Realizzazione dei processi

G. Lettieri

5 Aprile 2022

1 I processi

Vediamo ora come i processi sono realizzati in generale, e poi come sono stati implementati nel sistema didattico.

1.1 Stati di esecuzione dei processi

Durante la sua vita un processo si trova in uno degli *stati di esecuzione* illustrati in Figura 1[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 in "esecuzione", il processore sta eseguendo le sue istruzioni: il processo ha il controllo del processore e lo stato del processo (contenuto dei registri e della memoria) cambia nel tempo. Se abbiamo un solo processore (come stiamo supponendo in tutto il corso), un solo processo per volta può trovarsi in esecuzione. In tutti gli altri stati di esecuzione lo stato del processo resta costante nel tempo.

Mentre si trova in esecuzione un processo può chiedere di terminare, oppure 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 (di esecuzione) "pronto" sono processi che potrebbero proseguire se avessimo a disposizione sufficienti processori: sono

[1] Attenzione a non confondere lo stato *di esecuzione* di un processo (che è uno tra nuovo, pronto, esecuzione, bloccato e terminato) con lo *stato del processo*, di cui abbiamo parlato precedentemente, e che consiste nel contenuto dei registri e della memoria in un qualche punto della vita del processo.

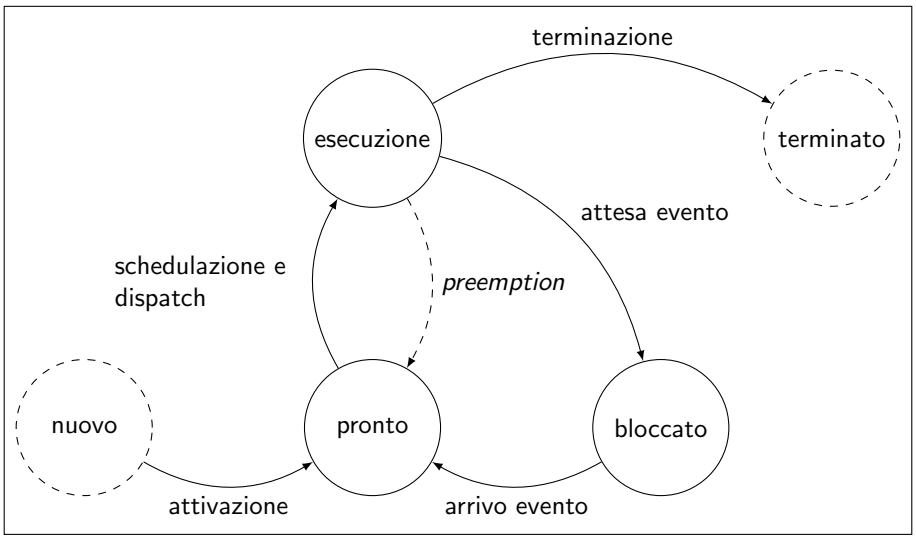

Figura 1: Stati di esecuzione un processo.

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". Il processo andrà effetivamente in esecuzione (prendendo il controllo del processore) tramite una successiva operazione che è detta di *dispatch*. In alcuni sistemi (ma non in quello che realizzeremo) schedulazione e dispatch sono combinate in un'unica azione. In ogni caso l'azione di dispatch segue dopo poco quella di schedulazione, e il fatto che siano distinte è solo un dettaglio implementativo.

Ovviamente, al momento del dispatch, 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 o eccezione, 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": il processo non sta attendendo un evento e non è più in esecuzione soltanto perché un altro processo sta occupando il processore.

Si noti che 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 strategie di schedulazione. Nella strategia *time-sharing* (a divisione di tempo), per esempio, il processore viene assegnato ad ogni pro-

cesso pronto per un tempo massimo, passato il quale un timer interrompe il processore causando una preemption con schedulazione e dispatch 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 da "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 (nel sistema didattico)

