I processi

G. Lettieri

9 Maggio 2017

1 I processi

L'astrazione più importante introdotta dal sistema è quella dei *processi*. Un processo è un programma in esecuzione. Riporto qui una vecchia riposta della mailing list sulla distinzione tra programmi e processi:

> Proviamo ad illustrare la differenza con una semplice metafora: in una pizzeria, il pizzaiolo riceve una ordinazione. Il pizzaiolo è inesperto e ha bisogno di avere la ricetta della pizza davanti a se. Stende la pasta, la condisce, la inforna, aspetta che sia cotta, la farcisce e serve la pizza.
>
> Cos'è il programma?
>
> Sono sicuro che nessuno di voi ha dubbi: il programma è la ricetta.
>
> Cos'è il processo?
>
> Qui credo che molti risponderanno: il pizzaiolo. No. Il pizzaiolo è il processore, cioè l'entità che interpreta la ricetta e la esegue.
>
> Allora è la pizza? No. La pizza sono i dati da elaborare.
>
> Il processo è un concetto un po' più astratto. È la *sequenza di stati* che il sistema "pizza + pizzaiolo" attraversa, passando dalla pasta alla pizza finale, secondo le istruzioni dettate dalla ricetta, eseguite pedissequamente dal pizzaiolo inesperto.
>
> Uscendo dalla metafora, un processo è un programma in esecuzione su dei dati di ingresso. Questa esecuzione la possiamo modellare come la sequenza degli stati attraverso cui il sistema processore + memoria passa eseguendo il programma, su quei dai, dall'inizio fino alla conclusione. Notate che questa definizione si applica bene ai programmi di tipo *batch*, in cui gli ingressi vengono specificati tutti all'inizio, e il processo (generato dal programma in esecuzione su quei dati) prosegue indisturbato fino ad ottenere le uscite (ad esempio, pensate ad un programma per ordinare alfabeticamente un file). Una volta afferrato il concetto, però, in questo caso semplice, credo che non vi sarà difficile estenderlo ai programmi "interattivi" a cui ormai

siete più abituati (praticamente tutti i programmi con interfaccia grafica, ma non solo quelli).

A prima vista, il processo potrebbe sembrare molto simile al programma: la ricetta dice "stendere la pasta" e nel processo vediamo il pizzaiolo stendere la pasta; nel rigo successivo la ricetta ricetta dice "versare il condimento" e nel processo vediamo, subito dopo, il pizzaiolo versare il condimento. È molto semplice confondere i due concetti, soprattutto se il programma è molto semplice, ma si tratta di due cose completamente distinte, per i seguenti motivi:

- uno stesso programma può essere associato a più processi: tanti clienti, in genere, chiedono lo stesso tipo di pizza. In questo caso, il programma è sempre lo stesso (la ricetta per quel tipo di pizza), ma ad ogni pizza corrisponde un processo distinto, che si svolge autonomamente nel tempo.
- in generale, non è esclusivamente il programma (la ricetta) a decidere attraverso quali stati il processo dovrà passare, ma anche la richiesta del cliente(l'input): la ricetta potrebbe, infatti, prevedere delle varianti (un `if`), che il cliente dovrà specificare. La ricetta conterrà istruzioni per entrambe le varianti (con o senza carciofi), ma un particolare processo seguirà necessariamente *una sola* variante.

Ma c'è dell'altro, nella metafora del pizzaiolo, che ci può aiutare a capire altri punti fondamentali. Il processo, se lo guardiamo nella sua interezza, si svolge necessariamente nel tempo. Possiamo anche, però, guardarlo ad un certo istante, facendone una fotografia. La fotografia che ne facciamo deve contenere tutte le informazioni necessarie a capire come il processo si svolgerà nel seguito. Nel nostro esempio, la foto dovrà contenere:

- la pizza nel suo stato semilavorato (la memoria dati del processo);
- il punto, sulla ricetta, a cui il pizzaiolo è arrivato (il contatore di programma);
- tutte le altre informazioni che permettono al pizzaiolo di proseguire (da quel punto in poi) con la corretta esecuzione della ricetta, come, ad esempio, il tempo trascorso da quando ha infornato la pizza (i registri del processore).

