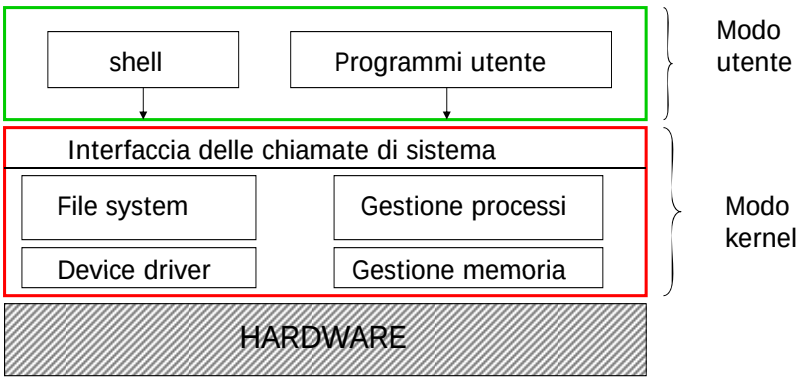
Struttura interna del sistema operativo Linux

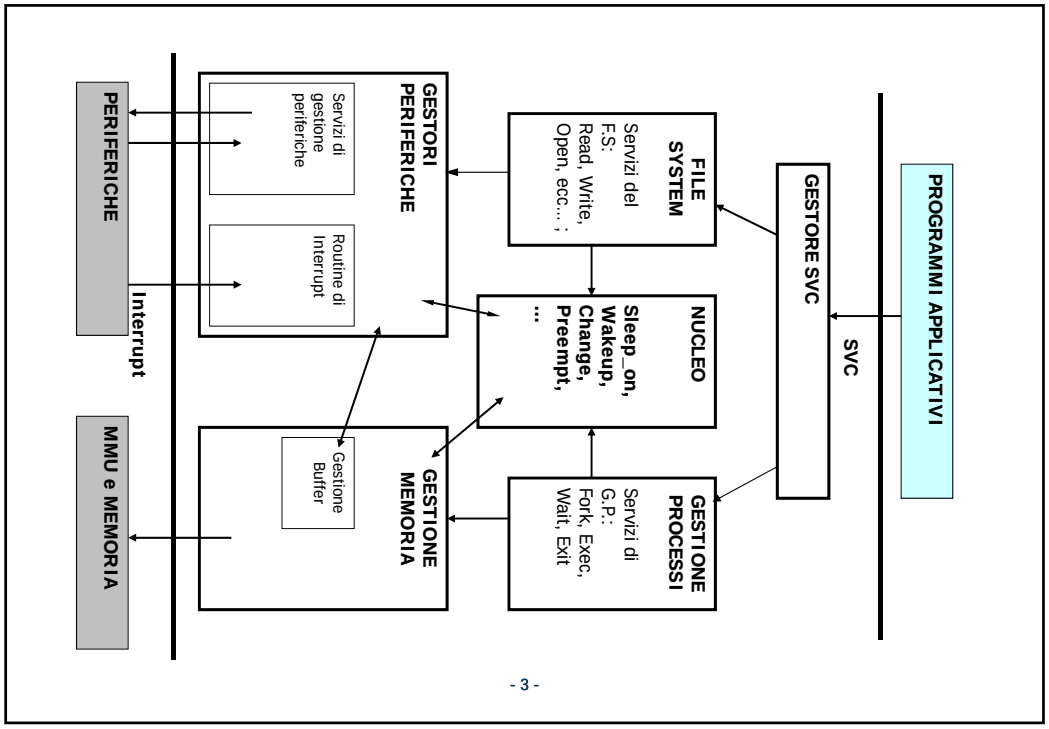
CAP. 6: Nucleo del sistema operativo

(La gestione dei processi)

Architettura del sistema operativo



Struttura a strati di LINUX



- 3 -

Funzionalità del Nucleo

- Mantenere in memoria il S.O. e i diversi programmi in spazi di indirizzamento separati. Durante l'esecuzione di un programma, accedere solo allo spazio di indirizzamento consentito.
- Permettere il passaggio dall'esecuzione di un programma a quella del S.O. e viceversa.
- Gestire le operazioni di I/O solo da S.O., e quindi tramite chiamate di sistema.
- Gestire i meccanismi di interrupt.
- Gestire la verifica del quanto di tempo associato ai processi.

- 4 -

Che cosa è un processo per il S.O.

