

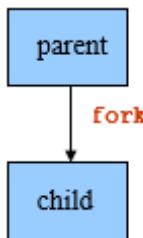
Processi

Il concetto di processo incorpora due nozioni ben distinte:

1. È un elemento che possiede delle risorse: infatti dispone di uno spazio di indirizzamento virtuale, dove si trova l'immagine del processo; saltuariamente, può richiedere zone di memoria aggiuntive e controllo di altre risorse come canali I/O, dispositivi o file
2. È un elemento che viene allocato: essendo una traccia di esecuzione entro uno o più programmi, può essere alternato con altri processi, quindi gli viene attribuito uno stato esecutivo e una priorità di schedulazione. Il compito di schedulare ed allocare risorse è del SO.

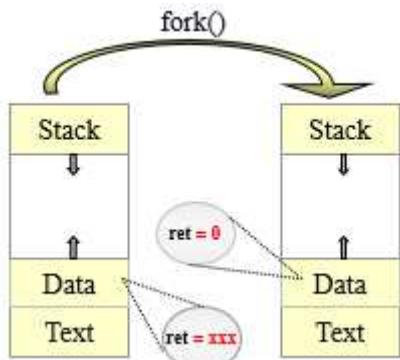
Chiamate a sistema UNIX per creare e terminare processi:

Un processo (PADRE) può generare un altro (FIGLIO) aggiungendo al suo main la chiamata a sistema FORK(). Essi vengono eseguiti concorrentemente ed il figlio è una duplicazione esatta del padre, da ciò si capisce che questa operazione è pesante. Il processo generato a sua volta può crearne di nuovi, così da creare un albero dove la radice è il primo processo padre aperto all'inizio dal SO stesso.



La fork() restituisce:

- o -1 se l'operazione non ha avuto successo
- o 0 dalla parte del figlio
- o L'identificatore del figlio dalla parte del padre



Il processo figlio eredita:

1. Una copia identica della memoria
2. I registri della CPU
3. Tutti i file che erano stati aperti dal genitore

Il codice è lo stesso e l'esecuzione del figlio parte con l'istruzione successiva alla fork(). Il suo contesto di esecuzione è la copia di quella del genitore al tempo della chiamata.

Il padre crea dei figli per fare delle operazioni quindi necessita dei loro risultati per continuare. Per entrare in attesa viene inserita la chiamata WAIT(). Il padre verrà sbloccato quando tutti i figli termineranno, egli però non può attendere la morte dei nipoti. La chiamata ritorna con l'identificatore del figlio o con -1 se esso non esiste. Nel caso in cui vuole attendere la terminazione di uno specifico figlio si usa WAITPID().

```

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
  
```

La fine dell'esecuzione del processo figlio avviene attraverso EXIT(numero). Essa genera un segnale asincrono che sblocca il padre dalla wait(). Questa chiamata a sistema esegue le seguenti operazioni:

- Salva i risultati
- Chiude tutti i file e le connessioni aperti
- Dealloca la memoria
- Controlla se il padre è vivo: se lo è, conserva i risultati finché non gli vengono richiesti (in questo caso il figlio non è morto ma entra nello stato zombie/defunto), se non lo è, il figlio muore (il SO cancella tutto definitivamente).

Nello stato zombie viene conservato solo lo stato, il pid (suo identificatore) e il risultato.

Quando il computer viene acceso o resettato, ci deve essere un programma di partenza che inizializza il sistema. Esso viene chiamato **BOOTSTRAP PROGRAM** e le sue operazioni sono:

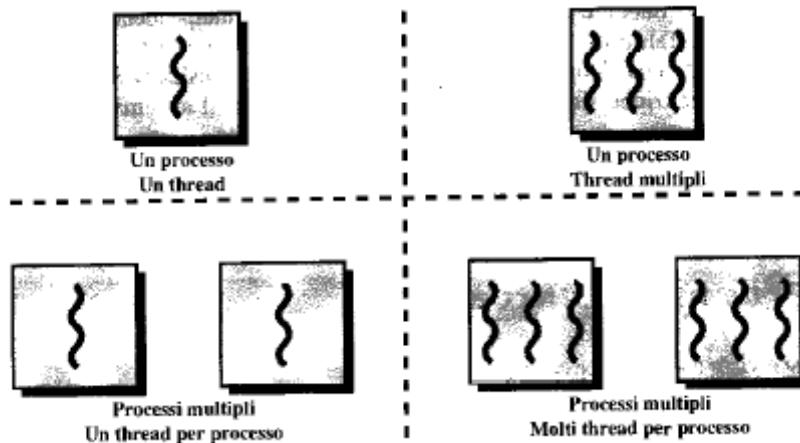
- Inizializza i registri della CPU, i device controllers e la memoria
- Carica il SO in memoria
- Porta allo stato running il SO

Il SO parte con un primo processo ed attende qualche evento, come un hw interrupts o un sw interrupts (traps).

