Sistemi Operativi – primo modulo
I sistemi a processi

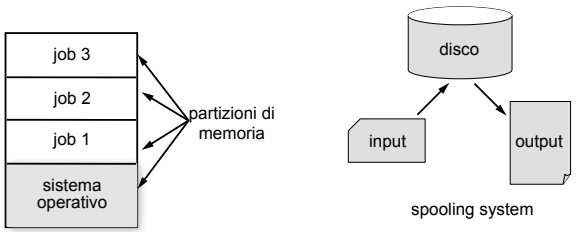
Augusto Celentano
Università Ca' Foscari Venezia
Corso di Laurea in Informatica

Il concetto di processo

- 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

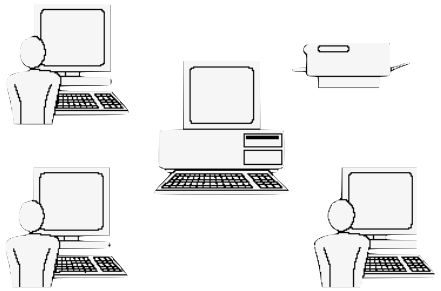
Batch multiprogrammato

- 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



Time sharing interattivo

- 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



Real-time

- 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

Problemi comuni

- 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

Definizione di processo

- 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

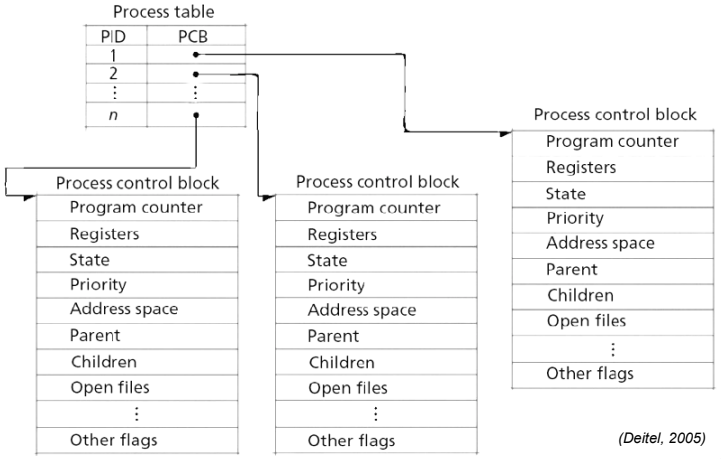
$$P = (C , S)$$

  - *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*)

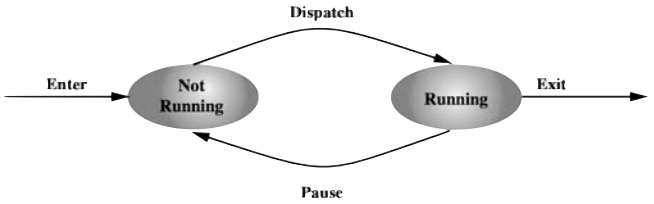
Descrittore di processo (2)



(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



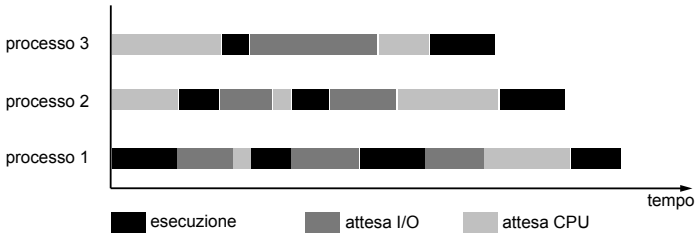
(Stallings, 2011)

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

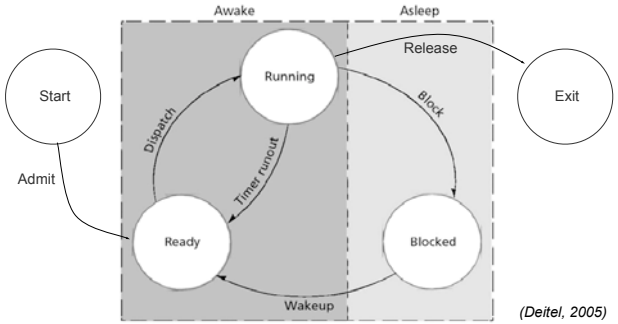
Stati di attività di un processo (3)

- 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)

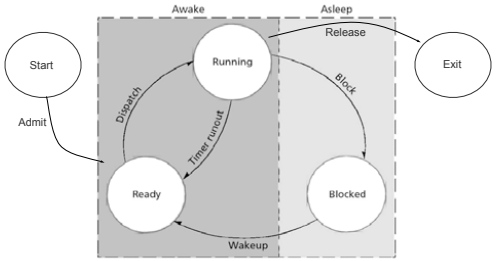
- Per identificare lo stato completo di un processo servono almeno 3 stati
  - attivo, in attesa, pronto
    + inizio, fine



(Deitel, 2005)

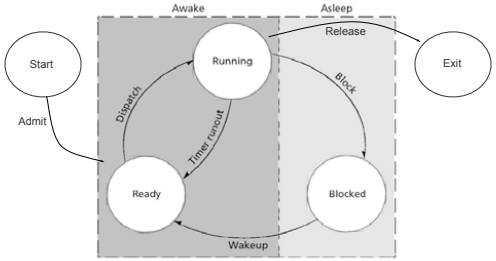
Un modello a 3+2 stati (2)

- *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



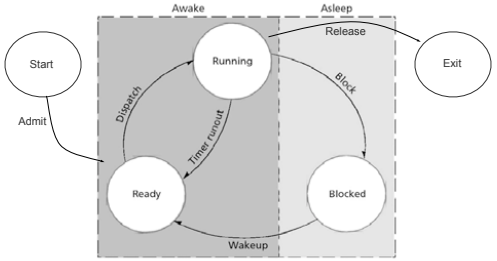
Un modello a 3+2 stati (3)

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



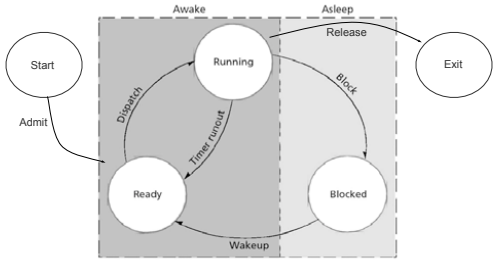
Transizioni di stato (1)

- 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



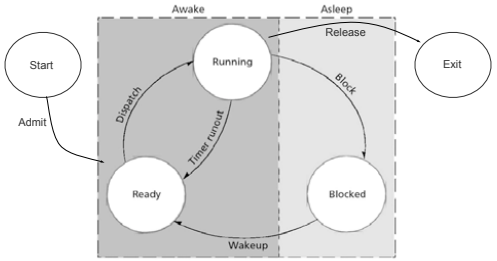
Transizioni di stato (2)