- Rappresenta una istanza del programma in esecuzione con tutte le informazioni necessarie:
  - codice eseguibile
  - dati del programma
  - spazio di indirizzamento che contiene il codice eseguibile, i dati e lo stack
  - informazioni relative al suo funzionamento (stato)

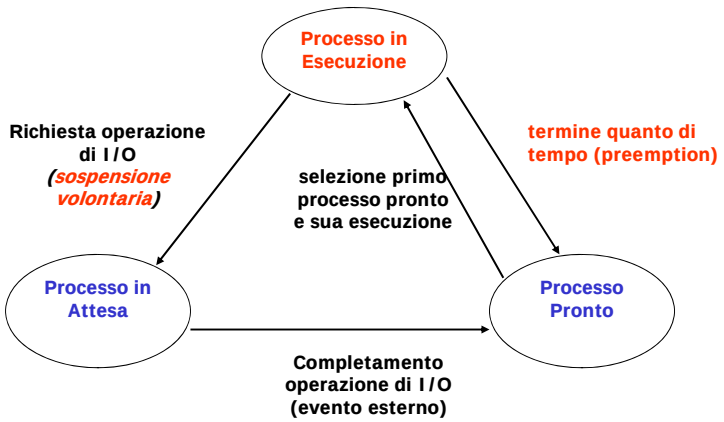
Meccanismo base di funzionamento del nucleo (1)

- Il S.O. alloca una zona di memoria ad ogni processo (spazio di indirizzamento del processo) che viene creato e lo carica in tale zona
- Esecuzione di un processo:
  - il nucleo del S.O. sceglie, secondo politiche di scheduling opportune, un processo da mandare in esecuzione (**stato di esecuzione**)
  - quando un processo in esecuzione richiede un servizio di sistema tramite una SVC:
    - viene attivata la funzione relativa e il S.O. esegue il servizio nel **contesto del processo** stesso
    - i servizi forniti dal S.O. operativo sono quindi parametrici in quanto vengono svolti *per conto del processo che li ha attivati*

Meccanismo base di funzionamento del nucleo (2)

- Un processo in stato di esecuzione lo abbandona:
  - per *sospensione volontaria*. Se il servizio di sistema richiesto deve attendere il verificarsi di un **evento** per completare il servizio stesso, il processo passa da stato di esecuzione **a** **stato di attesa**.
  - per la **scadenza del quanto di tempo** (*preemption*). Quando il S.O. verifica che il quanto di tempo è scaduto, pone il processo in esecuzione in **stato di pronto**
- Quando un processo in esecuzione è passato in stato di attesa o di pronto, lo scheduler del S.O. seleziona un processo, tra quelli pronti e lo porta in stato di esecuzione (**commutazione di contesto**).

Diagramma degli stati di un processo e relative transizioni di stato



Stati di un processo

- **In esecuzione**: è il processo che utilizza il processore (esiste un solo processo in esecuzione).
- **Pronto**: un processo è in stato di pronto se attende solo la risorsa processore per poter proseguire nell'elaborazione (possono esistere più processi in stato di pronto).
- **Attesa**: un processo è in stato di attesa se attende una risorsa non disponibile, un evento, il completamento di una operazione di I/O (possono esistere più processi in attesa su "eventi" diversi).

Transizioni di stato di un processo

ESECUZIONE ➔ PRONTO

- Al termine del quanto di tempo il SO deve **salvare** tutte le informazioni necessarie (**contesto**) per poter riprendere l'esecuzione del processo dal punto in cui è stata interrotta.

ESECUZIONE ➔ ATTESA

- Si verifica quando il processo richiede delle risorse che non sono disponibili o attende un evento;
- il SO **salva** tutte le informazioni necessarie (**contesto**) a riprendere l'esecuzione e l'informazione relativa all'evento atteso.

Transizioni di stato di un processo

ATTESA ➔ PRONTO

- Quando l'evento atteso da un processo si verifica, il SO sposta tutti i processi in attesa di quell'evento o di quella risorsa nella coda dei processi pronti.

PRONTO ➔ ESECUZIONE

- Lo scheduler del SO stabilisce quale dei processi accodati nello stato di PRONTO debba essere mandato in esecuzione (caricamento del contesto).
- La scelta è effettuata dall'algoritmo di scheduling dei processi.

