Architettura degli Elaboratori, a.a. 2006-07

Parte Terza: guida allo studio e integrazioni

La Terza Parte riguarda le gerarchie di memoria e principi di strutturazione di sistemi operativi.

Il materiale didattico è il seguente:

- **Gerarchie di memoria: memoria virtuale e memoria cache:**
  - *Sez. 1, 2, 3*di queste note;
  - *Cap. VIII;*
  - *Cap. VI, sez. 2;*
- **Principi di strutturazione di sistemi operativi:**
  - *Sez. 4*, *5* di queste note;
  - *Cap. VI*, Sez. 1.

La Terza Parte è accompagnata dalla **terza Esercitazione di Verifica Intermedia**.

1. Gerarchie di memoria e memoria cache

La trattazione di questa parte è contenuta nel Cap. VIII e nel Cap. VI, sez. 2, della Dispensa.

Come spiegato nella sez. 1 del Cap. VIII, il concetto di gerarchia di memoria, e la tecnica della gerarchia di memoria con *paginazione*, sono applicati a molti livelli in un sistema. Per gli scopi di questo corso, le gerarchie più importanti sono "**Memoria Virtuale – Memoria Principale**" e "**Memoria Principale – Memoria Cache**", anche se i concetti di *località* e *riuso* (che sono alla base delle gerarchie di memoria con paginazione) sono applicati in diversi altri casi (come nell'utilizzazione dei registri generali del processore).

- La parte **Gerarchia Memoria Virtuale – Memoria Principale** è trattata nella sez. 2 del Cap. VI, da studiare dopo la sez. 1 del Cap. VIII. La **realizzazione della MMU**, indispensabile per l'applicazione del concetto di gerarchia di memoria con paginazione a questo livello, è spiegata nella sez. 1.1. del Cap. VII.
- La parte **Gerarchia Memoria Principale – Memoria Cache** è contenuta nella sez. 2 del Cap. VIII, integrato dalla sez. 2 e dalla sez. 3 di queste note.

2. Metodi di indirizzamento della memoria cache

Nel seguito sono riportati alcuni chiarimenti riferiti alle pagine fotocopiate ("stampate per orizzontale") facenti parte della sez. 2.1 (tra la pagina VIII-9 e la pagina VIII-10). I numeri delle pagine (da 307 a 313) che verranno menzionati sono quelli delle pagine originali fotocopiate.

All'inizio di questa parte (pag. VIII-9) è precisato che il termine "indirizzo logico", nella gerarchia "memoria principale – memoria cache", è fuorviante. Questo termine va usato, come in tutto il corso, solo per indicare un indirizzo della Memoria Virtuale di un programma. Di conseguenza, ogni volta che si incontra il termine "indirizzo logico" (ad esempio, a pagina 308), questo va sostituito con il termine "**indirizzo di memoria principale**", o "indirizzo fisico", mentre il termine "indirizzo fisico" va sostituito con "**indirizzo di cache**".

2.1. Metodo diretto

Pagina 308: le prime 11 righe, fino alla formula inclusa, vanno sostituite come segue:

"In fig. 11.a è mostrato questo schema di corrispondenza, nel caso si adotti la funzione *modulo*, secondo la quale l'identificatore BC del blocco di cache è dato da:

$$BC = BM \ \textbf{mod} \ NC$$

dove BM indica l'identificatore del blocco di M e NC il numero di blocchi della memoria cache. Se, come di regola NC è una potenza di 2, il campo BM dell'indirizzo di M contiene l'informazione BC *direttamente* nei $lg_2 NC$ bit meno significativi, mentre i restanti bit di MB, indicati con TAG,

$$TAG = BM \ \textbf{div} \ NC$$

identificano il blocco di M all'interno dell'insieme di tutti quelli che corrispondono a quel particolare blocco di C identificato da BC."

Pagina 309: dove si afferma che "l'accesso a tale tabella può essere effettuato in parallelo all'accesso alla memoria cache: ne consegue che l'accesso stesso è completato in un ciclo di clock", aggiungere:

"Questo è reso possibile dall'adozione della tecnica del *controllo residuo* (Cap. III): il bit che indica l'eventuale presenza di fault è quello che condiziona la scrittura nella memoria cache (se richiesta) e la scrittura nell'interfaccia verso il processore (fig. 1, pag. VIII-8). Come **esercizio**, i realizzi in dettaglio l'applicazione di questa tecnica.

Con il metodo diretto, tenendo conto anche della presenza della MMU, il tempo di accesso alla memoria cache, in assenza di fault, come visto dal processore è:

$$t_c = 2\tau$$

che rappresenta il minimo valore possibile con il protocollo a domanda e risposta".

2.2. Metodo completamente associativo

Il componente logico "**memoria associativa"** è spiegato nel Cap. VII, sez. 1.1, a proposito dell'applicazione che di questo metodo viene fatta nella MMU per la gerarchia "Memoria Virtuale – Memoria Principale".

Pagina 310: nella figura, ogni uscita della memoria associativa AM è un singolo bit, uguale a 1 se il contenuto della locazione corrispondente è diverso dal campo BM dell'indirizzo di M. Di conseguenza, la condizione "fault" è data dall'OR di tutte le uscite di AM: se questo bit è uguale a zero, non c'è fault.

Il valore di BC è ottenuto da una rappresentazione su lg_2NC bit della configurazione degli NC bit all'uscita di AM. La rete combinatoria per implementare questa trasformazione è indicata con "Codificatore" nella figura: per **esercizio**, ricavare questa rete.

Tenendo conto anche della presenza della MMU, il tempo di accesso alla memoria cache, in assenza di fault, come visto dal processore è:

$$t_c = 3\tau$$

assumendo che in un ciclo di clock sia possibile la stabilizzazione di un solo componente logico memoria.

2.3. Metodo associativo su insiemi

Pagina 311: le 8 righe a partire da "La regola di corrispondenza scelta …" vengono sostituite come segue:

"Similmente a quanto fatto nel Metodo Diretto, adottiamo la funzione *modulo* per esprimere la corrispondenza tra blocchi di M e insiemi di cache; l'identificatore SET dell'insieme dei blocchi di cache è dato da:

$$SET = BM \ \mathbf{mod} \ NS$$

dove BM indica l'identificatore del blocco di M e NS il numero di insiemi di blocchi della memoria cache. Se, come di regola NS è una potenza di 2, il campo BM dell'indirizzo di M contiene l'informazione SET *direttamente* nei lg_2NS bit meno significativi, mentre i restanti bit di MB, indicati con TAG,

$$TAG = BM \ \mathbf{div} \ NS$$

identificano il blocco di M all'interno dell'insieme di tutti quelli che corrispondono a quel particolare insiemi di blocchi di C identificato da SET."

Pagina 312: lo schema della figura va realizzato più in dettaglio. Ogni locazione della Tabella di Corrispondenza (Tab), indirizzata con il campo SET dell'indirizzo di M, contiene, *nell'ipotesi di avere due blocchi per insieme,* due parti. Ogni parte contiene, oltre ad altri bit di controllo, la coppia (P_i, TAG_i), con i = 0, 1, dove P_i è il bit di presenza del blocco e TAG_i è il TAG dell'eventuale blocco di M presente nel blocco.

All'uscita di Tab, due reti combinatorie hanno l'una in ingresso BM.TAG e Tab[BM.SET].TAG_0, l'altra BM.TAG e Tab[BM.SET].TAG_1: si tratta di confrontatori, per ognuno di quali i bit di uscita sono posti in OR. Siano X_0 e X_1 le uscite di tali reti:

$$X_0 = OR(BM.TAG \oplus Tab[BM.SET].TAG_0) \ , \ X_1 = OR(BM.TAG \oplus Tab[BM.SET].TAG_1)$$

Occorre realizzare una funzione avente come dominio P_0, P_1, X_0, X_1, e come condominio F e B, entrambi di un bit, dove F è uguale a uno se c'è fault di blocco, e B esprime quale blocco indirizzare all'interno dell'insieme BM.SET nel caso non ci sia fault. La funzione è definita mediante la tabella di verità di pagina seguente, dalla quale è agevole ricavare l'implementazione della rete combinatoria. Se F = 0, allora l'indirizzo di cache è dato dalla concatenazione dei valori (BM.SET, B, BM.D).

