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 i 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

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)

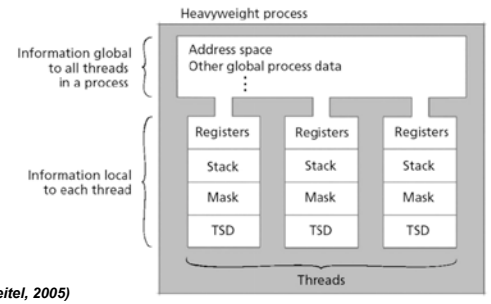
- Separare l'identificazione dell'allocazione risorse dall'identificazione delle proprietà di esecuzione porta a due concettidistinti
  - l'unità di allocazione delle risorse è identificata dal concetto di processo
  - l'unità di esecuzione è identificata dal concetto di *thread* (o *lightweight process*)
- 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)

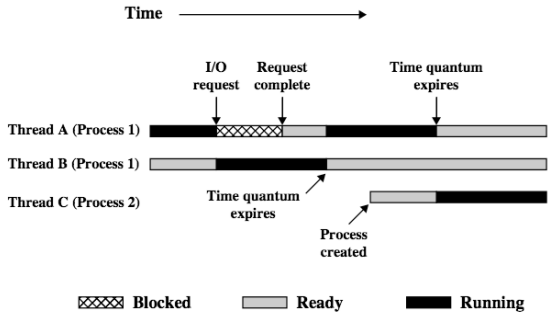
- Ogni thread è caratterizzato da uno stato di esecuzione
  - program counter
  - un insieme di registri
  - uno stack (dati locali)
- Condivide con gli altri thread dello stesso processo il codice, i dati globali e le risorse dell'ambiente esterno (I/O, file, …)
- Ogni thread viene eseguito in modo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

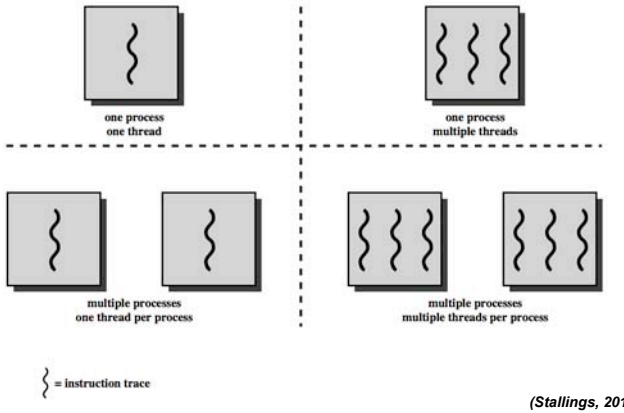


(Stallings, 2011)

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)

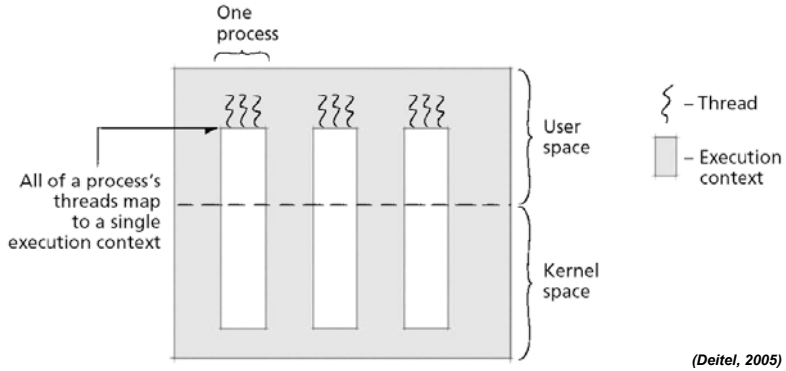
Thread vs processi

- 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

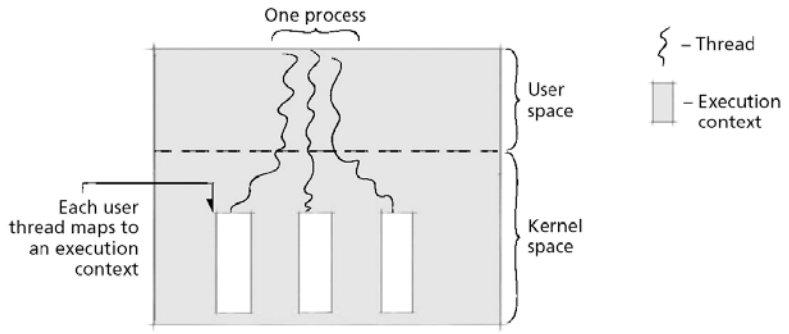
- Thread di utente
  - la gestione è completamente a carico dell'applicazione
  - il kernel non ha cognizione dei thread e non li gestisce
  - realizzato attraverso chiamate a funzioni di libreria
- Thread di kernel
  - il kernel gestisce le informazioni sul contesto dei processi e dei thread
  - lo 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



