    else if (task->status==ZOMBIE)
      nFatalError("nSend", "El receptor es un ``zombie''\n");

    /* En nReply se coloca ``this_task'' en la cola de tareas ready */
    PutTask(task->send_queue, this_task);
```

```
    this_task->send.msg= msg;
    this_task->status= WAIT_REPLY;
    ResumeNextReadyTask();

    rc= this_task->send.rc;
  }
  pending_sends--;
  END_CRITICAL();

  return rc;
}

void *nReceive(nTask *ptask, int timeout)
{
  void *msg;
  nTask send_task;

  START_CRITICAL();
  pending_receives++;
  { nTask this_task= current_task;

    if (EmptyQueue(this_task->send_queue) && timeout!=0)
    {
      if (timeout>0)
      {
        this_task->status= WAIT_SEND_TIMEOUT;
        ProgramTask(timeout);
        /* La tarea se despertara automaticamente despues de timeout */
      }
      else this_task->status= WAIT_SEND; /* La tarea espera indefinidamente */

      ResumeNextReadyTask(); /* Se suspende indefinidamente hasta un nSend */
    }

    send_task= GetTask(this_task->send_queue);
    if (ptask!=NULL) *ptask= send_task;
    msg= send_task==NULL ? NULL : send_task->send.msg;
  }
  pending_receives--;
  END_CRITICAL();

  return msg;
}

void nReply(nTask task, int rc)
{
  START_CRITICAL();

    if (task->status!=WAIT_REPLY)
      nFatalError("nReply","Esta tarea no espera un ``nReply''\n");

    PushTask(ready_queue, current_task);

    task->send.rc= rc;
    task->status= READY;
    PushTask(ready_queue, task);

    ResumeNextReadyTask();

  END_CRITICAL();
}
```

## 2.2. nTime.c

```
#include "nSysimp.h"
```

```c
#include <nSystem.h>
#include <sys/time.h> /* Para gettimeofday */

/* Procedimientos locales del modulo: */

static void RtimerHandler();
static void AwakeTasks();

/* Variables del modulo */

static Squeue wait_squeue;
static int init_time=0;

/*************************************************************
 * Prologo y epilogo:
 *************************************************************/

void TimeInit()
{
  wait_squeue= MakeSqueue();
  init_time= nGetTime();
}

void TimeEnd()
{
  if (!EmptySqueue(wait_squeue))
    nFprintf(2, "\nQuedan tareas con timeout programados\n");
}

/*************************************************************
 * Manejo de timeouts
 *************************************************************/

int nGetTime()
{
    struct timeval Timeval;

    gettimeofday(&Timeval, NULL);
    return Timeval.tv_sec*1000+Timeval.tv_usec/1000-init_time;
}

void ProgramTask(int timeout)
{
  VerifyCritical("ProgramTask");
  if (timeout>0)
  {
    int curr_time= nGetTime();
    int wake_time= curr_time+timeout;

    if (EmptySqueue(wait_squeue) || wake_time-GetNextTimeSqueue(wait_squeue)<0)
      SetAlarm(REALTIMER, wake_time-curr_time, RtimerHandler);

    PutTaskSqueue(wait_squeue, current_task, wake_time);
  }
  else
  {
    current_task->status= READY;
    PushTask(ready_queue, current_task);
  }
}

void CancelTask(nTask task)
{
  VerifyCritical("CancelTask");
  DeleteTaskSqueue(wait_squeue, task);
```

```
  AwakeTasks();
}

static void AwakeTasks()
{
  int curr_time= nGetTime();

  /* Despertamos todas las tareas con wake_time<=curr_time */

  while (! EmptySqueue(wait_squeue) &&
         GetNextTimeSqueue(wait_squeue)<=curr_time)
  {
    nTask task= GetTaskSqueue(wait_squeue);
    /* Ahora la tarea que dormia vuelve a estar READY */
    task->status= READY;
    PushTask(ready_queue, task);
  }

  /* Preparamos la proxima interrupcion */
  SetAlarm(REALTIMER,
           EmptySqueue(wait_squeue) ?
             0 : GetNextTimeSqueue(wait_squeue)-curr_time,
           RtimerHandler);
}

static void RtimerHandler()
{
  PreemptTask();

    AwakeTasks();

  ResumePreemptive();
}

/*************************************************************
 * nSleep
 *************************************************************/

void nSleep(int delay)
{
  START_CRITICAL();

  current_task->status= WAIT_SLEEP;
  ProgramTask(delay);
  ResumeNextReadyTask();

  END_CRITICAL();
}
```