Tenendo conto anche della presenza della MMU, il tempo di accesso alla memoria cache, in assenza di fault, come visto dal processore è:

$$t_c = 3\tau$$

assumendo che in un ciclo di clock sia possibile la stabilizzazione di un solo componente logico memoria.

P_0	X_0	P_1	X_1	F	B	Note
0	0	0	0	1	–	fault
0	0	0	1	1	–	fault
0	0	1	0	0	1	blocco di M presente nel secondo blocco dell'insieme di C
0	0	1	1	1	–	fault
0	1	0	0	1	–	fault
0	1	0	1	1	–	fault
0	1	1	0	0	1	blocco di M presente nel secondo blocco dell'insieme di C
0	1	1	1	1	–	fault
1	0	0	0	0	0	blocco di M presente nel primo blocco dell'insieme di C
1	0	0	1	0	0	blocco di M presente nel primo blocco dell'insieme di C
1	0	1	0	–	–	situazione impossibile o di errore (da segnalare come eccezione)
1	0	1	1	0	0	blocco di M presente nel primo blocco dell'insieme di C
1	1	0	0	1	–	fault
1	1	0	1	1	–	fault
1	1	1	0	0	1	blocco di M presente nel secondo blocco dell'insieme di C
1	1	1	1	1	–	fault

2.4. Trasferimento di blocchi

Questo aspetto è trattato nella sez. 2.2.2 del Cap. VIII, ed è esemplificato ampiamente nella sez. 3 di queste note.

Per quanto riguarda l'uso della memoria modulare interallacciata, con riferimento alle soluzioni indicate con 1) e 2) (e riportate in Fig. 2 nel Cap. VIII della Dispensa), si consideri solo la soluzione **2)** detta "**trasferimenti su parola lunga**".

2.5. Cache prefetching: opzioni nelle istruzioni Load/Store

Nella sez. 1.2.2 del Cap. VIII è spiegato il significato di "prefetching dei blocchi" in alternativa al metodo "su domanda". Nei casi in cui sia decidibile staticamente l'opportunità di avere un funzionamento con prefetching, il compilatore può indicare che, facendo accesso per la prima volta ad un certo blocco di cache, deve essere trasferito in cache anche il blocco di identificatore successivo.

Affinché il livello firmware sia a conoscenza del metodo da adottare, è sufficiente un bit (che verrà trasmesso all'unità cache) nelle istruzioni di LOAD e STORE per il prefetching dei *dati*, mentre il prefetching dei blocchi delle *istruzioni* può essere automatico da parte dell'unità cache.

3. Esercizi su architettura e valutazione di CPU con memoria cache

3.1. Risoluzione di un problema tipico

Valutare il tempo di completamento, il tempo medio di elaborazione e la performance di programmi assembler nel caso che la CPU contenga una cache primaria con le seguenti caratteristiche:

- operante su domanda,
- metodo di indirizzamento associativo su insiemi,

- capacità 64K parole,
- ampiezza di blocco uguale a 16 parole,
- scritture gestite con il metodo Write-Through,
- cache secondaria interallacciata di capacità 1M parole, con 8 moduli collegati direttamente alla CPU, e ciclo di clock uguale a 10τ,
- collegamenti inter-chip con ritardo uguale a 5τ,
- trasferimenti di blocchi con la tecnica "su parola lunga".

Si faccia l'ipotesi che la probabilità di fault della cache secondaria sia trascurabile.

Valutare anche l'**efficienza relativa**, ε, con $0 < \varepsilon \leq 1$, definito come il *rapporto tra tempo di completamento ideale, cioè in assenza di fault di cache, e tempo di completamento effettivamente ottenuto considerando i fault.*

Traccia di procedimento

Il tempo di completamento si può valutare come:

$$T_c = T_{c\text{-}id} + T_{fault}$$

dove $T_{c\text{-}id}$ è il tempo di completamento in assenza di fault di cache, e T_{fault} è il tempo speso per gestire tutti i fault di cache che hanno luogo durante l'esecuzione del programma. Nella valutazione di $T_{c\text{-}id}$ per il tempo di accesso in memoria va, quindi, preso il tempo di accesso alla cache t_c (come "visto" dal processore). T_{fault} si valuta come:

$$T_{fault} = N_{fault} * T_{trasf}$$

dove N_{fault} è il numero medio di fault di cache che hanno luogo durante tutta l'esecuzione del programma, e T_{trasf} è il tempo necessario per effettuare il trasferimento di un blocco di cache dal livello superiore (cache secondaria nel nostro caso) alla cache primaria. Si veda la dispensa, cap. VIII, sez. 2.2.2. Per una organizzazione con "trasferimenti su parola lunga" si ha che, se m è il numero di moduli della cache secondaria interallacciata, σ è l'ampiezza del blocco, con σ multiplo di m, e t_{C2} il tempo di accesso alla cache secondaria:

$$T_{trasf} = \frac{\sigma}{m} t_{C2} + m\tau$$

Da T_c si ricavano T, $\mathcal{P}$ e ε.

Nel caso che nel programma assembler alcune strutture dati siano utilizzate solo in *scrittura* (cioè, che siano presenti istruzioni di STORE e che queste operino su indirizzi che non sono utilizzati da istruzioni di LOAD), per l'accesso ai blocchi di tali strutture si può assumere $T_{trasf} = 0$ in quanto i blocchi *non* vengono trasferiti in cache: in caso di fault, l'unità cache si limita ad allocare un blocco nel quale avverranno le operazioni di scrittura.

Se si adotta la tecnica *Write-Through*, le scritture nel livello superiore avvengono *in parallelo* al calcolo nelle CPU; quindi, entro ampi limiti, le scritture non hanno impatto sulle prestazioni.

Nel caso di tecnica *Write-Back*, in T_{trasf} occorrerebbe tener conto della potenziale riscrittura in memoria di blocchi sostitutiti. Poiché questo evento ha, in molti programmi, una probabilità abbastanza bassa, in prima approssimazione siamo autorizzati a non considerare questa penalità su T_{trasf}.

Volendo calcolare prima T, e da questo T_c, occorre calcolare il *tempo di accesso in memoria equivalente*, dato da:

$$t_a = (1 - h)\, t_c + h\, T_{trasf}$$

che richiede il calcolo della *probabilità di fault*

$$h = \frac{N_{fault}}{N_{tot}}$$

con N_{tot} numero totale di accessi in memoria per tutta l'esecuzione del programma (istruzioni e dati).

3.2. Esercizio svolto

Problema

Si consideri il programma assembler Risc ottenuto dalla compilazione di un programma sorgente che, dato un vettore A di 256K interi, modifica A in modo che ogni elemento sia sostituito dal suo quadrato.

a) Con riferimento a tale programma, spiegare le proprietà di località e riuso.

b) Valutarne il tempo di completamento, il tempo medio di elaborazione, la performance e l'efficienza relativa per un calcolatore con le seguenti caratteristiche:

i) processore con ciclo di clock $\tau = 0{,}3$ nsec;

ii) cache primaria operante su domanda, con indirizzamento completamente associativo, blocchi di 8 parole, scritture con il metodo Write-Through;

iii) cache secondaria (con probabilità di fault trascurabile) organizzata con 4 moduli interallacciati collegati direttamente alla CPU, ciclo di clock di 10τ e latenza di trasmissione dei collegamenti inter-chip di 10τ.

Spiegare esplicitamente come si è tenuto conto di ognuna delle caratteristiche del punto *ii).*

c) Spiegare, in generale, le motivazioni dell'inserimento della cache secondaria.