La gestione degli interrupt

Quando si verifica un'interruzione generalmente esiste un processo in esecuzione e si possono avere 3 casi:

- l'interrupt può interrompere un processo in esecuzione in modalità U (utente);
- l'interrupt può interrompere un servizio di sistema invocato dal processo in esecuzione (è in esecuzione il SO - modalità S)
- l'interrupt può interrompere una routine di interrupt (è in esecuzione il SO - modalità S) – interrupt annidato o di secondo livello

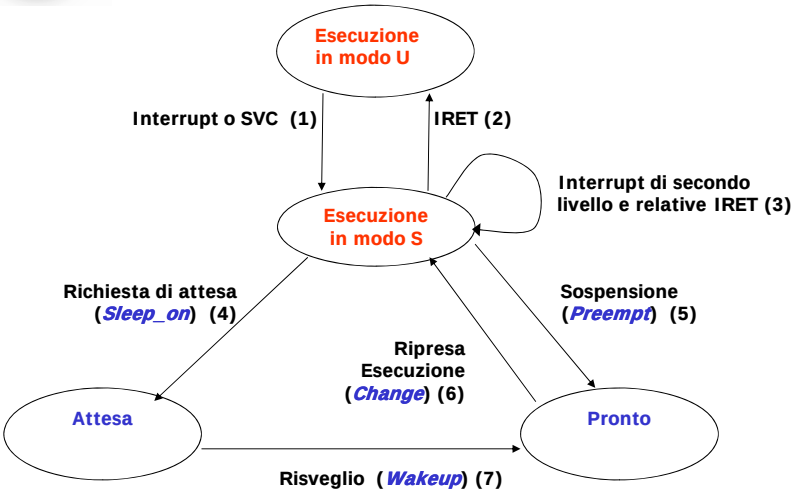
Meccanismo base di funzionamento del nucleo (4)

- In tutti i casi la routine di interrupt (cioè il S.O.) svolge il proprio compito in modo trasparente rispetto al processo in esecuzione
  - **non** viene mai svolta una commutazione di contesto durante l'esecuzione di un interrupt
- Se la routine di interrupt è associata al verificarsi di un certo evento E sul quale erano in attesa dei processi, allora la routine porta i processi dallo stato di attesa allo stato di pronto.
- Si osservi che *una routine di interrupt è associata ad un evento E atteso da un processo P, ma si svolge nel contesto di un altro processo Q*

Modalità user e modalità kernel

- I processi possono essere in esecuzione in modalità kernel o user → è necessario sdoppiare lo stato di esecuzione in due stati:
  - Esecuzione in modo user (**U)**
  - Esecuzione in modo kernel (**S)**
- Durante lo stato di esecuzione in modo kernel (S) non viene eseguito il codice del processo ma un servizio richiesto dal processo o una routine di interrupt.
- Durante l'esecuzione in modo kernel (S) possono verificarsi altri interrupt di maggior priorità (interrupt annidati o di secondo livello) – transizione **3**.
  - Si noti che durante questi interrupt annidati non può mai essere eseguita una commutazione di contesto.

Diagramma degli stati di un processo e relative transizioni di stato



Transizioni di stato

- La transizione **(1)** di stato da esecuzione in modo user a modo kernel avviene in due casi:
  - chiamata di sistema (SVC)
  - interrupt
- La transizione **(2)** di stato da esecuzione in modalità kernel a modalità user avviene per una IRET quando:
  - termina l'esecuzione della routine di risposta all'interrupt (se non ci sono interrupt annidati)
  - si rientra da SVC

Transizioni di stato

L'abbandono dello stato di esecuzione (modo S) può avvenire solo per uno dei seguenti due motivi:

1. Il processo P si sospende quando un servizio di sistema richiede un evento e invoca la funzione **`sleep-on`** (transizione **4**) con relativo salvataggio di contesto del processo P in esecuzione
2. Il processo P si sospende durante un servizio di sistema o un interrupt di primo livello (transizione 1) se si verifica che è scaduto il **quanto** di tempo e invoca la funzione **`preempt`** (transizione **5**) con relativo salvataggio di contesto

Si noti che un processo P può passare in stato di attesa o in stato di pronto solo quando è in esecuzione in modo S.

Transizioni di stato

- La ripresa dell'esecuzione di un processo P (transizione **6**) avviene solo se il processo P è pronto e se è quello con maggiore priorità di esecuzione tra tutti i processi pronti (algoritmo di scheduling dei processi). La funzione **`change`** esegue la commutazione del contesto.

- Si noti che il momento in cui avviene la transizione 6 per un processo P non dipende dal processo P stesso, ma da un altro processo che era in esecuzione (che starà eseguendo la transizione 4 oppure 5) e dall'algoritmo di scheduling.

Transizioni di stato

- Il risveglio di un processo P tramite la funzione **wakeup** (transizione **7**) avviene quando, nel contesto di un altro processo, si verifica un interrupt relativo all'evento E atteso dal processo P.
- Ovviamente, ad un certo istante di tempo, un solo processo P può essere in esecuzione (in modalità U oppure S), ma diversi processi possono essere pronti o in attesa di eventi.