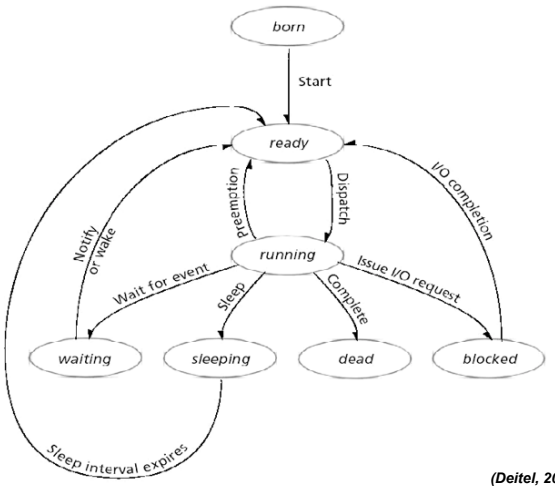
(Deitel, 2005)

Thread di kernel



(Deitel, 2005)

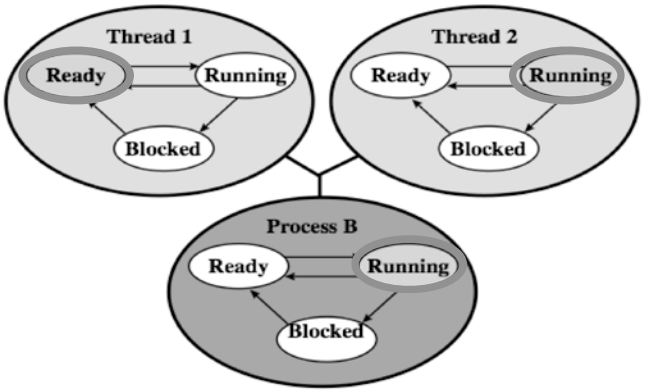
Gli stati di esecuzione di un thread



(Deitel, 2005)

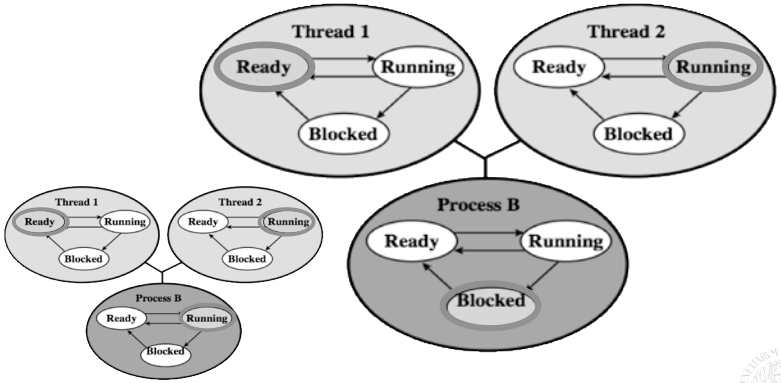
Relazioni tra gli stati di thread e di processo (1)

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



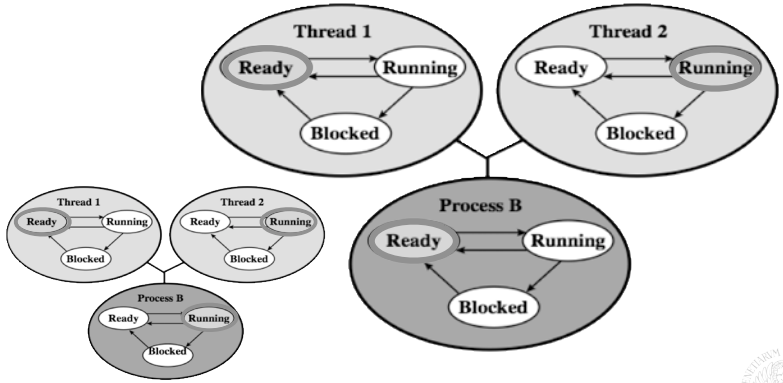
Relazioni tra gli stati di thread e di processo (2)