Se la fotografia è fatta bene, contiene tutto il necessario per sospendere il processo e riprenderlo in un secondo tempo. Il pizzaiolo fa questa operazione continuamente, quando condisce, a turno, più pizze, o quando lascia una serie di pizze nel forno e, nel frattempo, comincia a prepararne un'altra (multiprogrammazione).

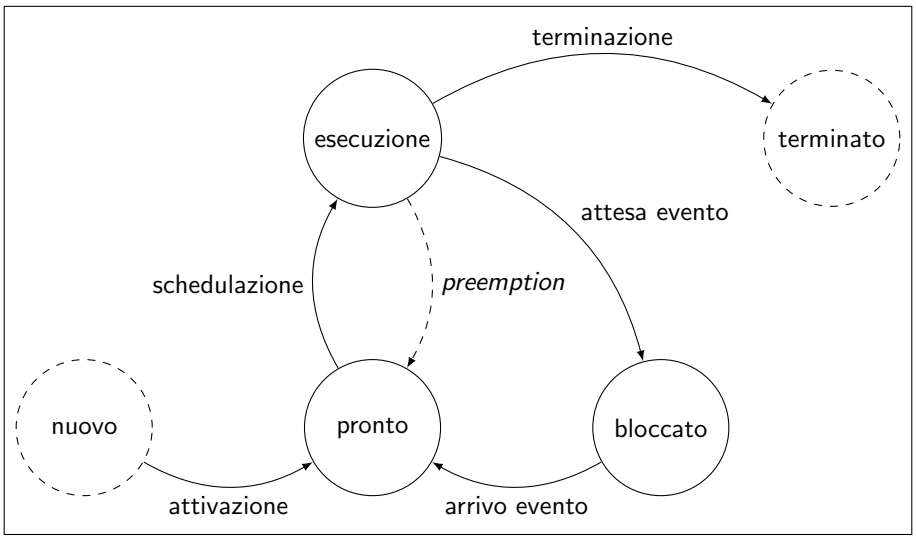

Figura 1: Stati di un processo.

1.1 Stati dei processi

Durante la sua vita un processo si trova in uno degli stati illustrati in Figura 1.

I processi devono essere prima di tutto "attivati", in modo che possano cominciare ad essere eseguiti. L'attivazione comporta la creazione di tutte le strutture dati necessarie (descrittore di processo, pile, etc.). In alcuni sistemi i processi da attivare sono decisi staticamente all'avvio del sistema. Noi realizzeremo il caso in cui i processi possono essere creati dinamicamente da altri processi (tranne ovviamente il primo processo, che sarà creato dal sistema stesso all'avvio).

Se un processo si trova nello stato "esecuzione", il processore sta eseguendo le sue istruzioni. Se abbiamo un solo processore (come stiamo supponendo in tutto il corso), un solo processo per volta può trovarsi in questo stato.

Mentre si trova in esecuzione, un processo può chiedere di terminare o di sospendersi in attesa di un evento. Esempi di evento sono il completamento di una operazione di I/O, o il passaggio di un determinato intervallo di tempo o anche, come vedremo, l'arrivo di un generico "segnale" da parte di un altro processo. Quando l'evento atteso si verifica, il processo diventa "pronto" (può anche accadere che vada direttamente in esecuzione, anche se ciò non è mostrato esplicitamente in figura).

