Programació de serveis i processos

TEMA 01 – PART III

Programació de processos i fils

1. Programació multiprocés i paral·lela
   1.4. Sincronització i comunicació entre processos

Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

IES DR. LLUÍS SIMARRO

1. Programació multiprocés i paral·lela

1.4 Sincronització i comunicació entre processos

Comunicar-se i **sincronitzar**-se són les dues accions més importants en l'execució de **processos concurrents**. Quan diversos processos estan en execució a la vegada no podem controlar l'ordre en què acabaran executant-se, ja que això dependrà de la política de planificació escollida i dels processos que es troben en execució a cada moment. Però l'**ordre de processament**, en algunes ocasions, pot arribar a ser fonamental per aconseguir un **resultat correcte**. Les tècniques que ens ajuden a controlar l'ordre d'execució dels diferents processos s'anomenen tècniques de comunicació i sincronització.

1.4.1. Competència de recursos i sincronització

Els **processos concurrents** es poden classificar en **independents** o **cooperants** segons el seu grau de col·laboració.

- Un procés **independent no necessita ajuda** ni cooperació d'altres processos.
- Els processos **cooperants** estan dissenyats per **treballar conjuntament amb altres processos** i, per tant, s'han de poder comunicar i interactuar entre ells.

Aquestes col·laboracions impliquen una **coordinació** de tots el **processos** implicats. A vegades la coordinació requereix de cert nivell de **comunicació** per poder-se **sincronitzar**. Tot i així, una comunicació detallada és **molt costosa** i sovint n'hi ha prou a assegurar només que els **processos no accedisquen a la vegada a un mateix recurs** o **executen una mateixa operació**. En aquest darrer cas direm que els **processos competeixen entre si** perquè no hi ha manera de saber l'ordre exacte amb què finalment s'executaran els processos.

En la **programació concurrent** el procés de sincronització permet que els **processos** que s'executen de forma simultània es **coordinen**, parant l'execució d'aquells que vagen més avançats fins que es complisquen les condicions òptimes per estar segurs

que els **resultats** finals seran **correctes**.

Per sincronitzar processos podem utilitzar diferents mètodes:

- **Sincronisme condicional**: o condició de sincronització. Un **procés** o fil es troba en estat d'execució i **passa a estat de bloqueig esperant** que una certa **condició** es complisca per continuar la seua execució. Un **altre procés** fa que aquesta **condició** es **complisca** i així el primer procés passa de nou a l'estat d'execució.

- **Exclusió mútua**: es produeix quan dos o més **processos** o fils volen **accedir** a un **recurs compartit**. S'han d'establir protocols d'execució perquè **no accedisquen** de **forma concurrent** al recurs.

- **Comunicació per missatges**: en la qual els **processos** s'intercanvien **missatges** per **comunicar**-se i **sincronitzar**-se. És la comunicació típica en sistemes distribuïts i és utilitzada també en sistemes no distribuïts.

1.4.2. Sincronització i comunicació

La **col·laboració** entre **processos** dóna lloc a un seguit de **problemes clàssics** de **comunicació** i **sincronització**. Sincronitzar i comunicar processos és, per tant, bàsic per tal de solucionar-los.

Seccions crítiques

Les seccions crítiques són un dels problemes que amb més freqüència es donen en programació concurrent. Tenim diversos **processos** que s'executen de forma **concurrent** i cadascun d'ells té una **part** de **codi** que s'ha d'**executar de forma exclusiva** ja que **accedeix** a **recursos compartits** com ara fitxers, variables comunes, registres de bases de dades, etc.

La solució passarà per obligar a **accedir als recursos** a través de l'execució d'un

codi que anomenarem **secció crítica** i que ens permetrà **protegir** aquells **recursos** amb mecanismes que **impedisquen** l'execució simultània de **dos** o més **processos** dins els límits de la **secció crítica**.

Aquests algoritmes de sincronització que eviten l'accés a una regió crítica per més d'un fil o procés i que garanteixen que **únicament un procés** estarà **utilitzant** d'aquest recurs i la **resta** que volen utilitzar-lo estaran **a l'espera** que siga alliberat, s'anomenen **algoritmes d'exclusió mútua**.

> ■ **Exclusió mútua** (**MUTEX**, mutual exclusion en anglès) és el tipus de sincronització que **impedeix** que dos **processos executen simultàniament** una **mateixa secció crítica**.

Un mecanisme de sincronització en forma de codi que protegisca la secció crítica haurà de tenir una forma com el següent:

```
Entrada_Secció_Critica /* Sol·licitud per executar Secció Crítica */

/* codi Secció Crítica */

Eixida_Secció_Critica /* altre procés pot executar la secció Crítica */
```

L'`Entrada_Secció_Crítica` representa la part del codi en la qual els **processos demanen permís** per entrar a la secció crítica. L'`Eixida_Secció_Critica` en canvi, representa la part que executen els processos quan ixen de la secció crítica **alliberant** la **secció** i **permetent** a d'**altres processos entrar**-hi.