- 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



Un semplice esempio

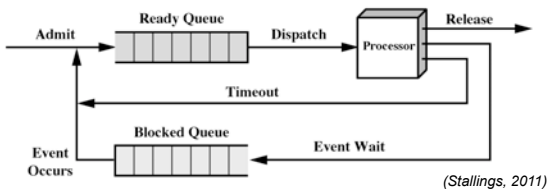
```
#include ...

int main(void)
{
char text[256];
time_t current_time;
char* time_text;

while (gets(text) != NULL) {
    current_time = time(NULL);
    c_time_string = ctime(&current_time);
    printf("%s: %s\n", time_text, text);
    }
exit(0);
}
```

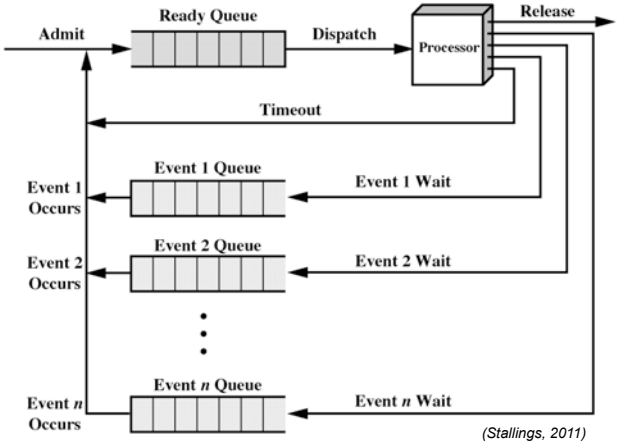

Scheduling dei processi (1)

- 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



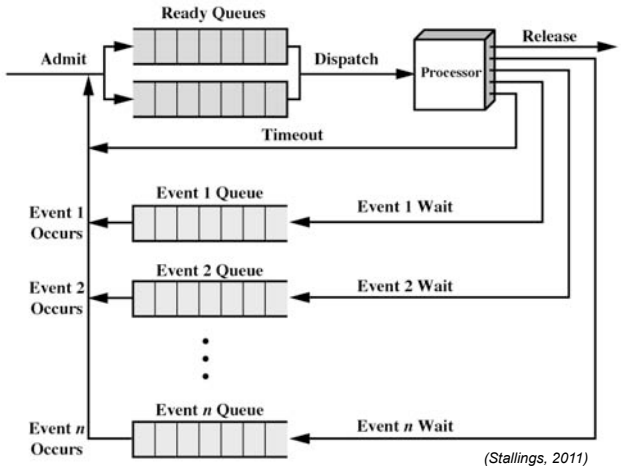
(Stallings, 2011)

Scheduling dei processi (2)



(Stallings, 2011)

Scheduling dei processi (3)

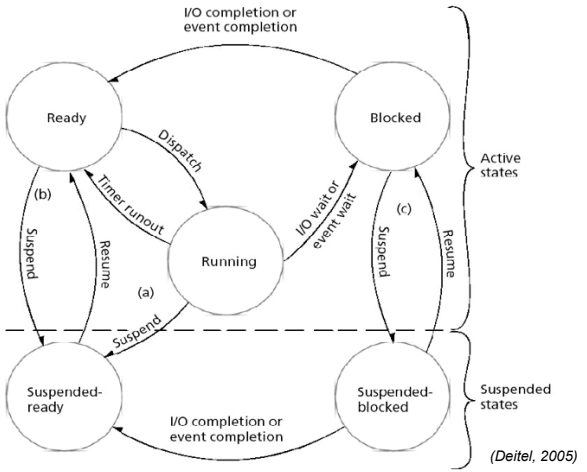


(Stallings, 2011)

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

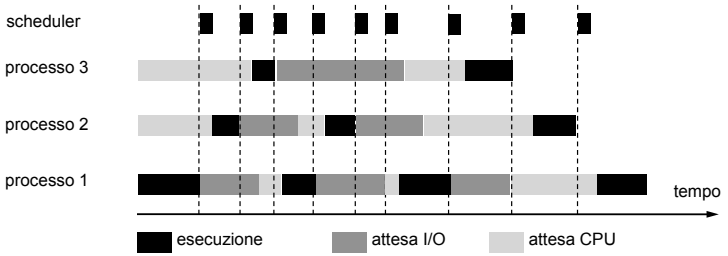
Sospensione dei processi fuori memoria



(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

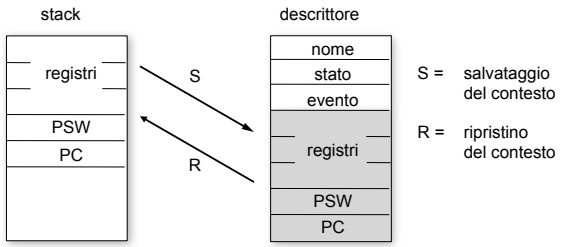


Commutazione di contesto (1)

- 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

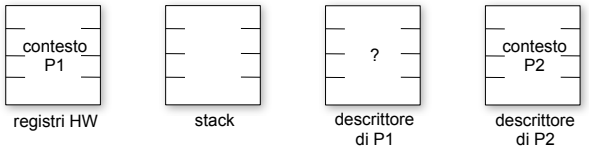
Commutazione di contesto (2)

- 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



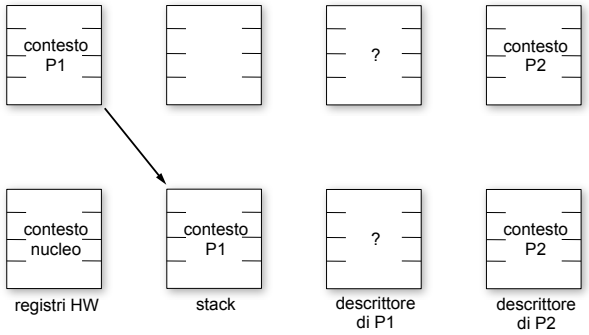
Commutazione di contesto (3)

- 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



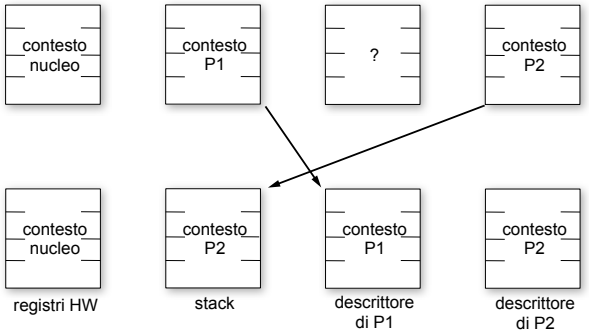
Commutazione di contesto (4)

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



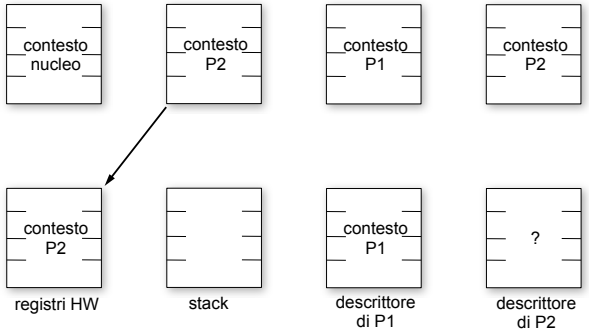
Commutazione di contesto (5)

