Azzolini Riccardo 2019-10-16

Processi – Context switch

1 Contesto di un processo

Il **contesto** di un processo (**process context**), detto anche **ambiente**, è costituito da:

register context: il contenuto dei registri della CPU;

user-level context: le aree di testo, dati e stack, cioè tutte le zone della RAM a cui il processo può accedere;

system-level context: le informazioni che riguardano il processo, ma che vengono gestite dal SO, e sono quindi inaccessibili dal programma user:

- PCB;
- informazioni per accedere a memoria, file e dispositivi (puntate dal PCB);
- **kernel stack**:[1] contiene i record di attivazione degli interrupt handler e delle procedure chiamate dagli interrupt handler (mentre i record di attivazione del programma user sono nell'area di stack, che è parte dello user-level context).

2 Operazioni sui processi

Il SO effettua tre operazioni fondamentali sui processi per gestirne l'esecuzione:

Context save: quando il processo running perde la CPU (andando in ready o waiting), il SO salva nel PCB tutti i dati che, in futuro, serviranno a far ripartire il processo.

Scheduling: in base alla politica di scheduling, il SO sceglie un processo da eseguire tra quelli ready.

Dispatching: il SO ripristina il contesto del processo schedulato, sfruttando i dati salvati nel PCB al momento del context save.

[1] Esiste un solo kernel stack (non uno per processo), ma, siccome viene usato nella gestione degli interrupt, per convenzione lo si considera parte del contesto del processo interrotto, così come tale processo si considera ancora running (kernel running, secondo UNIX).

3 Context switch

Si ha un **context switch** quando il SO:

1. effettua il context save per un processo P;
2. schedula un processo P' diverso da P;
3. effettua il dispatching per P'.

Nota: durante tutto il context switch, il processo P risulta essere ancora in running (kernel running, in UNIX).

Il context switch può causare un overhead significativo, soprattutto se l'algoritmo di scheduling è complicato. Inoltre, esso causa un rallentamento del sistema anche perché le varie cache contengono probabilmente i dati (immagini della memoria nella cache della CPU, immagini dei file nella cache del disco, ecc.) di P, e quindi non quelli di P': finché le cache non si popolano, P' avanzerà più lentamente del normale.

4 Esempi

Per illustrare nel dettaglio come il SO gestisce i processi, sono riportati in seguito alcuni esempi. Essi simulano un sistema nel quale:

- la CPU ha i registri di controllo PC, SP, e PSW, i registri generali R1 e R2, ed è a 16 bit (quindi i valori dei registri e gli indirizzi sono rappresentabili con 4 cifre esadecimali);
- *si assume che la SRIA sia nel kernel stack.*

4.1 Singolo interrupt

All'inizio, si suppone che siano presenti due processi:

- il processo 5, che è (user) running;
- il processo 9, che è ready.

Il processo 5 eseguirà una `TRAP`, che verrà gestita dall'apposito interrupt handler, e al ritorno da quest'ultimo riprenderà l'esecuzione del processo 5.

Subito prima che venga eseguita la `TRAP`, la situazione è:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	
1001	
1002	
1003	
1004	

Registri

PC = 1880
PSW = 16F2
SP = 2880
R1 = 4500
R2 = CD31

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

0FFF

Osservazioni:

- Tutti i dati raffigurati, tranne i registri, sono contenuti nella kernel area della RAM.
- Il kernel stack (che non è attualmente in uso) cresce dall'indirizzo `0FFF` in su, ma nella figura gli indirizzi crescenti sono raffigurati dall'alto verso il basso.
- Anche se non si vede dal valore `16F2` della PSW, il bit PM è a 0 (modalità user) e la Interrupt Mask è impostata in modo da abilitare tutti gli interrupt.
- È raffigurato solo l'interrupt vector relativo alla `TRAP`.

Quando la CPU riceve l'interrupt generato dalla `TRAP`, l'hardware:

- salva nel kernel stack (SRIA) i valori dei registri di controllo;
- imposta i registri PC e PSW in base ai valori dell'interrupt vector;
- aggiorna il registro SP, in modo che punti al top del kernel stack (mentre prima puntava al top dello stack del programma utente).

