Architettura degli Elaboratori, a.a. 2006-07

Note sul livello dei processi

Queste note completano la trattazione del livello dei processi rispetto a quanto contenuto nelle note integrative sulla Parte Terza.

Vengono definite le primitive di comunicazione per un semplice linguaggio concorrente a scambio di messaggi, fornendo esempi di utilizzo nella compilazione di linguaggi applicativi e nella gestione dei trasferimenti di I/O.

Nella sez. 3 viene completata la trattazione della gestione dei **trasferimenti di ingresso-uscita** e del **trattamento delle interruzioni**: il primo aspetto è ricondotto interamente al supporto di primitive di comunicazione con "processi esterni" (astrazione delle unità di I/O), il secondo aspetto è ricondotto a segnalazioni di sveglia processo effettuate da "processi esterni" nei confronti di processi da eseguire sulla CPU.

1. Linguaggio a scambio di messaggi 1
2. Compilazione di linguaggi applicativi 4
3. Trasferimenti di I/O: processi esterni e interruzioni trattate come sveglie di processi. 6

1. Linguaggio a scambio di messaggi

Il formalismo adottato in queste note è un linguaggio concorrente a scambio di messaggi.

Processi

Un programma concorrente è una collezione di *processi* dichiarata come:

> **parallel** < lista di nomi unici di processi >; < eventuale dichiarazione di nomi parametrici >;
>
> < definizione di processo >;
>
> …
>
> < definizione di processo >;

Il comando *parallel* non può figurare nella definizione dei processi (i processi non sono annidati).

Il *nome unico* di un processo è una qualunque stringa di caratteri. La definizione di un processo ha una struttura del tipo:

> < nome unico del processo > **::**
>
> < dichiarazioni >
>
> < codice >

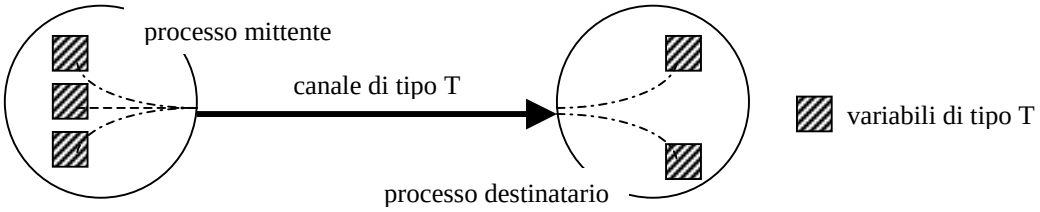
La *parte sequenziale* del linguaggio concorrente è quella di un usuale linguaggio imperativo standard. Per non appesantire la trattazione con aspetti che non sono essenziali per i nostri scopi, nel seguito useremo uno *pseudo*-linguaggio algoritmico con: assegnamento =; costrutti di controllo *if-then-else, case, for, while, repeat*; sequenze di comandi racchiuse tra {…}; procedure e funzioni. Lo tesso dicasi per i tipi di dato (usuali tipi base Pascal / C / Java).

Canali e primitive di comunicazione

I canali di comunicazione sono *con tipo*. Il tipo di un canale è quello dei messaggi che possono esservi inviati e delle variabili targa cui tali messaggi sono assegnati [1]:

Il tipo di un canale è quindi riconosciuto e controllato automaticamente a tempo di compilazione. I messaggi non possono contenere puntatori.

Il modello di comunicazione è con *porte unidirezionali* con identificatore unico, o *modello con nomi unici di canali*. Due primitive corrispondenti, *send* e *receive*, riferiscono lo stesso nome di canale. Il binding delle porte viene dunque effettuato automaticamente a tempo di compilazione, controllando al contempo la coincidenza dei tipi dei messaggi e delle variabili targa.



Il modello di cooperazione di questo linguaggio è *ad ambiente locale*, quindi i canali *non* sono oggetti condivisi (solo la conoscenza dei nomi è a comune di processi partner); ben diverso è dire che il supporto a tempo di esecuzione del linguaggio concorrente, su macchine a memoria condivisa, implementerà i descrittori dei canali come strutture dati condivise.