Per validar qualsevol mecanisme de **sincronització** d'una **secció crítica** s'han de complir els **criteris** següents:

- **Exclusió mútua**: no pot haver-hi més d'un procés simultàniament en la secció crítica.

- **No inanició**: un procés no pot esperar un temps indefinit per entrar a executar la secció crítica.
- **No interbloqueig**: cap procés de fora de la secció crítica pot impedir que un altre procés entre a la secció crítica.
- **Independència del maquinari**: inicialment no s'han de fer suposicions respecte al número de processadors o la velocitat dels processos.

Un **error** de **consistència** típic quan no hi ha control sobre una secció crítica es pot il·lustrar amb l'**exemple** de dos processos que volen modificar una variable comuna x. El procés A vol incrementar-la: x++. El procés B disminuir-la: x–;. Si els dos processos accedeixen a llegir el contingut de la variable al mateix temps, els dos obtindran el mateix valor, si fan la seua operació i guarden el resultat, aquest serà inesperat. Dependrà de qui salve el valor d'x en últim lloc.

La taula següent mostra un exemple similar. Un codi és accessible per dos fils o processos, veiem que si **no hi ha cap tipus de control sobre l'accés**, el primer fil accedeix a les instruccions i **abans d'arriba**r a la instrucció d'**increment** de la variable a++4, que inicialment val 4, **el segon procés entra** a executar el mateix codi. El resultat és que el segon procés pren encara el valor inicial d'a, per tant erroni, ja que el primer procés no hauria augmentat la variable.

Procés 1	Temps	Procés 2	Valor variable compartida a	Eixida al fitxer
			4	
pout.print(a);	**1**		4	"4"
	2	pout.print(a);	4	"4"
a=a+4;	**3**		8	
	4	a=a+4;	**12**	
pout.print("+4=");	**5**		12	"+4="
pout.println(a);	**6**		12	"12"
	7	pout.print("+4=");	12	"+4="
	8	pout.println(a);	12	"12"

En aquest exemple suposarem què pot representar l'eixida a un **fitxer**, que cada procés arriba a la zona crítica i que cada procés treballa amb un fitxer diferent. Suposarem també que la variable a té un valor inicial de 4 i que es tracta d'una **variable compartida** per ambdós processos. Podeu imaginar que les eixides seran prou sorprenents, ja que en ambdós fitxers indicaria: 4+4=12.

Per evitar aquest problema, **únicament un fil** hauria d'executar aquesta part de codi de forma **simultània**. Aquesta part de codi, que és susceptible a aquest tipus d'error, s'ha de declarar com a **secció crítica** per evitar aquest tipus d'error.

Les instruccions que formen part d'una **secció crítica** s'han d'**executar com** si foren **una única instrucció**. S'han de **sincronitzar** els **processos** per tal que un únic procés o fil puga **excloure** de forma temporal a la **resta de processos** d'un recurs compartit (memòria, dispositius, etc.) de tal manera que la integritat del sistema quede garantida.

Productor-consumidor

El problema productor-consumidor és un exemple clàssic on cal donar un **tractament independent** a un conjunt de **dades** que es van generant de forma més o menys aleatòria o almenys d'una forma en què **no** és **possible predir** en quin moment es **generarà** una **dada**. Per evitar un ús excessiu dels recursos de l'ordinador esperant l'arribada de dades, el sistema preveu dos tipus de processos: els **productors**, encarregats d'**obtenir** les **dades** a **tractar** i els **consumidors**, especialitzats en fer el **tractament de** les **dades obtingudes pels productors**.

En aquesta situació el **productor genera** un seguit de **dades** que el **consumidor recull**. Imaginem que és el valor d'una variable que el productor modifica i el consumidor l'agafa per utilitzar-la. El problema ve quan el **productor produeix** dades a un **ritme diferent** del que el **consumidor** les **agafa**. El productor crea una dada i canvia la variable al seu valor. Si el **consumidor** va més **lent**, el **productor** té temps a generar una nova dada i torna a **canviar** la **variable**. Per tant el procés del consumidor ha perdut el valor de

la primera dada. En el cas que siga el **consumidor** el que va més **ràpid**, pot passar que **agafe dues vegades** una **mateixa dada**, ja que el **productor no** ha tingut temps de **substituir**-la, o que **no trobe res per consumir**. La situació pot complicar-se encara més si disposem de diversos processos productors i consumidors.