I processi che si trovano nello stato "pronto" sono processi che potrebbero proseguire se avessimo a disposizione sufficienti processori: sono fermi solo perché il processore è già impegnato a portare avanti un altro processo. Un processo pronto può essere scelto per andare in esecuzione tramite una operazione che è detta di "schedulazione". Ovviamente, il processo che era precedentemente in esecuzione deve aver prima liberato il processore, per esempio chiedendo

di bloccarsi e passando nello stato "bloccato". Un processo in esecuzione può anche essere costretto a liberare forzatamente il processore, con una azione che è detta di "*preemption*" (prelazione). La preemption può avvenire sia in seguito ad una interruzione, sia come effetto collaterale di una azione richiesta dal processo in esecuzione stesso (vedremo che nel nostro caso ciò può accadere quando un processo invia un segnale ad un altro che lo stava aspettando). Il caso di preemption differisce dal caso di blocco, in quanto il processo passa nello stato "pronto" e non nello stato "bloccato". Nei sistemi senza preemption un processo può occupare il processore indefinitamente (per esempio, eseguendo un ciclo infinito) senza lasciare mai il processore agli altri processi.

Sono possibili tante trategie di schedulazione. Nella strategia *time-sharing* (a divisione di tempo), per esempio, il processore viene assegnato ad ogni processo pronto per un tempo massimo, passato il quale un timer interrompe il processore causando una preemption con schedulazione del prossimo processo pronto. Noi realizzeremo un sistema a *priorità fissa*: ad ogni processo è assegnata un priorità numerica al momento della creazione, e il sistema si impegna a garantire che, ad ogni istante, si trovi in esecuzione il processo che ha la massima priorità tra tutti quelli pronti. Questo ci permette di dover eseguire una azione di schedulazione solo quando un processo passa "esecuzione" a "bloccato" oppure da "bloccato" a "pronto". Nel primo caso il processore si libera, e dunque dobbiamo mettere in esecuzione il processo a maggiore priorità tra i pronti. Nel secondo caso c'è un novo processo pronto, che potrebbe avere priorità maggiore di quello attualmente in esecuzione: per rispettare la regola che abbiamo promesso di garantire potremmo fare preemption sul processo in esecuzione. Si noti che anche quando un processo P_1 ne attiva un altro P_2 ci troviamo in una situazione simile: abbiamo un nuovo processo pronto (P_2) mentre un altro (P_1) è in esecuzione. Noi però garantiremo che i processi non possano attivarne altri a priorità maggiore della propria, quindi non sarà mai necessaria una preemption in questo caso.

1.2 Realizzazione dei processi

Ogni processo ha un proprio descrittore di processo, una propria pila sistema e una propria memoria (contentente il suo codice, i suoi dati e la sua pila utente). Lo scopo del descrittore di processo è quello di contenere tutte le informazioni che il sistema deve ricordare relativamente al processo. In particolare, il descrittore contiene un campo "contesto" destinato a contenere l'ultima "fotografia" scattata sullo stato del processore (vale a dire, il contenuto di tutti i registri del processore). Si noti che una foto completa dello stato del processo deve contenere anche lo stato di tutta la memoria. Per il momento supponiamo che lo stato di tutta la memoria del processo sia conservato su un hard disk. Nel descrittore di processo ci saranno le informazioni necessarie a recuperare tale stato dall'hard disk.

Dal momento che abbiamo comunque deciso di avere un TSS per ogni processo, in modo che ogni processo abbia una sua pila distinta, utilizziamo i TSS stessi come parte dei descrittori di processo. La tabella GDT contiene una entrata per ogni processo attivo. L'entrata relativa ad un dato processo punta al TSS