- 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



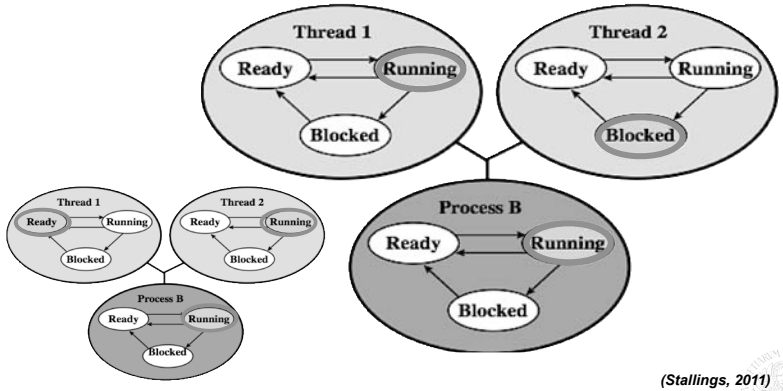
Relazioni tra gli stati di thread e di processo (3)

- 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



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)

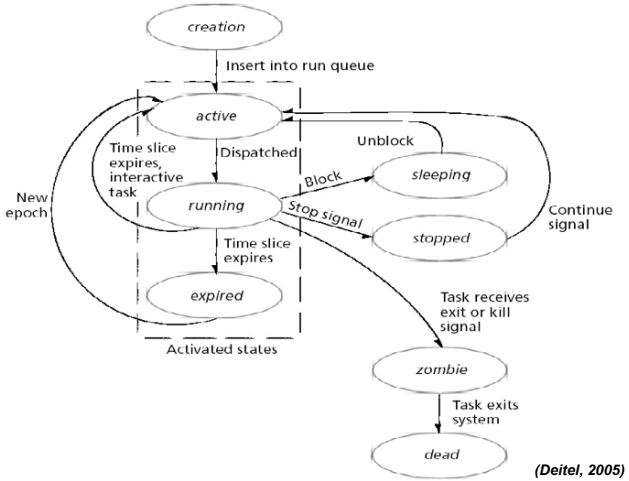
Thread di utente: vantaggi e problemi

- 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

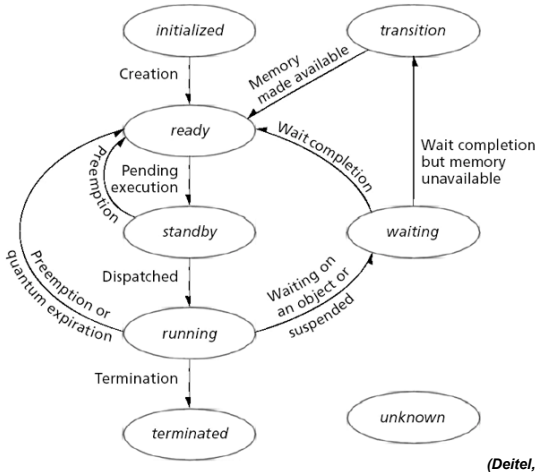
- Vantaggi
  - in un sistema multiprocessor il kernel può assegnare più thread dello stesso processo a processori diversi
  - la sospensione e l'esecuzione delle attività sono eseguite a livello thread
- Problemi
  - la commutazione di thread all'interno di un processo costa quanto la commutazione di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

Gli stati dei thread in Linux



(Deitel, 2005)

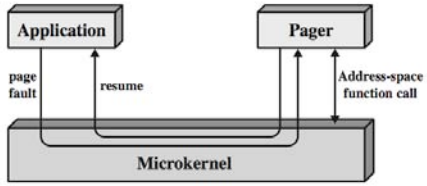
Gli stati dei thread in Windows 2000/XP



(Deitel, 2005)

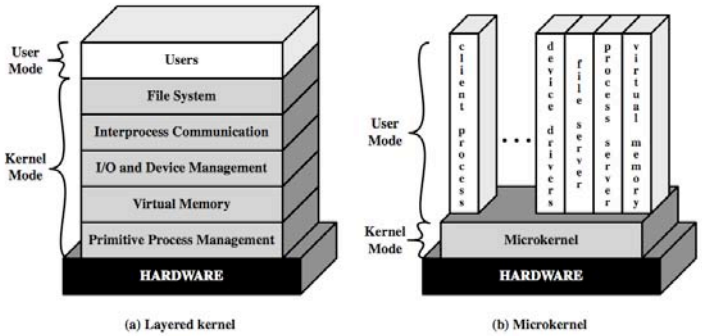
Architetture a microkernel

- 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