Thread

In molti sistemi operativi queste due parti vengono gestite separatamente. Per distinguerle, l'entità allocata si chiama **THREAD** o **LIGHTWEIGHT PROCESS**, invece quella possidente le risorse è detto **PROCESSO** o **TASK**

La capacità di un sistema di supportare thread di esecuzione multipli per ogni processo, viene chiamato **MULTITHREADING**.



La figura evidenzia 4 approcci diversi:

- Quadrante in alto a sinistra: è tipico di MS-DOS e supporta un singolo processo utente con un solo thread
- Quadrante in basso a sinistra: vengono supportati più processi ma ognuno ha un solo thread, usato negli ambienti UNIX

In questi prime due visioni, il concetto di thread non viene evidenziato

- Quadrante in alto a destra: in un processi sono presenti più thread, un esempio è la macchina runtime di JAVA
- Quadrante in basso a destra: è l'approccio adottato da Windows NT, Solaris, Mach e OS/2, cioè vengono sostegni più processi in cui girano più thread.

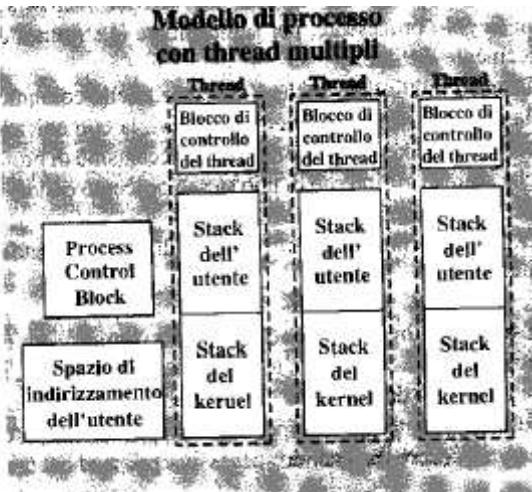
In un ambiente multi-thread, il processo è l'elemento di protezione e di allocazione delle risorse. Ad esso sono associati:

- Uno spazio di indirizzamento virtuale contenente l'immagine del processo
- Accesso protetto ai processori, ad altri processi (per la comunicazione), file e risorse di I/O

All'interno possono trovarsi diversi thread, ognuno di essi possiede:

- Uno stato di esecuzione del thread
- Un contesto, che viene salvato quando il thread corrispondente non è in esecuzione
- Uno stack di esecuzione
- Uno spazio di memoria statico per le variabili locali
- Accesso alla memoria e alle risorse del processo di cui fa parte. Quello in cui entra è uno spazio condiviso con altri thread.

Nell'immagine di sotto, vengono riportati i modelli di un processo con thread singolo e un processo con thread multipli:



È da notare che, nel secondo modello, i thread condividono lo stato, le risorse, lo spazio di indirizzamento ma soprattutto hanno accesso agli stessi dati (Process Control Block e Spazio di indirizzamento utente).

Il vantaggio dell'introduzione dei thread è nelle prestazioni: la loro creazione, eliminazione o scambio è molto più veloce (anche 10 volte) rispetto all'uso di processi. Essi migliorano anche l'efficienza della comunicazione tra programmi in esecuzione, infatti, non è richiesto l'intervento del kernel, in quanto condividono lo spazio di memoria.

I thread hanno due funzionalità principali:

1. Stato di esecuzione:

Gli stati sono gli stessi già visti per i processi. L'unico, però, che non ha senso associare ai thread è Suspend; infatti, se un processo viene caricato in memoria secondaria, anche tutti i suoi thread lo seguono.

Ci sono 4 operazioni associate ad un cambiamento di stato:

- ◊ Creazione: solitamente quando un processo viene creato, nasce anche un thread che, a sua volta, ne potrebbe generare altri.
- ◊ Blocco: un thread viene bloccato quando è in attesa di un evento, viene salvato il contesto (registri utente, program counter e stack pointer). Se un thread si blocca, è possibile l'esecuzione di un altro dello stesso processo.
- ◊ Sblocco: il thread, dopo che si è verificato l'evento, torna Ready
- ◊ Terminazione: accade quando un thread completa il suo compito e quindi i suoi registri e lo stack vengono deallocati

2. Sincronizzazione dei thread:

Ogni thread condivide lo spazio di indirizzamento ed altre risorse, quindi quando le modifica, potrebbe alterare l'ambiente degli altri. La sincronizzazione è necessaria per evitare che ci sia interferenza tra il lavoro di un thread e l'altro.

