Sistemi Operativi – primo modulo
I sistemi a processi

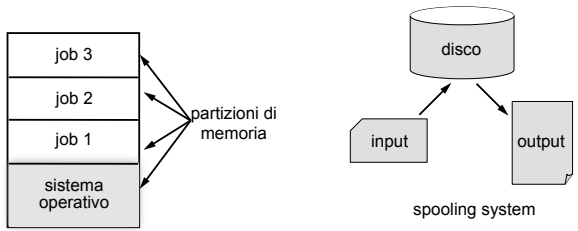
Augusto Celentano
Università Ca' Foscari Venezia
Corso di Laurea in Informatica

Il concetto di processo (1)

- Il concetto di *processo* è fondamentale nella teoria dei sistemi operativi
  - il termine è stato coniato negli anni '60 per il sistema operativo MULTICS; è quindi un concetto "antico"
- L'introduzione e il perfezionamento di questo concetto derivano dai problemi osservati in diversi modelli di gestione di attività *multitask* sviluppati progressivamente nel tempo
  - batch multiprogrammato
  - time sharing interattivo
  - real-time transazionale

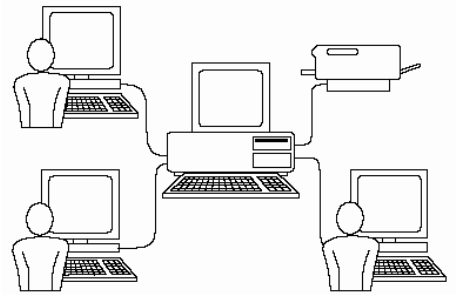
Il concetto di processo (2)

- Un sistema batch multiprogrammato serve in modo non interattivo un "lotto" di programmi caricati in memoria contemporaneamente, elaborandoli a turno
  - sovrapposizione dei lavori
  - serializzazione delle operazioni di ingresso / uscita
  - sfruttamento del tempo macchina



Il concetto di processo (2)

- Un sistema time sharing interattivo serve a turno più utenti ripartendo tra essi l'utilizzo delle risorse
  - ogni utente lavora indipendentemente dagli altri, avendo l'impressione di utilizzare una macchina dedicata
  - *fairness* nello sfruttamento delle risorse della macchina
  - protezione



Il concetto di processo (3)

- Un sistema real-time esegue più attività caratterizzate da vincoli di tempo sulla base di una scala di priorità, sospendendo i lavori meno urgenti a favore dei più urgenti
  - protezione, tempi di risposta
  - gestione conflitti e coerenza dei dati esterni
  - sfruttamento delle risorse compatibilmente con le esigenze dell'ambiente esterno

Il concetto di processo (4)

- In tutti e tre i casi si osservano due problemi relativi all'alternarsi di più attività distinte
  - la necessità di preservare lo *stato* di un'attività prima che sia terminata nel caso di passaggio ad un'altra attività
  - l'ottimizzazione nell'uso delle risorse
- Il meccanismo fondamentale per gestire l'alternanza di attività è l'interruzione
  - la sua gestione deve essere del tutto generale e indipendente dal tipo di attività corrente
- I problemi derivano dalla casualità con cui l'interruzione si presenta rispetto alle attività in corso
  - errori di sincronizzazione
  - errori di blocco indefinito
  - non determinismo nella successione delle operazioni

Il concetto di processo (5)

- Un processo è un'occorrenza (*instance*) di un programma in esecuzione
  - un'entità assegnata a un processore ed eseguita su di esso
  - un'unità di attività caratterizzata da un flusso di esecuzione unico e da un insieme di risorse associate (... *thread*)
- Formalmente, nella sua formulazione più essenziale, un processo *P* è una coppia di elementi