Il nostro sistema, per semplicitá, sarà mono-utente e mono-programma (perché prevediamo un singolo modulo `utente`), ma sarà comunque multiprocesso, nel senso che il programma potrá creare tanti processi a cui far eseguire funzioni definite nel programma stesso. I sistemi multi-processo si distinguono comunemente in "sistemi a memoria comune", in cui tutti i processi hanno accesso alla stessa memoria, e "sistemi a scambio di messaggi", in cui ogni processo ha una memoria completamente privata (e può comunicare con gli altri processi solo tramite messaggi). Noi realizzeremo un modello ibrido, perché questo ci permetterà di esplorare più dettagliatamente il meccanismo che rende possibile condividere o non condividere la memoria tra i processi (cioé la paginazione, come vedremo in seguito). Nel nostro sistema ogni processo utente ha una sua pila utente distinta da quella di tutti gli altri processi, a cui ha accesso esclusivo, mentre le sezioni **`.text`**, **`.data`** e **`.bss`**, e anche lo heap, sono condivise tra tutti. In pratica sono condivise tutte le entità globali definite nel modulo utente, più lo heap.

Dal punto di vista del sistema ogni processo ha inoltre un proprio descrittore di processo e una propria pila sistema. Lo scopo del descrittore di processo è 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 ricordi 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 e che nel descrittore di processo ci saranno le informazioni necessarie a recuperare tale stato dall'hard disk (almeno per la parte che non è a comune con gli altri processi).

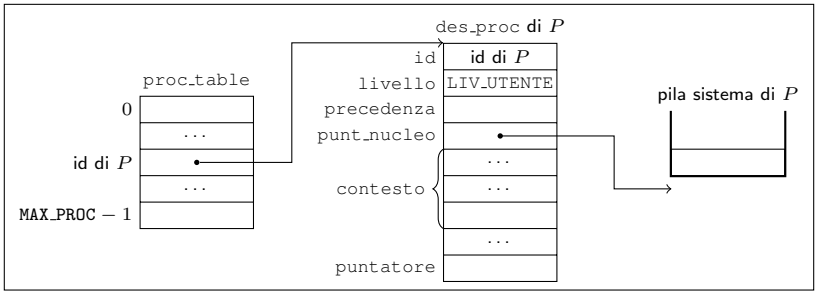

Figura 2: Strutture dati del sistema associate ad ogni processo utente.

Per gestire gli stati di Figura 1 faremo ricorso a *code di processi* realizzate come liste di descrittori di processo. 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` che memorizza l'identificatore numerico del processo (campo `id`), il suo livello di privilegio (utente o sistema) e la sua precedenza, come decisi al momento della sua creazione (campi `livello` e `precedenza`). Il campo `contesto` memorizza la copia privata dei registri del processore, che fanno parte appunto del contesto del processo. Questo campo 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. Osserviamo anche il campo `punt_nucleo`, che punta alla base della pila sistema del processo, e il campo `puntatore`, che serve a realizzare le code di processi a cui abbiamo accennato poco sopra.

Ricordiamo che il meccanismo di interruzione, nel caso in cui debba cambiare la pila corrente, prende l'indirizzo della nuova pila dal segmento TSS identificato dal registro TR. Noi allocheremo un unico segmento TSS e inizializzeremo il registro TR una sola volta all'avvio del sistema. Poi, per fare in modo che il meccanismo delle interruzioni utilizzi la pila sistema del processo corrente, dovremo sovrascrivere opportunamente il segmento TSS ogni volta che cambiamo processo.

Il processo attualmente in esecuzione sarà identificato dalla variabile globale `esecuzione`, di tipo puntatore a `des_proc`. I processi pronti si troveranno in una coda globale, la cui testa sarà puntata dalla variable `pronti`, anch'essa di tipo puntatore a `des_proc`. Predisporremo inoltre una coda diversa per ogni tipo di evento atteso dai processi bloccati. Manterremo tutte queste code ordinate in base al campo priorità. In questo modo, in particolare, la schedulazione di un processo pronto può essere eseguita semplicemente estraendo il `des_proc` 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 modulo 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 nel descrittore del processo identificato dalla variable `esecuzione`, mentre la seconda carica nel processore lo stato contenuto nel descrittore del processo identificato da `esecuzione`.