Esistono 3 diverse implementazioni dei thread:



- i. THREAD A LIVELLO UTENTE puro (fig. a): **USER LEVEL THREAD, ULT**

Tutto il lavoro di gestione dei thread viene fatto a livello utente, quindi il kernel non è a conoscenza, vede solo un blocco unico. La direzione da parte dell'applicazione avviene attraverso una libreria che contiene del codice per creare, distruggere, far comunicare, schedulare, salvare e ricaricare i thread. Ad esempio, la creazione, che può avvenire in qualsiasi momento basta che il processo sia in stato running, viene fatta tramite la chiamata di procedura spawn, utilità della libreria, che passa il controllo a quest'ultima. Durante il

passaggio, il contesto del thread (registri, program counter e stack pointer) viene salvato, e ripristinato alla fine.

I vantaggi di questo approccio sono:

- o Il cambio tra thread non richiede privilegi e quindi interventi da parte del kernel. Perciò si evita il sovraccarico di due cambiamenti di modalità
- o La schedulazione può essere diversa per ogni applicazione, quindi può essere scelto l'algoritmo migliore ed ottimizzarlo senza disturbare lo scheduler del SO
- o Gli ULT possono essere eseguiti su qualunque SO, senza cambiare il kernel, infatti la libreria, insieme di utilità, è condivisa da tutte le applicazioni

Gli svantaggi, invece, sono:

- Le chiamate a sistema, solitamente, sono bloccanti; quindi se un thread ne fa una, non blocca solo se stesso ma anche tutti quelli dello stesso processo
- Una applicazione multithread non può sfruttare il multiprocessing. Infatti il kernel vede il processo come un unico blocco quindi viene assegnato ad un solo processore, perciò due thread non possono essere eseguiti concorrentemente.

Per superare questi due punti, si potrebbe scrivere un'applicazione composta da processi multipli, ma ci sarebbe un peggioramento delle prestazioni.

Invece per aggirare il secondo punto, si potrebbe usare la tecnica chiamata JACKETING. Essa consiste nel trasformare una chiamata di sistema bloccante in non bloccante.

ii. THREAD A LIVELLO DI KERNEL puro (fig. b): (KERNEL LEVEL THREAD, KLT)

Definiti anche leggeri. Il lavoro di controllo dei thread viene fatto dal kernel. Nell'area dell'applicazione non vi è il codice di gestione ma un API (Application Programming Interface – Interfaccia per la Programmazione delle Applicazioni).

Tutti i thread di un'applicazione vengono supportati all'interno di uno stesso processo. Il kernel mantiene sia il loro contesto, sia quella del processo stesso.

I vantaggi:

- o Superamento dei due svantaggi dell'approccio ULT punto (il kernel può schedulare vari thread, e in più se uno è bloccato, ne può scegliere un altro)
- o Le routine dello stesso Kernel possono essere multithread

Lo svantaggio è:

- Il trasferimento del controllo da un thread all'altro, richiede il passaggio in modalità kernel

iii. APPROCCI MISTI (fig. c):

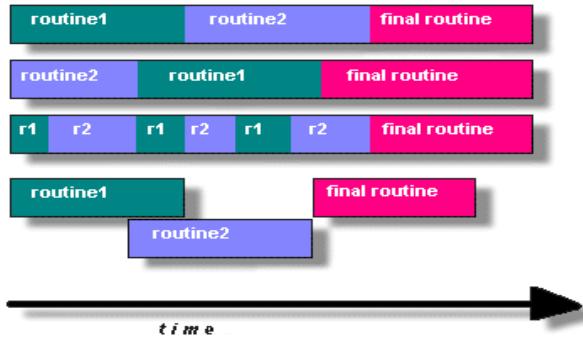
La creazione dei thread avviene a livello utente, come la schedulazione e la sincronizzazione. Gli ULT vengono mappati in un numero \leq di KLT. È da notare che operando in questo modo, vari thread di una stessa applicazione possono operare correntemente, in più una chiamata di sistema non blocca l'intero processo.