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

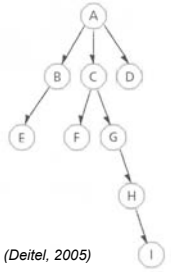


Creazione di un processo (1)

- 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)

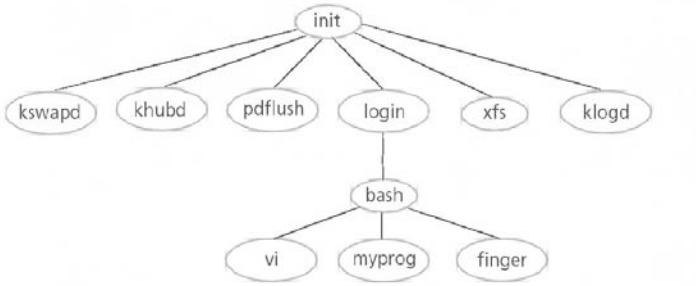
- Un processo può essere creato solo da un altro processo
  - utente
  - di nucleo (del sistema operativo)
- La differenza risiede nella autorizzazioni che il processo creato (*figlio*) eredita dal processo creante (*padre*)
- La creazione di processi può essere iterata a più livelli producendo una struttura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

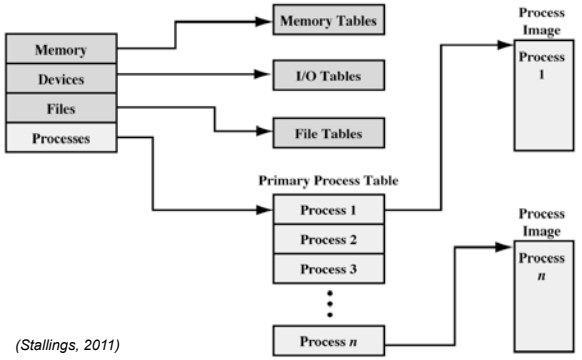
- Un processo termina con una richiesta al sistema operativo (*exit*) che causa
  - la conclusione delle operazioni di I/O bufferizzate
  - il rilascio delle risorse impegnate (memoria, dispositivi di I/O dedicati)
  - la (eventuale) trasmissione di dati di completamento al processo creante
  - la distruzione 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

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

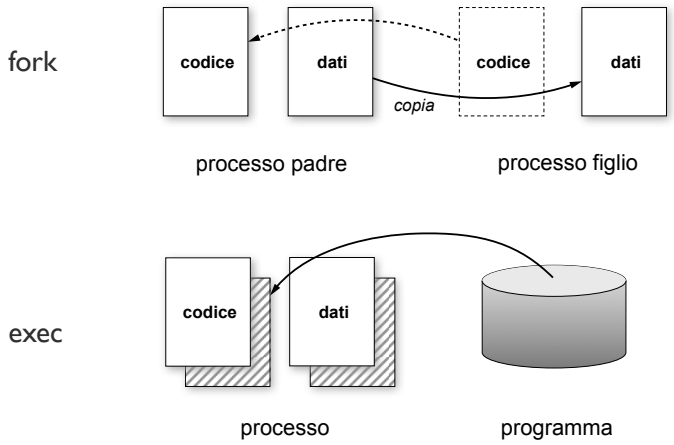
Gestione dei processi in Unix (3)

- 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)

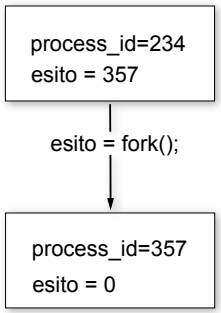


Relazioni tra i processi

- 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

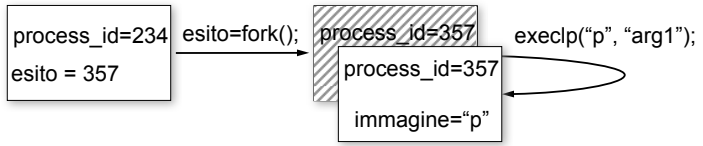
Un esempio di creazione di un processo

```
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 */
```



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
  {
    ...
  }
```


I processi e l'ambiente esterno (1)

- I processi di un sistema multiprogrammato scambiano informazioni con l'ambiente esterno ricevendo e producendo dati
  - da e verso altri processi
  - da e verso archivi permanenti
  - da e verso dispositivi periferici
  - da e verso linee di comunicazione
- Ogni genere di scambio e ogni categoria di dispositivi emettitori/ricettori ha specifiche proprietà e richiede specifiche operazioni
  - è fondamentale, finché possibile, virtualizzare dispositivi e operazioni per aumentare la flessibilità del sistema e ridurre il numero di implementazioni diverse a carico di ogni processo

I processi e l'ambiente esterno (2)

- Il sistema Unix introduce un buon livello di flessibilità attraverso il concetto di file, che rappresenta la virtualizzazione di un generico dispositivo in grado di produrre o ricevere dati in sequenza
  - archivi
  - canali di comunicazione
  - strutture di memoria
  - dispositivi fisici
- Alcune operazioni sono significative e permesse solo su alcuni tipi di "file"
  - solo lettura da una tastiera, solo scrittura su una stampante
  - lettura/scrittura rigorosamente sequenziale da un linea di comunicazione, in ordine sparso su un archivio magnetico su disco

I file in Unix (1)

- Un file Unix è una sequenza di byte identificabile come unità contenente informazioni
  - non c'è alcuna distinzione tra file di testo e file binari
  - la struttura logica è imposta dai programmi applicativi (es. record di un archivio anagrafico, segmento di un'immagine eseguibile, fotogramma di un filmato)
- Ogni file è identificato da un nome simbolico che lo distingue in modo univoco nel sistema (*path name*)
- Opportune convenzioni permettono a un utente umano di interpretare alcune proprietà del file derivandole dal nome
  - un suffisso ne caratterizza il contenuto (.c, .a, .h, .gz, .doc. .xls, .pdf, etc.)
  - una struttura composta di nomi di tipo gerarchico consente di raggruppare i file in cataloghi (*directory*)

I file in Unix (2)

- I file sono elaborati attraverso cinque classi di operazioni, corrispondenti a chiamate al sistema operativo, funzioni e programmi applicativi di base
  - utilizzo di un file esistente: *open*, *read*, *write*, *lseek*, *close*
  - creazione di un nuovo file: *creat*
  - manipolazione dell'associazione tra un file e la sua identificazione in un programma: *dup*
  - definizione delle proprietà del file: *chmod*, *chown*, *stat*, etc.
  - elaborazione della struttura organizzativa dei file: *mv*, *ln*, *mount*, *umount*
- Operazioni più complesse che modificano il contenuto di un file non sono considerate operazioni di base e dipendono dal tipo di file
  - cp, sort

Creazione di un file

- Un file può essere creato specificandone il nome e alcuni parametri che definiscono le sue proprietà

  fd = creat(pathname, mode)

  - *pathname* è il nome simbolico del file
  - *mode* è una composizione di flag che identificano i permessi attribuiti agli utenti (lettura, scrittura, esecuzione)
  - *fd* è un numero che viene associato al file per la sua identificazione nel processo (*file descriptor*)
- Il numero del descrittore è un indice in una tabella in cui sono conservati i descrittori dei file di un processo
  - ogni descrittore contiene informazioni sui diritti di accesso, sullo stato corrente (ultima posizione letta o scritta), nonché puntatori ai buffer di I/O utilizzati
  - ogni processo ha la propria tabella; il sistema mantiene una tabella globale per individuare i file condivisi tra più processi

Apertura di un file

- Prima di poter accedere ad un file esistente un processo deve *aprirlo*, cioè renderlo disponibile all'uso verificandone le proprietà e associandolo al processo

  fd = open(pathname, flags)

  - *pathname* è il nome simbolico del file
  - *flags* è una composizione di indicatori binari che definiscono le operazioni permesse e le modalità della loro esecuzione (es. lettura/ scrittura)
  - *fd* è un numero che viene associato al file per la sua identificazione nel processo (*file descriptor*)
- Esiste una forma più generale per aprire un file esistente oppure crearlo se non esiste

  fd = open(pathname, flags, mode)

  - *mode* è una composizione di flag che che identificano i permessi attribuiti agli utenti se il file viene creato perché non già esistente

Chiusura di un file

- Un file aperto da un processo può essere *chiuso* quando non deve più essere utilizzato

  close(fd)

  - *fd* è il numero del descrittore del file
- La chiusura di un file determina una serie di operazioni che garantiscono la coerenza del sistema rispetto al file e viceversa
  - vengono eseguite le operazioni di I/O bufferizzate e non completate
  - vengono liberate le risorse occupate dal provesso per la gestione del file (buffer di I/O)
  - viene liberato il corrispondente descrittore nella tabella dei file aperti del processo
  - dal momento che un file può essere condiviso occorre verificare, nella tabella globale dei file, se altri processi stiano utilizzando lo stesso file
- Tutti i file aperti vengono automaticamente chiusi alla fine del processo (ma non del programma, es. exec)

Lettura e scrittura da un file

- Un file può essere letto sequenzialmente a partire dalla posizione corrente

  readcount = read(fd, buffer, nbyte)

  - *fd* è il numero descrittore del file
  - *buffer* è l'indirizzo di un'area dati del processo dove verranno depositati i dati letti
  - *nbyte* è il numero di byte da leggere
  - *readcount* è il numero di byte effettivamente letti (<= nbyte, < se end-of-file o errore)
- Un file può essere aggiornato scrivendo sequenzialmente nuovi dati a partire dalla posizione corrente

  writecount = write(fd, buffer, nbyte)

  - *writecount* è il numero di byte effettivamente scritti

Un esempio di operazioni sui file

- Duplica il contenuto del file "from" sul nuovo file "to"