Il *nome di un canale* è espresso da una qualunque stringa di caratteri; l'implementazione codificherà i nomi come interi. Il nome di un canale è *unico* in tutto il programma espresso da *parallel*.

La sintassi dei comandi per inviare e ricevere messaggi è:

send (nome di canale, valore del messaggio)

receive (nome di canale, variabile targa)

I messaggi e la variabili targa possono essere espressi come *tuple*. Ad esempio: *(operazione, valore1, valore2, indice).*

I nomi dei canali sono dichiarati nei processi che li usano, distinguendo se si tratta di canali *in ingresso* o *in uscita*. Ad esempio:

parallel A, B;

A :: … **channel in** go, ch2; **channel out** compute; …

{ … **send** (compute, …); …; **receive** (go, …); …; **receive** (ch2, …); … }

B :: … **channel in** compute, ch3; **channel out** go, ch4, ch5; …

{ **send** (go, …); …; **receive** (compute, …); … }

[1] Più in generale, i tipi dei messaggi e delle variabili targa, per una stesso canale, possono essere compatibili.

Forme di comunicazione

Le forme di comunicazione possibili in questo linguaggio concorrente sono:

1) *simmetrica* o *asimmetrica in ingresso.* I due casi non vengono distinti da specifiche dichiarazioni. Per quanto detto sopra sui nomi di canali, i possibili mittenti di uno stesso canale asimmetrico sono riconosciuti in fase di compilazione;

2) *sincrona* o *asincrona.* In generale, detto *k* il *grado di asincronia* di un canale, può essere $k \geq 0$, dove il caso $k = 0$ corrisponde alla comunicazione sincrona. *k* deve essere una *costante*. La specifica del grado di asincronia di un canale viene associata alla dichiarazione del nome del canale (valore di *k* scritto tra parentesi dopo il nome del canale); è sufficiente scrivere questa dichiarazione nel solo processo *destinatario*. Nel caso $k = 0$ la specifica può essere omessa. Un esempio di dichiarazione è:

   channel in ch1 (4), ch2 (1), ch3, ch4(0); **channel out** …

   I canali di nome *ch1*, *ch2*, *ch3*, *ch4* hanno grado di asincronia uguale rispettivamente a 4, 1, 0, 0.

   Nel caso di canale *asimmetrico asincrono* con grado di asincronia *k*, *ogni* mittente ha grado di asincronia *k* nei confronti del destinatario.

In casi semplici, il *controllo del nondeterminismo nelle comunicazioni* è esprimibile mediante canali asimmetrici. Esiste una forma più potente e generale, detta dei *comandi alternativi con guardia*, che non rientra comunque negli scopi della presente trattazione.

Nomi di canali come variabili

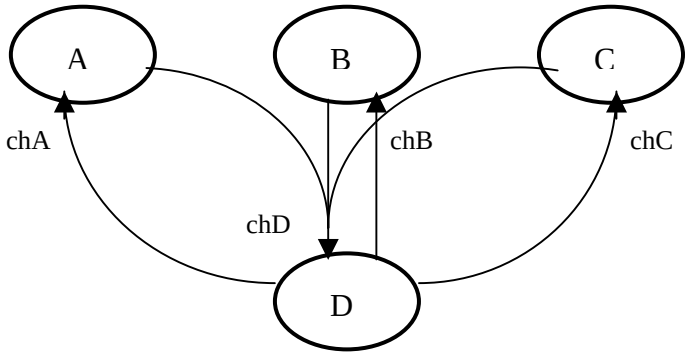
Oltre che costante, il nome di un canale può anche essere una *variabile*. A questo scopo, i tipi del linguaggio comprendono, oltre ai tipi della parte sequenziale, il tipo **channelname**. Nella dichiarazione di una variabile *channelname* viene usata la parola chiave **var**. Nel seguente esempio *ch1* e *ch5* sono nomi costanti, mentre *ch2*, *ch3* e *ch4* sono variabili di tipo *channelname*:

channel in ch1, **var** ch2; **channel out var** ch3, **var** ch4, ch5…

I possibili nomi di canale che possono essere assunti da una variabile *channelname* sono determinabili a tempo di compilazione da una analisi statica del programma parallelo.

