Apuntes de Ejercicios Prácticos (Sistemas Concurrentes y Distribuidos)

En estos apuntes se recogen **aspectos fundamentales**, **patrones**, **ejemplos** y **pseudocódigo** para abordar ejercicios como el **27**, **37** o similares del **Tema 4** (o de ejercicios de sincronización y tiempo real). El objetivo es tener un **guion mental** de cómo enfrentarse a cada tipo de problema, recordando que muchas de las ideas se repiten y se pueden adaptar.

---

1. Ejemplo de Sincronización con Recursos (Problema 27)

> **Enunciado resumido**
> - Hay *n* procesos `P1, P2, …, Pn` que **comparten** un único recurso `R`.
> - Solo puede usar `R` un proceso cada vez.
> - Si varios procesos esperan, el **proceso de mayor prioridad** se sirve primero.
> - Las prioridades están fijadas: proceso `Pi` tiene prioridad `i`. (El número más bajo indica más prioridad).

1.1. Ideas clave

1. **Monitor o Semáforos**:
   - Si usamos **monitores**, la cola asociada al recurso puede ser de **prioridad** (variable condición con prioridad).
   - Si usamos **semáforos**, debemos simular manualmente las prioridades.

2. **Estructura mínima** de sincronización:
   - **Pedir** recurso → se bloquea si está en uso.
   - **Liberar** recurso → se libera y se despierta quien corresponda.

3. **Variables fundamentales**:
   - `ocupado`: boolean (`true` si `R` está en uso).
   - `cola` con prioridad, p.e. `cond.wait(p)` donde `p = i` (la prioridad).

1.2. Código ilustrativo con un **monitor** (semántica SU simplificada)

```pseudo
monitor Recurso {
  var ocupado : boolean := false
  var recurso : condition  // cola con prioridad

  procedure Pedir(i : integer)  // i es la prioridad
  begin
    if (ocupado) then
      recurso.wait(i)   // se bloquea con prioridad i
    ocupado := true
  end

  procedure Liberar()
  begin
    ocupado := false
```

```
    recurso.signal()    // despierta al de mayor prioridad esperando
  end
}
```

Cómo se usa:

```
process Pi
begin
  while (true) do
  begin
    Recurso.Pedir(i)
    // --- usar R ---
    Recurso.Liberar()
    // --- resto del código ---
  end
end
```

> **Patrón**: el **protocolo de entrada** (Pedir) bloquea al proceso si `R` está ocupado. El **protocolo de salida** (Liberar) despierta al siguiente proceso con mayor prioridad.

---

2. Ejemplo de Paso de Mensajes (Problema 37)

> **Enunciado resumido**
>
> - Tribu de antropófagos (salvajes) + cocinero, comparten una olla de capacidad `M` misioneros.
> - **Salvajes**:
>   - Si la olla no está vacía → comen directamente.
>   - Si está vacía → despiertan al cocinero (bloquean al salvaje hasta que se rellene).
> - **Cocinero**: se "duerme" hasta que es despertado porque la olla está vacía, rellena la olla con `M` de nuevo.

2.1. Versión con **paso de mensajes** (asíncrono / síncrono)

Imaginemos un proceso central `Olla` que:

- Recibe **peticiones** de salvajes para comer (`peticion`).
- Envía al cocinero la señal de "rellenar" cuando se queda `contador = 0`.
- Recibe de cocinero un "ok" cuando la olla está lista otra vez.

Procesos:

2.1.1. Salvaje `i`

```
process Salvaje[i]
var peticion : integer := 1 // un valor simbólico
begin
  while true do
```

```
  begin
    // s_send(peticion, Olla) si es paso síncrono
    s_send(peticion, Olla)
    // comer (consumir 1 misionero)
    Comer()
  end
end
```

2.1.2. Cocinero

```
process Cocinero
var dummy : integer
begin
  while true do
  begin
    // recibir la “orden de rellenar”
    receive(dummy, Olla)
    // rellena la olla
    // manda confirmación a la olla
    send(dummy, Olla)
  end
end
```

2.1.3. Proceso central `Olla`

- `contador` indica cuántos misioneros quedan.
- Si `contador > 0`, aceptamos la petición del salvaje y decrementamos.
- Si `contador == 0`, enviamos una señal al cocinero, esperamos su confirmación, y recargamos.

```
process Olla
var contador : integer := M
    peticion, dummy : integer

begin
  while true do
    select
      // rama 1: hay comida
      for i in Salvajes when (contador > 0) receive(peticion, Salvaje[i]) do
          contador := contador - 1

      // rama 2: si la olla está vacía
      when (contador == 0) do
          send(dummy, Cocinero)
          receive(dummy, Cocinero)
          contador := M
    end select
end
```

Patrón:

- Espera selectiva que combina "aceptar salvajes" si `contador>0` con la acción de "llamar y esperar al cocinero" cuando `contador==0`.
- Evita el interbloqueo (no hay semáforos mal gestionados, sino mensaje + control de `contador`).