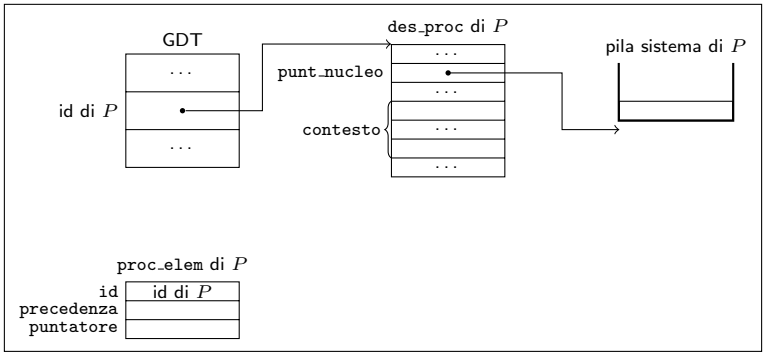

Figura 2: Strutture dati del sistema associate ad ogni processo.

di quel processo, che dunque è anche il suo descrittore. Possiamo identificare ciascun processo tramite l'offset della sua entrata nella GDT.

Per gestire gli stati di Figura 1 faremo ricorso a *code di processi*. Gli elementi delle code saranno oggetti di tipo `proc_elem` con un campo contenente l'identificatore di un processo (`id`), un capo contenente la priorità del processo (`precedenza`), e un puntatore al prossimo `proc_elem` in coda. La Figura 2 mostra le strutture dati che il modulo sistema deve associare ad ogni processo (con l'esclusione delle strutture dati relative alla memoria del processo). Per il descrittore di processo definiamo una struttura `des_proc` con un campo `contesto` dove memorizzare il contenuto dei registri del processore. Il campo `contesto` è un array di `natq` (interi senza segno a 64 bit). Ogni registro ha un posto assegnato all'interno di questo array: definiremo delle costanti (`I_RAX`, `I_RBX`, etc.) per evitare di ricordare gli indici numerici.

Si noti che sia la GDT che la parte superiore del descrittore di processo (coincidente con il TSS) hanno un formato che è stabilito dall'hardware, più precisamente dal meccanismo di interruzione. Ricordiamo che il meccanismo di interruzione, nel caso in cui debba cambiare la pila corrente, prende l'indirizzo della nuova pila dal descrittore di processo puntato dal registro TR. Per fare in modo che tale meccanismo utilizzi la pila sistema che abbiamo allocato per P, dobbiamo

- predisporre un campo `punt_nucleo` che punti alla pila sistema di P e si trovi, all'interno del descrittore di processo, all'offset a cui lo leggerà l'hardware;
- cariare `TR` con l'id di P quando P è in esecuzione.

Dal punto di vista del software, il processo attualmente in esecuzione sarà memorizzato nella variabile globale `esecuzione`, di tipo puntatore a `proc_elem`. I processi pronti si troveranno in una coda globale, la cui testa sarà puntata dalla variable `pronti`, anch'essa di tipo puntatore a `proc_elem`. Predisporremo inoltre una coda diversa per ogni tipo di evento atteso dai processi bloccati.

Manterremo tutte queste code ordintate in base al campo priorità. In questo modo, in particolare, la schedulazione di un processo pronto può essere eseguita semplicemente estraendo il `proc_elem` in testa alla coda `pronti`.

Per evitare di dover gestire in modo speciale il caso in cui tutti i processi sono bloccati è sufficiente che il sistema crei un processo a priorità minima che sia sempre pronto, detto processo `dummy`. Tale processo può eseguire un ciclo infinito. Nel nostro caso il processo `dummy` controllerà ciclicamente il numero di processi attualmente esistenti nel sistema. Quando scopre che tutti gli altri processi sono terminati, il processo `dummy` può eseguire lo *shutdown* del sistema.

1.3 Cambio di processo

Il cambio di processo può avvenire solo quando il processo in esecuzione si porta a livello sistema, cosa che può avvenire solo nei seguenti modi:

- se il processo stesso esegue una istruzione `INT`;
- se il processore genera una eccezione (per es., page fault);
- se il processore accetta una interruzione esterna.