Con il meccanismo delle variabili *channelname* si implementa una forma di *comunicazione simmetrica parametrica*.

Sulle variabili di tipo channelname sono definite le *operazione di assegnamento*. Questo permette, in particolare, di gestire parametricamente i canali, usando *nomi di canali in messaggi ed in variabili targa*. Ad esempio in un caso come:



si può scrivere:

parallel A, B, C, D;

A :: … **channel in** chA; **channel out** chD; … {… **send** (chD, (chA, valore)); …; **receive** (chA, variabile); … }

B :: … **channel in** chB; **channel out** chD; … {… **send** (chD, (chB, valore)); …; **receive** (chB, variabile); … }

C :: … **channel in** chC; **channel out** chD; … {… **send** (chD, (chC, valore)); …; **receive** (chC, variabile); … }

D :: … **channel in** chD; **channel out var** ch_out; …

{… **receive** (chD, (ch_out, valore)); …; **send** (ch_out, variabile); … }

I processi A, B, C operano come "clienti" del processo "server" D. Ogni cliente invia a D, su un canale (in questo caso asimmetrico) identificato staticamente (*chD*), messaggi contenenti la richiesta di servizio e il nome del canale su cui desiderano ricevere la risposta al servizio. La variabile *ch_out* del processo D viene assegnata (al valore *chA*, oppure *chB*, oppure *chC*) come variabile targa di un comando *receive*, e potrà essere usata successivamente in un comando *send* per identificare parametricamente il canale (simmetrico) su cui inviare un messaggio (il risultato del servizio) [2].

2. Compilazione di linguaggi applicativi

La compilazione di un programma applicativo produce un processo (o più processi se il programma applicativo è concorrente) che contiene anche tutte le "chiamate al sistema operativo", cioè *invocazioni all'interprete* delle funzionalità di gestione delle risorse necessarie al programma. L'interprete è il sistema operativo stesso.

Ad esempio, consideriamo un sistema operativo *a nucleo minimo*: in esso il nucleo è esclusivamente il **supporto a tempo di esecuzione dei meccanismi di concorrenza**, mentre *tutta la gestione delle risorse (eccetto quella dei processori) è effettuata da appositi processi: gestori*: dello spazio di memoria principale, dello spazio di memoria secondaria, file system, driver delle unità di I/O, gestore delle applicazioni.

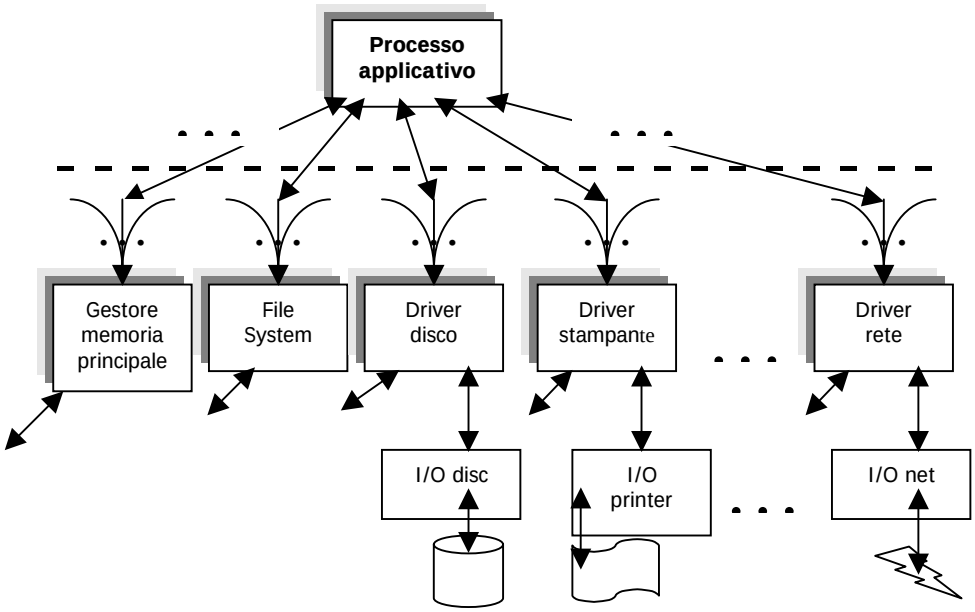
Le stesse unità di I/O sono da considerare processi ("**processi esterni**") eseguiti da processori ad essi dedicati.