Relazione tra thread e processi:

- 1:1 → Ogni thread di esecuzione è un processo unico con il proprio spazio di indirizzamento e le proprie risorse
- M:1 → Ogni processo ha associato un proprio spazio di indirizzamento e delle risorse. In ogni processo si possono creare ed eseguire molti thread
- 1:M → Un thread può spostarsi da un processo all'altro; ciò permette di spostare facilmente i thread fra sistemi diversi
- M:M → Combina le proprietà degli approcci M:1 e 1:M

POSIX THREADS (PThreads):

I PThread sono tipi del linguaggio C e sono definiti in pthread.h. Essi permettono di migliorare le performance del programma, infatti la creazione comporta un minor overhead di SO. Necessita anche di un minor numero di risorse per l'esecuzione. I thread conviene usarli quando i tasks di un'applicazione possono essere parallelizzati. Ad esempio se abbiamo due routine (riga 1):



1. Possono scambiarsi (riga 2)
2. Possono essere interleaved (riga 3)
3. Possono sovrapporsi (riga 4)

I modelli più comuni per rappresentare la struttura dei thread sono:

1. MANAGER/WORKER (o MASTER/SLAVE): il manager gestisce l'input ed assegna il lavoro ai vari thread, i workers
2. PIPELINE: il compito è suddiviso in una serie di sotto-operazioni, ognuna di esse gestita in serie, ma concorrentemente, da diversi thread

Tutti i thread hanno accesso ad una zona comune: la memoria condivisa. È compito del programmatore sincronizzare l'accesso e, quindi proteggere, i dati globalmente condivisi. Comunque hanno una parte privata, molto piccola.

Si definiscono THREAD-SAFENESS quei thread non condizionati dalla race condition, cioè sono sincronizzati in modo da non corrompere i dati condivisi. L'uso di questo tipo di processi leggeri provoca un decadimento delle performance in quanto è necessario salvaguardare determinate celle. Infatti senza sincronizzazione, i thread sarebbero sempre attivi e quindi la CPU lavorerebbe costantemente. Per garantire che non siano corrotte le celle scritte da un determinato thread, a livello applicativo, blocca gli altri presenti sulla stessa applicazione. Però per sfruttare la concorrenzialità, è necessario usare la sincronizzazione su blocchi di poche righe e non su tutto il codice.

Il main() è un singolo thread predefinito. Per generare un altro, e renderlo eseguibile, si utilizza l'istruzione: pthread_create(). Esso a sua volta ne può creare altri. Il numero massimo di thread generabili in un processo è legato alla sua implementazione.

Modi per chiudere un thread:

- È stato completato e quindi ritorna
- Viene invocato pthread_cancel()
- Viene chiamato exit()
- Viene richiesta la routine pthread_exit(), dopo che il thread è stato completato e non necessita di continuare a vivere

Se viene chiuso il main, i thread possono continuare la loro esecuzione se è presente il pthread_exit(). È da sottolineare che questo metodo non chiude i file aperti, quindi si deve tenere conto della pulizia da fare.

Multiprocessing Simmetrico

Tradizionalmente il computer viene visto come una macchina sequenziale, ma questa descrizione non è completamente corretta. Un campo di grande interesse è il parallelismo, il quale ha il fine di migliorare le prestazioni e, in alcuni casi, l'affidabilità. Esso si basa sul controllo del sistema su più elaborazioni contemporaneamente. (Per sistemi paralleli si intendono quei computer con più CPU che lavorano contemporaneamente)

Introduciamo la classifica più famosa sulle varie architetture dei calcolatori: la TASSONOMIA DI FLYNN.

Egli ha enucleato quattro categorie di sistemi di elaborazioni, basandosi sulla molteplicità del flusso di istruzioni e del flusso di dati gestibile:

- I. Istruzione Singola, Dato Singolo (SISD): un singolo processore esegue una singola sequenza di istruzioni operando su dati memorizzati in una singola memoria (Macchina di Von Neumann)
- II. Istruzione Singola, Dati Multipli (SIMD): una singola istruzione macchina controlla l'esecuzione simultanea di più elaborazioni, sincronizzate passo a passo. Ogni elemento di elaborazione ha associata una propria memoria per i dati, così ogni istruzione è eseguita su dati diversi da diversi processori.
- III. Istruzioni Multiple, Dato Singolo (MISD): una sequenza di dati viene trasmessa in un insieme di processori, ognuno dei quali esegue una diversa sequenza di istruzioni. Mai realizzata
- IV. Istruzioni Multiple, Dati Multipli (MIMD): un insieme di processori esegue simultaneamente diverse sequenze di istruzioni su dati diversi

I sistemi che operano in parallelo sono SIMD e MIMD.