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

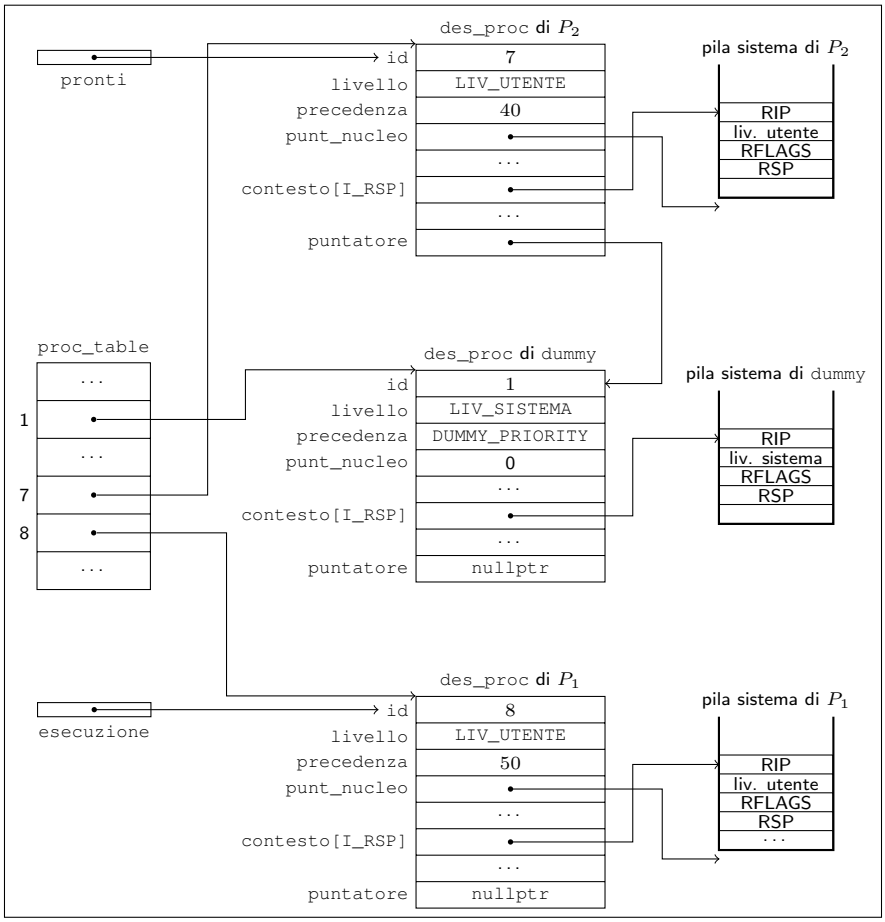

Figura 3: Esempio di istanziazione delle strutture dati per la gestione dei processi.

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 quadruple. La funzione `carica_stato` ripristinerà anche questo registro in modo che l'istruzione **iretq** finale possa estrarre dalla pila sistema le cinque parole quadruple, 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 in uno stato simile: 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 meccanismo delle interruzioni, in uno dei tre modi possibili. Tutti saranno dunque passati anche 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 **iretq**[2] 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 la primitiva `activate_p()`. La Figura 4 mostra un esempio di programma utente che usa la primitiva per attivare un nuovo processo. La primitiva accetta i seguenti parametri:

- un puntatore `f` alla funzione che il nuovo processo dovrà eseguire; la funzione deve essere di tipo **void** `(`**int**`)`, cioè accettare un unico parametro, di tipo **int**, e non restituire niente (`mioproc` in Figura);
- un'espressione di tipo `natq`, che sarà il valore ricevuto come argomento dalla funzione eseguita dal processo (10 in Figura);

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

```
1   #include <all.h>
2
3   void mioproc(natq i)
4   {
5       printf("mioproc: i = %d", i);
6       pause();
7       terminate_p();
8   }
9
10  int main()
11  {
12      natl id = activate_p(mioproc, 10, 20, LIV_UTENTE);
13      printf("Ho creato il processo %d", id);
14      terminate_p();
15  }
```

Figura 4: Un esempio di programma utente (file `utente/utente.cpp`).

- un'espressione di tipo `natl` che indica la priorità del processo (20 in Figura);
- uno tra `LIV_UTENTE` o `LIV_SISTEMA`, per indicare se il nuovo processo deve essere di tipo utente o sistema.

Si noti che un processo utente non può creare un processo di livello sistema e non può creare un processo a priorità maggiore della propria. La primitiva restituisce l'identificatore numerico del nuovo processo, o `0xFFFFFFF` in caso di errore.