Supponiamo che il linguaggio concorrente di sistema quello sia a scambio di messaggi definito in queste note. I processi di sistema prevedono canali di comunicazione (oltre che con altri processi di sistema) con un certo numero *massimo* di processi applicativi. Questo schema permette di realizzare a tempo di compilazione, in modo agevole e modulare, l'aggancio tra processi applicativi e processi di sistema. I processi applicativi, pur essendo in genere creati e distrutti dinamicamente, utilizzano sempre i canali definiti per i processi gestori, come schematizzato nelle figura seguente.

Tra i processi di sistema ai quali, di regola, ogni processo applicativo viene collegato, vanno segnalati quelli che provvedono alla *gestione delle eccezioni*, ad esempio il gestore della memoria principale per il trattamento dell'eccezione di fault di pagina.

---

[2] **IMPORTANTE**: il significato di "richiesta di servizio" e "risposta al servizio" non è primitivo in questo linguaggio concorrente, per il quale la comunicazione "a domanda e risposta" non corrisponde ad un comando specifico, come ad esempio RMI in Java. Nel nostro caso, "domande" e "risposte" sono implementate con comunicazioni indipendenti, in generale via canali asincroni, e senza che il comando *receive*, per l'attesa della "risposta", sia necessariamente il comando immediatamente successivo alla *send* per effettuare la "richiesta".



Il processo derivante dalla compilazione del programma applicativo contiene, in corrispondenza dei comandi per operare sulle risorse, chiamate delle procedure send e/o receive necessarie per effettuare richieste ai processi gestori e ricevere eventuali risposte.

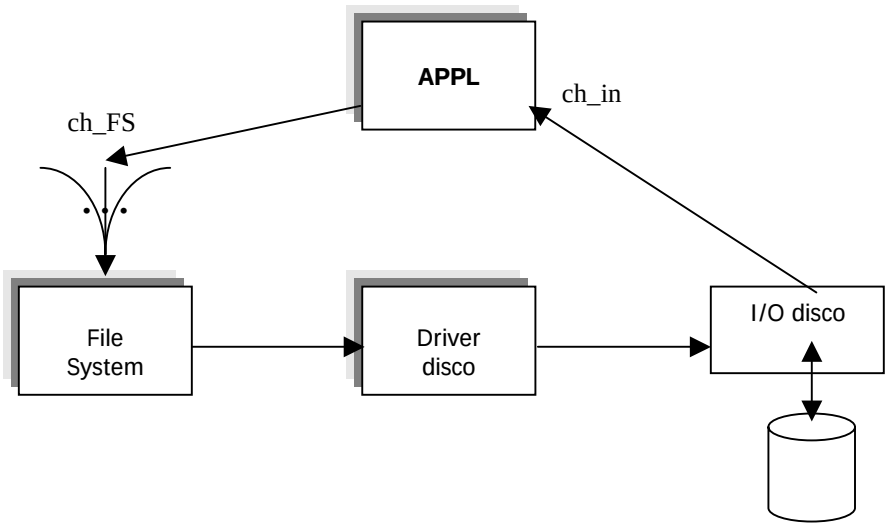
Ad esempio, consideriamo la compilazione di un comando applicativo *read_file(F, N, X)* per la lettura di un file di nome *F*, di dimensione *N* byte, in un array di *N/4* interi.

La compilazione da linguaggio applicativo in linguaggio concorrente di sistema sostituisce a tale comando la sequenza:

APPL **::** … **channel in** ch_in ….; **channel out** ch_FS, ….
…
send (ch_FS, (read, F, N, ch_in));
…
receive (ch_in, (esito, X));
if negativo(esito) **then** < trattamento_eccezione >;
…

dove *ch_FS* è un canale d'ingresso al processo File System mediante il quale il processo applicativo (sia APPL) richiede a tale gestore l'operazione di lettura del file con i relativi parametri. Il messaggio al File System contiene anche il nome del canale *ch_in* dal quale APPL attende il risultato dell'operazione, consistente dell'esito dell'accesso al file e del valore dell'array. Se l'esito è negativo (ad esempio, violazione di protezione, errore di lettura da disco, dimensione errata del file, ecc) viene eseguita una lista di comandi per trattare l'eccezione, altrimenti APPL prosegue nell'elaborazione normale.