A la figura.13 podem veure com dos **processos comparteixen** un **recurs comú** (la memòria) i la il·lustració del problema quan no està sincronitzat el procés productor i el consumidor, i el **productor** és **més ràpid que** el **consumidor**.

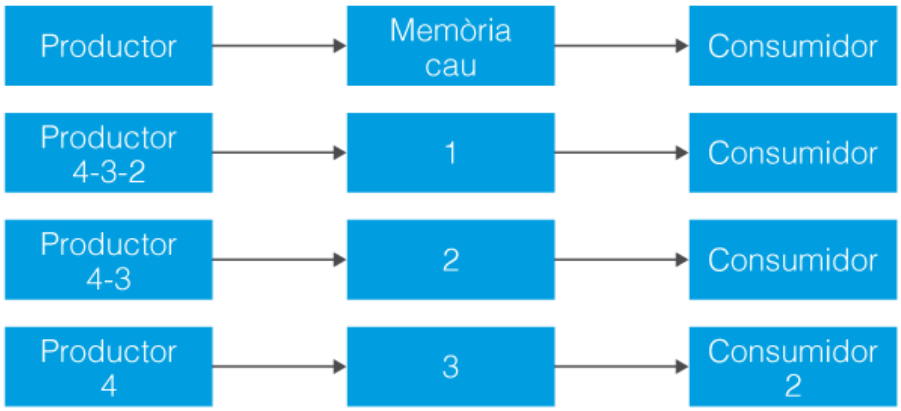

Figura 13. Productor-consumidor

Imaginem que en la figura.13 la memòria comuna és com una caixa que és capaç de guardar una única dada, un enter. Existeix un procés **productor** que **genera** els números **enters** i els **deixa a la caixa**. Mentrestant hi ha un procés **consumidor** que **agafa** el **número** enter **de la caixa**.

Com és el cas, el productor deixa a la memòria el número 1 i abans que el procés consumidor agafe la dada, genera un altre número, el 2, que substitueix l'anterior. El consumidor ara sí que agafa el número 2 però, tal com podem veure, el **número 1 s'ha perdut**, produint segurament uns resultats erronis.

Una manera de solucionar el problema consisteix a **ampliar** la **ubicació** on el

productor escriu les **dades** de manera que siga possible mantenir diverses dades a l'espera que siguen consumides mentre els processos consumidors estiguen ocupats. És a dir, els **productors emmagatzemaran** les **dades en un llista** (array), que tradicionalment es coneix com a **buffer** i els consumidors les aniran extraient.

> ■ Un **buffer** és un espai de memòria per emmagatzemar dades. Es poden implementar en forma de **cues**.

La figura.14 és l'esquema d'un procés productor que deixa informació a un buffer i que és agafada per un procés consumidor.

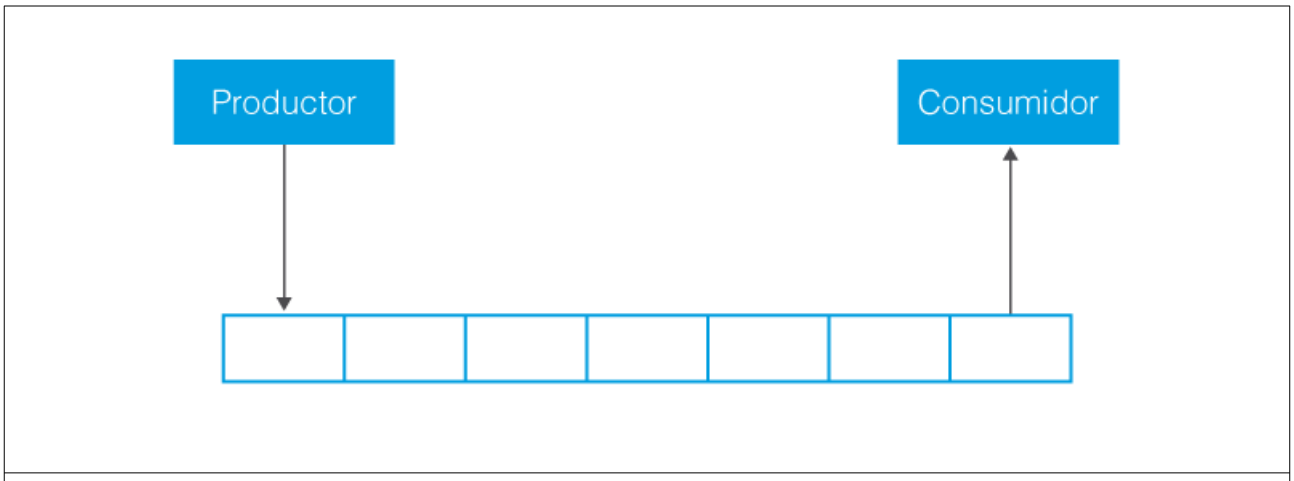

Figura 14. Buffer

Malauradament aquest mecanisme **no soluciona tots els problemes**, ja que pot ser que un **consumidor** intente **accedir** a les **dades** malgrat el **productor no n'haja** encara **escrit cap**, pot passar que l'**espai** destinat a emmagatzemar la col·lecció de dades **s'omple** a causa que la **producció** de dades siga sempre **molt més ràpida que** els processos **consumidors**, o bé podria donar-se el cas que **dos** processos **productors coincidiren** a l'hora de **deixar** una **dada** o que diversos processos **consumidors intentaren accedir a l'hora**.