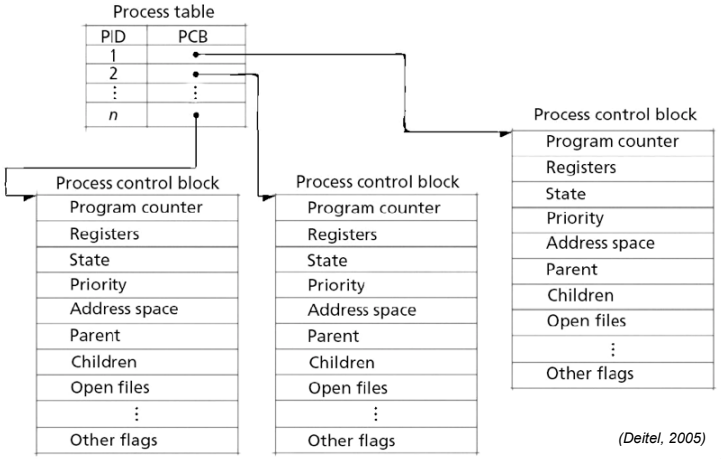
$$P = (C , S)$$

  - *C* è il *codice* eseguito dal processo (il programma, costante)
  - *S* è il *vettore di stato* del processo, cioè l'insieme dei dati variabili: valore dei registri, valore delle celle di memoria associate ai dati, stato dei dispositivi di ingresso e uscita, punto in cui si trova l'esecuzione (*program counter*)

Descrittore di processo (1)

- Per gestire più processi, ad ognuno di essi viene associato un *descrittore* (*PCB , Process Control Block*)
  - i descrittori contengono le informazioni necessarie per individuare e ripristinare lo stato dei processi
  - ogni descrittore contiene due tipi di informazioni
    - quelle necessarie quando il processo è in esecuzione (*ambiente*)
    - quelle necessarie quando il processo non è in esecuzione (*contesto*)
  - i descrittori sono mantenuti dal sistema operativo in aree protette (*tabella dei processi*)

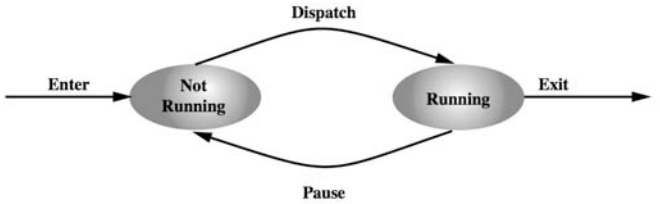
Descrittore di processo (2)



(Deitel, 2005)

Stati di attività un processo (1)

- In una situazione ideale caratterizzata da un processore dedicato, un processo può trovarsi in uno tra due stati: attivo, o in attesa di un evento esterno
  - un processo attivo va in attesa (si sospende) quando chiede un servizio del S.O. (es. una operazione di I/O)
  - un processo in attesa ritorna attivo quando il servizio del sistema operativo è terminato



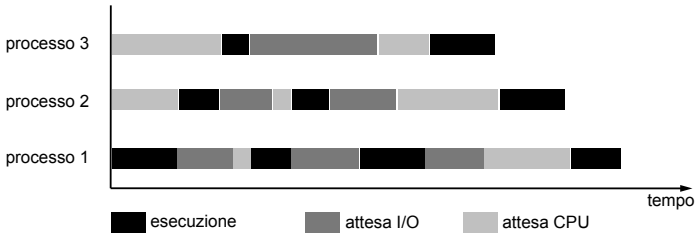
(Stallings, 2011)

Stati di attività di un processo (2)

- In un sistema multiprogrammato vengono eseguiti più processi su un solo processore. La situazione è più complessa
  - un processo può essere logicamente attivo o in attesa di un evento esterno
  - un processo logicamente attivo è in esecuzione quando occupa il processore
  - un processo logicamente attivo può non essere in esecuzione perché il processore non è disponibile (processo inattivo)
- Un solo stato di inattività non è sufficiente
- … oppure: un solo stato di attività non è sufficiente

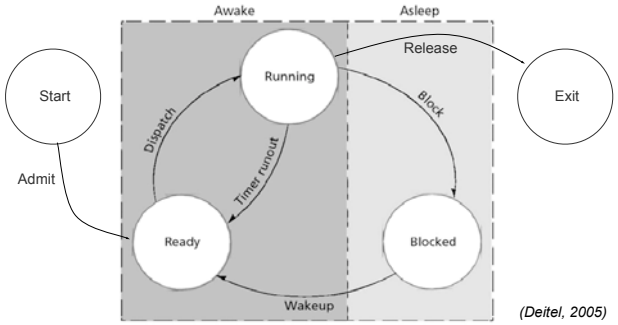
Stati di attività di un processo (3)

- In un sistema multiprogrammato i processi si alternano nell'esecuzione in base a
  - proprio stato di esecuzione (attivo - in attesa)
  - disponibilità di risorse



Un modello a 3+2 stati (1)