Nell'esempio di Figura 4 il nuovo processo eseguirà `mioproc(10)`. Si noti che nulla vieta di invocare `activate_p()` più volte, anche usando la stesso puntatore a funzione. Ogni invocazione attiverà un nuovo processo (si ricordi la distinzione tra processo e programma: `mioproc` è solo il programma).

1.5 Realizzazione della `activate_p()`

La primitiva `activate_p()` dovrà fare in modo che il nuovo processo, quando verrà messo in esecuzione la prima volta, parta dalla prima istruzione della funzione `f`, usando una nuova pila utente.

Per attivare un processo è necessario allocare e inizializzare tutte le sue strutture dati: descrittore di processo (`des_proc`) 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 `des_proc`, 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 processo e

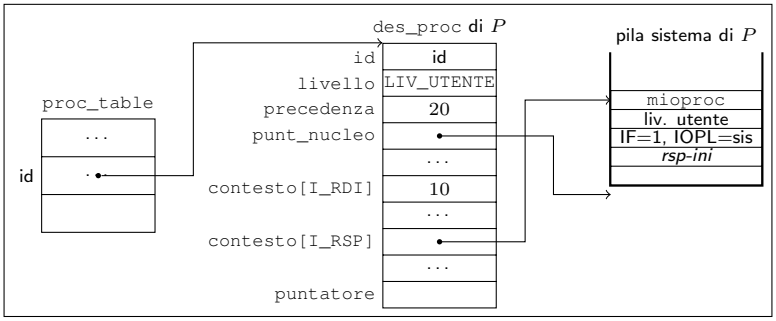

Figura 5: Attivazione del processo utente di Figura 4. Tutte le strutture dati mostrate si trovano nella parte M1 della memoria.

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 `mioproc`, nell'esempio di Figura 4). La Figura 5 mostra il risultato finale. In pila sistema scriviamo l'indirizzo di partenza (`mioproc` nell'esempio), il codice del livello utente, un campo **`rflags`** cof flag IF=1 e IOPL=sistema, e *rsp-ini*, che punta alla pila utente opportunamente allocata. 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 `mioproc`, 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.

Si noti che per fare in modo che la funzione che il processo eseguirà riceva il parametro che l'utente ha chiesto (secondo argomento della `activate_p ()` dobbiamo fare in modo che questo si trovi nel registro **`rdi`**. Per ottenere questo è sufficiente che all'attivazione del processo scriviamo il parametro attuale nel campo `contesto[I_RDI]`. La prima `carica_stato` lo porterà poi nel registro **`rdi`**.

Per completare l'attivazione dobbiamo anche occupare una nuova entrata della `proc_table`, scrivendovi dentro il puntatore al nuovo `des_proc`. L'indice di questa entrata nella `proc_table` funge anche da identificatore del processo. Scriviamo l'identificatore nel campo `id` del nuovo `des_proc`. Il campo `precedenza` del `des_proc` deve contenere la priorità (`prio`) del nuovo processo, che è semplicemente quella ricevuta come terzo parametro dalla `activate_p()`.

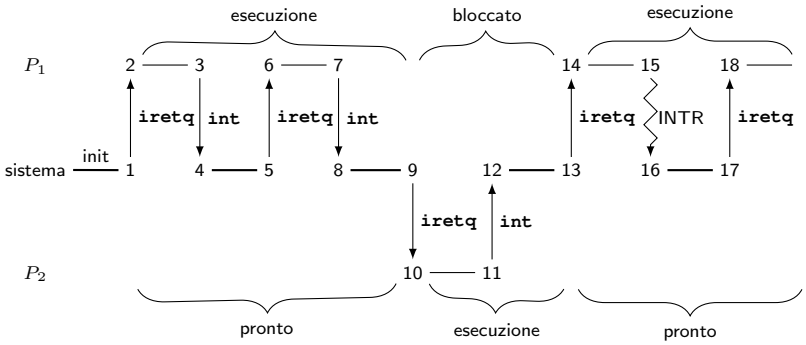

Figura 6: Un esempio di evoluzione del sistema con due processi.

1.6 Esempio

La Figura 6 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. 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.