```
fd1 = open("from", O_RDONLY);
if (fd1 < 0)
   { printf("Cannot open file from\n"; exit(fd1); }
fd2 = creat("to",S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH);
if (fd2 < 0)
   { printf("Cannot create file to\n"; exit(fd2); }
while (readcount = read(fd1, buffer, MAXBUFFER) > 0)
   writecount = write(fd2, buffer, readcount);
close(fd1);
close(fd2);
```

- Possono verificarsi errori
  - in creazione (permessi)
  - in apertura (permessi, file non esistente)
  - in lettura (permessi, errori di dispositivo, end-of-file)
  - in scrittura (permessi, dispositivo saturo, errori di dispositivo)

Standard input e standard output (1)

- Tutti i processi in Unix hanno accesso a tre file predefiniti
  - *standard input*: associato al descrittore 0, è normalmente il terminale (tastiera) da cui viene eseguito il processo
  - *standard output*: associato al descrittore 1, è normalmente il terminale (video) da cui viene eseguito il processo
  - *standard error*: associato al descrittore 2, è normalmente il terminale (video) da cui viene eseguito il processo
- Questi file sono già aperti in fase di inizializzazione del programma eseguito dal processo e non devono essere aperti né creati esplicitamente
  - sono ereditati dal processi padre
  - sono condivisi con il processo padre
- Possono essere chiusi esplicitamente e vi si può operare in modo del tutto normale

Standard input e standard output (2)

- Gli interpreti dei comandi di Unix (shell) definiscono una convenzione per "dirottare" i file standard di ingresso e uscita verso altri dispositivi o verso altri file
  - il processo non sa dove sono realmente collegati i suoi input e output standard (può saperlo utilizzando altre funzioni del file system)
- Questi meccanismo consente di realizzare in modo semplice programmi "filtro" che elaborano sequenzialmente dati da input a output senza gestire esplicitamente l'associazione ai file
- L'associazione avviene con una opportuna sintassi nella linea di comando di shell
  - standard input è dirottato sul file che segue il carattere '<'
  - standard output è dirottato sul file che segue il carattere '>'
  - standard error è dirottato sul file che segue i caratteri ">2"

filter <from_file >to_file >2 log_file

Un esempio di operazioni su input e output standard

- Duplica il contenuto di un generico file su un nuovo file utilizzando standard input e standard output ridiretti da shell verso i file desiderati

```
while (bytecount = read(0, buffer, MAXBUFFER) > 0) {
    writecount = write(1, buffer, readcount);
    if (writecount != readcount) }
        write(2, "Errore nella scrittura, \n", 25);
        break;
        }
    }
```

cp <from_file >to_file

Pipe e pipeline (1)

- Un'applicazione interessante e efficace del concetto di input e output standard è dato da *pipe* e *pipeline*
  - sono strumenti sintattici di una shell che permettono di comporre più programmi in sequenza collegando l'ouput di ciascuno all'input del successivo
  - i programmi realizzano un'elaborazione a più stadi (*pipeline*) in cui ogni programma rappresenta uno stadio, e i dati fluiscono da uno stadio all'altro in modo sequenziale e sincronizzato

prog1 < input | prog2 | prog3 > output

- Pipe e pipeline sono file da un punto di vista concettuale, ma la loro implementazione non coinvolge archivi memorizzati (file in senso classico)
  - i trasferimenti avvengono attraverso aree di memoria gestite come file condivisi

Pipe e pipeline (2)

- Esempio: produrre nel file "listing.dat" l'elenco dei file che non contengono codice in linguaggio C (*.c) o in linguaggio java (*.java)

```
ls -1 >temp1
grep -v ".c$" <temp1 > temp2
grep -v ".java$" <temp1 >listing.dat
rm temp1 temp2

ls -1  |  grep -v ".c$"  |  grep -v ".java$"   >listing.dat
```


Pipe e pipeline (3)

- Esempio: produrre un elenco ordinato delle parole presenti in un testo ciascuna preceduta dal numero di occorrenze
- Si utilizzano alcuni filtri di Unix
  - tr: traslittera caratteri, parole e simboli
  - sort: ordina le righe di un testo
  - uniq: rimuove le righe duplicate in un testo

```
tr -sc "[:alpha:]" "\n"  < testo.txt  |
tr "[:upper:]" "[:lower:]"  |
sort   |
uniq -c
```


Pipe e pipeline (4)

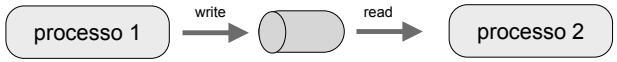
- Esempio: produrre un elenco ordinato delle parole presenti in un testo ciascuna preceduta dal numero di occorrenze

