Processi

- Immagine di un processo (codice, variabili locali e globali, stack). Risorse possedute: (file aperti, processi figli, dispositivi di I/O..)
- L'immagine di un processo e le risorse da esso possedute costituiscono il suo spazio di indirizzamento.
- La locazione dello spazio di indirizzamento dipende dalla tecnica di gestione della memoria adottata. Potra' essere contenuto in tutto o solo in parte nella memoria principale (registri base ed indice, impaginazione, segmentazione)
- I processi hanno spazi di indirizzamento distinti, cioe` "non condividono memoria".
- Complessita' delle operazioni di cambio di contesto tra due processi: comportano il salvataggio ed il ripristino dello spazio di indirizzamento (overhead). Analogamente per le operazioni di creazione e terminazione di un processo.

Processi

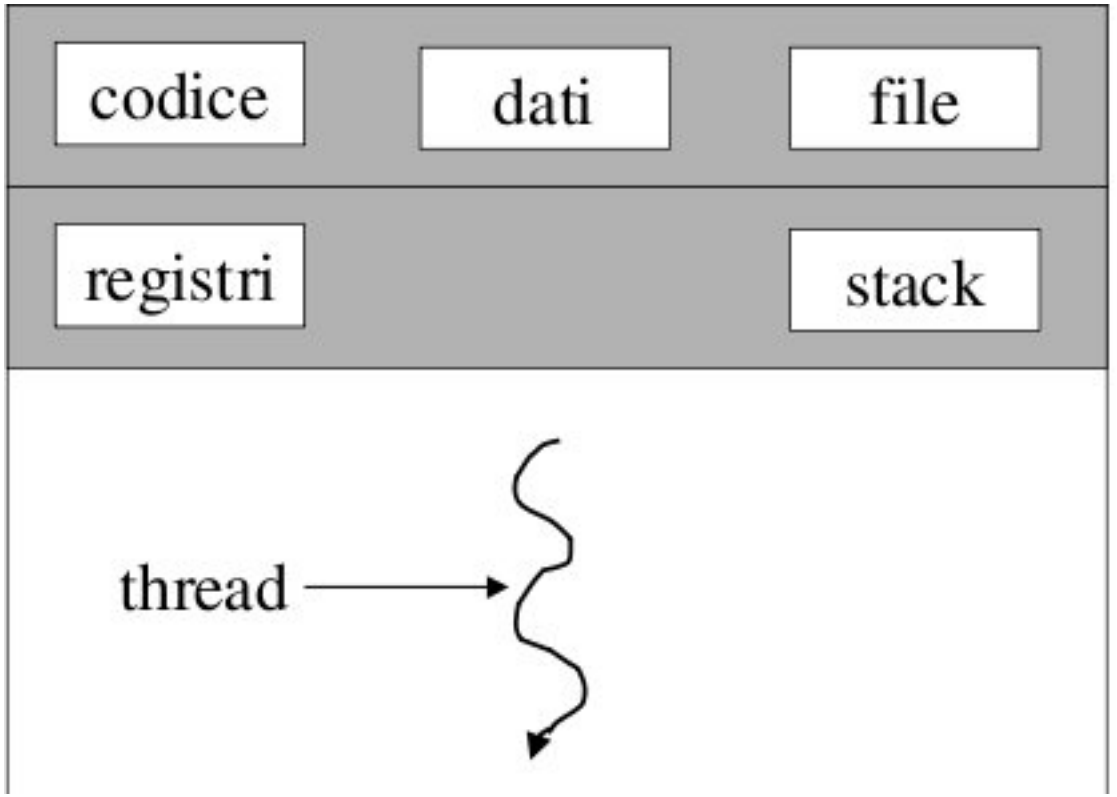
Consideriamo due processi che devono lavorare sugli stessi dati. Come possono fare, se ogni processo lavora su un'area di memoria diversa?

- i dati possono essere scambiati mediante messaggi o memoria condivisa, ma questa soluzione e' inefficiente perche' richiede l'intervento del SO

- i dati possono essere tenuti in un file che viene accedutoa turno dai due processi. Ancora più inefficiente!