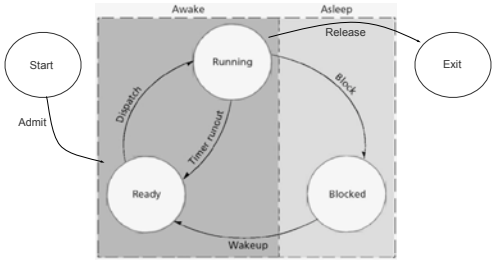
- Per identificare lo stato completo di un processo servono almeno 3 stati
  - attivo, in attesa, pronto
    + inizio, fine



(Deitel, 2005)

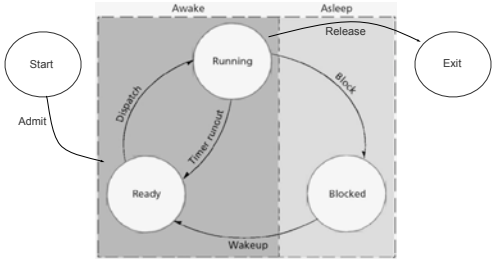
Un modello a 3+2 stati (2)

- *In esecuzione*: un processo che utilizza l'unità centrale
- *Pronto*: un processo che potrebbe essere eseguito se avesse l'uso dell'unità centrale
- *In attesa*: un processo che non può essere eseguito perché richiede che si verifichi un evento esterno



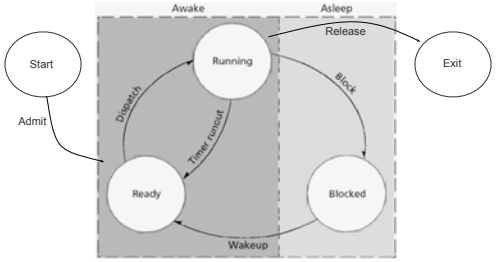
Un modello a 3+2 stati (3)

- *Nuovo*: un processo che inizia l'esecuzione
- *Uscita*: un processo che termina l'esecuzione



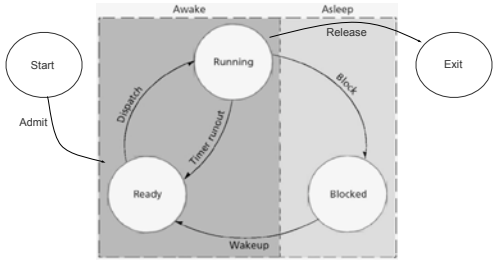
Transizioni di stato (1)

- Un processo in esecuzione va *in attesa* (si sospende) quando chiede l'intervento del S.O. (es. per una operazione di I/O)
- Un processo in attesa va *in stato di pronto* quando l'evento per cui si era sospeso si verifica



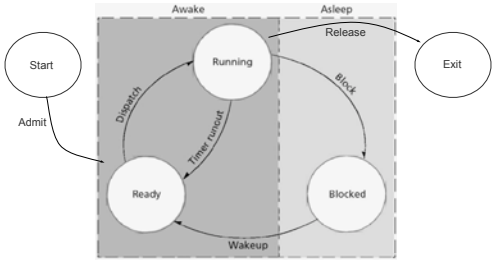
Transizioni di stato (2)

- Un processo pronto va *in esecuzione* quando il nucleo gli assegna l'uso dell'unità centrale (*dispatch*)
- Il processo in esecuzione va *in stato di pronto* quando il nucleo gli toglie l'uso dell'unità centrale (*timeout*, *priorità*)



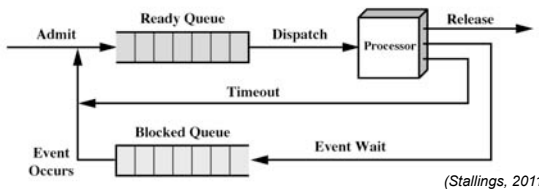
Transizioni di stato (3)

- Un *nuovo* processo viene creato in stato di *pronto*
  - andrà in esecuzione quando gli sarà assegnata l'unità centrale
- Un processo *termina* quando esegue una funzione di terminazione (*exit*) e va nello stato di *uscita*
  - vengono rimosse le risorse occupate



