Auxiliar 6: Estrategias de Scheduling e Implementación de Mensajes en nSystem

CC4302 - Sistemas Operativos

José Astorga

Estrategias de Scheduling

Conceptos Importantes

- Procesos livianos (threads) vs procesos pesados (procesos Unix)
- Preemptiveness
- Interrupciones y el timer
- Estados de un proceso
- El descriptor de proceso
- Cambio de contexto
- Colas de scheduling
- Cambios de contexto implícito vs. cambio de contexto explícito (cuando el cambio de contexto ocurre producto de una acción del mismo proceso, y no producto de una interrupción del timer o del disco)
- El identificador de proceso (pid)
- Ráfagas de CPU

Estrategias de Scheduling:

- First Come First Served
- Shortest Job First
- Prioridades
- Round Robin

Pueden ser de tipo preemptive o non-preemptive.

Estrategias de Scheduling:

Estrategia	Pros	Cons
First Come First Served		
Shortest Job First		
Prioridades		
Round Robin		

Estrategias de Scheduling:

Estrategia	Pros	Cons
First Come First Served	Simpleza	Mal tiempo de despacho promedio. No sirve para sistemas interactivos
Shortest Job First		
Prioridades		
Round Robin		

Estrategias de Scheduling:

Estrategia	Pros	Cons
First Come First Served	Simpleza	Mal tiempo de despacho promedio. No sirve para sistemas interactivos
Shortest Job First	Menor tiempo de despacho promedio	Hambruna para los procesos intensivos en CPU
Prioridades		
Round Robin		

Estrategias de Scheduling:

Estrategia	Pros	Cons
First Come First Served	Simpleza	Mal tiempo de despacho promedio. No sirve para sistemas interactivos
Shortest Job First	Menor tiempo de despacho promedio	Hambruna para los procesos intensivos en CPU
Prioridades	Se atienden procesos “importantes” primero.	Hambruna. (Se puede solucionar aumentando prioridad de procesos en estado READY)
Round Robin		

Estrategias de Scheduling:

Estrategia	Pros	Cons
First Come First Served	Simpleza	Mal tiempo de despacho promedio. No sirve para sistemas interactivos
Shortest Job First	Menor tiempo de despacho promedio	Hambruna para los procesos intensivos en CPU
Prioridades	Se atienden procesos “importantes” primero.	Hambruna. (Se puede solucionar aumentando prioridad de procesos en estado READY)
Round Robin	Minimiza el tiempo de respuesta	Muchos cambios de contextos implícitos (se soluciona fijando la el tamaño de la tajada de manera adecuada)

Diagrama de scheduling

El diagrama muestra las decisiones de scheduling para 3 procesos. A la izquierda se indica para cada proceso las duraciones de sus ráfagas de CPU y entre paréntesis las duraciones de sus estados de espera.

La línea punteada indica que el estado del proceso es READY. Si el proceso está en estado de espera el espacio aparece en blanco.

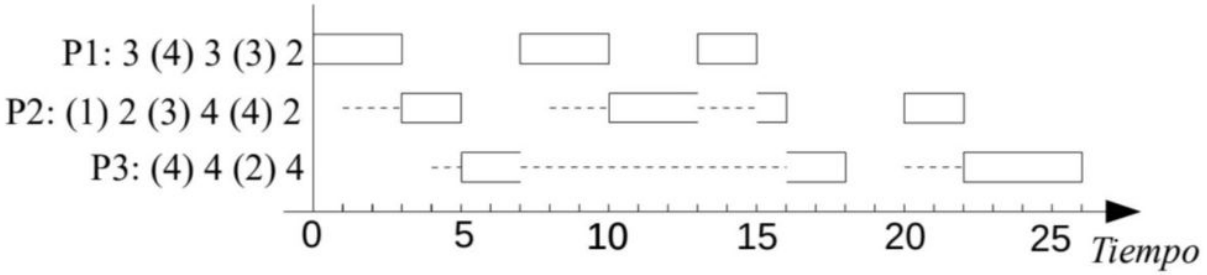


Diagrama de scheduling

¿De qué estrategia de scheduling se trata?