La gestione del quanto di tempo

- Il **quanto di tempo** è gestito da un particolare **interrupt** generato dall'orologio di sistema
  - ad una frequenza definita, il dispositivo che realizza l'orologio di sistema genera un interrupt. La routine di risposta relativa incrementa una opportuna variabile che contiene il tempo di esecuzione del processo corrente
  - se il quanto di tempo non è scaduto la routine termina e, se non ci sono interrupt annidati, il processo prosegue nell'esecuzione
  - se invece il quanto di tempo è scaduto, e l'interrupt da real-time clock non è annidato
    - **viene invocata la funzione (`preempt`) che cambia lo stato del processo da esecuzione a pronto, salva il contesto del processo e**
    - **lo scheduler manda in esecuzione un processo pronto e viene invocata la funzione (`change`) che esegue una commutazione di contesto e**

Regole relative alla preemption (1)

- Si è visto che la commutazione di contesto non deve avvenire durante una routine di risposta all'interrupt, per evitare che venga sostituita la pila di sistema associata al contesto del processo originariamente in esecuzione
- Per non dilazionare troppo l'operazione di preemption e quindi la relativa commutazione di contesto, nel caso in cui la routine di interrupt da orologio abbia interrotto altre routine di interrupt (e quindi per evitare che il processo si appresti a ritornare in modo U senza aver sentito la preemption)
- **Tutte le routine di interrupt e il gestore dei servizi di sistema controllano, prima di terminare, se il ritorno con istruzione IRET riporta al modo di esecuzione U e in tal caso viene invocata la funzione `preempt` che controlla se per caso il processo corrente non debba essere sospeso.**

Regole relative alla preemption (2)

- **Il SO, in particolare tutte le routine di interrupt e il gestore dei servizi di sistema, prima di eseguire una IRET che rientra in modo U, invoca la funzione `preempt`**
  - Viene cioè verificato se il quanto di tempo del processo corrente è scaduto e, in caso positivo, dopo il salvataggio del contesto si esegue la commutazione di contesto invocando la **change**
- **In pratica, per imporre la preemption, il SO deve attuarla prima di ritornare al modo U,** perché un processo in modo U non possiede nessun controllo relativo al superamento del quanto di tempo.

Strutture dati fondamentali del nucleo per la gestione dei processi (1)

1. Il SO dispone di una **tabella dei processi** (**`Proctable`**) che contiene un elemento per ogni processo che è stato creato.
   - Gli elementi della tabella possono essere visti come *record* (di tipo ProcRec): il record contiene i campi indispensabili a caratterizzare il processo relativamente alla gestione dei processi, gestione della memoria e gestione dei file.
   - Una variabile **`CurProc`** contiene l'indice del record relativo al processo correntemente in esecuzione.

La tabella dei processi

```
ProcRec ProcTable[MAXPROC] /* vettore di elementi ProcRec       */
int     CurProc            /* puntatore al processo corrente   */
struct ProcRec {    /* elemento della tabella dei processi      */
    Stato           /* indica lo stato del processo             */
    Pila            /* contiene il valore del puntatore alla
                       pila di modo S (sSP) salvato quando
                       l'esecuzione del processo viene sospesa  */
    Evento          /* contiene l'identificatore dell'evento
                       sul quale il processo è in attesa        */
    Base            /* contiene la base del processo da usare
                       per la rilocazione dinamica             */
    File Aperti     /* tabella contenente i riferimenti ai file
                       aperti del processo; pure le periferiche
                       (file speciali) sono definite qui        */
    /* altri campi che per il momento non interessano           */
} /* end ProcRec */
```

Strutture dati fondamentali del nucleo (2)

2. Lo **stack (pila) di sistema**
   - Ogni processo ha un suo **stack utente** (che appartiene al suo spazio di indirizzamento) e un suo **stack di sistema**
   - Lo **stack di sistema** può essere visto come una "tabella" di stack di sistema: **esiste quindi uno stack di sistema per ogni processo che è stato creato.**
   - Il registro **sSP** punta di volta in volta allo stack di sistema del processo correntemente in esecuzione.
   - Il registro **uSP** punta di volta in volta allo stack utente del processo correntemente in esecuzione.
   - Il modo di esecuzione corrente (U/S) determina quale stack utilizzare (uSP/sSP).

Funzioni del nucleo per la gestione dei processi

- Tali funzioni del nucleo gestiscono il cambiamento di stato dei processi e la commutazione di contesto.
- Tali funzioni del nucleo possono scrivere nella tabella dei processi **(ProcTable)** per aggiornare le informazioni relative allo stato dei processi e al salvataggio di contesto
- Si noti che normalmente i vari servizi di sistema e le routine di interrupt possono accedere alla tabella dei processi solo per leggerne il contenuto ma non per modificarlo.