```
tr -sc "[:alpha:]" "\n"  < bohemian_rapsody.txt  |
tr "[:upper:]" "[:lower:]"  | sort | uniq -c
```

10 a	1 at	3 blows	1 dead
1 aching	1 away	1 body	1 devil
1 again	2 baby	1 born	1 didn
1 against	1 back	3 boy	2 die
4 all	1 because	1 but	2 do
6 and	1 beelzebub	4 can	1 doesn
3 any	1 been	2 carry	1 don
1 anyone	1 begun	1 caught	1 down
1 as	1 behind	3 come	4 easy
1 aside	3 bismillah	1 cry	1 escape

Creazione e utilizzo di pipe (1)

- Una *pipe* è un canale di comunicazione tra due processi che appare ad essi come un *file* su cui sono definiti due descrittori:
  - uno per la lettura
  - uno per la scrittura
- Ciò che viene scritto attraverso il descrittore per la scrittura può essere letto attraverso il descrittore per la lettura



Creazione e utilizzo di pipe (2)

- La funzione *pipe* crea una coppia di descrittori di file che corrispondono ai due lati di una pipe

```
int pipefd[2];
esito = pipe(pipefd);
```

  - pipefd[0] è il descrittore per la lettura
  - pipefd[1] è il descrittore per la scrittura
  - i dati scritti su pipefd[1] possono essere letti da pipefd[0]
  - lettura e scrittura sono automaticamente sincronizzati in modo corretto
  - la funzione restituisce 0 se la pipe è stata creata, -1 in caso di errore

Creazione e utilizzo di pipe (3)

- Esempio: un processo genera un figlio e gli invia dati su una pipe

```
int main(int argc, char* argv[])
{
   int pfd[2];
   int pid;
   char s[512];

   if (pipe(pfd) == -1) {
      ... errore...
      }
   if ((pid=fork()) < 0 ) {
      ...errore...
      }
   if (pid==0) {    /* figlio */
      close(pfd[1]);
      while ((read(pfd[0],s,512)) != 0) {
         printf("%s\n", buf);
         }
      close(pfd[0]);
      }

   else {                  /* padre */
      close(pfd[0]);
      strcpy(buf,"...");
      write(pfd[1], buf, strlen(buf)+1);
      close(pfd[1]);
      }
   exit(0);
}
```


Creazione e utilizzo di pipe (4)

- Una pipe consente la comunicazione solo tra processi legati da una relazione padre-figlio
  - il descrittore di ciascuno dei due lati della pipe deve essere lo stesso nei due processi che comunicano
  - i descrittori dei file vengono ereditati da padre a figlio
- Problema: nel caso in cui il processo figlio esegua una exec la comunicazione non può più avvenire
  - il descrittore di pipe è una variabile locale
  - sopravvive alla fork, quindi padre e figlio possono comunicare, ma non sopravvive all'exec, per cui il nuovo programma non sa che descrittore usare
  - non è possibile in questo modo far comunicare due processi se eseguono programmi distinti
- Soluzione: si ricorre a una funzione che consente di manipolare i numeri dei descrittori

Pipelining (1)

- Le funzioni *dup* e *dup2* permettono di duplicare un descrittore di file assegnandogli un numero diverso

```
newfd = dup(oldfd)
newfd = dup2(oldfd, newfd)
```

  - dup crea uno nuova copia del descrittore oldfd usando il più piccolo numero disponibile
  - dup2 crea uno nuova copia del descrittore oldfd usando un nuovo numero specificato
  - il file è condiviso tra vecchio e nuovo descrittore
- Si possono usare numeri di descrittori attribuiti secondo una logica "standard" nota ai due processi che devono comunicare
  - le pipeline a livello shell ocmunicano tramite input e output standard (descrittori numero 0 e 1)

Pipelining (2)

- Esempio: realizzare una pipeline per contare il numero di file presenti nella directory corrente: ls | wc -w

```
int main() {
   int pfd[2], pid;

   if (pipe(pfd) == -1) {
      ... errore ...
      }
   if ((pid = fork()) < 0 ) {
      ... errore ...
      }
   if (pid == 0) {      /* figlio */
      close(pfd[1]);
      close(0);          dup2(pfd[0],0);
      dup(pdf[0]);       close(pfd[0]);
      execlp("wc","wc","-w",NULL);
      ... errore ...
      }
   else {               /* padre */
      close(pfd[0]);
      close(1);          dup2(pfd[1],1);
      dup(pdf[1]);       close(pfd[1]);
      execlp("ls","ls",NULL);
      ... errore ...
    }
    exit(0);
}
```


Il ciclo principale di esecuzione di una shell (1)

```
/* read command line until "end of file" */
while (read(stdin, buffer, numchars))
{
      /* parse command line */
      if (/* command line contains & */)
            amper = 1;
      else
            amper = 0;
      /* for commands not part of the shell command language */
      if (fork() == 0)
      {
            /* redirection of IO? */
            if (/* redirect output */)
            {
                  fd = creat(newfile, fmask);
                  close(stdout);
                  dup(fd);
                  close(fd);
                  /* stdout is now redirected */
            }
            if (/* piping */ )
            {
                  pipe(fildes);
```

(Bach, 1986: fig. 7.28)

Il ciclo principale di esecuzione di una shell (2)