---

3. Ejemplo de Tema 4 (Sistemas de Tiempo Real)

> **Se pueden enfrentar problemas de "planificación cíclica", "RMS" (Rate Monotonic Scheduling) o "EDF" (Earliest Deadline First)**.
> **Objetivo**: Dado un conjunto de tareas periódicas con `(Ci, Ti, Di)`, verificar la planificabilidad y construir cronogramas.

3.1. Planificación Cíclica

Idea: Se define un "macro-ciclo" (hiperperiodo) = `mcm(T1, T2, …)`. Dentro de él se divide en "marcos" (slots) de longitud `TS`.

- Restricciones típicas:
  1. `TS >= max(Ci)`.
  2. `TS <= min(Ti)`.
  3. `TS` divide el hiperperiodo.

Ejemplo (Problema 48 o 49):

- T1 con (C1=2, T1=6)
- T2 con (C2=2, T2=8)
- T3 con (C3=3, T3=12)
- Se busca `TS`, se diseña la **tabla** de en qué marco corre cada tarea.

3.2. Planificación con RMS

RMS = Rate Monotonic Scheduling

- **Prioridad** ∝ **1/Ti** (periodo menor → prioridad mayor).
- **Test** de factor de utilización: [U = \sum \frac{C_i}{T_i} \quad\text{y}\quad U \le U_0(n) = n \left(2^{\frac{1}{n}} - 1\right).]
- **No** es "condición necesaria" (puede ser planificable aunque no cumpla `U <= U0(n)`, hace falta análisis más fino).

3.3. Planificación con EDF

Earliest Deadline First:

- Cada vez se elige la tarea con plazo más próximo.
- **Test**: si (\sum C_i/T_i \le 1), entonces es planificable. (Ese test **sí** es condición suficiente y necesaria cuando no hay sobrecargas locales).

3.4. Ejemplo rápido

Tareas:

- T1: C=1, T=5
- T2: C=1, T=10
- T3: C=2, T=10
- T4: C=10, T=20
- T5: C=7, T=100

Se verifica (U = 1/5 + 1/10 + 2/10 + 10/20 + 7/100 = 0.97).

- Con **RMS** no pasa el test de factor de utilización (en n=5, `U0(5)≈0.7437 < 0.97`).
- Con **EDF**, si `U=0.97 < 1.0`, **sí** se garantiza planificable.

Para dibujar el **cronograma**:

1. Con RMS, se asignan prioridades: T1 > T2 > T3 > T4 > T5.
2. Con EDF, en cada instante, vemos cuál es la tarea con deadline más próximo.

Conclusión: la tarea con menor periodo o menor deadline se prioriza. Si la simulación en [0, hiperperiodo] no falla (no hay "deadline miss"), es planificable.

---

4. Resumen de Patrones y Consejos

1. **Monitores** y **Semáforos**:
    - Usa "**procedure pedir**" y "**procedure liberar**" para encapsular la sección crítica.
    - Para **prioridades**, recurre a `cond.wait(prio)` si el lenguaje/entorno lo permite.
2. **Paso de mensajes**:
    - "select" con guardas → muy útil para mezclar varias condiciones de recibir.
    - Alterna "send/receive" con la lógica de contadores, colas, etc.
    - Cuidado con interbloqueos síncronos: a veces hay que cambiar el orden `receive → send` para no quedarse todos esperando.
3. **Tiempo Real**:
    - Conjunto de tareas periódicas con `(Ci, Ti, Di)`.
    - **Planificación cíclica**: define slots, verifica `TS >= max(Ci)`, etc.
    - **RMS**: prioridades fijas por periodo. Factor de utilización `U ≤ U0(n)`.
    - **EDF**: el más efectivo si `U ≤ 1`.
    - Si un test no se cumple, **no** significa automáticamente "no planificable"; en RMS requiere análisis adicional.
4. **Visual**:
    - **Cronograma (Gantt)**: dibuja ejes de tiempo, rellena huecos con la tarea prioritaria disponible.
    - **Tablas**: si es un "ejecutivo cíclico", divide hiperperiodo en marcos y rellena con "T1, T2, ...".

---

5. Conclusión

- **Problema 27** (exclusión mutua con prioridades) → Monitor con colas de prioridad.
- **Problema 37** (olla antropófagos con paso de mensajes) → Proceso `Olla` + `select`.
- **Tema 4** (tiempo real) → Tres principales algoritmos: cíclico, RMS, EDF. Saber test de planificabilidad y dibujar cronogramas.

Al final, cada "familia de ejercicios" se resuelve siguiendo **los mismos patrones**.

1. **Semáforos/monitores** para sincronización local.
2. **Paso de mensajes** (send/receive/select) cuando hay varios procesos y un "proceso central" que coordina.
3. **Planificación** (ejecutivo cíclico / RMS / EDF) con tests y cronogramas en **Sistemas de Tiempo Real**.

¡Con esto, tendrás un **mapa mental** y ejemplos para repasar los **códigos típicos** y **evitar confusiones**!