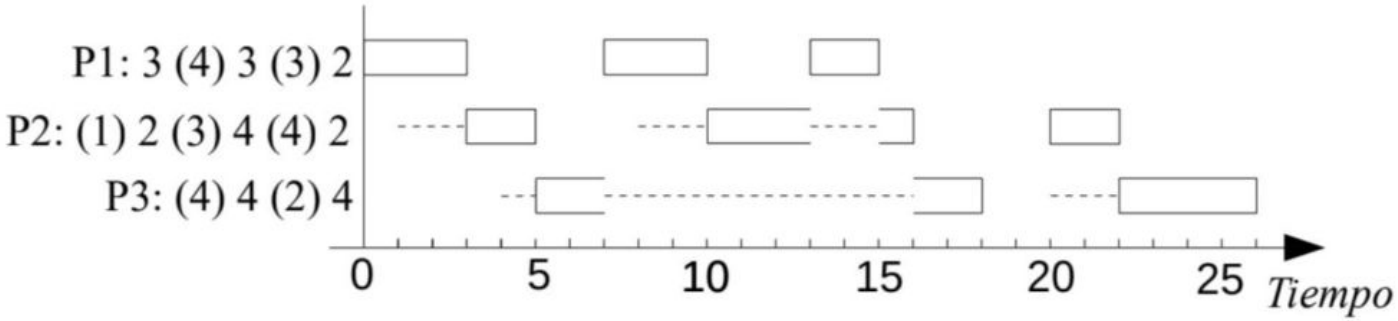
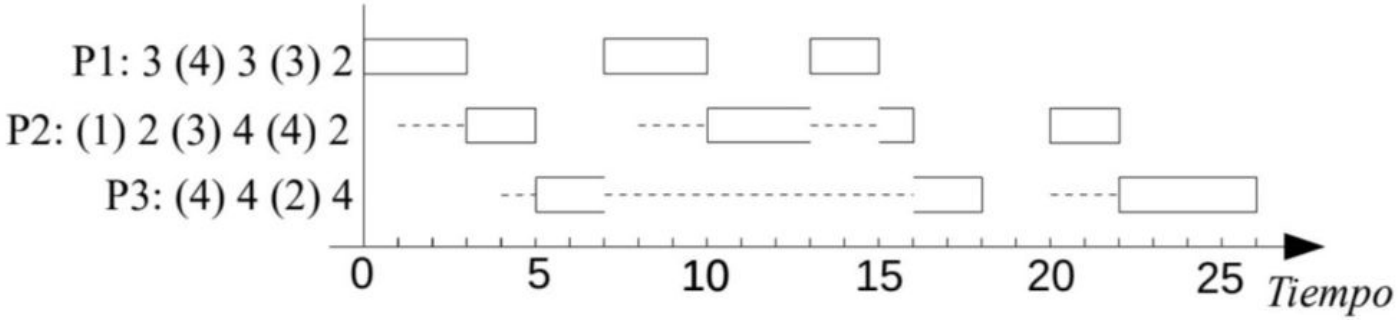


Diagrama de scheduling

¿De qué estrategia de scheduling se trata?



- Preemptive
- Estrategia de prioridades

Diagrama de scheduling

Reescribamos este diagrama considerando una estrategia FCFS :