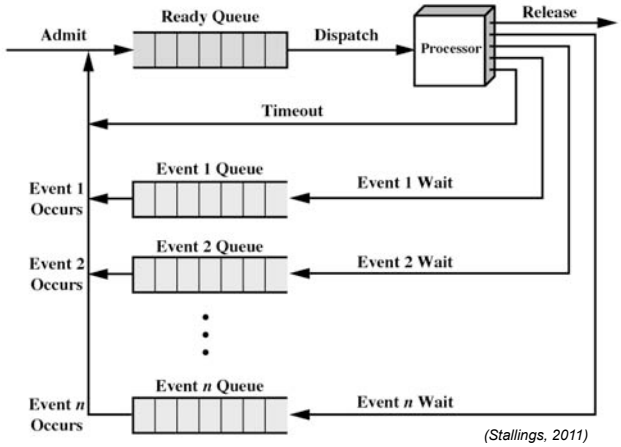
Scheduling dei processi (1)

- I processi pronti sono organizzati in una o più code
  - in base alle politiche di gestione dell'unità centrale
  - la gestione delle code può essere statica o dinamica
- I processi in attesa su dispositivi di I/O normalmente sono organizzati in code, una per ogni dispositivo
  - la gestione delle code è normalmente FIFO



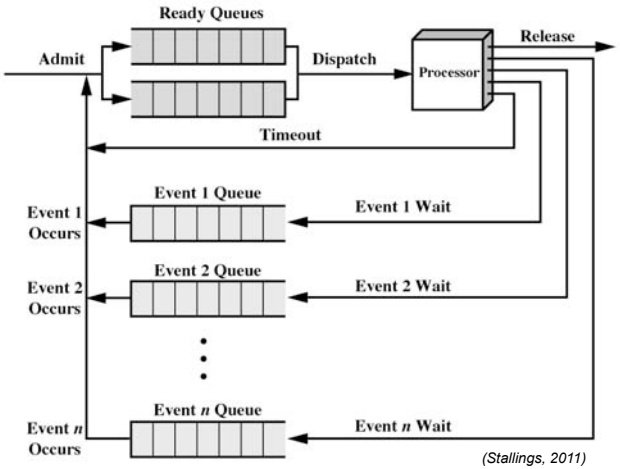
(Stallings, 2011)

Scheduling dei processi (2)



(Stallings, 2011)

Scheduling dei processi (3)

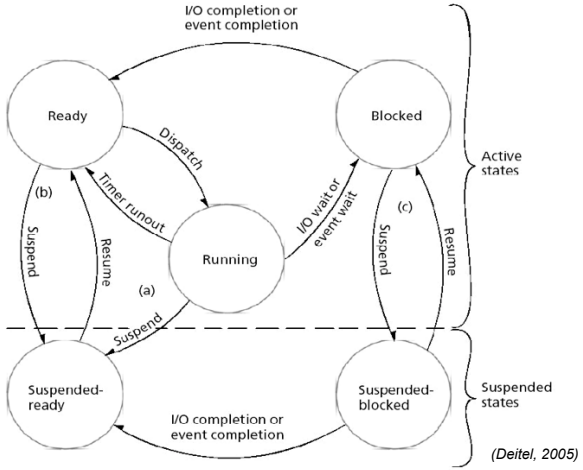


(Stallings, 2011)

Gestione degli stati di attesa

- Il processore è più veloce dei dispositivi di I/O
  - tutti i processi in memoria potrebbero essere in attesa di eventi esterni
  - potrebbero esserci altri processi fuori memoria ma in grado di essere eseguiti
- I processi in attesa di I/O lento potrebbero essere portati fuori dalla memoria
  - la memoria si libera per l'esecuzione di altri processi
- Si introduce lo stato di *processo sospeso fuori memoria*
  - in attesa e sospeso
  - pronto e sospeso

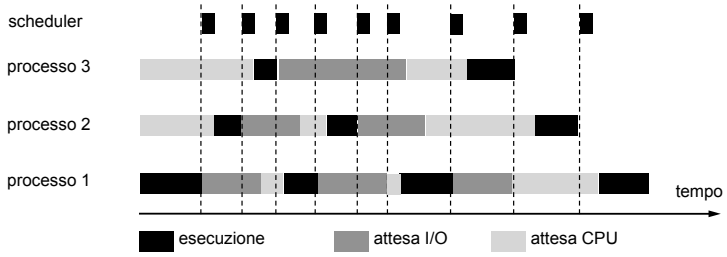
Sospensione dei processi fuori memoria



(Deitel, 2005)