Soluzione

Il programma da compilare è:

```
int A[256K];
for (i = 0; i < N; i++)
        A[i] = A[i] * A[i];
```

Il compilatore provvede a inizializzare RG[RA] all'indirizzo logico base di A, RG[Ri] a zero, RG[RN] alla costante N = 256K, ed a riservare RG[Ra] come temporaneo non inizializzato. Adottando la regola di compilazione di un *for* con almeno una iterazione, il programma assembler richiesto è il seguente:

```
LOOP:  LOAD  RA, Ri, Ra
       MUL   Ra, Ra, Ra
       STORE  RA, Ri, Ra
       INCR  Ri
       IF<  Ri, RN, LOOP
       END
```

a) Località e riuso

Queste proprietà possono essere spiegate osservando una *traccia di indirizzi logici generati dal programma* in esecuzione. Supponendo, ad esempio, che, nella memoria virtuale, il codice sia allocato a partire dall'indirizzo 0 e l'array A sia allocato a partire dall'indirizzo 1024, la seguente è la traccia della fase iniziale dell'esecuzione del programma (in assenza di interruzioni ed eccezioni):

0, 1024, *1*, *2*, 1024, *3*, *4*, *0*, 1025, *1*, *2*, 1025, *3*, *4*, *0*, 1026, *1*, *2*, 1026, *3*, *4*, *0*, 1027, …

Si osserva che, all'interno di una finestra temporale sufficientemente ampia, si intrecciano più (due nel caso specifico) sequenze di indirizzi logici ([0, 1, 2, 3, 4] e [1024, 1024, 1205, 1025, 1026, 1026, 1027, …]) dove in ogni sequenza gli indirizzi sono tra loro relativamente vicini rispetto all'ampiezza della finestra temporale (consecutivi e/o coincidenti nel caso specifico): questa proprietà è detta *località* dei riferimenti. Inoltre, alcune sequenze si ripetono molte volte all'interno della finestra temporale (la prima nel caso specifico), oppure alcuni riferimenti si ripetono all'interno della stessa sequenza (come nel secondo caso), dando luogo a *riuso* delle informazioni.

Organizzando le informazioni a pagine (blocchi), risulta relativamente elevata la probabilità che, per un periodo di tempo relativamente lungo, le informazioni riferite appartengano ad un numero limitato di pagine e che le pagine riferite variino lentamente: nel caso specifico, la stessa pagina di codice è usata per tutta la durata del programma, mentre la stessa pagina di dati è usata per σ riferimenti consecutivi all'array A, con σ ampiezza della pagina.

Quanto ora detto vale qualitativamente *per qualunque* (sotto-)gerarchia di memoria. Passando da una sotto-gerarchia (ad esempio, Memoria Virtuale – Memoria Principale) ad un'altra (ad esempio, Memoria Principale – Memoria Cache), cambiano quantitativamente i valori di σ e della probabilità di fault *h* (ad esempio, $s \sim 10^3$ parole e $h \sim 10^{-4} \div 10^{-5}$ nel primo caso, $s \sim 10^0 \div 10^1$ parole e $h \sim 10^{-2}$ nel secondo caso).

b) Valutazione delle prestazioni

Essendo la cache operante su domanda, il trasferimento di un blocco dal supporto superiore della gerarchia viene effettuato ad ogni fault sequenzialmente all'elaborazione. Il tempo di completamento si valuta come:

$$T_c = T_{c-id} + T_{fault} = T_{c-id} + N_{fault} * T_{trasf}$$

dove

- $T_{c\text{-}id}$ è il tempo di completamento ideale, in assenza di fault. Esso è quindi valutando assumendo che il tempo di accesso alla memoria sia uguale al tempo di accesso alla cache primaria; nel nostro caso, essendo usato il metodo completamente associativo, $t_c = 3\tau$;
- N_{fault} è il numero di fault che si verificano durante l'esecuzione del programma. Le istruzioni provocano un solo fault iniziale, dopo di che, grazie alla proprietà del riuso, esse sono sempre presenti in un blocco di cache; si può dunque trascurare la probabilità di fault delle istruzioni. Per i dati (elementi dell'array A), si verifica un fault ogni σ iterazioni, quindi

$$N_{fault} = \frac{N}{\sigma} = \frac{N}{8}$$

- T_{trasf} è il tempo per trasferire un blocco dalla cache secondaria C2 alla cache primaria C1. Essendo C2 interallacciata con $m = 4$ moduli, occorrono due letture da C2 per un blocco di $\sigma = 8$ parole, delle quali la seconda lettura si sovrappone alla scrittura in C1 delle 4 parole ottenute con la prima lettura:

$$T_{trasf} = \frac{\sigma}{m} t_{C2} + m\tau = 2t_{C2} + 4\tau = 2(\tau_{C2} + 2T_{tr}) + 4\tau = 64\tau$$

Quindi:

$$T_{fault} = N_{fault} * T_{trasf} = 8N\tau$$

Con i simboli noti per i tempi medi di chiamata istruzione e decodifica (T_{ch}) e della fase di esecuzione (T_{ex}), e ricordando le prestazioni dell'interprete microprogrammato del linguaggio assembler Risc, il tempo di completamento ideale è dato da:

$$T_{c-id} = N\ [5T_{ch} + 2T_{ex-LD/ST} + T_{ex-MUL} + T_{ex-INCR} + T_{ex-IF}\ ] =$$
$$= N\ [5(2\tau + t_c) + 2(2\tau + t_c) + 50\tau + 1\tau + 2\tau\ ] = 88N\tau$$

Quindi:

$$T_c = T_{c-id} + T_{fault} = 96N\tau = 7{,}55 \text{ msec}$$

Una singola iterazione è eseguita in 96τ, quindi la scrittura nella cache secondaria (che richiede un tempo $t_{C2} = 30\tau$) è mascherata completamente.

L'efficienza relativa della CPU con cache è data da:

$$\varepsilon = \frac{T_{c-id}}{T_c} = \frac{88}{96} = 0{,}92$$

che è un buon risultato (cache sfruttata al 92% delle proprie potenzialità), dovuto anche al fatto che il programma è "rallentato" dall'istruzione MUL.

Essendo *5N* il numero di istruzioni complessivamente eseguite, il tempo medio di elaborazione per istruzione è dato da:

$$T = \frac{T_c}{5N} = \frac{96}{5}\tau = 19{,}2\tau = 5{,}76 \text{ nsec}$$

La performance vale:

$$\wp = \frac{1}{T} = 173{,}6\,MIPS$$

c) Motivazioni dell'inserimento della cache secondaria

Nonostante si ricorra ad una organizzazione interallacciata, essendo la probabilità di fault dell'ordine di 10^{-2} nel caso più favorevole, l'efficienza relativa sarebbe ugualmente piuttosto bassa se il supporto di memoria immediatamente superiore alla cache (primaria) fosse direttamente la memoria principale, a causa del tempo di accesso, e quindi del valore di T_{trasf}, troppo elevato.

Ad esempio, nel nostro caso, conservando $m = 4$, ma passando da $t_{C2} = 30\tau$ a $t_M \sim 300\tau$ (anche a causa dell'aumento della latenza dei collegamenti), avremmo $T_{fault} = 75{,}5N\tau$, $T_c = 163{,}5N\tau$, $\varepsilon = 0{,}54$.

3.3. Testi di esercizi

1) Come l'Esercizio della sez. 3.1, 3.2, ma prevedendo una memoria cache operante con *prefetching* invece che su domanda. Valutare prima se la tecnica del prefetching è applicabile, e spiegare se un compilatore può riconoscere tale applicabilità.

2) Spiegare la seguente affermazione: data una gerarchia di memoria contenente i supporti M_0, …, M_{n-1}, la dimensione dei blocchi della sotto-gerarchia M_i - M_{i-1} è minore di quella della sotto-gerarchia M_{i+1} - M_i (i = 0 … N-1).

3) Spiegare in dettaglio il funzionamento di una unità cache con le seguenti caratteristiche:

- capacità di 128K parole,
- blocchi ampi 16 parole,
- indirizzamento associativo su insiemi con insiemi di 4 blocchi,
- scritture gestite con il metodo Write-Through.

La memoria principale ha capacità massima di 4G parole.

La spiegazione deve includere la realizzazione della Parte Operativa dell'unità cache almeno per quanto riguarda le risorse (registri e reti logiche) necessarie all'implementazione della traduzione dell'indirizzo e della verifica della condizione di fault.

3.4. Esercizi su unità di elaborazione specializzate con memoria cache

1) Una unità di elaborazione U è connessa ad una gerarchia di memoria costituita da una memoria principale M di capacità 512M parole e da una memoria cache C di capacità 64K parole. U e l'unità che contiene C appartengono allo stesso chip.