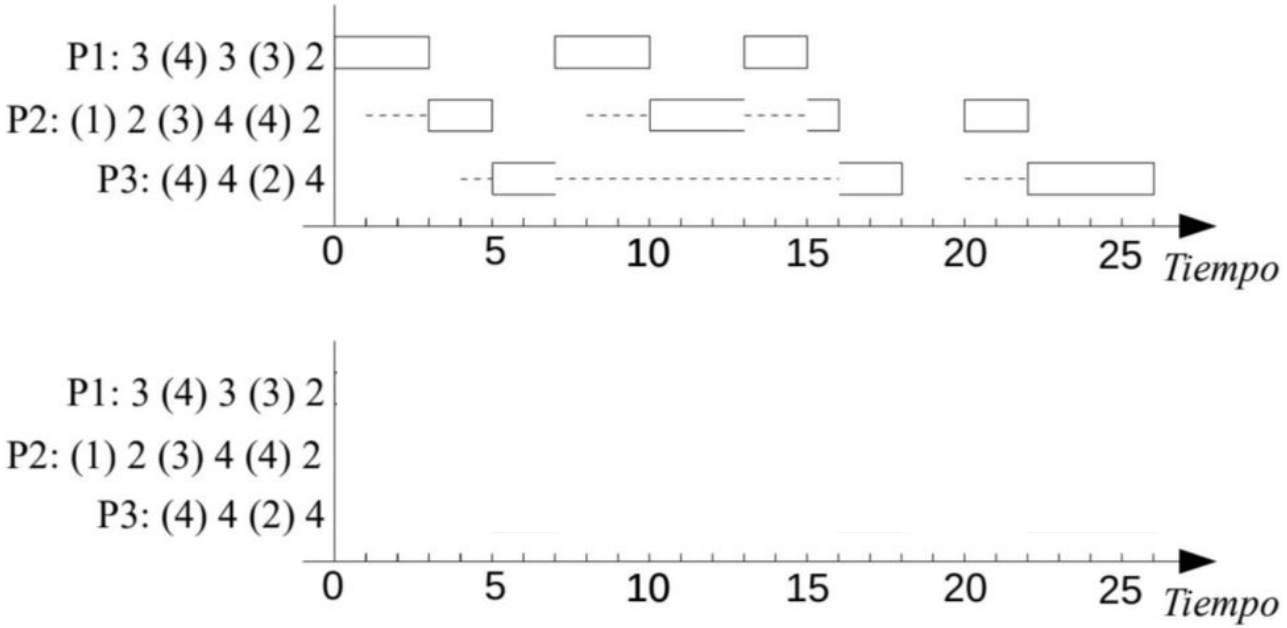
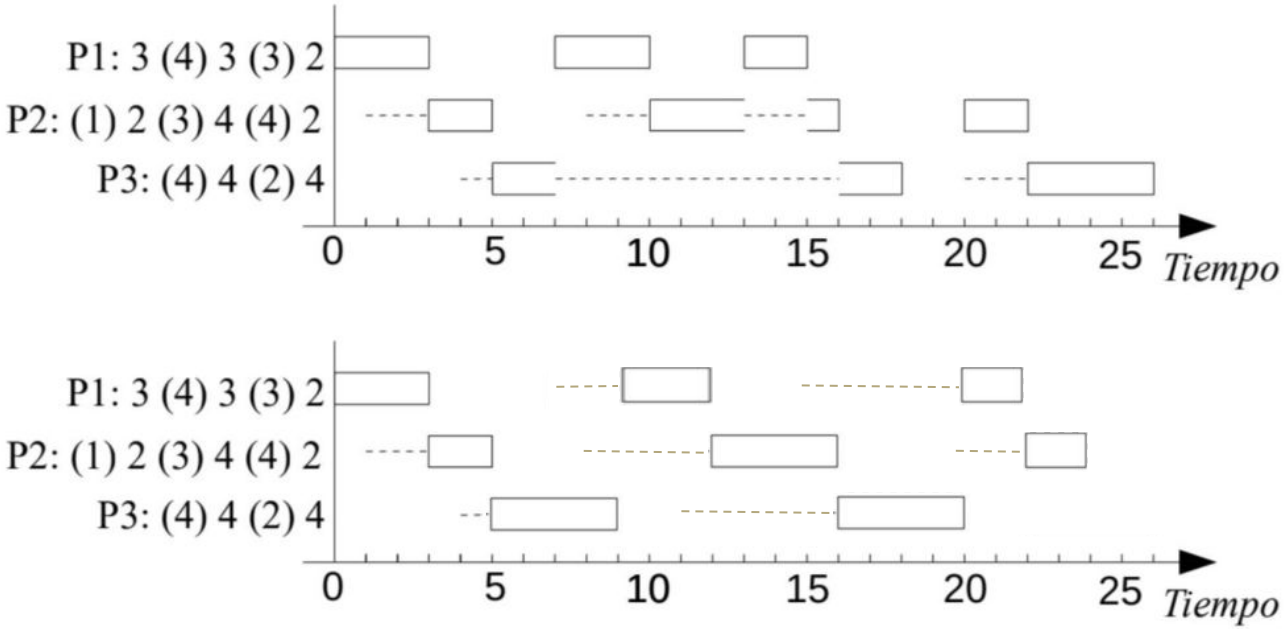