Secondo UNIX, in questo momento il processo 5 va in kernel running. La situazione immediatamente dopo queste operazioni è:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	
1004	

Registri

PC = 5000
PSW = 0045
SP = 1002
R1 = 4500
R2 = CD31

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

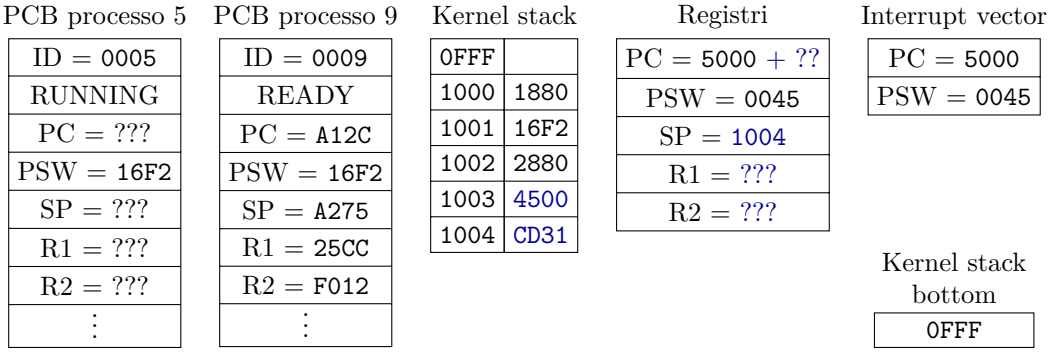
0FFF

Nota: i valori colorati sono quelli cambiati rispetto alla situazione precedente.

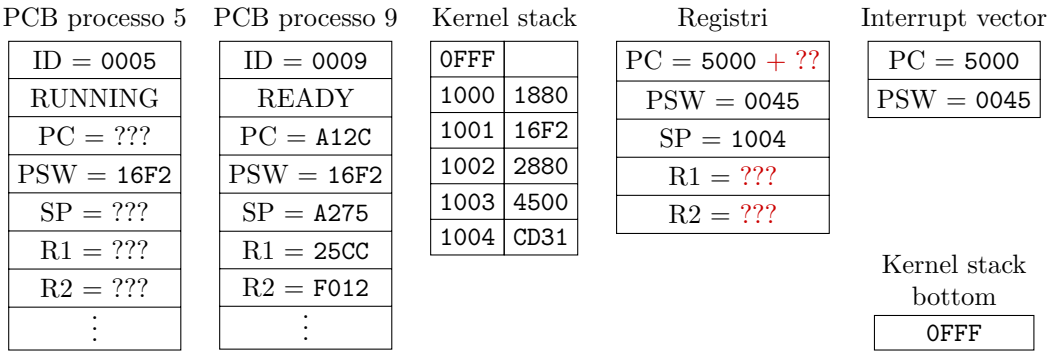
Con il nuovo valore della PSW, `0045`, il bit PM è stato impostato a 1, mettendo la CPU in modalità kernel, e sono stati disattivati gli interrupt di priorità minore o uguale alla `TRAP`.

In seguito, poiché il PC è stato impostato al valore contenuto nell'interrupt vector, inizia l'esecuzione dell'handler. Per prima cosa, esso salva nel kernel stack anche i registri generali, aggiornando lo SP per indicare il nuovo top dello stack. Ciò completa il context save.

Per effettuare queste operazioni, l'handler ha dovuto eseguire delle istruzioni, quindi il PC è cambiato (in questo esempio, si suppone che sia stato incrementato di un valore sconosciuto). Inoltre, l'handler potrebbe aver usato i registri generali (dopo averne salvato i contenuti originali), quindi anche i loro valori potrebbero essere cambiati. A questo punto, la situazione complessiva è:



L'handler continua poi a eseguire, effettuando la gestione vera e propria dell'interrupt, che comporta ulteriori cambiamenti dei valori di PC, R1 e R2. Quando questa è terminata, la situazione è:



Dopo aver gestito l'interrupt, l'handler chiama lo scheduler, che, come stabilito per questo esempio, riseleziona il processo 5 (cioè non si ha un context switch). Bisogna quindi effettuare il dispatching per farlo ripartire.

Nella prima fase del dispatching, vengono ripristinati i valori dei registri generali, che erano salvati nel kernel stack, e vengono cancellati logicamente da esso decrementando il registro SP, in modo da aggiornare il top dello stack. Siccome questo ripristino avviene via software, sono state eseguite ulteriori istruzioni, quindi il valore del PC è nuovamente cambiato. Adesso, la situazione è:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31

Registri

PC = 5000 + ??
PSW = 0045
SP = 1002
R1 = 4500
R2 = CD31

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

0FFF

Infine, con l'esecuzione dell'istruzione `IRET`, i registri di controllo vengono ripristinati via hardware dal kernel stack:

- il PC punta nuovamente all'istruzione del processo 5 successiva alla `TRAP`;
- la CPU viene rimessa in modalità user, e vengono riattivati tutti gli interrupt;
- lo SP punta di nuovo al top dello stack del programma utente, quindi è stata di fatto eseguita una cancellazione logica di tutti i dati presenti sul kernel stack.