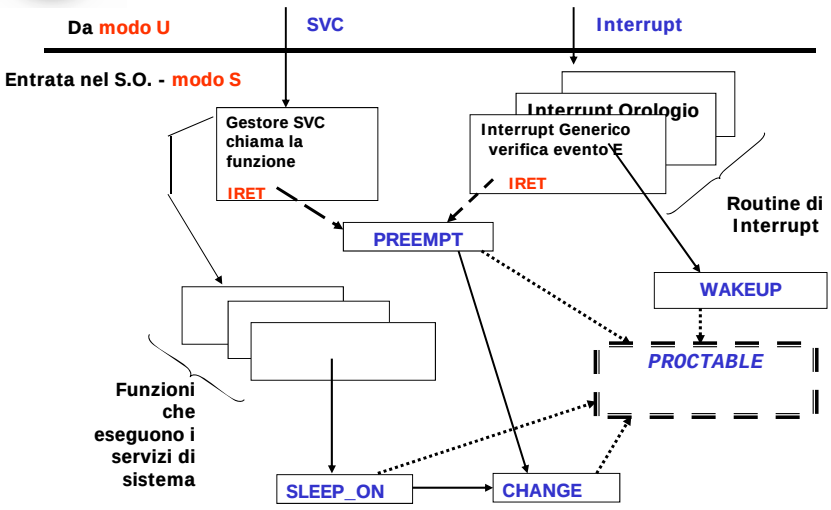
Funzioni del nucleo per la gestione dei processi

- **Sleep_on**: pone il processo corrente in stato di attesa, con salvataggio del contesto.
- **Change**: esegue una commutazione di contesto.
- **Wake_up**: risveglia un processo passando il suo stato da stato di attesa a stato di pronto.
- **Preempt**: sospende il processo in esecuzione per scadenza del quanto di tempo, con salvataggio del contesto.

Contesto di un processo

- Contesto: insieme delle informazioni che caratterizzano lo stato di un processo:
  - se il processo è fisicamente in esecuzione **parte del contesto** si trova nei registri della CPU (Program Counter, Stack Pointer, PSW, registri utente)
  - se il processo non è in esecuzione il contesto è tutto salvato in memoria
- È caratterizzato dalle seguenti informazioni:
  - codice
  - valori delle variabili globali e strutture dati a livello utente
  - i valori contenuti nei registri del processore
  - le informazioni memorizzate nella tabella dei processi
  - il contenuto dello stack user e dello stack kernel e relativi puntatori (usP e sSP)

Moduli del Sistema Operativo



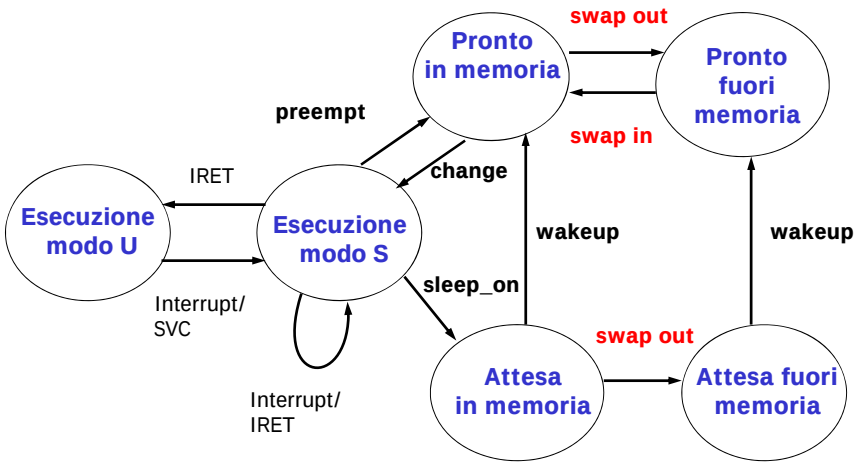
Stati di un processo e memoria virtuale

- Nel caso di memoria virtuale i processi possono trovarsi in memoria o fuori memoria (su disco).
- Gli stati di attesa e pronto si sdoppiano:
  - attesa in memoria e attesa fuori memoria
  - pronto in memoria e pronto fuori memoria
- Lo spostamento di un processo fuori o in memoria centrale si chiama **swapping.**
- Il sistema operativo sposta un processo fuori memoria se necessita di spazio in memoria.