Scheduling dei processi pronti

- La gestione della coda (delle code) dei processi pronti è effettuata da uno *scheduler* a breve termine (*scheduler di CPU*)
  - algoritmi di scheduling diversi infuiscono non solo sull'ordine di esecuzione dei processi ma anche sulle prestazioni complessive del sistema

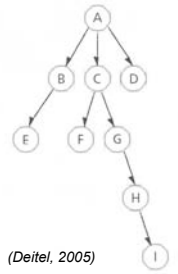


Creazione di un processo (1)

- Assegnazione di un identificatore unico
- Allocazione di memoria per il processo
  - codice
  - dati
- Allocazione di altre risorse nello stato iniziale
  - privilegi, priorità
  - file, dispositivi di I/O
- Inizializzazione del descrittore
- Collegamento con le altre strutture dati del sistema operativo
- Contabilizzazione

Creazione di un processo (2)

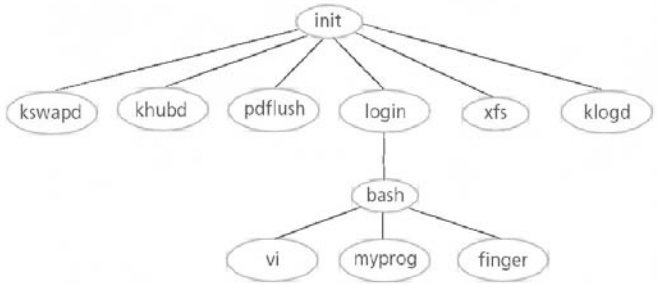
- Un processo può essere creato solo da un altro processo
  - utente
  - di nucleo (del sistema operativo)
- La differenza risiede nella autorizzazioni che il processo creato (*figlio*) eredita dal processo creante (*padre*)
- La creazione di processi può essere iterata a più livelli producendo una struttura ad albero



(Deitel, 2005)

Creazione di un processo (3)

- in Unix tutti i processi nel sistema sono generati a partire da un solo processo iniziale



(Deitel, 2005)

Creazione di un processo (4)

- Relazioni dinamiche con il processo creante
  - il processo padre prosegue
  - il processo padre aspetta
  - il processo figlio non conserva relazioni con il padre (processo *detached)*
- Relazioni di contenuto con il processo creante
  - il processo creato è una copia del processo padre
  - il processo creato esegue un programma diverso

Terminazione di un processo (1)

- Un processo termina con una richiesta al sistema operativo (*exit*) che causa
  - la conclusione delle operazioni di I/O bufferizzate
  - il rilascio delle risorse impegnate (memoria, dispositivi di I/O dedicati)
  - la (eventuale) trasmissione di dati di completamento al processo creante
  - la distruzione del descrittore
- Un processo può terminare per effetto di un altro processo (*kill*), in modo controllato rispetto a privilegi e protezioni
- Un processo può terminare per errore

Terminazione di un processo (2)

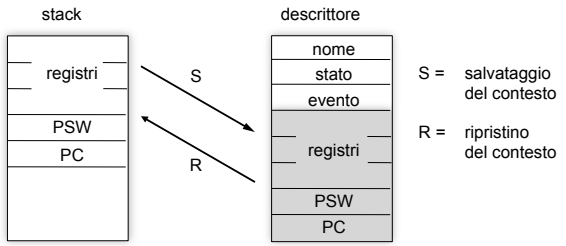
- Le relazioni dinamiche tra processo creante e creato si riflettono sulla terminazione
  - la terminazione di un processo figlio "risveglia" il processo padre in attesa
  - la terminazione di un processo padre può causare la terminazione dei processi figli, oppure
  - i processi orfani possono essere "adottati" da un altro processo (in Unix è il processo "init")
  - i processi *detached* non sono influenzati dalla terminazione del processo che li ha creati

Commutazione di contesto (1)

- La transizione di stato di un processo è una operazione complessa che, a fronte di una interruzione, modifica il contesto nel quale il processore lavora
- Si assumono le seguenti ipotesi:
  - il verificarsi di una interruzione provoca il salvataggio dei registri del processore (PC, PSW, altri) sullo stack
  - durante il servizio dell'interruzione le interruzioni sono disabilitate
  - il ritorno da una interruzione ripristina i registri del processore dallo stack e riabilita le interruzioni