Diagrama de scheduling

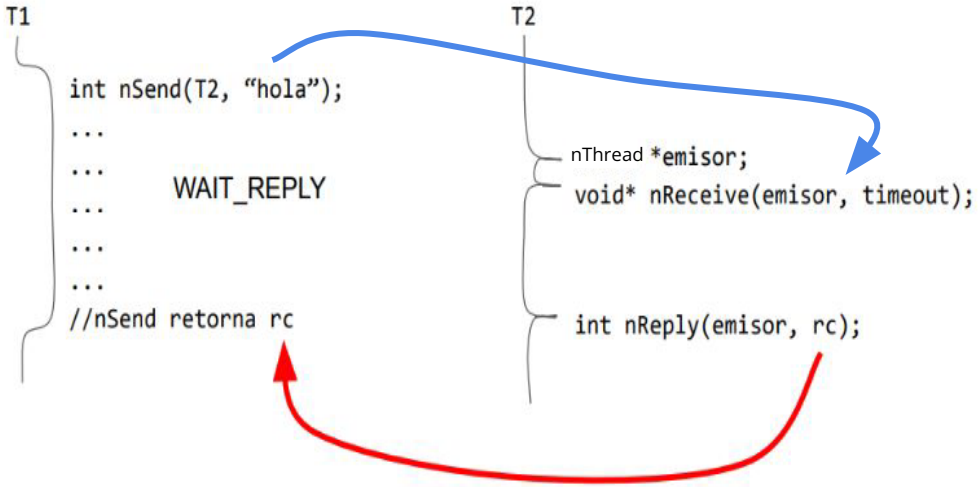
Reescribamos este diagrama considerando una estrategia FCFS :



Herramientas de Sincronización: Implementación de Mensajes

Herramientas de Sincronización

- A continuación vamos a programar dentro del Sistema Operativo.
- Mensajes es un herramienta de sincronización que se utiliza para que threads compartan información entre ellos. Es una alternativa a semáforos y mutex con condiciones.



Mensajes

```
int nSend(nThread th, void *msg)
```

Envía un mensaje msg al thread th.
Suspende al thread hasta que se reciba una respuesta desde th.

Retorna el valor recibido.

```
int nReply(nThread th, int rc)
```

Envía una respuesta con rc a un mensaje recibido de th.

No suspended el thread.

```
void* nReceive(nThread *pth, int timeout_ms)
```

Suspende al thread hasta recibir un mensaje. Registra al thread desde donde viene el mensaje en el puntero *pth.

Retorna el mensaje recibido. La suspensión dura timeout_ms, si no se recibe un mensaje retorna de todas formas, si este valor es 0 entonces no se suspende.

Si es <0 la suspensión es infinita hasta recibir un mensaje.

Mensajes: Cómo funcionan

```
int nSend(nThread th, void *msg)
```

Envía un mensaje msg al thread th.
Suspende al thread hasta que se reciba una respuesta desde th.

Retorna el valor recibido.

```
int nReply(nThread th, int rc)
```

Envía una respuesta con rc a un mensaje recibido de th.

No suspended el thread.

```
void* nReceive(nThread *pth, int timeout_ms)
```

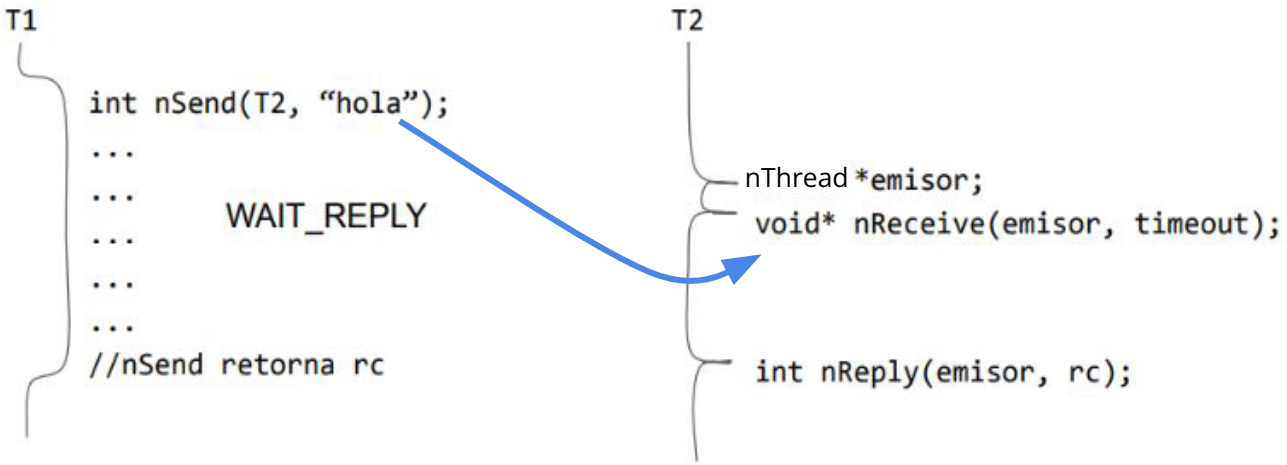
Suspende al thread hasta recibir un mensaje. Registra al thread desde donde viene el mensaje en el puntero *pth.