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31

Registri

PC = 1880
PSW = 16F2
SP = 2880
R1 = 4500
R2 = CD31

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

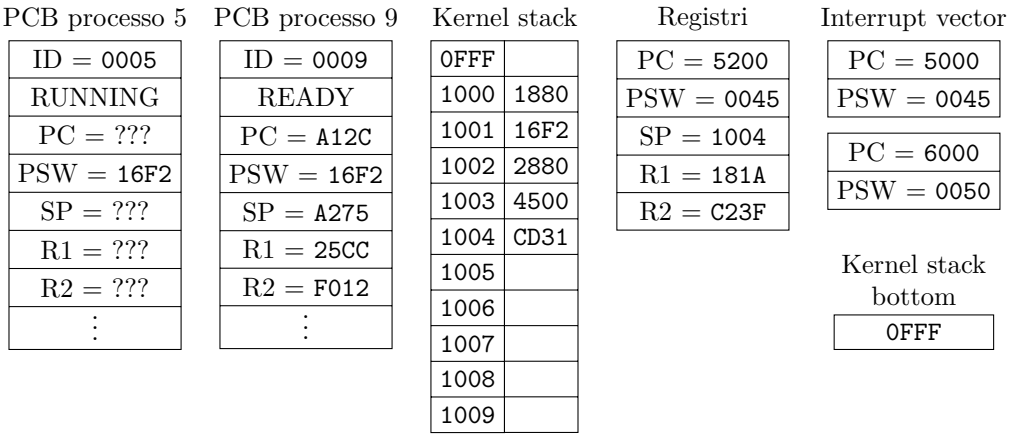
0FFF

Il processo 5 può quindi riprendere l'esecuzione: secondo UNIX, esso torna in user running.

4.2 Interrupt a cascata

Questo esempio parte dallo stesso scenario iniziale dell'esempio 1, ma durante l'esecuzione dell'handler della `TRAP` arriva un disk interrupt, che ha priorità maggiore e deve quindi essere trattato.

Fino alla partenza dell'handler della `TRAP`, l'esecuzione procede in modo uguale all'esempio 1. Poi, si suppone che il disk interrupt arrivi in un momento in cui PC = `5200` (l'indirizzo di una delle istruzioni dell'handler), R1 = `181A` e R2 = `C23F`. La situazione complessiva è quindi:



Osservazione: In questa figura è raffigurato anche un secondo interrupt vector, che è quello per i disk interrupt.

Quando riceve il disk interrupt, la CPU:

- salva PC, PSW e SP nel kernel stack, sopra ai valori dei registri salvati al momento della prima interruzione;
- aggiorna SP alla nuova cima del kernel stack;
- imposta PC e PSW dall'interrupt vector per i disk interrupt.

Secondo lo schema UNIX, il processo 5 transiziona da kernel running a kernel running.

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31
1005	5200
1006	0045
1007	1004
1008	
1009	

Registri

PC = 6000
PSW = 0050
SP = 1007
R1 = 181A
R2 = C23F

Interrupt vector

PC = 5000
PSW = 0045

PC = 6000
PSW = 0050

Kernel stack bottom

0FFF

Inizia così l'esecuzione del disk interrupt handler, che per prima cosa salva i registri generali, aggiornando ancora SP:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31
1005	5200
1006	0045
1007	1004
1008	181A
1009	C23F

Registri

PC = 6000 + ??
PSW = 0050
SP = 1009
R1 = ???
R2 = ???

Interrupt vector

PC = 5000
PSW = 0045

PC = 6000
PSW = 0050

Kernel stack bottom

0FFF

In seguito, l'handler effettua la gestione vera e propria del disk interrupt. Quando ha finito, siccome l'interruzione corrispondente alla `TRAP` risulta ancora *pendente*, l'handler non chiama lo scheduler, ma invece ripristina i registri generali dal kernel stack (aggiornando SP),

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31
1005	5200
1006	0045
1007	1004
1008	181A
1009	C23F

Registri

PC = 6000 + ??
PSW = 0050
SP = 1007
R1 = 181A
R2 = C23F

Interrupt vector

PC = 5000
PSW = 0045

PC = 6000
PSW = 0050

Kernel stack bottom

0FFF

e infine esegue l'istruzione `IRET`, con la quale l'hardware ripristina i registri di controllo. Secondo lo schema UNIX, il processo 5 transiziona nuovamente da kernel running a kernel running.