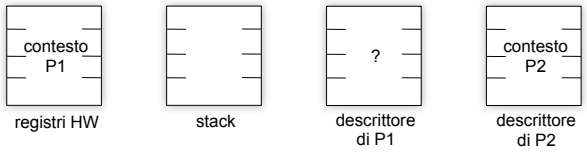
Commutazione di contesto (2)

- La commutazione tra due processi richiede che i loro contesti di esecuzione siano salvati e ripristinati
  - la commutazione avviene solo a seguito di interruzione
  - in cima allo stack c'è il contesto del processo corrente
  - la commutazione può avvenire scambiando informazioni tra lo stack e il descrittore del processo



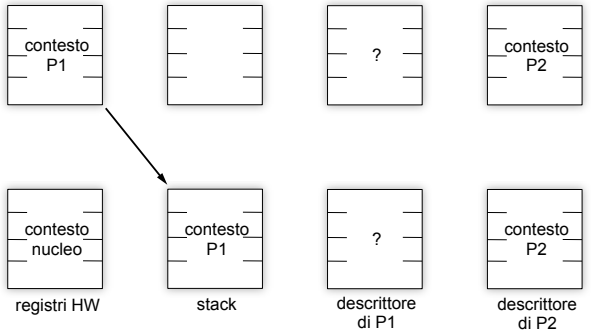
Commutazione di contesto (3)

- La commutazione di contesto dal processo P1 (da esecuzione a attesa) al processo P2 (da pronto a esecuzione) avviene in quattro fasi:
  1. inizialmente il processo P1 è in esecuzione, il processore opera nel contesto di P1, lo stack contiene dati locali di P1, il descrittore di P1 non è significativo, il descrittore di P2 contiene il contesto di P2 salvato quando P2 ha interrotto l'esecuzione



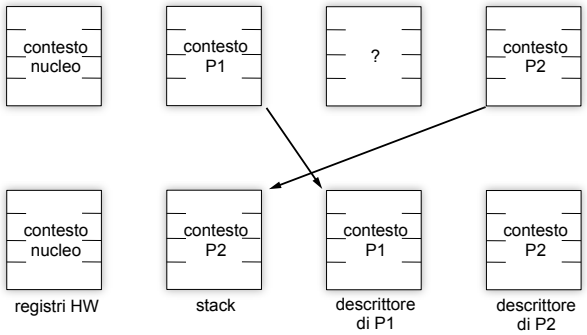
Commutazione di contesto (4)

2. P1 esegue una SVC per richiedere una operazione di I/O. Il suo contesto viene posto in cima allo stack e il processore opera nel contesto del nucleo



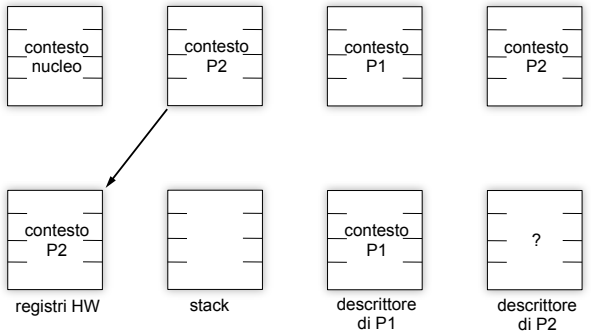
Commutazione di contesto (5)

3. Il nucleo porta P1 in stato di attesa e P2 in stato di esecuzione, salvando il contesto di P1 nel descrittore di P1, e ripristinando dal descrittore di P2 il contesto di P2



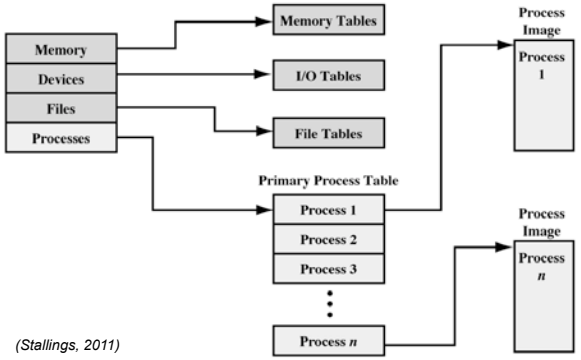
Commutazione di contesto (6)

4. Il nucleo termina la SVC eseguendo un ritorno da interruzione che ripristina il contesto del processore con il contenuto dello stack. Il processore opera nel contesto di P2