Processi

Un normale processo e' contraddistinto da un unico Thread (filo) di computazione



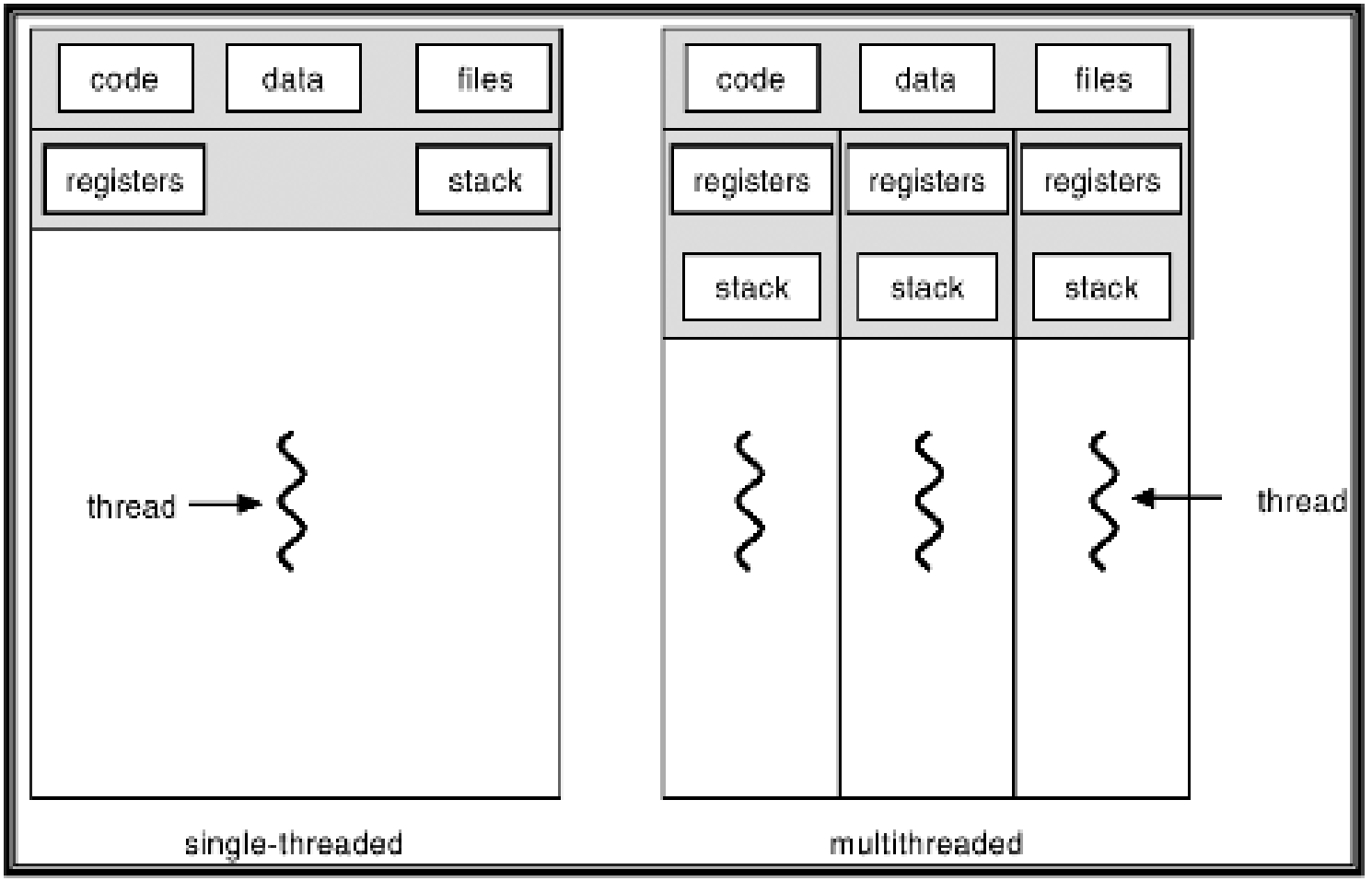
Thread

I threads sono anche chiamati processi leggeri “Lightweight”:

- Un thread rappresenta un flusso di esecuzione all’interno di un processo pesante.
- Multithreading: molteplicita' di flussi di esecuzione all’interno di un processo pesante.
- Tutti i thread definiti in un processo condividono le risorse del processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso a dati comuni.

Thread

Un processo Multi-Thread e' fatto di piu' thread, detti peer thread:



Thread

Ogni thread ha:

- uno stato di esecuzione (runnig, ready, blocked)
- un contesto che e' salvato quando il thread non e' in esecuzione
- uno stack di esecuzione
- uno spazio di memoria privato per le variabili locali
- accesso alla memoria e alle risorse del task condiviso con gli altri thread.

Vantaggi:

- maggiore efficienza: le operazioni di contex switch, creazione etc., sono piu' economiche rispetto ai processi (IL CONTEXT SWITCH TRA PEER THREAD E' MOLTO PIU' VELOCE).
- maggiori possibilita' di utilizzo di architetture multiprocessore.

Thread e Processi

- La differenza fondamentale tra un processo e un insieme di peer thread (o task), e' quindi che:

  . la vita di un processo si sviluppa lungo un unico percorso (o “filo”) di computazione, fatto della sequenza di istruzioni eseguite dal processo.