Lo schema di tutti i processi interessati può essere quello mostrato nella figura seguente:



Invece di ricevere il messaggio risultato dal File System, è molto più efficiente che il nome del canale *ch_in* sia trasmesso (dal File System tramite Driver del disco) al *processo esterno "I/O disco"* (assieme ad informazioni sull'esito generate dal File System e dal Driver del disco). Attraverso tale canale, il processo I/O disco provvederà a comunicare direttamente con APPL riducendo drasticamente il numero di trasferimenti dei dati. In tal modo, si sfrutta adeguatamente l'architettura in DMA dell'unità di I/O. L'argomento dell'I/O a processi, e l'applicazione a questo esempio, verrà approfondito in una sezione successiva.

La versione di APPL in linguaggio concorrente di sistema verrà, ovviamente, compilata in assembler per dar luogo all'eseguibile.

Si osservi come il "doppio passo di compilazione", prima da linguaggio applicativo a linguaggio concorrente di sistema e poi da quest'ultimo a linguaggio assembler, sia puramente *concettuale*: di fatto è sufficiente il primo passo si riduca soltanto a riconoscere i punti in cui inserire le librerie delle primitive di comunicazione; nel secondo passo anche le librerie delle primitive sono sostituite dalla rispettiva versione assembler (supporto a tempo di esecuzione). Ciò nonostante, *questo schema concettuale è importante per collegare in modo chiaro la tecnologia della compilazione con quella del sistema operativo,*oltre che *per capire come il supporto dei meccanismi di concorrenza possa nascondere tutta una lunga serie di dettagli che, altrimenti, verrebbero trattati come casi a se stanti e tutti diversi* (ad esempio, trasferimenti di I/O).

3. Trasferimenti di I/O: processi esterni e interruzioni trattate come sveglie di processi.

La soluzione ora vista permette anche di risolvere brillantemente i problemi di efficienza nel *trasferimento di grosse moli di dati da dispositivi*, senza affidarsi a funzionalità di più basso livello del sottosistema di I/O, come introdotto alla fine della sez. 2.

Approcci di basso livello

In un caso come l'esempio della sez. 2, operando secondo un approccio "tradizionale" al trattamento dell'I/O, all'unità di I/O viene passato l'indirizzo fisico della struttura dati X e il processo APPL viene fatto passare in attesa con una primitiva ad hoc. L'unità di I/O effettua, in

DMA, la scrittura dei dati in X ed invia una interruzione con la quale comunica l'esito del trasferimento ed una informazione che permette di capire per quale processo è stato effettuato il trasferimento stesso. Il trattamento dell'interruzione risveglia APPL e passa il valore dell'esito in una sua variabile. Questo approccio è basato su meccanismi di basso livello non integrati in un formalismo uniforme; un qualunque cambiamento nelle unità di I/O, o nel processore, o nelle librerie del nucleo, o nel trattamento delle interruzioni, o nei driver obbliga a riprogettare gran parte del sistema e a modificare apprezzabilmente il compilatore.

Processi esterni e schema uniforme per il trattamento interruzioni

Un primo passo verso una soluzione più strutturata è quello di considerare le unità di I/O come *processi* ("processi esterni"), cooperanti con gli altri processi mediante le stesse primitive del linguaggio concorrente di sistema.

Per prima cosa, occorre aver chiaro che (in base al concetto generale di "modulo di elaborazione") ogni unità di I/O può realmente essere considerata un processo: il processo può corrispondere al funzionamento del microprogramma che descrive l'unità stessa, nel caso di una unità specializzata a firmware, oppure può essere l'oggetto della compilazione da linguaggio concorrente in assembler nel caso che l'unità di I/O sia realizzata come un nodo di elaborazione (CPU, memoria locale) con tecnologia general-purpose. La *condivisione della memoria* tra processi I/O e processi eseguibili dal processore centrale è ottenuta con la tecnica del DMA e/o del Memory Mapped I/O.