M è interallacciata con 8 moduli ed ha ciclo di clock uguale a 5 volte quello di U. C ha blocchi di 8 parole, opera su domanda ed è indirizzata con il metodo diretto. Tutti i collegamenti inter-chip hanno latenza di trasmissione uguale a 5 volte il ciclo di clock di U.

U riceve in ingresso messaggi (J, N) dove J è un indirizzo di M, e N è un intero positivo. U invia in uscita il numero degli elementi strettamente positivi dell'array avente indirizzo base J e dimensione N.

Valutare il tempo medio di elaborazione di U in funzione di t_p, spiegando adeguatamente la risposta.

2) Una unità di elaborazione U ha come unica operazione esterna la somma di array di interi. Il messaggio di ingresso include la dimensione N degli array (con N potenza di due ≤ 128K) e gli indirizzi base dei tre array (INDA, INDB, INDC). Il messaggio di uscita consiste semplicemente in una comunicazione che l'operazione richiesta è stata completata. Gli array sono memorizzati in una memoria esterna M di capacità molto maggiore di N. Il sottosistema di memoria fa uso di cache con le caratteristiche dell'Esercizio C1.

Progettare U e valutarne il tempo medio di elaborazione.

4. Processi cooperanti

4.1. Dai programmi ai processi

Per svariate ragioni (costo, efficienza, messa a disposizione di servizi, gestione di risorse), in un sistema di elaborazione general-purpose *coesistono*, istante per istante, più programmi, alcuni applicativi (utenti), altri di sistema (sistema operativo), altri ancora per lo sviluppo di applicazioni (compilatori, debugger, monitor, editor, interfacce grafiche, ecc).

La convenienza di questa visione si può vedere considerando il ciclo di vita di un generico programma applicativo; ad esempio:

- quando ne viene chiesta la compilazione, il sistema sarà generalmente già occupato ad eseguire altri programmi;
- quando ne viene chiesta l'esecuzione, come sintetizzato nella sez. 4, l'allocazione dinamica delle risorse (memoria principale) viene effettuata da altri programmi (di sistema) che coesistono con una o più applicazioni correnti;
- quando, durante l'esecuzione, il programma rileva la mancanza di risorse necessarie (ad esempio, fault di pagina della memoria virtuale), occorre che esso cooperi con programmi di sistema opportuni (gestore della memoria); analogamente se, durante l'esecuzione, un programma necessita di accedere a risorse gestite dal sistema (ad esempio, files).

Questa coesistenza, detta anche *multiprogrammazione* (o altri termini), è implementata anche in un calcolatore *con un solo processore*. Sono i "tempi morti" che si verificano frequentemente durante l'elaborazione di un programma a far sì che possano subentrare, al suo posto nell'utilizzo della "risorsa Processore", altri programmi.

Ovviamente, avendo *più Processori* (più CPU), l'occasione di dar luogo a multiprogrammazione è ancora più evidente: nello stesso istante, tanti programmi per quante sono le CPU sono effettivamente in esecuzione; cioè, si ha *parallelismo a livello di programmi.* Avendo invece un unico Processore, la multiprogrammazione contribuisce a *sfruttare meglio le risorse* del sistema (Processore e memoria principale, in primis), *dando l'illusione*, su un certo arco di tempo, che più programmi siano in esecuzione contemporaneamente: in realtà non si ha vero parallelismo, ma *parallelismo simulato*.

Il concetto di processo sta a significare un programma capace di essere eseguito contemporaneamente ad altri, ed in generale di *cooperare* con altri, dove la contemporaneità può essere effettiva (*parallelismo* in senso stretto) o simulata (in questo caso si usa il termine *concorrenza*, più generale di parallelismo).

Il concetto di processo è quindi un caso particolare del concetto di *modulo di elaborazione* (Cap. I, sez. 2), visto ai livelli delle applicazioni e del sistema operativo, così come lo sono le unità di elaborazione a livello firmware. Così come le unità, anche i processi si possono *comporre* affinché *cooperino* ad un fine comune (una applicazione).

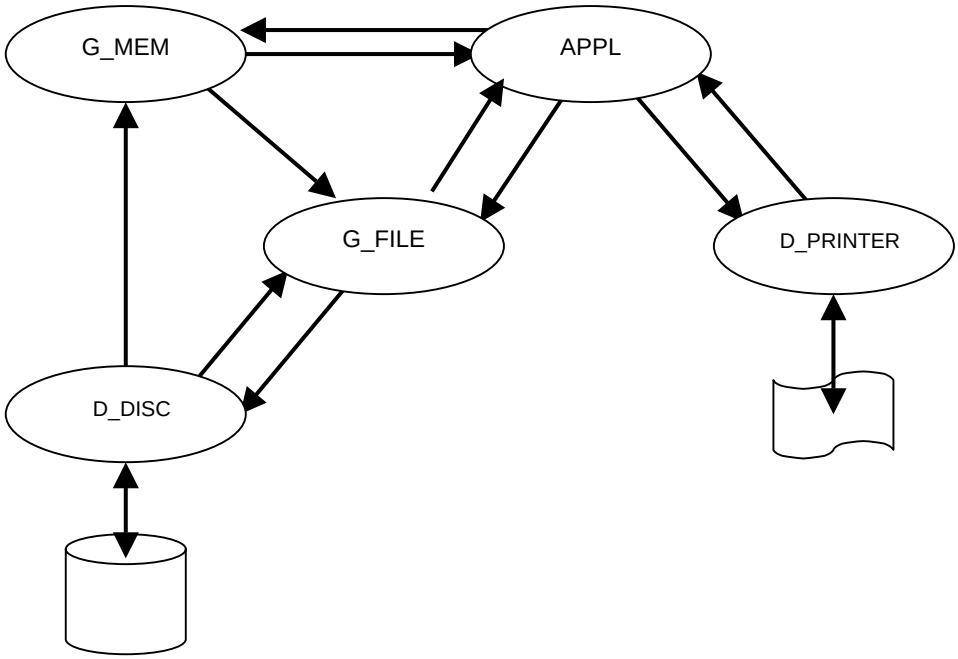
A livello dei processi, un sistema di elaborazione va quindi visto come una collezione (in generale dinamica) di processi cooperanti. Una tipica situazione è quella in cui si distinguono due sottoinsiemi di processi:

- *processi di sistema* (operativo): sono processi che esistono *permanentemente* nel sistema, e che sono delegati alla gestione di risorse e servizi nei confronti di richieste delle applicazioni, ad esempio gestione della memoria principale, gestione della memoria secondaria, gestione dei file, gestione di dispositivi periferici (driver), gestione delle applicazioni (shell). Oltre che con le applicazioni, i processi di sistema cooperano tra loro: ad esempio, il processo shell coopera almeno con il gestore della memoria principale e con il gestore dei file, il gestore della memoria principale può cooperare con il gestore dei file e con il driver del disco, ecc. In sintesi, il sistema operativo può esser visto come un *programma parallelo*;
- *processi applicativi*, che derivano dalla compilazione e dalla richiesta di esecuzione di programmi applicativi. Questi processi , in generale, "nascono e muoiono". Se si tratta di processi derivati da programmi sequenziali, i processi applicativi non cooperano direttamente tra loro, ma solo con i processi di sistema. È però possibile avere anche *programmi paralleli* a livello di applicazioni, pur di adottare opportuni linguaggi ed ambienti di programmazione per il calcolo parallelo e distribuito: in questo caso, una applicazione è costituita da più processi tra loro cooperanti (oltre che cooperare con i processi di sistema).

4.2. Processi cooperanti a scambio di messaggi

Un modo elegante (non l'unico, ma del tutto reale e molto frequente) di vedere la cooperazione tra processi è quella di considerare, come modalità di cooperazione, lo *scambio di messaggi* nel modello ad ambiente locale (Cap. I, sez. 3); in altri termini, ragionare come a livello di unità: la composizione di più processi, ognuno con il proprio ambiente locale, si ottiene facendoli comunicare attraverso *canali di comunicazi*one.