Transizioni fuori memoria

- Lo spazio di memoria può essere richiesto nei seguenti casi:
  - l'esecuzione di una fork richiede l'allocazione di spazio di memoria per il processo figlio
  - una richiesta di memoria dell'area dati (mediante *brk*) incrementa le dimensioni del processo
  - crescita del segmento stack implica la crescita delle dimensioni globali del processo
  - il Sistema Operativo vuole liberare spazio in memoria per i processi che ha portato fuori memoria precedentemente e che sono pronti

Stati di un processo e memoria virtuale



La funzione *Sleep_on*

```
Sleep_on (identificativo_evento) {
/* Sleep_on_1: salva contesto processo in esecuzione */
/* salva su pila di sistema tutti i registri del processore */
/* incluso uSP */
...
/* salva le info di stato in ProcTable[curproc] */
ProcTable[curproc].evento = identificativo_evento
ProcTable[curproc].stato = attesa
Change ( ) /* invoca Change che esegue commutaz. del contesto */
/* Sleep_on_2: ripristina il contesto del processo sospeso */
/* carica uSP e i registri del processore dalla pila di sistema */
...
/* torna al chiamante */
RFS /* rientro da routine ordinaria */
} /* end Sleep_on */
```


La funzione *Change*

```
Change ( ) {
/* completa salvataggio di contesto del processo in esecuzione */
/* salvando il valore corrente sSP in ProcTable */
ProcTable[curproc].pila = sSP
/* lo scheduler determina il nuovo processo da mandare in esecuz. */
curproc = Scheduler ( )
/* ripristino del contesto prelevandolo da ProcTable */
/* e manda in esecuzione il nuovo processo */
RegBase = ProcTable[curproc].base
sSP = ProcTable[curproc].pila
ProcTable[curproc].stato = esecuzione
need_resched =0;
/* esegui ritorno all'ind. presente in cima alla pila di sistema */
RFS /* rientro da routine ordinaria */
} /* end Change */
```

La funzione *Wake-up*

```
Wake_up (identificativo_evento) {
/* cerca processo in attesa dell'evento */
/* e porta il processo in stato di pronto nella tab. dei processi*/
ProcTable[processo].stato = pronto;
if (ProcTable[processo].priorità > ProcTable[curproc].priorità.)
     { need_resched = 1;
      } /* end if */
RFS /* rientro da routine ordinaria */
} /* end Wake_up */
```


La funzione *Preempt*

```
Preempt ( ) {
/* verifica se è scaduto un quanto di tempo */
if (need_resched) { /* esegui salvataggio e commutaz. contesto */
     /* Preempt_1: salva contesto processo in esecuzione   */
     /* salva su pila di sistema i registri del processore */
     /* incluso uSP */
    ...
   /* salva le info di stato in ProcTable[curproc] */
     ProcTable[curproc].stato = pronto
     Change ( )
     /* Preempt_2: ripristina il contesto del processo sospeso */
     /* carica uSP e i registri del processore dalla pila di sistema */
      ...
      /* torna al chiamante */
} /* end if */
RFS /* rientro da routine ordinaria                    */
} /* end Preempt */
```

Routine di Risposta Interrupt

```
R_int (E) {
     /* R_int_1: gestisci interrupt */
     if (occorre_risvegliare_processo/i) {
          Wake_up(E)
     } /* end if */

     /* R_int_2: */
     if (modo_ritorno == U) {
          Preempt ( ) / * invoca Preempt */
     } /* end if */

     /* R_int_3: rientro da routine servizio interrupt */
     IRET
} /* end R_int */
```


Routine di Risposta Interrupt da Orologio

```
R_int (CK) {
     /* R_int_1: gestisci interrupt da orologio */
     /* aggiorna il quanto di tempo del processo in esecuzione */
     if (quanto di tempo scaduto) {
          need_resched = 1;
     } /* end if */

     if (modo_ritorno == U) {
          Preempt ( ) / * invoca Preempt */
     } /* end if */

     /* R_int_2: rientro da routine servizio interrupt */
     IRET
} /* end R_int */
```

G_SVC

```
G_SVC {
      .............
       /* G_SVC_1: scelta del servizio da invocare */
      case "fork": {
         fork ( )
         altro eventuale codice sempre legato a fork ...
         break
      } /* end case */
      case "exit": {
         exit ( )
         altro eventuale codice sempre legato a exit ...
         break
      } /* end case */
      .............
       /* G_SVC_2: invoca Preempt */
       Preempt ( )
        /* G_SVC_3: rientro da SVC tramite IRET */
       IRET
} /* end G_SVC */
```