  . la vita di un task e' fatta di piu' thread, ognuno dei quali e' contraddistinto dal proprio “filo” di computazione

Thread

Ma come si fa a distinguere tra i thread di un task (in modo da far fare a ciascuno una cosa diversa), se questi condividono lo stesso codice?

Con delle opportune funzioni di libreria (system call), che all'interno del codice permettono di creare i thread necessari, e specificano per ciascun thread quale pezzo di codice deve eseguire quel thread

```
int n, in, out;
type item = ...;
item buffer [n];
```

ora le variabili sono automaticamente condivise

```
void  *produttore( );          // contiene il codice del thread produttore
void  *consumatore( );         // contiene il codice del thread consumatore
```

```
main ( ) /* produttore e consumatore */
{
...
pthread_create(produttore);              // crea il thread produttore
pthread_create(consumatore);             // crea il thread consumatore
}
```

Il main specifica quanti thread ci sono nel task e associa ad ogni thread il suo codice

```
void *produttore ( )
{
qui il codice del produttore
}
```

il codice del produttore è lo stesso di prima

```
void *consumatore ( )
{
qui il codice del consumatore
}
```

il codice del consumatore è lo stesso di prima

Thread realizzazione

User-level threads (es. Java):

- Libreria di funzioni (thread package) che opera a livello utente e fornisce il supporto per la creazione, terminazione, sincronizzazione dei thread e per la scelta di quale thread mettere in esecuzione (scheduling).
- Il sistema operativo ignora la presenza dei thread continuando a gestire solo i processi.
- Quando un processo e' in esecuzione parte con un solo thread che può creare nuovi thread chiamando una apposita funzione di libreria.
- La soluzione e' efficiente (tempo di switch tra thread), flessibile (possibilità di modificare l'algoritmo di scheduling), scalabile (modifica semplice del numero di thread).

Thread realizzazione

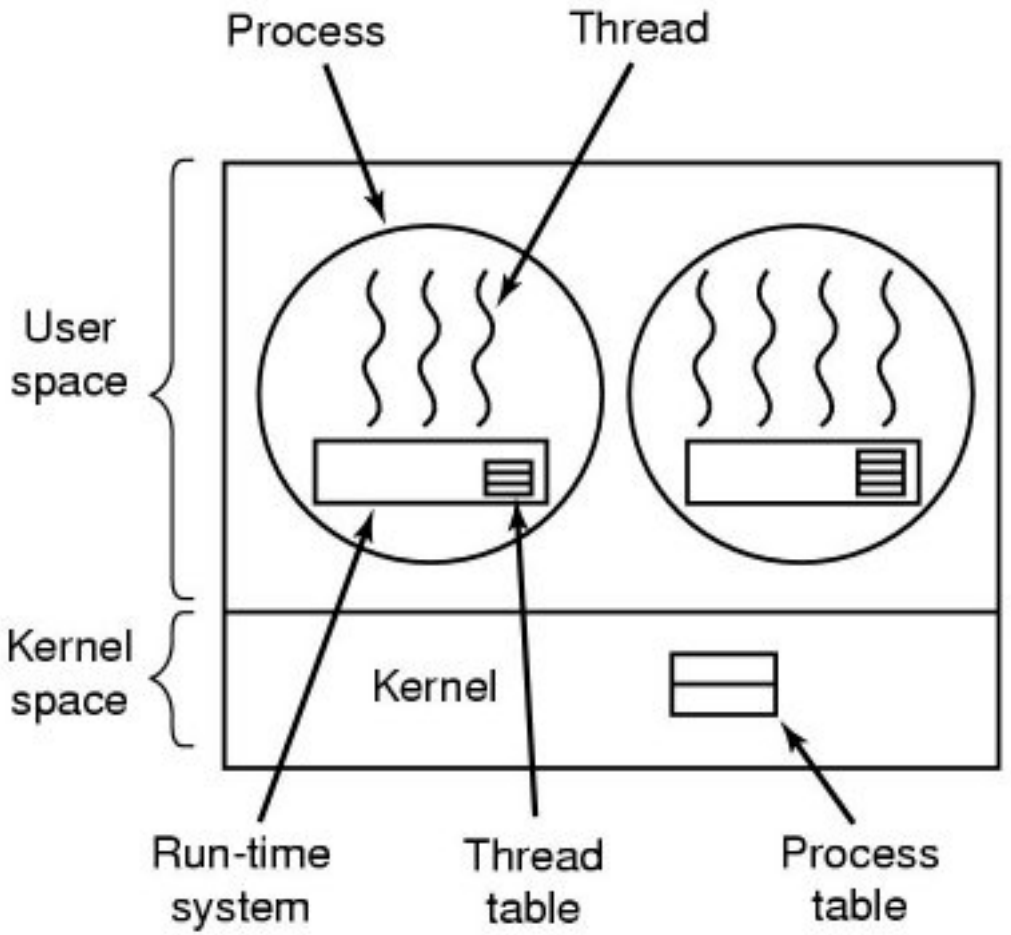
- Se un thread si blocca in seguito ad una chiamata ad una funzione del package (es. wait), va in esecuzione un altro thread dello stesso processo.