```
                  if (fork() == 0)
                  {
                        /* first component of command line */
                        close(stdout);
                        dup(fildes[1]);
                        close(fildes[1]);
                        close(fildes[0]);
                        /* stdout now goes to pipe */
                        /* child process does command */
                        execlp(command1, command1, 0);
                  }
                  /* 2nd command component of command line */
                  close(stdin);
                  dup(fildes[0]);
                  close(fildes[0]);
                  close(fildes[1]);
                  /* standard input now comes from pipe */
            }
            execve(command2, command2, 0);
      }
      /* parent continues over here...
       * waits for child to exit if required
       */
      if (amper == 0)
            retid = wait(&status);
}
```

(Bach, 1986: fig. 7.28)

Processi e attività concorrenti (1)

- Molti problemi sono risolti in modo più adeguato da un insieme di attività distinte che cooperano per raggiungere uno scopo comune
  - programmi interattivi che aprono finestre multiple
  - server di attività concorrenti
  - sistemi che elaborano eventi e segnali provenienti da fonti diverse secondo tempistiche non prevedibili
- In questi casi la soluzione del problema è più efficace se viene ottenuta da un insieme di algoritmi più o meno sincronizzati ognuno dei quali implementa una specifica attività
  - es. ogni algoritmo è realizzato da un processo diverso (processi cooperanti o concorrenti)
  - i processi non sono tra loro rigidamente sequenziali ma si alternano secondo tempistiche proprie, nel rispetto della coerenza complessiva dalla soluzione

Processi e attività concorrenti (2)

- Un insieme di processi concorrenti
  - può condividere una parte dei dati
  - può sincronizzarsi in momenti selezionati dell'esecuzione
- Ciascun processo evolve anche indipendentemente dagli altri
  - i processi operano su dati privati e su dati condivisi
- L'assegnazione della CPU ai processi è regolata dal sistema operativo
  - in base alle politiche di scheduling
  - attraverso meccanismi di commutazione di contesto

Processi e attività concorrenti (3)

- Se i processi hanno poche interrelazioni il cambiamento di contesto non altera in modo sostanziale le prestazioni complessive
  - lo scambio di dati tra processi richiede la gestione di aree di memoria condivise secondo schemi di protezione più complessi
- Se i processi hanno molte interrelazioni e/o condividono molti dati, la realizzazione con processi separati può introdurre un overhead sensibile
  - scheduling
  - sincronizzazione
  - gestione memoria

  } attraverso il kernel

Processi e attività concorrenti (4)

- Il processo è un'unità di allocazione di risorse
  - memoria virtuale per l'immagine del processo
  - controllo su altre risorse esterne (dispositivi I/O, file, …)
- Il processo è un'unità di esecuzione (*dispatching*)
  - identifica un flusso di esecuzione attraverso uno o più programmi
  - l'esecuzione può essere intervallata / sincronizzata con quella di altri processi
  - un processo ha uno stato di esecuzione e alcuni attributi che ne determinano le modalità di esecuzione (es. priorità)
- Queste due proprietà possono essere gestite in modo indipendente

Processi e attività concorrenti (5)

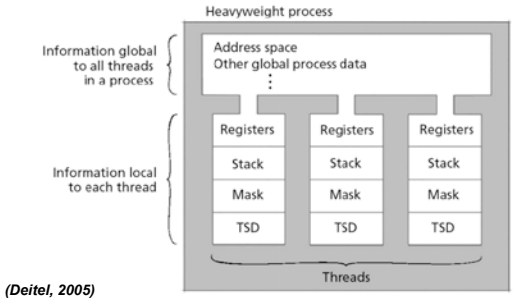
- Separare l'identificazione dell'allocazione risorse dall'identificazione delle proprietà di esecuzione porta a due concettidistinti
  - l'unità di allocazione delle risorse è identificata dal concetto di processo
  - l'unità di esecuzione è identificata dal concetto di *thread* (o *lightweight process*)
- L'intruduzione dei thread nel progetto di un sistema operativo genera una struttura di esecuzione più articolata, basata su
  - condivisione ragionata di risorse