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31
1005	5200
1006	0045
1007	1004
1008	181A
1009	C23F

Registri

PC = 5200
PSW = 0045
SP = 1004
R1 = 181A
R2 = C23F

Interrupt vector

PC = 5000
PSW = 0045

PC = 6000
PSW = 0050

Kernel stack bottom

0FFF

Riprende allora l'esecuzione dell'handler della `TRAP`, che continua come nell'esempio 1.

4.3 Context switch

In questo esempio, si suppone che il processo 9:

1. sia inizialmente waiting;
2. venga risvegliato dalla `TRAP` eseguita dal processo 5 (ad esempio, il processo 5 potrebbe inviare al 9 un messaggio che quest'ultimo stava aspettando);
3. venga schedulato al termine della `TRAP`.

La situazione iniziale, prima della `TRAP`, è quindi:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
WAITING
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	
1001	
1002	
1003	
1004	

Registri

PC = 1880
PSW = 16F2
SP = 2880
R1 = 4500
R2 = CD31

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

0FFF

Da qui, l'esecuzione avviene come nell'esempio 1, fino alla fine del context save. La situazione al termine di quest'ultimo è:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
WAITING
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31

Registri

PC = 5000 + ??
PSW = 0045
SP = 1004
R1 = ???
R2 = ???

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

0FFF

Viene poi eseguito il codice di gestione vera e propria della `TRAP`, che sblocca il processo 9, cambiando il suo stato da waiting a ready:

PCB processo 5

ID = 0005
RUNNING
PC = ???
PSW = 16F2
SP = ???
R1 = ???
R2 = ???
⋮

PCB processo 9

ID = 0009
READY
PC = A12C
PSW = 16F2
SP = A275
R1 = 25CC
R2 = F012
⋮

Kernel stack

0FFF	
1000	1880
1001	16F2
1002	2880
1003	4500
1004	CD31

Registri

PC = 5000 + ??
PSW = 0045
SP = 1004
R1 = ???
R2 = ???

Interrupt vector

PC = 5000
PSW = 0045

Kernel stack bottom

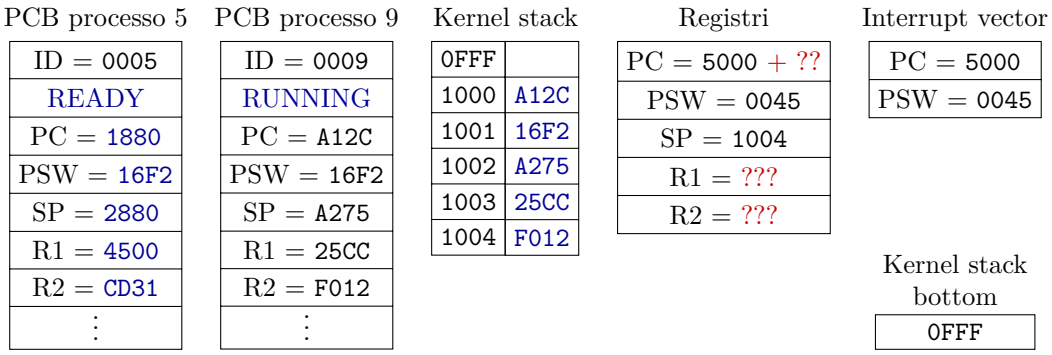
0FFF

L'handler chiama poi lo scheduler, che seleziona il processo 9 ed esegue il context switch:

- copia nel PCB del processo 5 i valori dei registri salvati nel kernel stack;

- carica nel kernel stack i valori dei registri salvati nel PCB del processo 9;
- aggiorna gli stati dei due processi, impostando il 5 a ready (in questo esempio, ma potrebbe invece essere waiting) e il 9 a (kernel) running.

L'esecuzione delle istruzioni che effettuano il context switch comporta anche il cambiamento di PC, R1 e R2.



Infine, viene ripristinato lo stato della CPU dal kernel stack (come negli altri esempi), e inizia quindi l'esecuzione del processo 9.