In tutti e tre i casi il processore esegue azioni simili: consulta una entrata ("cancello") della tabella IDT per prelevare l'indirizzo a cui saltare, salvando in pila l'indirizzo a cui ritornare. In base ai flag contenuti nel cancello, il processore può eseguire anche altre azioni: innalzare il livello di privilegio del processore (in base al valore del campo L); disabilitare le interruzioni (in base al valore del campo I/T). Se deve innalzare il livello di privilegio, provvede anche a cambiare pila (prelevando il puntatore alla nuova pila campo `punt_nucleo` del descrittore di processo puntato dal registro `TR`). Tutti i cancelli del nostro sistema prevedono l'innalzamento del livello di privilegio, quindi in tutti e tre i casi il processore passerà ad usare la pila sistema del processo corrente. In questa pila salverà il puntatore alla vecchia pila, il contenuto del registro `RFLAGS`, il livello di privilegio precedente l'innalzamento e l'indirizzo della prossima istruzione da eseguire. Vedremo anche che tutti i cancelli che portano al modolo sistema prevedono la disabilitazione delle interruzioni (il campo I/T deve dunque specificare il tipo Interrupt e non Trap).

Inoltre, predisporremo le cose in modo che il codice a cui si salta in tutti e tre i casi segua il seguente schema:

```
CALL salva_stato
...
CALL carica_stato
IRETQ
```

L'ingresso nel modulo sistema, subito dopo le operazioni svolte dal meccanismo di interruzione, passa quindi per la funzione `salva_stato`. L'uscita dal modulo sistema (con successivo ritorno al modo utente tramite `IRETQ`) passa invece dalla funzione `carica_stato`. La funzione `salva_stato` salva lo stato del processore

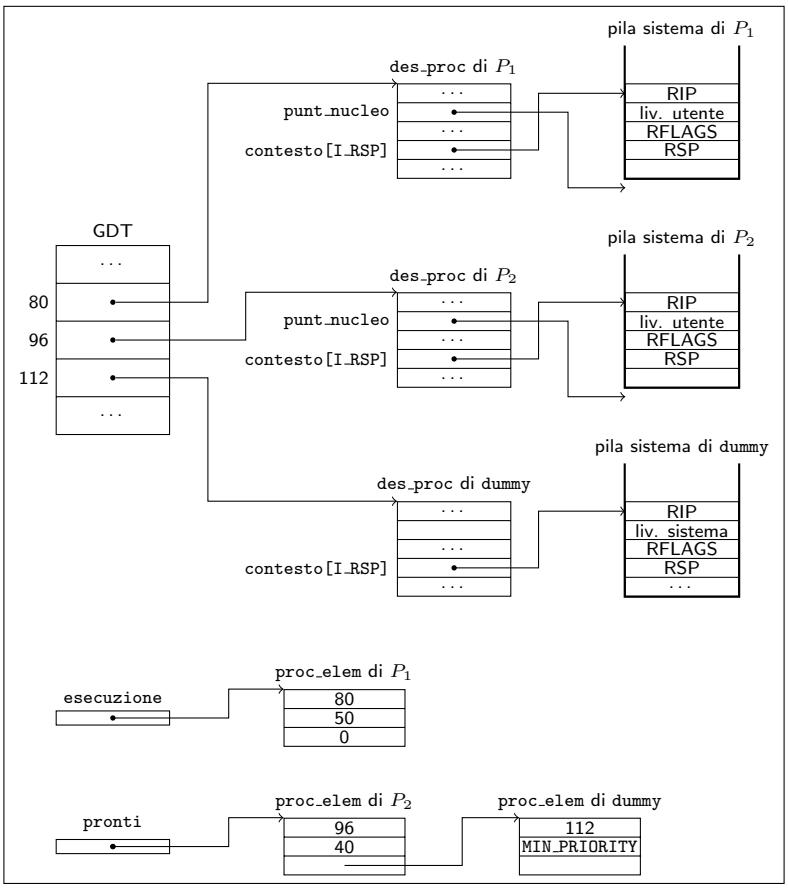

Figura 3: Esempio di istanziazione delle strutture dati per la gestione dei processi.

nel descrittore del processo identificato dalla variable `esecuzione`, mentre la seconda carica nel processore lo stato contenuto nel descrittore del processo identificato da `esecuzione`. Vedremo in seguito che in alcuni casi le chiamate a `salva_stato` e `carica_stato` si possono, oppure *devono*, essere omesse, ma per il momento possiamo assure che siano sempre presenti.