I MIMD possono essere ulteriormente divisi, a seconda della modalità di comunicazione tra processori:

- 1) Ogni processore ha una memoria dedicata quindi ogni unità è un computer indipendente. La comunicazione avviene tramite percorsi fissi o rete (bus). Essi sono i CLUSTER o MULTICOMPUTER.
- 2) Ogni processore condivide la memoria, quindi ognuno di essi accede ai programmi e ai dati degli altri. La comunicazione avviene proprio tramite la memoria condivisa. Questo sistema è detto MULTIPROCESSORE A MEMORIA CONDIVISA.

I Multiprocessori a Memoria Condivisa si dividono, a loro volta, a seconda di come i processi vengono assegnati ai processori:

- a) MASTER/SLAVE: il kernel risiede solo su un processore particolare (master), gli altri possono eseguire unicamente programmi utente e utilità del SO. Il master si occupa della schedulazione dei processi o thread. In più uno slave, se necessita di un servizio, deve mandargli una richiesta poi attendere che sia effettuato. L'approccio è semplice ma comporta un duplice svantaggio: se il master fallisce, l'intero sistema cade; il master potrebbe diventare un collo di bottiglia
- b) MULTIPROCESSORE SIMMETRICO (SMP): il kernel può essere eseguito su tutti i processori, ed ognuno di essi schedula autonomamente i processi o i thread disponibili. Il kernel può essere costituito da molti thread o processi che, quindi, possono essere eseguiti in parallelo. Da ciò si può dedurre che il SO viene complicato, infatti bisogna garantire che uno stesso processo non venga schedulato su più processori, o che quelli in coda vengano persi.

Descriviamo in dettaglio l'organizzazione dei SMP.

- Sono presenti più processori
- Ognuno di essi ha: unità di controllo, unità aritmetico- logica e registri
- L' accesso alla memoria condivisa avviene grazie ad un meccanismo di interconnessione, solitamente un bus
- La comunicazione può avvenire attraverso la stessa memoria condivisa oppure direttamente tramite segnali
- La memoria è organizzata in modo che sia possibile effettuare contemporaneamente più accessi allo stesso blocco
- Nei sistemi moderni, i processori hanno, almeno, un livello di memoria cache privata

Quest'ultimo punto, porta ad un ulteriore problema di progettazione, infatti se una parola viene modificata in una cache, potrebbe invalidare altre cache in altri processori. Questo problema è detto di COERENZA DELLA CACHE, e viene risolto a livello hw.

Un SO SMP gestisce i processori e le altre risorse in modo che la visione dell'utente sia di un sistema monoprocesso con multiprogrammazione. Quindi un SO multiprocessore dovrà fornire le stesse funzionalità di uno multiprogrammazione. Caratteristiche fondamentali:

- Processi o Thread concorrenti, lo stesso codice del kernel deve essere eseguito da diversi processori
- Schedulazione, fatta da qualsiasi processore
- Sincronizzazione, necessaria in quanto i processi attivi possono accedere allo stesso spazio di indirizzamento condiviso o alle risorse di I/O comuni
- Gestione della memoria
- Affidabilità e tolleranza dei guasti, il SO deve permettere un decadimento graduale in caso di fallimento dei processori

Microkernel

È un piccolo nucleo di SO che fornisce le basi per estensioni modulari. Il suo utilizzo è divenuto popolare con il sistema operativo Mach. Un altro SO che lo sfrutta è Windows NT. I benefici principali, oltre la modularità, anche la flessibilità o la portabilità.

Architettura:

I primi SO, definiti monolitici, non ponevano molta attenzione sulla strutturazione; essi si basavano sull'idea che ciascuna procedura poteva chiamare ogni altra procedura. In seguito iniziarono a svilupparsi i SO a strati in cui le funzioni sono organizzate gerarchicamente e l'interazione avviene solo tra strati adiacenti. A causa delle molteplici interazioni è difficile garantire la sicurezza.

La filosofia del microkernel è che solo le funzioni assolutamente essenziali del nucleo si dovrebbero essere nel kernel, le applicazioni sono eseguite sopra, cioè a livello utente. I componenti del SO esterni al microkernel sono implementati come processi server, essi interagiscono tramite il passaggio di messaggi su base di parità attraverso il microkernel stesso.