  - differenziazione di più flussi di esecuzione all'interno di un unico processo

Processi e attività concorrenti (6)

- Programmi interattivi che aprono finestre multiple
  - ogni finestra è gestita da un thread diverso, può contenere dati diversi e riferirsi a funzioni diverse (es. edit vs. stampa)
  - il processo nella sua globalità gestisce i menu, riceve memoria e priorità adeguate, garantisce la protezione degli accessi ai file, etc.
- Server di attività concorrenti e indipendenti (es. server Web)
  - ogni attività è servita da un thread diverso
  - la tempistica relativa non deve essere programmata esplicitamente ma deriva dai tempi di servizio di ogni thread
  - i servizi sono normalmente molto brevi; l'overhead della creazione e eliminazione di processi sarebbe troppo oneroso
- Sistemi che elaborano eventi e segnali provenienti da fonti diverse secondo tempistiche non prevedibili
  - segnali diversi sono elaborati da thread diversi
  - il processo nella sua globalità provvede alla gestione dei risultati delle elaborazioni (es. sintesi, report)

Thread (1)

- Un thread è una unità di impiego di CPU all'interno di un processo
- Un processo può contenere più thread, ciascuno dei quali evolve in modo logicamente separato dagli altri thread



(Deitel, 2005)

Thread (2)

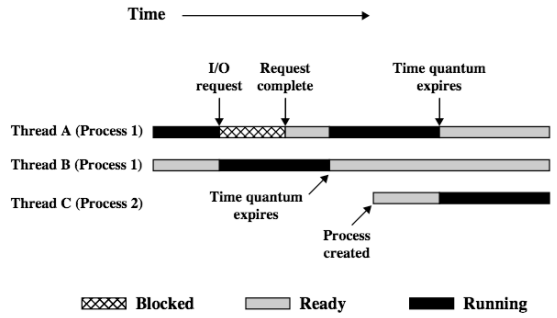
- Ogni thread è caratterizzato da uno stato di esecuzione
  - program counter
  - un insieme di registri
  - uno stack (dati locali)
- Condivide con gli altri thread dello stesso processo il codice, i dati globali e le risorse dell'ambiente esterno (I/O, file, …)
- Ogni thread viene eseguito in modo logicamente indipendente dagli altri
  - lo spazio di indirizzi è unico
  - i thread possono interagire tra loro (in modo controllato)

Thread (3)

- Un thread può essere attivo, in attesa, pronto, terminato, come un processo
  - non esiste lo stato *suspended* (la memoria è una risorsa del processo)
  - la sospensione di un processo sospende tutti i suoi thread
  - la terminazione di un processo termina tutti i suoi thread
- Un processo è una struttura del sistema operativo, un thread è una sottostruttura del processo (*lightweight process)*
- I thread possono essere implementati a livello utente (librerie) o a livello kernel (*system call)*

Scheduling dei thread

- Lo scheduling dei tread segue in gran parte le problematiche dello scheduling dei processi

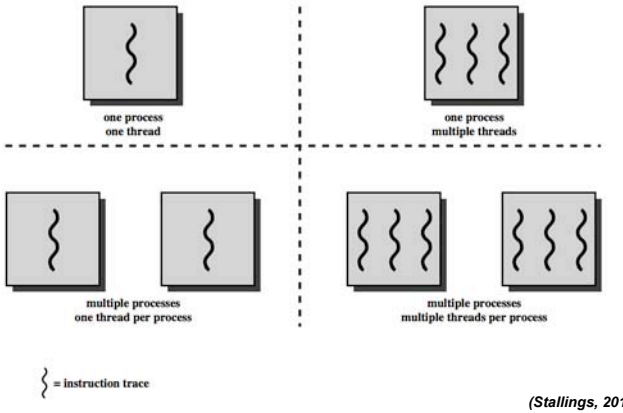


(Stallings, 2011)

Single threading vs. multithreading (1)

- Single threading
  - il sistema operativo non supporta il concetto di thread come entità separata dal processo
  - MS-DOS supporta(va) un solo processo utente con un solo thread di esecuzione
  - UNIX SVR4 e MAC OS supportano più processi utente ma solo un thread per processo
- Multithreading
  - il sistema operativo supporta l'esecuzione di più thread all'interno di un processo
  - Windows 2000/XP/Vista/7, Solaris, Linux, MAC OS X supportano thread multipli per ogni processo

Single threading vs. multithreading (2)



(Stallings, 2011)

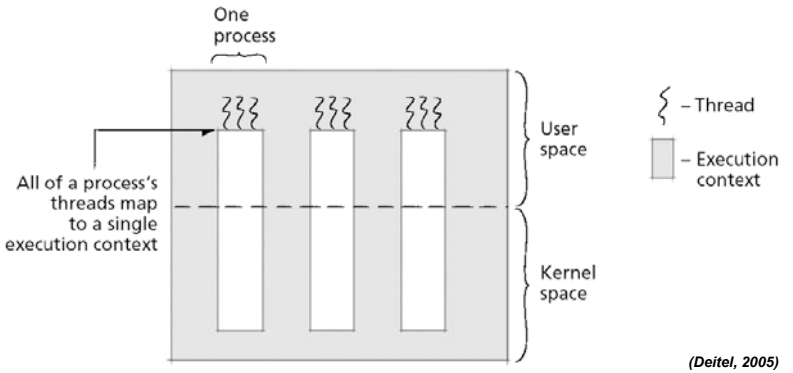
Thread vs processi

- La gestione dei thread è più veloce rispetto alla gestione dei processi
  - creazione
  - terminazione
  - commutazione di contesto
- I thread possono comunicare attraverso i dati locali invece che attraverso meccanismi di *interprocess communication* (*IPC*)
  - l'accesso ai dati locali deve essere regolamentato
- Si elimina il cambiamento di contesto dovuto all'intervento del sistema operativo
  - solo per i thread di utente

Thread di utente vs. thread di kernel

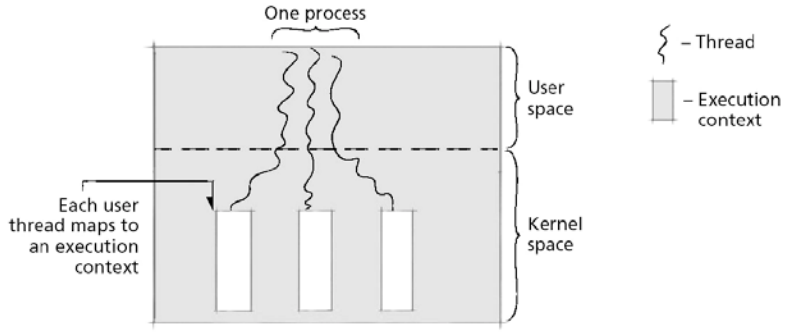
- Thread di utente
  - la gestione è completamente a carico dell'applicazione
  - il kernel non ha cognizione dei thread e non li gestisce
  - realizzato attraverso chiamate a funzioni di libreria
- Thread di kernel
  - il kernel gestisce le informazioni sul contesto dei processi e dei thread