Chiamata di sistema invocata da un generico processo

```
G_SVC_1
...
servizio ( )    servizio
G_SVC_2         ...
Preempt ( )     Sleep_on (E)    Sleep_on_1
IRET            servizio        ...
                RFS             Change ( )      Change
                                Sleep_on_2      /* salva contesto        */
                                RFS             /* carica nuovo contesto */
                                                /* commuta               */
                                                RFS

                Preempt_1
                ...
                Change ( )      Change
                Preempt_2       /* salva contesto        */
                RFS             /* carica nuovo contesto */
                                /* commuta               */
                                RFS
```

Creazione di un processo (fork) - 1

Strutture dati del Sistemo Operativo

La fork crea un nuovo elemento in **ProcTable** per il figlio e restituisce indice allo scheduler

La fork crea un nuovo elemento in **area stack di sistema** per la pila di sistema del figlio e restituisce il valore dello sSP del figlio (indirizzo effettivo)

lo stack di sistema del figlio è copia di quello del padre

Memoria del processo (Linux)

La fork **duplica** il segmento codice, il segmento dati e il segmento di sistema del padre

segmento codice: condiviso

segmento dati (di cui fa parte la pila utente): allocato fisicamente solo quando il padre o il figlio eseguono una scrittura

segmento di sistema: "agganciato" all'elemento della ProcTable e vengono aggiornate le informazioni associate ai file

Stato dei processi

dopo l'esecuzione della fork è **sempre il padre** che va in **esecuzione**, il figlio è in stato di pronto (vedi codice fork( )).

Creazione di un processo (fork) - 2

Valore restituito da fork

- PID del figlio al processo padre e 0 al processo figlio salvati nei rispettivi stack di sistema

Terminata la fork, in G_SVC_2 il valore restituito dalla fork e presente sullo stack di sistema viene copiato nello stack utente. Si noti che la fork viene invocata una volta, ma ritorna due volte!

```
G_SVC {
   .......
   case "fork": {
      fork ( ) /* esegui la fork */
      /* G SVC 2 */
      prelievo del valore restituito dalla fork e caricato sullo
      stack di sistema e copia sullo stack utente
      break
   } /* end case */
   ............
IRET /* rientro da routine servizio interrupt */
} /* end G_SVC */
```

Creazione di un processo (fork) - 2

- Il codice della fork, copiando il contenuto della pila di sistema di P in quella di F, copia anche l'indirizzo di ritorno di G_SVC posto in cima a tale pila
- In ritorno da G_SVC avviene quindi **due volte:**
  1. Quando termina G_SVC_2 del padre;
  2. Quando la funzione Change sceglie il processo F e lo lancia in esecuzione per la prima volta (G_SVC_2 di F)
- Naturalmente la fork dopo aver copiato il contenuto della pila di sistema di P in quella di F deve modificare le due pile, salvando il PID del figlio sulla pila di P e il valore 0 sulla pila di F (ovviamente nella posizione del valore restituito cioè subito sotto l'indirizzo di ritorno – che è lo stesso per padre e figlio).

Creazione di un processo (fork) - 3a

```
pid_t fork ( ) {
/* crea le varie strutture dati S.O. per il figlio */
/* 1 - crea elemento in ProcTable per figlio       */
/* 2 - alloca memoria utente e assegna base al figlio   */
ProcTable[figlio].base = base_figlio
/* creazione (salvataggio) del contesto del figlio
   - simile a Preempt                                */
/* salva su pila di sistema del figlio lo uSP e i registri
   del processore */
…
/* salva informazioni di stato in ProcTable[figlio] */
/* il figlio viene creato in stato di pronto con priorità */
/* del figlio uguale a quella del padre */
ProcTable[figlio].stato = pronto
```

Creazione di un processo (fork) - 3b

```
/* 3 - alloca stack di sistema per figlio e ottieni
   il valore iniziale di sSP figlio                    */
/* copia sStack padre in sStack figlio e aggiorna
   sSP figlio                                          */
/* come prima parte Change per concludere il
   salvataggio del contesto                            */
ProcTable[figlio].pila = sSP figlio (assoluto)
```


Creazione di un processo (fork) – 3c

```
/* 4 - predisponi il ritorno: gestisci i valori
   restituiti secondo le convenzioni dello hardware   */
/* carica il valore restituito (PID figlio) in sStack padre  */
/* carica il valore restituito (0) in sStack figlio   */
/* sSP deve puntare alla pila di sistema del padre
   in esecuzione                                       */
RFS /* rientro da routine ordinaria                    */
} /* end fork */
```

Terminazione di un processo (exit) - 1

- La exit dealloca la memoria utente del processo, lo stack di sistema del processo e l'elemento in ProcTable associato al processo.
- Deve gestire il PID del processo e il valore restituito da exit per possibile wait su figlio già terminato (cioè deve gestire lo stato di zombie).
- Per come sono strutturati i moduli del S.O. visti, la exit si comporta come la seconda parte della change (**carica il contesto del nuovo processo** che andrà in esecuzione) e quindi viene eseguita nel contesto di 2 processi.