Per tant, ha d'existir un **mecanisme** que **detinga l'accés** a la **dada** dels

productors i dels **consumidors** en cas necessari. **Una secció crítica**. Malauradament no n'hi ha prou a restringir l'accés a les dades, perquè podria donar-se el cas que un procés **consumidor esperant** l'arribada d'una **dada impedisca** l'**accés** dels processos **productors** de forma que la **dada no arribara mai**. Una situació com la descrita es coneix amb el nom de problemes d'**interbloqueig** (**deadlock** en anglès).

■ Anomenem **interbloqueig** a la situació extrema que ens trobem quan **dos** o més **processos** estan esperant l'**execució** de **l'altre** per poder continuar de manera que **mai aconseguiran desbloquejar-se**.

■ També anomenem **inanició** a la situació que es produeix quan **un procés no pot continuar** la seua execució **per falta** de **recursos**. Per exemple si férem créixer el buffer il·limitadament, arribant a ocupar tota la memòria: el productor patiria inanició.

La **inanició no és sinònim d'interbloqueig**, encara que l'**interbloqueig produeix la inanició dels processos involucrats**. La inanició pot (encara que no té per què) acabar, mentre que un interbloqueig no pot finalitzar sense una acció de l'exterior.

Per exemple, si un sistema multitasca (mal dissenyat) sempre canvia entre les dues primeres tasques mentre que **una tercera no s'executa mai**, aleshores **la tercera tasca està patint inanició** de temps de CPU

Per solucionar el problema cal **sincronitzar l'accés al buffer**. S'ha d'**accedir en exclusió mútua** per tal que els **productors no alimenten** el **buffer si** aquest **ja està ple** i els **consumidors no** hi **puguen accedir si està buit**. Però a més caldrà **independitzar** les **seccions crítiques** d'accés dels **productors de** les **seccions crítiques** d'accés dels **consumidors evitant** així que es puga produir l'**interbloqueig**.

Això obligarà a crear un **mecanisme** de **comunicació entre seccions crítiques** de manera que cada vegada que un **productor** deixe una dada disponible, **avise** els

consumidors que puguen estar esperant que almenys un d'ells puga **iniciar** el **processament** de la **dada**.

Lectors-escriptors

Un altre tipus de **problema** que apareix en la **programació concurrent**, és el produït quan tenim un **recurs compartit** entre diversos processos concurrents com poden ser un **fitxer**, una **base de dades**, etc. que va **actualitzant-se periòdicament**. En aquest cas els **processos lectors** no consumeixen la dada sinó que **només** l'**utilitzen** i per tant es permet la seua **consulta** de forma **simultània**, malgrat que **no** la seua **modificació**.

Així, els processos que accedeixen al **recurs** compartit per **llegir** el seu contingut s'anomenaran **lectors**. En canvi, els que accedeixen per **modificar-lo** rebran el nom de **escriptors**.

Si la **tasca** de lectors i escriptors no es realitza de forma **coordinada** podria passar que un **lector llegira** diverses vegades la **mateixa dada** o que l'**escriptor modificara** el **contingut abans** que hagueren **llegit** tots els **lectors**, o que la **dada actualitzada** per un **escriptor malbaratés l'actualització d'un altre**, etc. A més, la falta de coordinació obliga als **lectors** a **comprovar** periòdicament si els **escriptors** han fet **modificacions**, cosa que farà **augmentar l'ús** del **processador** i per tant podria **minvar l'eficiència**.

Aquest problema de sincronització de processos s'anomena p**roblema de lectors-escriptors**. Per evitar-lo cal assegurar que els processos **escriptors accedeixen** de **forma exclusiva** al **recurs** compartit i que a cada **modificació** s'**avisa** als processos **lectors** interessats en el canvi.

Així, els processos **lectors** poden restar en **espera fins** que són **avisats** que hi ha **noves dades** i poden començar a llegir; d'aquesta manera s'evita que els lectors estiguen constantment accedint al recurs sense que l'escriptor haja posat cap nova dada, optimitzant, d'aquesta manera, els recursos.