Si consideri il seguente esempio: un processo applicativo (APPL), sequenziale al suo interno, ha bisogno di accedere a certi file, e quindi di cooperare con il processo gestire dei file (G_FILE), e di inviare dati ad una stampante, e quindi di cooperare con il driver della stampante (D_PRINTER); inoltre, come tutti i processi, può aver bisogno di cooperare con il processo gestore della memoria principale (G_MEM), in seguito ad eventi che attivano l'allocazione dinamica della memoria principale stessa. A loro volta, i processi di sistam coinvolti hanno bisogno di cooperare con altri, come ad esempio il driver del disco (D_DISC). La configurazione di processi e canali in gioco può essere quella mostrata in figura:



Supponiamo che, in un certo punto del programma, APPL debba leggere in file di nome MY_FILE, e sia BUF la variabile locale in cui copiare il valore del file. Nel modello di Fig. 2, la compilazione del comando di lettura-file consisterà in un comando per inviare al processo G_FILE un messaggio contenente l'identificatore dell'operazione richiesta (read_file) e il nome del file da leggere (MY_FILE). Sarà interamente compito del processo G_FILE effettuare tutte le azioni per controllare la legittimità di APPL ad accedere in lettura a quel certo file, per leggere fisicamente il file da disco, interagendo allo scopo con D_DISCO, e quindi inviare il valore del file ad APPL con una indicazione di esito (se negativo, il valore del file non è significativo). APPL, da parte sua, dopo aver inviato a G_FILE il messaggio di lettura-file, ed eventualmente avere eseguito altre azioni indipendenti dal valore del file, attende di ricevere un messaggio da G_FILE contenente appunto l'esito e l'eventuale valore del file che assegnerà alla variabile BUF.

Supponiamo di disporre di un formalismo (linguaggio di programmazione concorrente) in cui sia possibile scrivere processi mediante gli usuali comandi sequenziali ed, in più, mediante i comandi per scambiare messaggi con altri processi (si veda Cap. I, sez. 3.4). Una sintassi di tali comandi può essere:

send (identificatore di canale, valore)

receive (identificatore di canale, identificatore di variabile targa)

Oltre a questi comandi, normalmente ne sono presenti altri, in particolare ne occorre almeno un altro per il controllo del nondeterminismo nelle comunicazioni (Cap. V, sez. 3.5).

I canali, *con tipo*, sono contraddistinti da un *identificatore unico*. I valori di messaggi e le variabili targa possono essere strutturati come tuple.

Nel nostro caso, una possibile **compilazione** di APPL può essere la seguente:

… C1 …

send (CH_FILE_OUT, (read_file, MY_FILE))

… C2 …

receive (CH_FILE_IN, (esito_file, BUF)

… test di “esito_file” e gestione dell’eventuale eccezione …

… C3 (da qui in poi il codice è eseguito solo se “esito_file” = OK) …

send (CH_PRINTER_OUT, RESULT)

… C2 …

receive (CH_FILE_IN, esito_printer)

… test di “esito_printer” e gestione dell’eventuale eccezione …

… C4 …

C1, C2, C3, C4 sono sequenze di istruzioni assembler relative alla compilazione dell’elaborazione propria di APPL. RESULT è una struttura dati il cui valore deve essere stampato. CH_FILE_OUT, CH_FILE_IN, CH_PRINTER_OUT, CH_PRINTER_IN sono nomi mnemonici di canali che, in realtà, consisteranno in identificatori unici interi.

I comandi *send* e *receive* vanno intesi come sostituiti da sequenze generali di istruzioni che implementano le primitive di invio e di ricezione di messaggi: cioè, queste sequenze sono *l’interprete* dai comandi *send* e *receive*. Equivalentemente, nei punti in cui compaiono i comandi send e receive il compilatore può avere inserito *chiamate alle procedure* che consistono nel suddetto interprete.

Infine, si noti come che la strutturazione a processi comunicanti e la strutturazione ad oggetti: nel primo caso, le “interfacce”, mediante le quali viene invocato un “metodo” di una istanza di classe, corrispondono ai canali di comunicazione. Ad esempio, il canale CH_FILE_OUT è l’entità attraverso la quale un processo applicativo può invocare la funzionalità di lettura/scrittura/creazione di file; per avere una analogia ancora più forte, avremmo potuto introdurre più canali in ingresso a G_FILE, ognuno corrispondente ad una specifica operazione, eliminando dal messaggio la presenza del parametro “operazione”.

4.3. Sistema operativo e compilazione di programmi applicativi

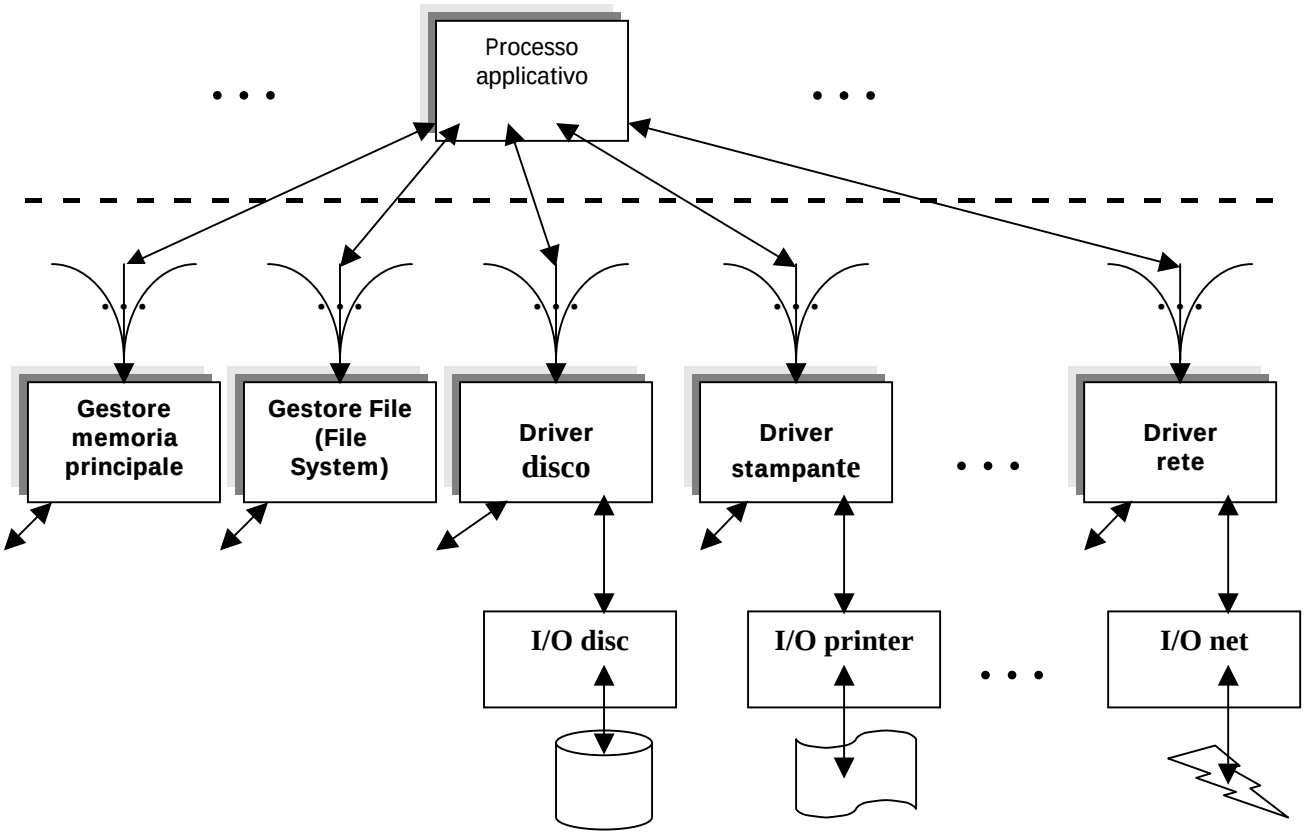
Quanto esemplificato nella sezione precedente ha carattere di generalità: la compilazione di un programma applicativo produce un processo che contiene tutte le “chiamate al sistema operativo”, cioè *invocazioni all’interprete* delle funzionalità di gestione delle risorse necessarie al programma stesso. L’interprete è il sistema operativo stesso.

Consideriamo un sistema operativo *a nucleo minimo,* nel quale tutta la gestione delle risorse (eccetto dei processori) è effettuata da appositi processi (gestori dello spazio di memoria principale e dello spazio di memoria secondaria, file system, driver delle unità di I/O).