In Figura 3 mostriamo un esempio con due processi, P_1, P_2 (oltre al processo `dummy`, sempre presente) nel momento in cui P_1, che è in esecuzione, si porta a livello sistema per uno qualunque dei tre motivi di cui sopra, mentre P_2 era pronto. La Figura mostra lo stato delle strutture dati dopo il ritorno dalla `CALL salva_stato`.

La "fotografia" dello stato di P_1 e P_2 è costituita in parte dalle informazioni salvate in pila sistema dal meccanismo di interruzione (in particolare, il contenuto dei registri `RIP` e `RSP`) e in parte dal contenuto del descrittore di processo

come aggiornato dalla `salva_stato`.

Attenzione ai due valori salvati di `RSP`: quello che fa parte dello stato del processo è quello che punta alla pila utente e che si trova salvato in pila sistema. La `salva_stato` salva anche (nel campo `contesto[I_RSP]` del descrittore di processo) il contenuto del registro `RSP` come lasciato dal processore dopo il cambio di pila e il successivo salvataggio in questa delle cinque parole lunghe. La funzione `carica_stato` ripristinerà anche questo registro in modo che l'istruzione `IRETQ` finale possa estrarre dalla pila sistema queste informazioni, facendo dunque proseguire il processo dal punto in cui era stato interrotto.

Si noti che, quando il processore sta eseguendo codice del modulo sistema, *tutti* i processi si trovano a livello sistema: non solo quello in esecuzione, ma anche tutti i pronti e tutti quelli bloccati. Infatti, se questi erano stati precedentemente in esecuzione, l'unico modo in cui possono esserne usciti è passando dal livello sistema, in uno dei tre modi possibili. Tutti questi saranno dunque passati dal meccanismo di interruzione e dalla `salva_stato`, e dunque i loro descrittori di processo e pile sistema saranno in una configurazione adatta ad essere interpretati da una `carica_stato` seguita da una `IREQT`[1] In Figura 3 questo è illustrato dal fatto che tutte le pile sistema contengono lo stesso tipo di informazioni. Per passare da un processo ad un altro è dunque sufficiente cambiare il valore della variable `esecuzione` in un momento qualunque tra la `CALL salva_stato` e la `CALL carica_stato`. La `carica_stato` finale caricherà (insieme agli altri registri) il valore di `RSP` che punta alla pila sistema del nuovo processo, e la successiva `IRETQ` farà ripartire quest'ultimo. In questo modo possiamo entrare nel sistema eseguendo un processo e, al ritorno, saltare ad un altro.

1.4 Attivazione di un processo

Nel nostro sistema un processo ne può attivare un altro invocando una opportuna primitiva. Questa dovrà fare in modo che il nuovo processo, quando verrà messo in esecuzione la privma volta, parta da un istruction pointer `rip_ini`, con una pila utente puntata da `rsp_ini`.

Per attivare un processo è necessario allocare e inizializzare tutte le sue strutture dati: descrittore di processo (`des_proc`), `proc_elem` e pila sistema. Per capire come inizializzarle, pensiamo a cosa accadrà al processo dopo averlo attivato: sarà inserito in coda pronti (si veda la Figura 1), dove prima o poi verrà schedulato e portato in esecuzione. Sappiamo che ciò avverrà facendo puntare `esecuzione` al nuovo `proc_elem`, quindi eseguendo `CALL carica_stato` e infine `IRETQ`. Per i processi che erano arrivati in coda pronti essendo stati precedentemente in esecuzione, il descrittore letto dalla `carica_stato` e la pila sistema letta dalla `IRETQ` erano stati opportunamente inizializzati (dalla `salva_stato` e dal meccanismo di interruzione) l'ultima volta che il processo era uscito dallo stato esecuzione. Se vogliamo che il nuovo processo si comporti come tutti gli altri che sono già in coda pronti, possiamo inizializzare il descrittore di proces-

[1] Per i processi che non erano ancora mai andati in esecuzione si veda la sezione successiva.