Retorna el mensaje recibido. La suspensión dura timeout_ms, si no se recibe un mensaje retorna de todas formas, si este valor es 0 entonces no se suspende.

Si es <0 la suspensión es infinita hasta recibir un mensaje.

⚠️ Por ahora ignoraremos los timeout ⚠️

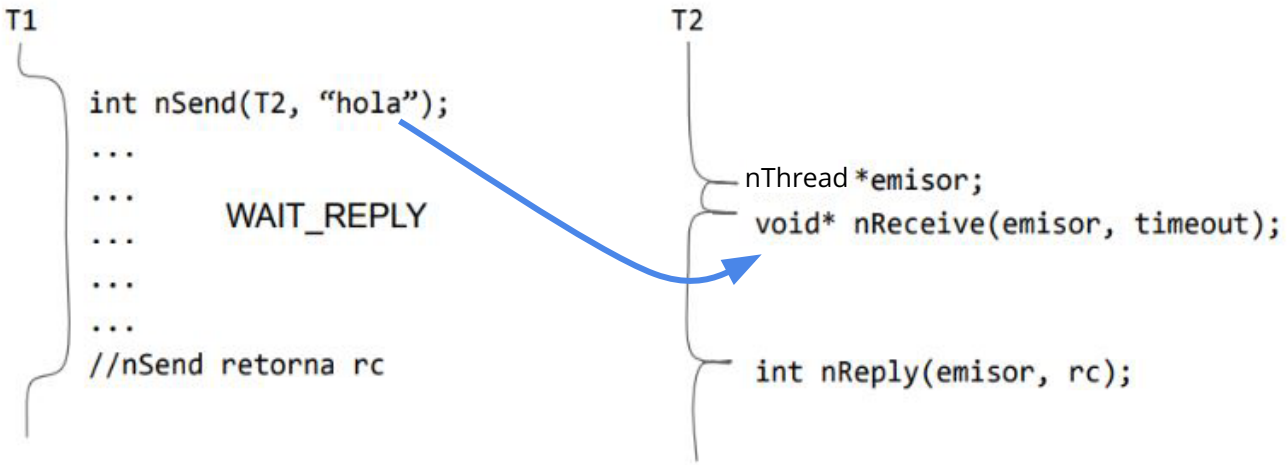
Mensajes: Cómo funcionan



T1 utiliza nSend para enviar un mensaje a T2.

nSend no retornará hasta que T2 invoque nReply.