- I thread possono chiamare delle system call (es. I/O): intervento del sistema operativo che blocca il processo e conseguentemente l'esecuzione di tutti i suoi thread.

- Il S.O. interviene anche nel caso allo scadere del quanto di tempo assegnato ad un processo (sistemi time sharing).

- Non e' possibile sfruttare il parallelismo proprio di architetture multiprocessore: un processo (con tutti i suoi thread) e' assegnato ad uno dei processori.

User-Level Thread

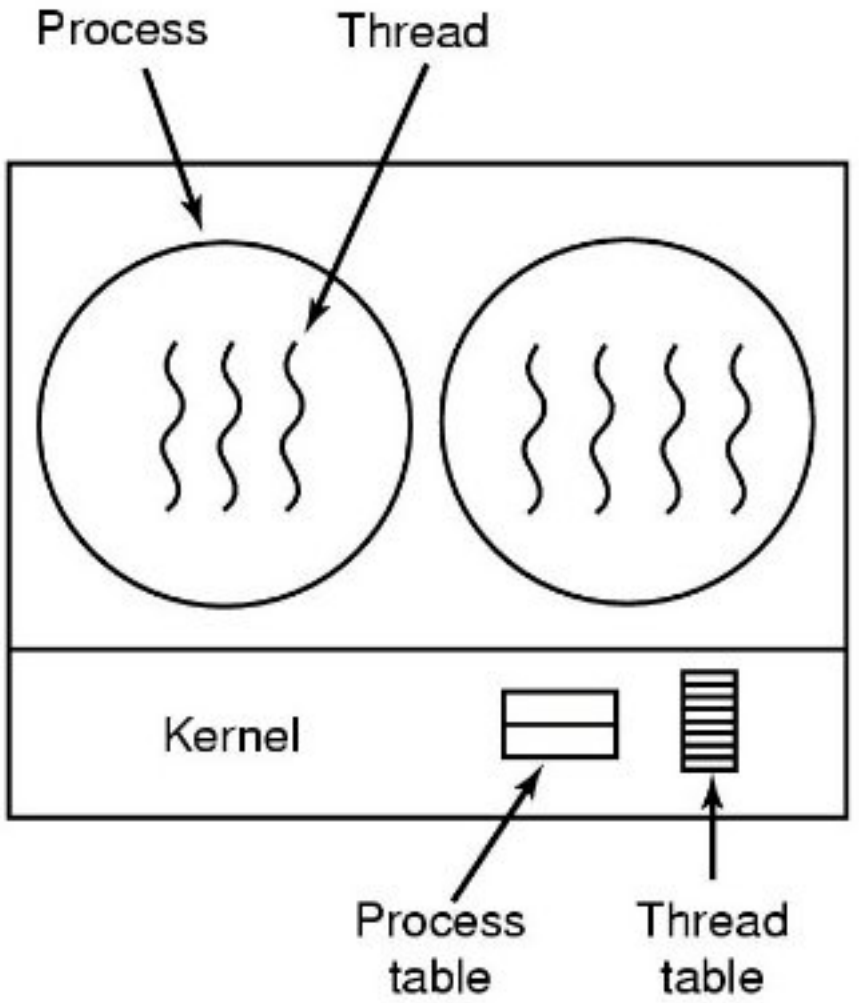


Thread realizzazione

Kernel-level threads

- Il S.O. si fa carico di tutte le funzioni per la gestione dei thread. Mantiene tutti i descrittori dei thread (oltre a quelli dei processi).
- A ciascuna funzione corrisponde una system call.
- Quando un thread si blocca il S.O. puo' mettere in esecuzione un altro thread dello stesso processo.
- Possibilita' di eseguire thread diversi appartenenti allo stesso processo su unita' di elaborazione differenti (architettura multiprocessore).
- La commutazione fra thread user-level non richiede l'intervento del SO ed è quindi piu' veloce
- La commutazione fra thread richiede il SO, ma un thread bloccato non blocca tutto il task

Kernel-Level thread



Thread realizzazione

Soluzione mista (es. Solaris):

- Creazione di thread, politiche di assegnazione della CPU e sincronizzazione a livello utente.
- I thread a livello utente sono mappati in un numero (minore o uguale) di thread a livello nucleo.

Vantaggi:

- Thread della stessa applicazione possono essere eseguiti in parallelo su processori diversi.
- Una chiamata di sistema bloccante non blocca necessariamente lo stesso processo