Se il linguaggio concorrente di sistema è a scambio di messaggi, i processi di sistema prevedono canali di comunicazione (oltre che con altri processi di sistema) con un certo numero massimo di processi applicativi. Questo permette di realizzare a tempo di compilazione, in modo agevole e modulare, *l’aggancio tra processi applicativi e processi di sistema*. Ad esempio, se un programma applicativo contiene comandi per leggere/scrivere certi file, o per stampare certi dati, *il processo derivante dalla sua compilazione contiene, in corrispondenza di tali comandi, chiamate delle procedure send e/o receive necessarie per effettuare richieste ai processi gestori* (nell’esempio, file system, driver della stampante) *e ricevere eventuali risposte*, come schematizzato nella figura a pagina seguente.

I canali sono tutti identificati da interi *noti a priori* e a disposizione del compilatore per produrre il codice dei comandi di invio e ricezione messaggi.

Ad esempio, il compilatore sa che il canale CH_FILE_IN in ingresso al processi gestore dei file ha identificatore 37. Questo valore verrà utilizzato in tutti i comandi *send* corrispondenti alla richiesta di lettura file, qualunque sia il processo applicativo in cui tale comando è inserito (dal compilatore stesso); in questa visione, i canali sono in gran parte asimmetrici.



Allo stesso modo, ai canali d'ingresso dei processi applicativi vengono assegnati, a compilazione, identificatori appartenenti ad un insieme di interi noto a priori. Si tenga conto che questa soluzione è corretta in quanto gli stessi processi applicativi sono contraddistinti da identificatori unici, e questi sono *in numero limitato* in quanto esiste un numero massimo di processi (ad esempio, 20000) che possono essere ammessi al sistema contemporaneamente; ogni volta che un processo applicativo termina, il suo identificatore verrà riutilizzato per contraddistinguere un processo creato successivamente.

5. Introduzione ai sistemi operativi: il nucleo

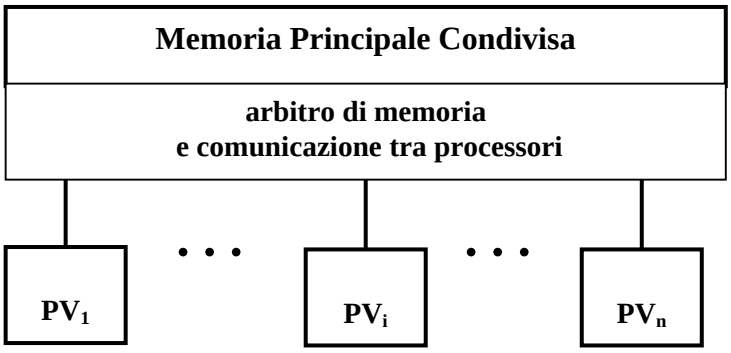
Dopo l'introduzione al concetto di processo e di processo cooperante della sez. 4, lo studente può passare allo studio del Cap. VI della Dispensa. In particolare nella sez. 1.3 si studia come implementare il concetto di processo, dando luogo al così detto *nucleo del sistema operativo* visto come *supporto a tempo di esecuzione (cioè, l'interprete) del linguaggio concorrente di sistema*, ad esempio come interprete di un linguaggio sequenziale arricchito di comandi per la cooperazione a scambio di messaggi.

Le sezioni che seguono servono da ulteriore chiarimento di quanto esposto nella sez. 1.3 del Cap. VI.

5.1. Scheduling a basso livello, architettura astratta e architetture concrete

Consideriamo una computazione, ai livelli delle Applicazioni e del Sistema Operativo, costituita da un certo numero di *processi* cooperanti P_1, P_2, …, P_n.

L'*architettura astratta* che supporta l'elaborazione di tale computazione è costituita da tanti *processori virtuali*, PV_1, PV_2, …, PV_n, per quanti sono i processi: il generico PV_i è delegato all'elaborazione di P_i. Inoltre, supporremo che tale architettura astratta sia un *multiprocessor a memoria condivisa*, nel quale (come mostrato nella figura seguente) tutti i processori (CPU con propria memoria locale o cache) sono connessi ad una stessa memoria principale, nella quale sono allocate, in particolare, strutture dati condivise tra i processi, come le strutture dati di nucleo:



Con questa architettura astratta gli *stati di avanzamento* sono solo quelli di *Esecuzione* e di *Attesa*. Quest'ultimo è di *Attesa Attiva*, in quanto, una volta che il generico P_i attenda il verificarsi di un certo evento, PV_i non ha alcun altro processo da eseguire e può solo essere sbloccato da una esplicita segnalazione da parte di un altro processore virtuale (in modo del tutto analogo a quanto avviene tra unità comunicanti a livello firmware).

Il nucleo, ed in particolare la funzionalità di scheduling a basso livello, ha il compito di *emulare l'architettura astratta su una specifica architettura concreta*. Quest'ultima potrà essere ancora un *multiprocessor*, ma con un numero di processori minore di *n* o,come caso particolare, un *uniprocessor*. Per rendere possibile tale emulazione:

- agli stati di avanzamento viene aggiunto lo stato di *Pronto* (vedi Dispensa, Cap. VI, sez. 1.3): i descrittori (PCB) di tutti i processi in stato di pronto sono collegati da un'unica Lista Pronti (nel caso più semplice con disciplina FIFO, più spesso con disciplina a priorità);
- lo stato di Attesa è ora di *Attesa Passiva*: il processore su cui veniva eseguito il processo che si sospende viene reso disponibile al primo processo in Lista Pronti. Questa modalità di attesa viene introdotta per ragioni di efficienza (miglior sfruttamento dei processori) e per ragioni di correttezza (se l'architettura concreta è uniprocessor, non appena un processo si sospendesse in attesa di un evento che deve essere provocato da un altro processo, il sistema risulterebbe bloccato in eterno).

L'elaborazione della computazione a processi sull'architettura concreta procede quindi con il meccanismo della *multiprogrammazione*, o *interleaving*:

- in un uniprocessor quei processi, che sarebbero eseguibili simultaneamente, vengono eseguiti *in un qualsiasi ordine* sull'unico processore disponibile, sfruttando le transizioni di stato di avanzamento (*concorrenza*);
- in un multiprocessor si ha anche un certo *parallelismo effettivo*, ma viene ancora sfruttato l'effetto concorrenza su ogni processore.

5.2. Interruzioni ed eccezioni

Per il trattamento di questi eventi, che è parte integrante del nucleo, è necessario un minimo di supporto a livello firmware ed assembler: si vedano le **Note su Ingresso-Uscita e Interruzioni**.

5.3. Prerilascio e quanti di tempo

Nella sez. 1.3.1 del Cap. VI è indicato come il *prerilascio di un processore* si possa verificare in due occasioni distinte:

i) un processo A in Attesa viene svegliato da un processo B ed A ha priorità maggiore di B (la priorità di un processo è indicata da un campo del proprio PCB): A passa direttamente in Esecuzione sul processore utilizzato da B, e B passa in stato di Pronto;

ii) un processo A in Esecuzione passa in stato di Pronto per permettere al primo processo Pronto di passare in Esecuzione: questo meccanismo viene utilizzato nella gestione del processore *a quanti di tempo* (time sharing), in modo da bilanciare l'utilizzazione del processore stesso da parte di processi "lunghi", e/o con scarse occasioni di cooperazione con altri processi, e di processi "corti", e/o con frequenti interazioni con altri processi.

Lo scheduling a quanti di tempo utilizza una apposita unità di I/O che funge da *Timer*: a distanza di un quanto di tempo (scandito da un certo numero di cicli di clock dell'unità; ad esempio, circa un milione di cicli di clock per un quanto di tempo dell'ordine del *msec*), tale unità invia una *interruzione* di "fine quanto di tempo": *il suo trattamento* consiste nella *sequenza di azioni per effettuare il prerilascio del processore.*

Il funzionamento a quanti di tempo, insieme al meccanismo delle interruzioni ed all'ordine casuale con cui i processi si pongono in Lista Pronti, contribuisce a rendere ancora sostanzialmente impredicibile l'ordine con il quale i processi utilizzano i processori in un funzionamento in multiprogrammazione o interleaving.