Strutture dati del sistema operativo (1)

- Mantengono informazioni sullo stato corrente del sistema in termini di processi e risorse
  - tabella dei processi
  - tabella di allocazione di memoria
  - tabella dei dispositivi di I/O
  - tabella dei file aperti



(Stallings, 2011)

Strutture dati del sistema operativo (2)

- Tabella dei processi
  - identificatore di processo
  - allocazione in memoria (segmenti)
  - file utilizzati
  - programma eseguito
  - informazioni di stato
  - informazioni contabili
- Tabella di allocazione di memoria
  - allocazione della memoria centrale ai processi
  - allocazione di memoria secondaria ai processi
  - attributi di protezione per l'accesso a zone di memoria condivisa
  - informazioni necessarie per la gestione della memoria virtuale

Strutture dati del sistema operativo (3)

- Tabella dei dispositivi di I/O
  - stato dei dispositivi di I/O: disponibile, occupato, assegnato esclusivamente ad un processo
  - stato delle operazioni di I/O
  - informazioni sui buffer utilizzati per il trasferimento dei dati da/verso la periferia
- Tabella dei file aperti
  - identificazione dei file
  - locazione sulla memoria secondaria
  - stato corrente di accesso / condivisione / posizione di lettura e scrittura
  - attributi
  - l'informazione può essere gestita attraverso il file system

I processi in Unix

- La gestione dei processi in Unix si basa sui concetti di *processo* e di *immagine*
  - il *processo* è l'entità attiva che esegue un programma (l'immagine); è descritto da un identificatore di processo, da una struttura dati (descrittore), e corrisponde all'insieme di codice, dati utente e dati di nucleo
  - l'*immagine* è il testo del programma eseguito dal processo; è composta da un'area contenente il codice, e da un'area riservata per i dati del programma eseguito (dati utente e stack)
  - quando un processo è in esecuzione la sua immagine deve essere presente in memoria centrale

Gestione dei processi in Unix (1)

- La creazione di un processo e la definizione della sua immagine avvengono attraverso un meccanismo combinato
  - *duplicazione* di un processo esistente (*fork*), che dà origine ad un processo (detto processo *figlio*) copia del processo creante (detto processo *padre*)
  - *sostituzione* dell'immagine eseguita (*exec*), che permette ad uno dei due processi di evolvere separatamente dall'altro

Gestione dei processi in Unix (2)

- La funzione *fork* duplica un processo creandone uno nuovo che esegue la stessa immagine del processo creante

esito = fork();

  - l'area dati viene duplicata, l'area codice viene condivisa
  - tutte le risorse utilizzate dal processo creante sono accessibili dal processo creato
  - il processo creato (figlio) riceve esito = 0
  - il processo creante (padre) riceve esito > 0 e uguale all'identificatore di processo del processo creato
  - se l'operazione fallisce esito < 0

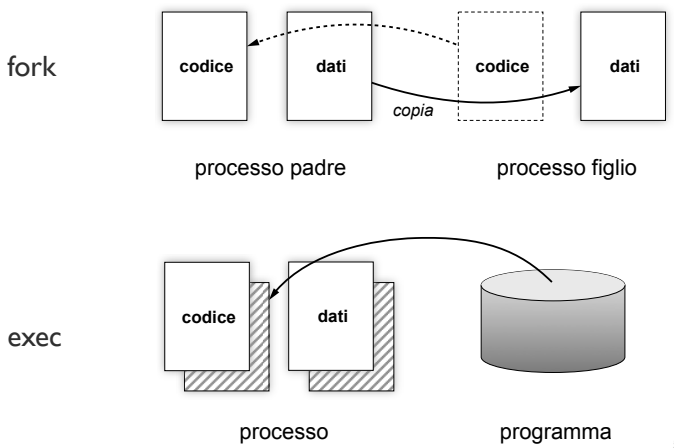
Gestione dei processi in Unix (3)

- La funzione *exec* sostituisce l'immagine del processo che la esegue con il contenuto di un altro file eseguibile

exec(nome file, lista di argomenti);

  - l'esecuzione prosegue con il nuovo programma a cui vengono trasmessi gli argomenti specificati
  - esistono più varianti della funzione che differiscono per la struttura dei parametri nella chiamata
    - execl(const char *file, const char *arg1, const char *arg2, …, (char *)0)
    - execv(const char *file, const char *argv[ ])
    - execvp(const char *file, const char *argv[ ])
    - execve(const char *file, const char *argv[ ], const char *env[ ])
    - …