Terminazione di un processo (exit) - 2

- Inoltre
  - chiude tutti i file aperti (close) e rilascia gli i-node associati al direttorio corrente
  - scrive informazioni relative a statistiche run-time riguardanti l'esecuzione del processo in un file globale
  - in Unix il processo passa nello stato *zombie*
  - sgancia il processo dall'albero dei processi facendo in modo che il processo 1 adotti tutti i suoi processi figli (se esistono)

Terminazione di un processo (exit) – 3a

```
exit (codice_uscita) {
/* 1 – dealloca memoria utente processo           */
/* 2 – dealloca stack sistema processo            */
/* 3 – "elimina" processo che ha eseguito la exit */
if (esiste padre in wait o waitpid) {
    /* dealloca l'elemento del processo (figlio)
       in ProcTable (e il processo non esiste)    */
    /* consegna codice di uscita al padre         */
    Wake_up (evento_exit)
} else { /* non esiste padre già in wait ...      */
    /* il processo diventa zombie e il codice di
       uscita viene salvato per usi futuri        */
} /* end if */
```


Terminazione di un processo (exit) – 3b

```
/* 4 – carica contesto del nuovo processo da mandare
   in esecuzione – come seconda parte della change */
curproc = Scheduler ( ) /* invoca Scheduler */
/* carica nuovo contesto e commuta  */
RegBase = ProcTable[curproc].base
sSP = ProcTable[curproc].pila
ProcTable[curproc].stato = esecuzione
need_resched = 0;
/* esegui ritorno all'indirizzo presente in cima alla pila di sistema */
RFS /* rientro da routine ordinaria */
} /* end exit */
```

Attesa della terminazione di un processo

- Un processo può sincronizzare la propria esecuzione con la terminazione di un processo figlio mediante l'esecuzione della chiamata di sistema wait
- L'esecuzione della wait comporta la ricerca da parte del kernel di un figlio zombie del processo, e se il processo non ha figli, restituisce errore
  - Se identifica un figlio in stato di zombie, ne estrae il PID e il parametro fornito dalla exit del processo figlio e ritorna questi valori
  - Il kernel libera la riga della tabella dei processi occupata dal processo figlio zombie
- Se il processo che esegue la wait ha processi figli ma nessuno è zombie
  - Il processo padre esegue una sleep e verrà risvegliato al momento della terminazione di un suo processo figlio
  - Al momento della terminazione di un figlio, il processo padre riprende l'esecuzione della wait, come descritto in precedenza

Inizializzazione del sistema operativo (i)

- Obiettivo della fase di inizializzazione è caricare una copia del sistema operativo in memoria e iniziarne l'esecuzione
- La procedura di bootstrap sulle macchine UNIX legge il blocco di bootstrap (blocco 0) da un disco e lo carica in memoria
- Il programma contenuto nel blocco di bootstrap carica il kernel dal file system e poi trasferisce il controllo all'indirizzo di inizio del kernel che va in esecuzione

Inizializzazione del sistema operativo (ii)

- Il kernel inizializza le strutture dati interne per la gestione del file system e della memoria centrale
  - Costruisce le liste di i-node e buffer, inizializza la tabella delle pagine,….
- Viene "caricato" parte del file system di root e creato il contesto per il *processo 0*, inizializzando la riga 0 della tabella dei processi.
- Quando il contesto del processo 0 è pronto il sistema è in esecuzione come processo 0.
- Il processo 0 esegue una *fork* e crea il processo 1 (processo *init*).
- Dopo aver creato il processo 1, il processo 0 diventa il processo di *swapper* (per la gestione della memoria virtuale).

Inizializzazione del sistema operativo (iii)

- Il processo 1, in esecuzione in modo kernel, crea il suo spazio di indirizzamento utente e copia il codice da eseguire dallo spazio di indirizzamento kernel a quello utente
- Il processo 1 esegue il codice copiato in modo utente
- Questo codice consiste una chiamata di sistema *exec* per mandare in esecuzione il programma **`/etc/init`** che è responsabile dell'inizializzazione dei nuovi processi
  - Il processo 1 è anche chiamato **`init`**
- Il processo **`init`** legge il file **`/etc/inittab`** che indica quali processi creare
- In un sistema multiutente **`inittab`** richiede la creazione di tanti processi **`getty`** quanti sono i terminali disponibili