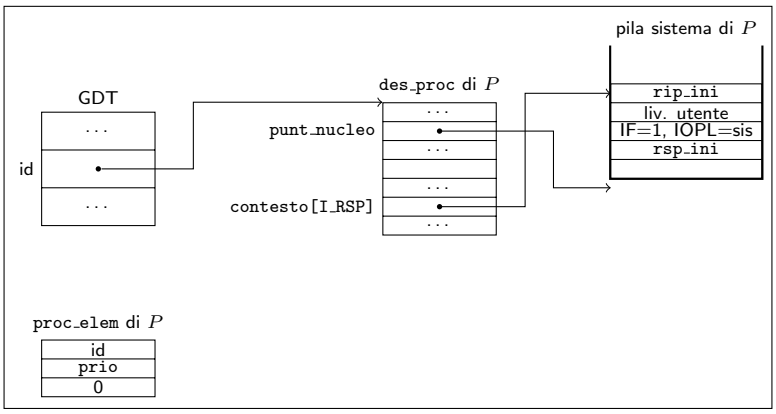

Figura 4: Attivazione di un processo utente. Tutte le strutture dati mostrate si trovano nella parte M1 della memoria.

so e pila sistema *come se* anch'esso si fosse precedentemente portato da livello utente a livello sistema, subito prima di eseguire la sua prima istruzione (cioè quella all'indirizzo `rip_ini`). La Figura 4 mostra il risultato finale. In pila sistema scriviamo `rip_ini`, il codice del livello utente, un campo `RFLAGS` cof flag IF=1 e IOPL=sistema, e `rsp_ini`. Inizializziamo il campo `contesto[I_RSP]` in modo che, in seguito alla `carica_stato`, il registro `RSP` del processore punti alle informazioni che abbiamo scritto in pila sistema. In questo modo la `IRETQ` che segue sempre la `CALL carica_stato` farà saltare il processore all'istruzione di indirizzo `rip_ini`, a livello utente, con le interruzioni abilitate (e l'impossibilità di disattivarle) e pronto a usare la pila utente puntata da `rsp_ini`.

Si noti che il campo `punt_nucleo` deve puntare comunque alla base della pila sistema come se questa fosse vuota. Questo campo, infatti, verrà utilizzato dal meccanismo delle interruzioni quando il processo sarà ormai in esecuzione a livello utente. Quando un processo si trova a livello utente, la sua pila sistema è sempre vuota: si riempie passando da utente a sistema e si svuota al ritorno.

Per completare l'attivazione dobbiamo anche occupare una nuova entrata della GDT, scrivendovi dentro il puntatore al nuovo `des_proc`. L'offset di questa entrata nella GDT funge anche da identificatore del processo. Scriviamo l'identificatore nel campo `id` del nuovo `proc_elem`. Il campo `precedenza` del `proc_elem` deve contenere la priorità (`prio`) del nuovo processo. Nel sistema finale questa sarà un ulteriore parametro da passare alla primitiva che attiva i processi.

1.5 Esempio

La Figura 5 mostra un esempio di una possibile evoluzione del sistema con due processi, P_1 e P_2, con P_1 che ha una priorità maggiore di P_2. Durante l'inizializzazione supponiamo che vengano attivati sia P_1 che P_2 e inseriti in coda pronti.