La sveglia, da parte di processi di I/O, di processi eseguibili dal processore centrale avviene attraverso il meccanismo delle interruzioni, **il cui trattamento può quindi essere sempre ricondotto alla funzionalità di sveglia processo**. *L'interruzione viene inviata* **solo** *rilevando che deve essere effettuata la sveglia di un processo, e non comunque alla fine di un trasferimento dati.*

Una soluzione intermedia

In queste ipotesi (I/O a processi), un modo usuale per realizzare il trasferimento di I/O dell'esempio della sez. 2 è il seguente: APPL passa nel messaggio anche il riferimento alla variabile X; questo riferimento può essere un indirizzo logico, se questo è significativo per il processo I/O, altrimenti una sua trasformazione opportuna, oppure l'indirizzo fisico tradotto da APPL a programma. Il processo I/O effettua la scrittura dei dati in DMA, dopo di che esegue la primitiva *send* ad APPL per comunicare l'esito del trasferimento. L'implementazione della *send* di I/O provvederà, se APPL era in attesa, ad inviare al processore l'interruzione di sveglia, accompagnata dal riferimento al PCB del processo da svegliare (riferimento ricavato nel descrittore di canale).

Questa soluzione utilizza in parte le primitive del linguaggio concorrente per far cooperare APPL e gli altri processi con il processo I/O, *ma utilizza ancora altri meccanismi a basso livello*, nell'esempio quelli per effettuare il trasferimento dati in DMA in modo esplicito, o per manipolare indirizzi.

Soluzione finale: trasferimento di I/O interamente implementato nel supporto delle comunicazioni tra processi

La soluzione definitiva è quella di *utilizzare esclusivamente primitive del linguaggio concorrente* anche nella descrizione del processo I/O. Il programma che ne risulta è quello mostrato nella sez. 2. Un potenziale inconveniente è che, con questa descrizione, il file sia copiato due volte, rispetto all'unica copia effettuata nelle versioni precedenti che utilizzano, in tutto o in parte, meccanismi di basso livello.

Questo problema è risolto con una *implementazione delle comunicazioni tra processi, **ottimizzata** rispetto a quanto visto nelle Note sulla Parte Terza (sez. 5.7)*: nel caso che il processo destinatario sia in attesa, l'implementazione della primitiva *send* provoca la **copia del messaggio direttamente nella variabile targa**, provvedendo anche a svegliare il destinatario stesso; in tal modo, quando il destinatario ritorna in esecuzione, trova già il messaggio nella variabile targa senza rieseguire la *receive*.

Questa ottimizzazione vale *in generale* per qualunque canale di comunicazione. In particolare, nel caso dell'I/O, una volta che APPL si è messo in attesa del risultato (*receive* dal processo I/O), l'esecuzione della *send* da parte del processo I/O provocherà la copia dei blocchi del file direttamente nella variabile targa di APPL, *minimizzando così il numero di copie come nelle soluzioni di più basso livello*.

Il punto importante è che vengono eseguite le stesse azioni che, nell'altra soluzione, venivano effettuate con meccanismi di basso livello (passaggio esplicito dell'indirizzo di X, trasferimento dei dati in DMA), solo che ora *tali azioni non sono descritte esplicitamente nei processi, ma sono effettuate in quanto previste dall'implementazione del supporto alle primitive di concorrenza*: l'indirizzo di X non è passato esplicitamente da APPL, ma viene reperito nel descrittore di canale; la copia in DMA dei dati nella variabile X è effettuata in seguito all'ottimizzazione del supporto della *send* che prevede la scrittura nella variabile targa. Inoltre, rilevando APPL in attesa, il supporto della *send* di I/O provvede a trasmettere una interruzione ed il messaggio di interruzione contiene il riferimento del PCB del processo APPL da svegliare.

In conclusione, con lo schema visto, l'efficienza dell'implementazione rimane inalterata pur passando da una soluzione non strutturata (I/O gestito in modo indipendente dal linguaggio concorrente) ad una strutturata (*I/O a processi e trasferimenti di I/O espressi interamente nel supporto delle primitive di comunicazione*).