  - lo scheduling delle attività si basa sui thread e non sui processi
  - implementato in Windows 2000/XP e Linux
- Una soluzione mista combina le proprietà di entrambi
  - implementato in Solaris

Thread di utente



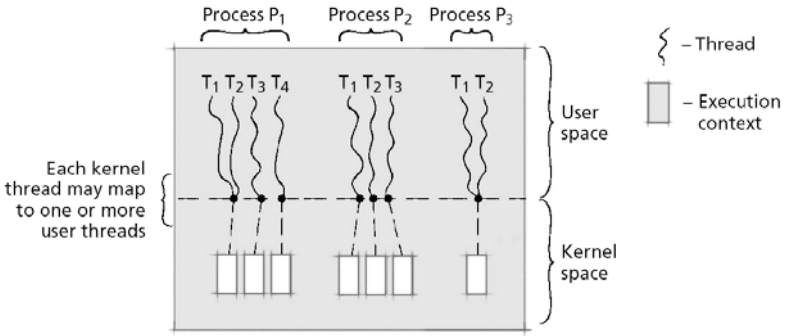
(Deitel, 2005)

Thread di kernel



(Deitel, 2005)

Modello misto di thread



(Deitel, 2005)

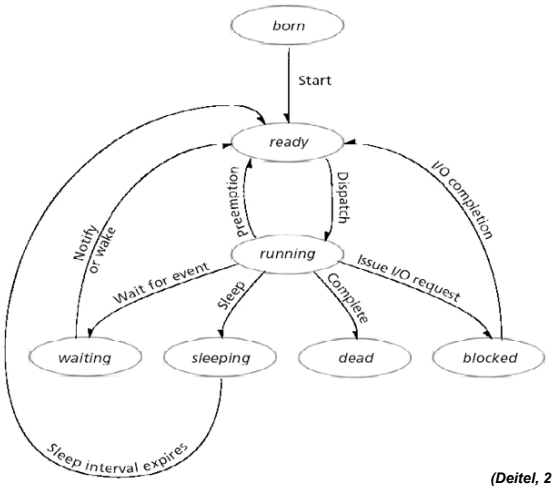
Thread di utente: vantaggi e problemi

- Vantaggi
  - la commutazione tra i thread non richiede l'intervento del kernel
  - lo scheduling dei processi è indipendente da quello dei thread
  - lo scheduling può essere ottimizzato per la specifica applicazione
  - possono essere implementati su qualunque sistema operativo attraverso una libreria
- Problemi
  - la maggior parte delle system call sono bloccanti, il blocco del processo causa il blocco di tutti i suoi thread
  - nei sistemi multiprocessor il kernel non può assegnare due thread a due processori diversi

Thread di sistema: vantaggi e problemi

- Vantaggi
  - in un sistema multiprocessor il kernel può assegnare più thread dello stesso processo a processori diversi
  - la sospensione e l'esecuzione delle attività sono eseguite a livello thread
- Problemi
  - la commutazione di thread all'interno di un processo costa quanto la commutazione di processo
  - cade uno dei vantaggi dell'uso dei thread rispetto ai processi

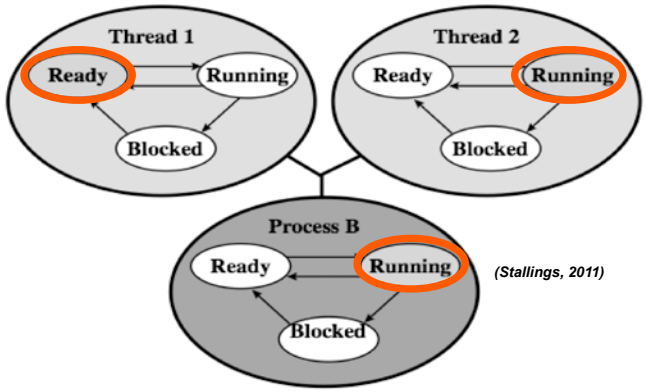
Gli stati di esecuzione di un thread



(Deitel, 2005)

Relazioni tra gli stati di thread e di processo (1)

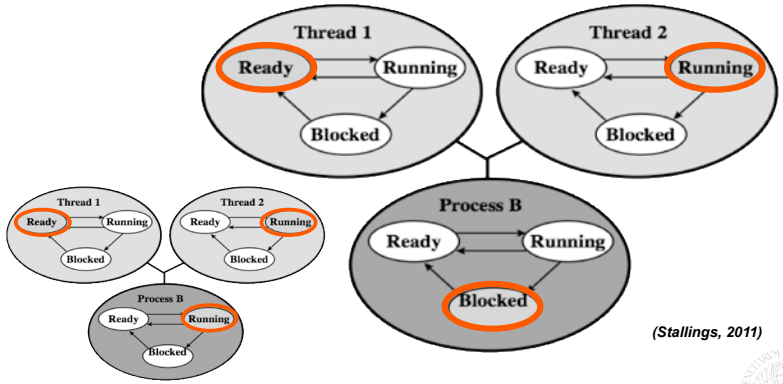
- Il processo B ha due thread, gestiti a livello utente
  - Thread1 è *ready*, Thread2 è *running*
  - il processo è *running*



(Stallings, 2011)

Relazioni tra gli stati di thread e di processo (2)

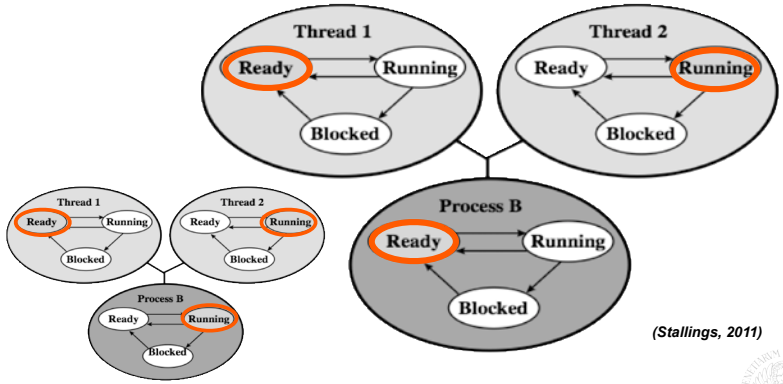
- Thread2 esegue un I/O, non gestito dalla libreria di thread
  - Thread2 continua a essere (logicamente) *running*, il processo è *blocked*
  - quando il processo torna *running*, Thread2 continua l'esecuzione



(Stallings, 2011)

Relazioni tra gli stati di thread e di processo (3)

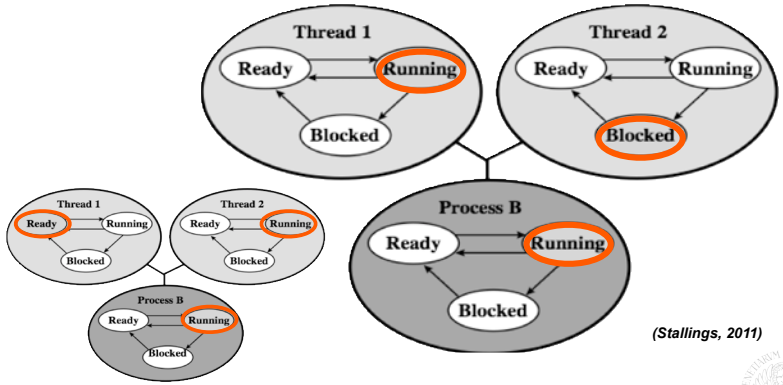
- Il processo va in timeout (scheduling del S.O.) e diventa *ready*
  - Thread2 è ancora (logicamente) *running*
  - quando il processo torna *running*, Thread2 continua l'esecuzione



(Stallings, 2011)

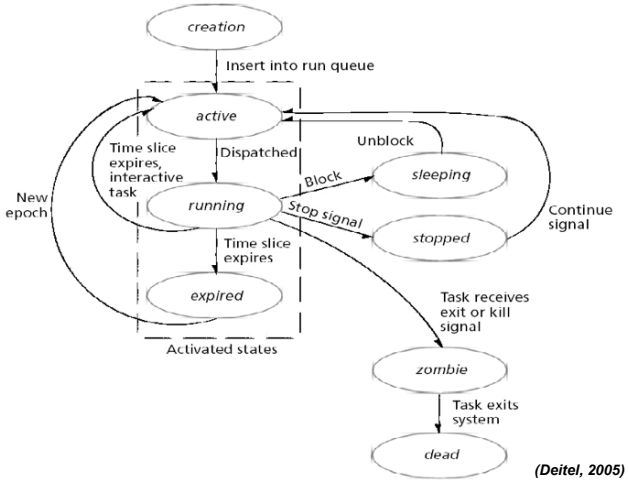
Relazioni tra gli stati di thread e di processo (4)

- Thread2 ha bisogno di un dato da Thread1 e va in stato *blocked*
  - Thread1 è *running*
  - il processo continua a essere *running*



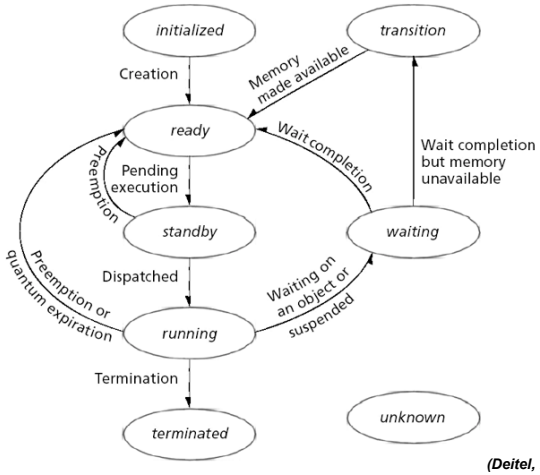
(Stallings, 2011)

Gli stati dei thread in Linux



(Deitel, 2005)

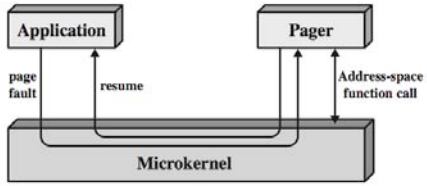
Gli stati dei thread in Windows 2000/XP



(Deitel, 2005)

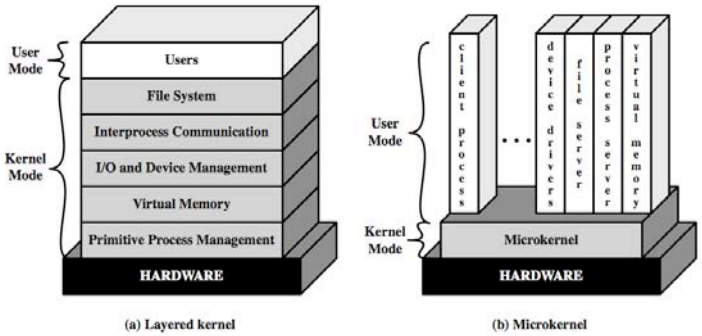
Architetture a microkernel

- Il sistema operativo è composto da un piccolo nucleo che contiene le funzioni fondamentali di gestione dei processi
- Molti servizi tradizionalmente compresi nel sistema operativo sono realizzati come sottosistemi esterni
  - device driver
  - file system
  - gestore della memoria virtuale
  - sistema di windowing e interfaccia utente
  - sistemi per la sicurezza



(Stallings, 2011))

Kernel convenzionale vs. microkernel



(Stallings, 2011)

Vantaggi di un'architettura a microkernel

- Interfaccia uniforme delle richieste di servizio da parte dei processi
  - tutti i servizi sono forniti attraverso lo scambio di messaggi
- Estendibilità, flessibilità, portabilità
  - è possibile aggiungere, eliminare, riconfigurare nuovi servizi senza toccare il kernel
- Affidabilità
  - progetto modulare, *object oriented design*
  - verificabilità (kernel limitato nelle dimensioni e nelle funzioni)
- Supporto per i sistemi distribuiti
  - lo scambio di messaggi è indipendente dall'organizzazione reciproca di mittente e destinatario