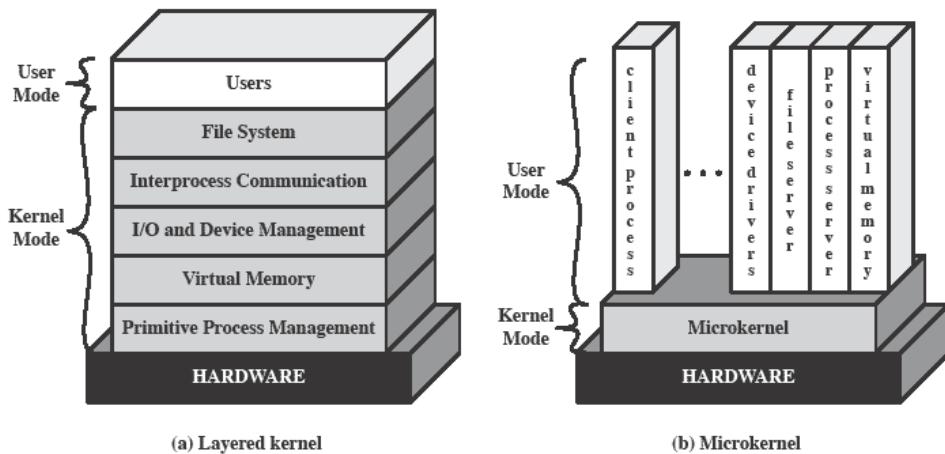


Figure 4.10 Kernel Architecture

Vantaggi:

- Interfaccia Uniforme: legato al passaggio di messaggi
- Estensibilità: facilità di estensione al seguito dell'aggiunta di nuovi servizi
- Flessibilità: alcune caratteristiche possono essere rimosse così da ottenere un'implementazione snella ed efficiente
- Portabilità: il codice che dipende dal tipo di processore è contenuto nel kernel, ciò rende più facili le modifiche necessarie per il trasferimento di su un nuovo processore
- Affidabilità: maggiore è la dimensione del prodotto, più è difficile garantire l'affidabilità
- Supporto ai sistemi distribuiti: usato ad esempio nei cluster da un SO distribuito
- SO orientati agli oggetti: tale approccio permette di sviluppare le estensioni modulari del SO

Prestazioni:

Lo svantaggio principale è quello delle prestazioni. Infatti lo scambio di messaggi richiede più tempo rispetto ad una semplice chiamata a sistema. Esistono due approcci per superare il problema:

1. Estendere il microkernel reintegrando nel SO i server critici e i driver così da ridurre il numero di cambiamento di stato utente-kernel. Ma ciò ridurrebbe i vantaggi come le interfacce minime o la flessibilità.
2. Ridurre il microkernel, molti studi hanno evidenziato il miglioramento di prestazioni e l'aumento di flessibilità ed affidabilità

Progettazione:

Purtroppo non esistono regole facili per decidere quali funzioni vadano fornite dal microkernel e quali invece implementate. Esso deve contenere le funzioni che dipendono direttamente dallo hw e quelle per supportare i server e le applicazioni in modalità utente. Esse vengono raggruppate nelle categorie generali:

- ◊ Gestione primitiva della memoria: il microkernel controlla lo spazio di indirizzamento per rendere possibile l'implementazione della protezione a livello di processo. Finché esso si occupa di mappare ogni pagina virtuale in fisica, il grosso della gestione della memoria si può implementare all'esterno del kernel.

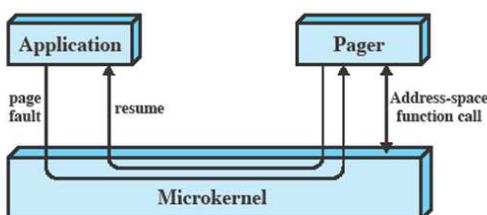


Figure 4.11 Page Fault Processing

- ◊ Comunicazione interprocesso: per comunicare i processi o thread nel microkernel utilizzano i messaggi. Esso contiene:
 - intestazione -> identificatore del processo mittente e ricevente
 - corpo -> contiene i dati, un puntatore ad un blocco di dati o informazioni di controllo relative al processo stesso
- ◊ I/O e gestione degli interrupt: con l'architettura microkernel gli interrupt hw vengono gestiti come messaggi e le porte di I/O sono contenute nello spazio di indirizzamento. Il microkernel può riconoscere gli interrupt ma

non gestirli. Esso genera un messaggio per il processo a livello utente a cui è associato quel determinato interrupt e il kernel ne mantiene la mappatura.