5.4. Indivisibilità

Tutti problemi di sincronizzazione e consistenza di strutture dati condivise, che si possono verificare sull'architettura astratta, si verificano anche su una qualunque architettura concreta. La soluzione di tali problemi sarà di volta in volta specifica dell'architettura concreta considerata.

Si consideri il seguente esempio: un processo A intende svegliare un processo B, e contemporaneamente un processo C intende svegliare un processo D. Nell'architettura astratta A e C provano ad appendere alla Lista Pronti B e D rispettivamente: è evidente che queste due operazioni devono essere eseguite *in modo mutuamente esclusivo* (in un ordine qualsiasi, purchè una dopo l'altra) in quanto, altrimenti, il valore finale della struttura dati Lista Pronti potrebbe risultare impredicibile. Occorre garantire che la *sequenza di azioni* per effettuare la fase di sveglia sia *indivisibile*, cioè che nessun altro processo effettui contemporaneamente, sulla stessa struttura dati, una sequenza di azioni incompatibile. Altri esempi di sequenze di azioni incompatibili se eseguite contemporaneamente, e dunque da rendere indivisibili, sono:

- la manipolazione del buffer, o di altri campi, di un canale di comunicazione da parte del processo mittente e del processo destinatario di comunicazioni;
- la manipolazione della Lista Pronti per eseguire una sveglia e per eseguire contemporaneamente una commutazione di contesto (a seconda della realizzazione della Lista Pronti).

Nell'architettura astratta il meccanismo per rendere indivisibili sequenze di operazioni, eseguite da processori virtuali distinti, *utilizza la struttura di arbitraggio della memoria condivisa*: un processore virtuale segnala all'arbitro (mediante una apposita operazione di *lock*) che intende iniziare una sequenza indivisibile e, alla fine della sequenza, segnala all'arbitro che la sequenza si è conclusa (*unlock*). Durante tutta la sequenza di accessi indivisibile, l'arbitro non permette ad altri processori virtuali di accedere alla memoria o, almeno, alla specifica struttura dati sulla quale è in corso la sequenza indivisibile.

Nell'architettura uniprocessor il problema dell'indivisibilità si presenta allo stesso modo, a causa dell'impredicibilità con la quale avviene l'utilizzazione del processore da parte dei processi eseguibili: come detto, tale impredicibilità è dovuta al meccanismo delle interruzioni, al meccanismo dello scheduling a quanti di tempo ed all'ordine con cui i processi si pongono in Lista Pronti. Nell'esempio in cui A vuole svegliare B e C vuole svegliare D, supponiamo che A sia in Esecuzione e C in Pronto e che, prima di aver concluso la sequenze di azioni per appendere il PCB di B alla Lista Pronti, A venga portato in stato di Pronto in seguito allo scadere del suo quanto di tempo oppure in seguito ad una qualunque altra interruzione che provochi un prerilascio. Se è proprio C ad entrare in Esecuzione, C si trova a lavorare sulla Lista Pronti lasciata da A in uno stato non consistente (il risultato finale probabilmente è che né B né D passeranno in stato di Pronto). Analoghe considerazioni valgono per tutti gli altri esempi.

Nell'architettura uniprocessor, per quanto ora detto il meccanismo per assicurare l'indivisibilità consiste nella *disabilitazione delle interruzioni*.

In una architettura multiprocessor concreta, oltre al meccanismo della *disabilitazione delle interruzioni per ogni processore*, deve anche essere adottata una tecnica, come quella descritta per l'architettura astratta, basata su *operazioni lock-unlock sulla memoria condivisa* per impedire che più processori contemporaneamente effettuino sequenze incompatibili sulle stesse strutture dati. Durante tutto il tempo in cui ad un processore è impedito l'accesso alla memoria, il processo da esso eseguito effettua dunque Attesa Attiva.

5.5. Spazi di indirizzamento

Come discusso a più riprese, lo spazio di indirizzamento di un processo è l'insieme di tutti i possibili indirizzi *logici* che il processo può generare trovandosi in stato di Esecuzione.

Tale spazio comprende gli indirizzi per le seguenti informazioni:

a) codice del programma;

b) dati privati del programma;

c) codice del nucleo: primitive di comunicazione / sincronizzazione usate, scheduling a basso livello, trattamento interruzioni ed eccezioni;

d) strutture dati di nucleo definite dal processo stesso: il proprio PCB, i canali / i semafori da esso utilizzati;

e) strutture dati di nucleo definite da altri processi ma che il processo può trovarsi a utilizzare a causa del funzionamento in multiprogrammazione sull'architettura concreta. Questo aspetto verrà ripreso nella successiva sezione.

Tutte le informazioni che corrispondono ai casi suddetti sono *collegate* dal compilatore nel file che rappresenta l'eseguibile del processo.

5.6. Indirizzamento di strutture dati di nucleo

Riprendiamo il caso *e)* della sezione precedente. Si consideri l'esempio in cui un processo A effettua la commutazione di contesto: A deve staccare dalla Lista Pronti il PCB del primo processo pronto, sia C, e quindi utilizzare certi campi del PCB_C (per inizializzare registri generali e contatore istruzioni con i valori presenti appunto nel PCB_C). Ciò significa che

i) un qualunque processo, come A, deve poter indirizzare il PCB di qualunque altro processo. Dunque tutti i PCB fanno parte dello spazio di indirizzamento di tutti i processi;

ii) ogni processo, come A, utilizza propri indirizzi logici per indirizzare i vari PCB degli altri processi.

Un altro caso si ha nella fase di sveglia: se A sveglia B, A deve poter indirizzare PCB_B. Non solo: per collegare PCB_B alla Lista Pronti, A deve poter indirizzare altri elementi di tale lista, e dunque altri PCB.

Per tener conto del punto *ii)* precedente in modo semplice, nel seguito supporremo che:

iii) tutte le strutture dati condivise che appartengono allo spazio di indirizzamento di tutti i processi, come in particolare i PCB , abbiano lo stesso indirizzo logico per tutti i processi.

Ad esempio, PCB_A ha lo stesso indirizzo logico nello spazio di indirizzamento di A, di B, di C, … In questo modo, se A deve permettere a B di utilizzare PCB_A (ad esempio, passandolo attraverso un canale di comunicazione o un semaforo nel caso A si sospenda), A può scrivere nella struttura condivisa (canale o semaforo) l'indirizzo logico di PCB_A nello spazio di A. Questo coincide con l'indirizzo di PCB_A nello spazio di B, e dunque B può agevolmente utilizzare PCB_A una volta prelevato l'indirizzo dalla struttura condivisa (canale o semaforo).

5.7. Esempio di implementazione di primitive di comunicazione

In alternativa a quanto contenuto nella sez. 1.3.3 del cap. VI, vediamo una implementazione di primitive di comunicazione su canale simmetrico, *valida sia per il caso sincrono che per quello asincrono*.

Per un canale con grado di asincronia $k \geq 0$ ($k = 0$ è il caso sincrono), è prevista una coda FIFO di messaggi. Il numero di elementi di tale coda è $k + 1$, in modo che, nell'esecuzione della *send*, il mittente possa depositare sempre il messaggio in coda e quindi si sospenda nel caso abbia riempito la coda stessa.

La struttura dati *descrittore di canale di comunicazione* è costituita come in figura seguente, dove, per ogni campo, le etichette numeriche indicano gli indici dei campi rispetto alla base del canale.

0. LUN: numero di parole del messaggio;
1. SENDERWAIT: boolean; se uguale a *true* indica che il mittente è in attesa; inizializzato a *false*;
2. RECEIVERWAIT: boolean; se uguale a *true* indica che il destinatario è in attesa; inizializzato a *false*;
3. PCB_SENDER: costante = indirizzo logico del PCB del processo mittente (nell'ipotesi della sez. precedente, questo indirizzo è lo stesso nello spazio logico del mittente e del destinatario);
4. PCB_RECEIVER: costante = indirizzo logico del PCB del processo destinatario (nell'ipotesi della sez. precedente, questo indirizzo è lo stesso nello spazio logico del mittente e del destinatario);
5. N: numero di posizioni della coda, ognuna ampia LUN parole;
6. INS: indice della posizione di BUFFER in cui inserire; inizializzato a zero;
7. ESTR: indice della posizione di BUFFER da cui estrarre; inizializzato a zero;
8. SIZE: integer, numero di posizioni di BUFFER occupate; inizializzato a zero;
9. BUFFER: array di N elementi, ognuno di LUN parole.