Gestione dei processi in Unix (4)

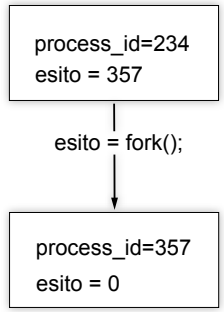


Relazioni tra i processi

- Il processo figlio non condivide memoria con il processo padre (ne condivide il codice)
  - dalla creazione in poi i due processi evolvono separatamente eseguendo la stessa immagine in modo indipendente
- La creazione avviene per duplicazione completa (logica) del processo padre
  - il processo figlio eredita l'ambiente di lavoro: file aperti, privilegi, directory di lavoro, etc.
  - l'ambiente di lavoro è legato al processo e non all'immagine, quindi sopravvive all'esecuzione della funzione *exec* e viene trasferito al nuovo programma

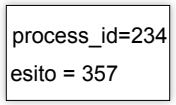
Un esempio di creazione di un processo

```
esito = fork();
if (esito < 0)
 {
  /* la fork ha fallito ... */
 }
else if (esito > 0)
 {
  /* codice del processo padre */
 }
else
 {
  /* codice del processo figlio */
 }
```



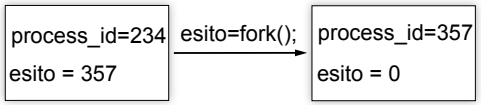
Un esempio di esecuzione di un'immagine

```
esito = fork();              /* crea un processo figlio */
if (esito == 0)              /* se è il figlio */
  { execlp("p","arg1");      /* esegue il programma "p" con argomento "arg1"*/
    error(...);              /* ...a meno di errori */
  }
...                          /* qui procede solo il padre */
```



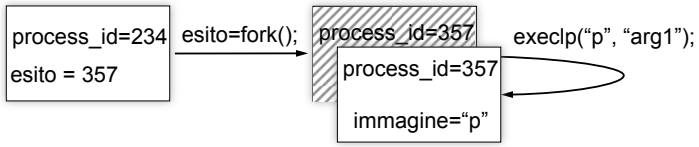
Un esempio di esecuzione di un'immagine

```
esito = fork();              /* crea un processo figlio */
if (esito == 0)              /* se è il figlio */
  { execlp("p","arg1");      /* esegue il programma "p" con argomento "arg1"*/
    error(...);              /* ...a meno di errori */
  }
...                          /* qui procede solo il padre */
```



Un esempio di esecuzione di un'immagine

```
esito = fork();              /* crea un processo figlio */
if (esito == 0)              /* se è il figlio */
  { execlp("p",”arg1”);      /* esegue il programma "p" con argomento “arg1”*/
    error(...);              /* ...a meno di errori */
  }
...                          /* qui procede solo il padre */
```



Terminazione di processi in Unix

- La terminazione di un processo consiste in una serie di operazioni che lasciano il sistema in stato coerente
  - chiusura dei file aperti
  - rimozione dell'immagine dalla memoria
  - eventuale segnalazione al processo padre
- Per gestire quest'ultimo aspetto Unix impiega due funzioni in modo coordinato
  - terminazione dell'esecuzione di un processo (*exit*)
  - attesa della terminazione di un processo da parte del processo che lo ha creato (*wait*)

Le funzioni exit e wait

- La funzione *exit* termina l'esecuzione di un processo
  - segnala al processo che lo ha creato un valore numerico che rappresenta l'esito sintetico (*stato*) dell'esecuzione

```
exit(stato);
```

- La funzione *wait* mette un processo in attesa della terminazione di un processo figlio
  - restituisce l'identificativo del processo terminato e il suo stato di esecuzione

```
pid = wait(&stato);
```


Un esempio riassuntivo

```
esito = fork();              // crea un processo figlio
if (esito < 0)               // creazione OK?
  { error("fork() non eseguita");
    ...
  }
if (esito == 0)              // è il processo figlio ?
  { execlp("p",...);         // sì, esegue il programma "p"
    error("exec non eseguita");      // ...a meno di errori
    ...
  }
id = wait(&stato);           // il processo padre attende la fine
if (stato == ...)            // del figlio e ne analizza l'esito
  {
    ...
  }
```