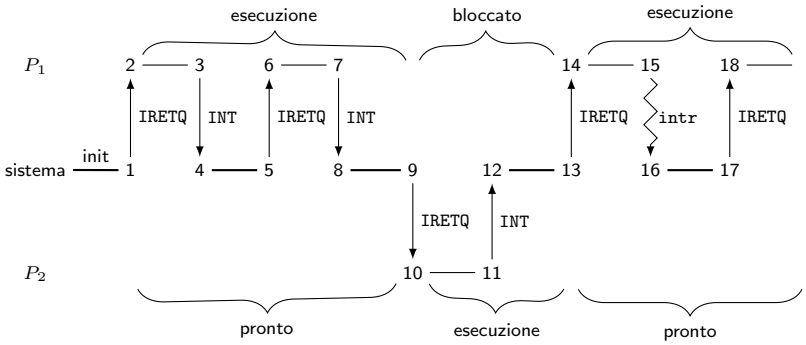

Figura 5: Un esempio di evoluzione del sistema con due processi.

All'istante 1 il sistema esegue la `IRETQ` che salta alla prima istruzione di P_1. All'istante 3 P_1 invoca una primitiva del sistema tramite una istruzione `INT`. Subito dopo, all'istante 4, il sistema salva lo stato di P_1 nel suo descrittore di processo. Una volta terminata, la primitiva torna al processo P_1 senza cambiare processo, all'istante 5. Si noti che il descrittore di processo viene aggiornato solo quando un processo si porta da livello utente a sistema. Nel caso di P_1 ciò avviene agli istanti 4, 8, e 16. Per tutto il resto del tempo il descrittore di processo continua a memorizzare l'ultimo stato salvato. Inoltre, si faccia attenzione a *quale* stato viene salvato: all'istante 4 viene salvato lo stato che permette a P_1 di ripartire dal punto 6, dove ci sarà la prima istruzione successiva alla `INT` che P_1 aveva eseguito all'istante 3. Ciò è semplice ed intuitivo da ricordare quando il sistema *non* cambia processo tra la `salva_stato` e la `carica_stato`. Ma si ricordi che ciò avviene sempre: all'istante 7 vediamo P_1 invocare di nuovo una primitiva, portandosi a livello sistema (istante 8). La situazione delle strutture dati del sistema è ora quella descritta in Figura 3. Il RIP salvato nalla pila sistema di P_1 è quello che punta all'istruzione successiva alla `INT` appena eseguita. Questa volta supponiamo che la primitiva blocchi P_1 e scheduli P_2: la pila sistema attiva (cioè, puntata dal registro RSP del processore) diventa quella di P_2 e la `IRETQ` eseguita all'istante 9 ritorna a P_2 (alla sua prima istruzione, dal momento che P_2 non era ancora mai andato in esecuzione). Successivamente, P_2 invoca anch'esso una primitiva, all'istante 11. All'istante 12 il sistema salva il nuovo stato di P_2 e decide di rimettere in esecuzione P_1 (istante 13). Da dove riparte P_1? Dall'ultimo stato salvato, che è sempre quello salvato all'istante 8. Questo stato fa ripartire P_1 dall'istruzione successiva alla `INT` che aveva eseguito all'istante 7. In particolare, P_1 riparte in stato utente (istante 14), come se fosse tornato adesso dalla primitiva che aveva invocato in 7.

All'istante 15 il processore accetta una interruzione esterna e si porta in modo sistema. Anche questa volta viene salvato il nuovo stato di P_1. In questo caso, il RIP salvato in pila sistema punta all'istruzione successiva all'ultima eseguita prima dell'accettazione dell'interrupt (si ricordi che il processore ascolta gli interrupt solo dopo aver terminato una istruzione). In 17 il sistema termina la

gestione dell' interrupt e torna in modo utente senza cambiare processo, quindi la `IRETQ` torna in P_1 in 18.

Si noti che anche la routine di interruzione usa la pila sistema del processo che si trovava in esecuzione al momento dell'accettazione dell'interruzione.