I comandi *send* e *receive* sono implementati come procedure i cui parametri sono indirizzi logici della struttura dati descrittore di canale, della variabile messaggio e della variabile targa.

La procedura

send (ch, msg)

con *ch* indirizzo logico del canale e *msg* indirizzo logico del messaggio, ha il seguente funzionamento:

```
{
    { deposita messaggio nel BUFFER };
    if RECEIVERWAIT then {RECEIVERWAIT = false; sveglia destinatario};
    if (SIZE = N) then {SENDERWAIT = true; commutazione di contesto}
}
```

La procedura

receive (ch, var)

con *ch* indirizzo logico del canale e *var* indirizzo logico della *variabile targa* cui assegnare il messaggio, ha il seguente funzionamento:

```
{
    if (SIZE > 0) then { estrai messaggio dal BUFFER e copialo nella variabile targa;
                         if SENDERWAIT then {SENDERWAIT = false; sveglia mittente};
                       }
                  else {RECEIVERWAIT = true; commutazione di contesto}
};
```

Vediamo l'implementazione della *send* in assembler Risc per una architettura uniprocessor.

Supponiamo che la procedura *send* abbia l'indirizzo di ritorno in R40, e che i parametri d'ingresso *ch* e *msg* siano passati attraverso i registri generali R41 e R42.

L'allocazione degli altri registri verrà indicata via via. Nel caso che qualcuno dei registri usati dalla procedura *send* sia usato anche dal programma chiamante, questo avrà provveduto a salvarli in memoria prima della chiamata e, al ritorno dalla procedura, a ripristinarli.

```
/ definizione della procedura send (R40, R41, R42) /

        DI                      /disabilita interruzioni /

/ fase deposita messaggio nel BUFFER /

        LOAD R41, 0, R43        / R43 contiene LUN /
        ADD R41, 9, R45         / R45 contiene l'indirizzo base del BUFFER /
        LOAD R41, 5, R46        / R46 contiene il valore di N /
        LOAD R41, 6, R47        / R47 contiene INS /
        MUL R47, R3, R50        / R50 contiene INS * LUN /
        ADD R45, R50, R45       / R45 contiene ora la base dell'elemento della coda in cui inserire /

    /va ora eseguito un for ripetuto LUN volte; all'i-esima iterazione si copia la i-esima parola del messaggio nella i-esima parola della posizione individuata del BUFFER; alla fine si incrementa INS modulo N e si incrementa SIZE /

        CLEAR R44               / R44 funge da indice del for /
LOOP:   LOAD R42, R44, R48      / R48 : temporaneo per la i-esima parola del messaggio /
        STORE R45, R44, R48     / scrittura della parola del messaggio nella parola di BUFFER /
        INCR R44                / incrementa indice del for /
        IF< R44, R43, LOOP      / if i < LUN goto LOOP /
        INCR R47                / incrementato INS /
        MOD R47, R46, R47       / le ultime due istruzioni calcolano INS = (INS +1) mod N /
        STORE R41, 6, R47       / aggiornato INS nel canale /
        LOAD R41, 8, R47        / ora R47 contiene il vecchio valore di SIZE /
        INCR R47                / incrementato il valore di SIZE /
        STORE R41, 8, R47               / aggiornato SIZE nel canale /
/ fine della fase deposita messaggio nel buffer /

/ fase di controllo stato del destinatario ed eventuale sveglia /

        LOAD R41, 2, R43        / ora R43 è usato per contenere RECEIVERWAIT /
        IF=0  R43, NEXT         / il destinatario non è in attesa; si passa alla prossima fase /
        STORE R41, 2, R0        / il destinatario è in attesa; RECEIVERWAIT è stato rimesso a "false" /
        LOAD R41, 4, R44        / R44 contiene l'indirizzo del PCB del destinatario; è usato per passare il parametro alla
                                  procedura di sveglia processo /
        CALL R30, R32           / R30 contiene sempre l'indirizzo della procedura di sveglia; l'indirizzo di ritorno è in R32 /
/ fine della fase di controllo stato del destinatario ed eventuale sveglia /

/ fase di controllo dello stato del mittente ed eventuale commutazione di contesto /

NEXT:   IF<  R47, R46, FINE     / if SIZE < N goto FINE /
        ADD R0, 1, R49          / il mittente va messo in attesa, R49 contiene "true" /
        STORE R41, 1, R49       / SENDERWAIT = true /
        LOAD R41, 3, R44        / R44 contiene l'indirizzo del PCB del mittente; è usato per passare il parametro alla procedura
                                  di commutazione di contesto /
        CALL R31, R32           / R31 contiene sempre l'indirizzo della procedura di commutazione di contesto; l'indirizzo di
                                  ritorno è in R32; sarà la procedura a riabilitare le interruzioni prima di ritornare /
/ fine della fase di controllo dello stato del mittente ed eventuale commutazione di contesto /

FINE:   GOTO R40, EI            / fine dell'esecuzione della procedura send; ritorno da procedura, con riabilitazione delle
        interruzioni /
```

/ fine della definizione della procedura send (R40, R41, R42) /

6. Esercizi di ricapitolazione sulla parte III

Domanda 1

Un elaboratore general-purpose OPS ha il set di istruzioni Risc del capitolo V arricchito dall'istruzione *INCR_MEM Rbase, Rindice*, che ha come effetto di incrementare di uno il contenuto di una locazione di memoria. La cardinalità del set di istruzioni rimane minore o uguale a 256.

La CPU di OPS, con ciclo di clock τ, ha una cache primaria C1 operante su domanda, metodo associativo su insiemi, capacità di 64K parole, ampiezza del blocco di 8 parole, e scritture gestite con il metodo Write-Through. OPS ha una cache secondaria C2 di capacità 1M parole, interallacciata con 4 moduli collegati direttamente alla CPU, ciclo di clock uguale a 4τ, e latenza di trasmissione dei collegamenti inter-chip di 5τ.

a) Scrivere l'interprete di *INCR_MEM*, inclusa la fase di chiamata e decodifica, e valutarne il tempo medio di elaborazione in funzione di τ e del tempo di accesso in memoria t_a.

b) Si consideri un programma applicativo che opera su due array *A, B*, ognuno di *N* interi, con *N* dell'ordine delle migliaia. Per ogni *i = 0, …, N-1, A[i]* è incrementato di uno se *F(A[i]) > F(B[i])*, con *F* funzione intera disponibile come procedura.

La procedura *F* consta di 10 istruzioni, opera solo su dati in registri generali, ed ha tempo medio di completamento uguale a 50τ. I parametri di ingresso e di uscita sono passati per valore attraverso registri generali.

Si indichi con *p* la probabilità che *F(A[i]) > F(B[i])* per qualsiasi *i*.

Valutare, in funzione di τ, *p* e *N*, il *tempo medio di completamento* del programma *su OPS*, supponendo trascurabile la probabilità di fault della gerarchia di memoria M-C2, con M memoria principale.

Dire e spiegare quanto è ampio *l'insieme di lavoro (working set)* del programma.

c) Dire se la seguente affermazione è vera o falsa, spiegando la risposta: "se C1 rileva un fault di blocco, il trattamento di questo evento può generare fault anche in M".

Domanda 2

Il programma della Domanda 1 è completato in modo che gli array *A, B* siano letti da un certo dispositivo D e che l'array *A* modificato sia scritto sullo stesso dispositivo.

A livello del linguaggio delle applicazioni, D è visto attraverso i comandi *get(n, x)* e *put (n, x)*, dove *x* è il nome di una variabile composta da *n* byte; *get* è il comando per ottenere *x*, e *put* il comando per inviare *x*.

Il linguaggio concorrente con cui è scritto il sistema operativo di OPS è a scambio di messaggi.

a) Mostrare e spiegare la *compilazione* del programma *in un processo* APPL, supponendo che all'unità I/O_D sia associato un processo Driver_D. Per semplicità, non si considerino eccezioni generate dalle operazioni *get* e *put*.

b) Spiegare in seguito a quali eventi il processo APPL può *transire in stato di attesa*, ed in seguito a quali eventi può essere *risvegliato*.