Mensajes: Cómo funcionan

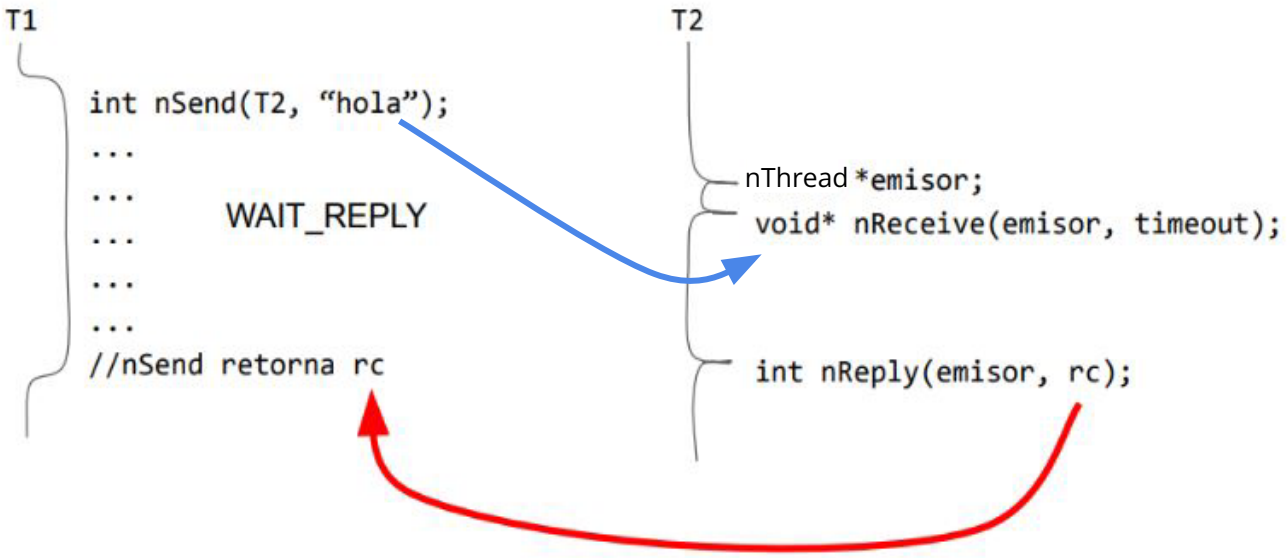


T2 invoca nReceive para esperar un mensaje.

nReceive no retornará hasta recibir un mensaje o hasta que se venza el tiempo *timeout.*

nReceive retorna el mensaje recibido y escribirá en *emisor* el descriptor del thread que envió el mensaje.

Mensajes: Cómo funcionan



T2 invoca nReply para responder a T1.

T1 finalmente retorna el valor rc que le fue entregado a través de nReply.

Impresora Compartida: Comparación

Patrón Request

```
void obtenerImpresora (){
 pthread_mutex_lock (&m);
 Request req = {FALSE, PTHREAD_COND_INITIALIZER};
 put(q, &req);
 pthread_cond_signal (&obtener);
 while (!req.ready) {
   pthread_cond_wait (&req.w, &m);
 }
 pthread_mutex_unlock (&m);
}

void devolverImpresora (){
 pthread_mutex_lock (&m);
 ocupada = FALSE;
 pthread_cond_signal (&devolver);
 pthread_mutex_unlock (&m);
}
```

Mensajes

```
enum Mensaje {OBTENER, DEVOLVER};

void obtenerImpresora (){
  int msg = OBTENER;
  // Bloquear thread hasta recibir respuesta
  nSend(impresora, &msg);
}


void devolverImpresora (){
  int msg = DEVOLVER;
  nSend(impresora, &msg);
}
```

Impresora Compartida: Comparación

Patrón Request

```
void ImpresoraServer()
{
 while (TRUE)
 {
   pthread_mutex_lock(&m);
   if (emptyQueue(q))
   {
     modoBajoConsumo();
     while (emptyQueue(q))
     {
       pthread_cond_wait(&obtener, &m);
     }
     modoUsoNormal();
   }
```

```
if (!emptyQueue(q))
   {
     Request *req = get(q);
     req->ready = TRUE;
     ocupada = TRUE;
     pthread_cond_signal(&req->w);
   }
   while (ocupada)
   {
     pthread_cond_wait(&devolver, &m);
   }
   pthread_mutex_unlock(&m);
 }
}
```

Impresora Compartida: Comparación

Mensajes

```
int impresoraServer(void *_ignored) {
 Queue *q = makeQueue();
 int ocupado = FALSE;
 nThread t;
 int *msg;
 while (TRUE) {
   // Si no esta ocupada, esperar
   if (!ocupado) {
     modoBajoConsumo();
     // esperar de manera indefinida
     msg = (int *)nRecieve(&t, -1);
     modoUsoNormal();
   }
   else { // Esperar a que se desocupe impresora
     msg = (int *)nReceive(&t, -1);
   }
```

```
   if (*msg == OBTENER) {
     // Encolar si esta ocupada,
     if (ocupado)
       put(q, t);
     else { // responder si esta disponible
       ocupado = TRUE;
       nReply(t, 0);
     }
   }
   else if (*msg == DEVOLVER) {
     nReply(t, 0); // "ok, la devolviste"
     if (EmptyFifoQueue(q)) {
       ocupado = FALSE;
     }
     else {
       // Entregar impresora a siguiente thread
       nThread *t2 = (nThread *)get(q);
       nReply(t2, 0);
     }
    }
  }
}
```

Implementar Mensajes

- Debemos implementar en nSystem:
  - `int nSend(nThread th, void *msg)`
  - `int nReply(nThread th, int rc)`
  - `void* nReceive(nThread *pth, int timeout_ms)`

- Para eso debemos modificar también el descriptor del thread para añadir los campos que necesitemos y los nuevos posibles estados, en el archivo nthread-impl.h