Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura

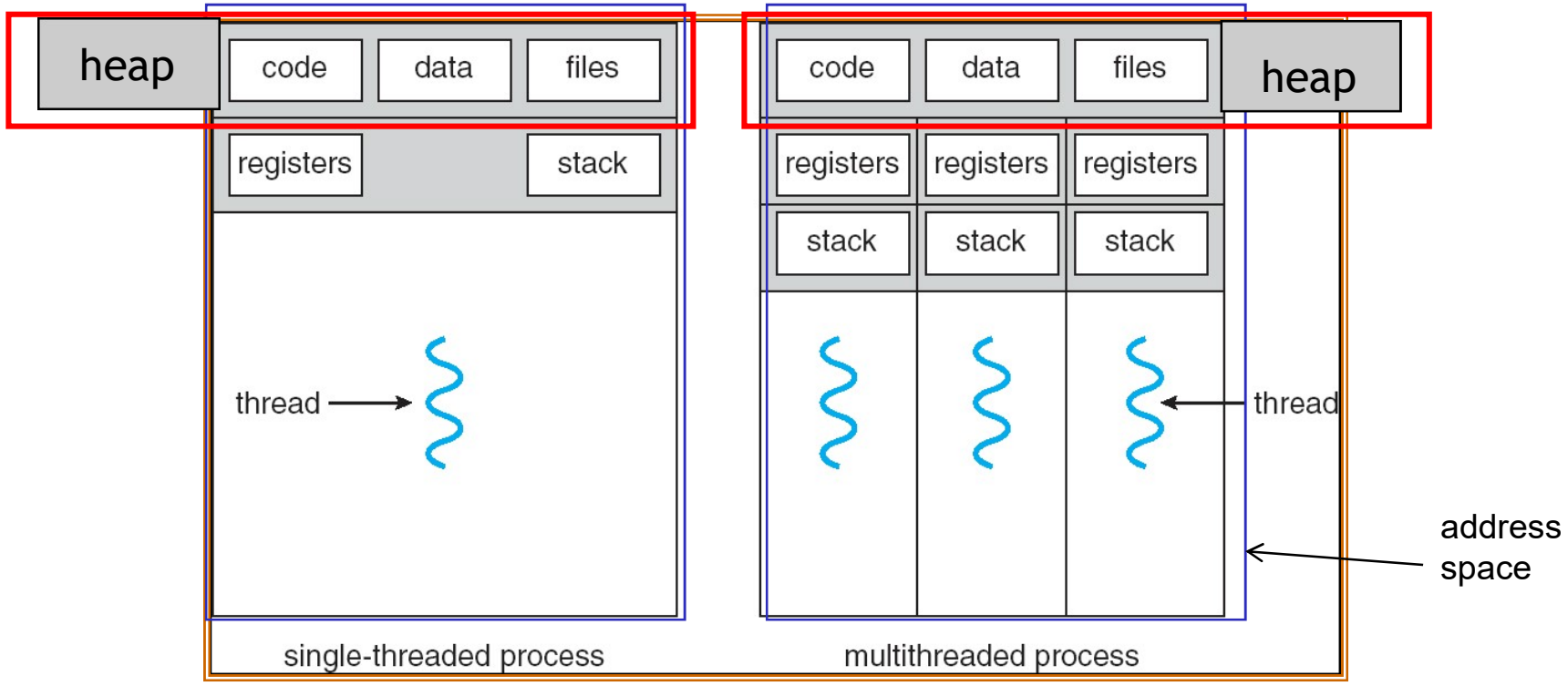
Sistemi operativi e in tempo reale – a.a. 2023/24

Meccanismi per l'esecuzione di thread multipli

Processi e thread

- Concetto moderno di *processo con thread multipli*:
  - *Processo*: astrazione del sistema operativo che rappresenta ciò che è necessario per eseguire *un singolo programma, nel caso più generale multithread*
- Il processo è costituito da due parti:
  - Thread multipli:
    - Flussi sequenziali di esecuzione, schedulabili individualmente
  - Risorse protette:
    - Stato della memoria principale
    - Stato dell'I/O

Processi con thread multipli



- Thread -> concorrenza, componente attiva
  - più thread nello stesso spazio di indirizzamento
  - registri e stack specifici del thread
- Spazi di indirizzamento -> protezione, componente passive
  - Code, data, files, heap condivisi tra i thread del processo

Meccanismi per il multithreading

- Sono parte dei meccanismi del SO per realizzare la multiprogrammazione
- Qui ci focalizziamo su come supportare e realizzare l'esecuzione di più thread

- Ignoriamo per ora gli aspetti relativi alla protezione

Programmare attività concorrenti con thread

- Cosa accade in presenza di un programma fortemente *compute-bound?*
- Esempio - Programma C eseguito da un solo thread; la prima procedura esegue un calcolo lunghissimo e al termine stampa un risultato:

```
main() {
 ComputeBound("results.txt"); //compute-bound, no I/O
 PrintSomething("list.txt"); //altro
}
```

- Quale comportamento si osserva?
- Probabilmente:
  - Il programma non stampa alcuna lista
  - `ComputeBound()` non termina …

Programmare attività concorrenti con thread

- Programma con due thread:

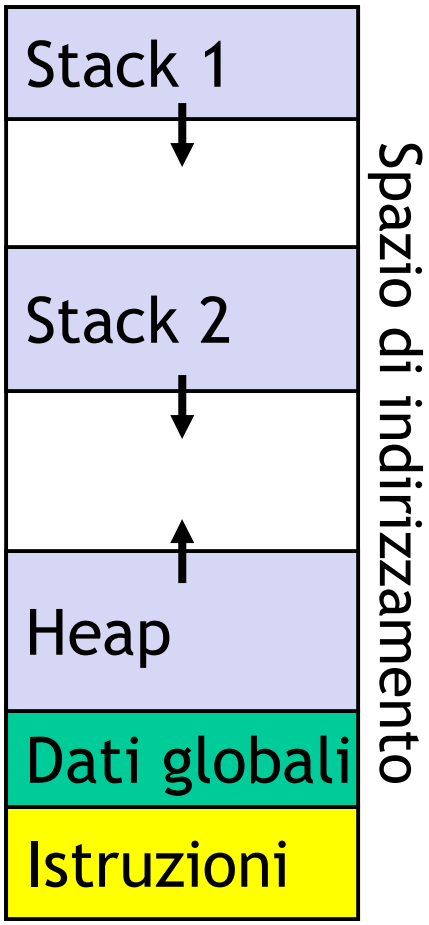
```
main() {
    CreateThread(ComputeBound("results.txt"));
    CreateThread(PrintSomething("list.txt"));
}
```

- `CreateThread()` è una primitiva della libreria di multithreading che *attiva un thread indipendente* per eseguire la procedura assegnata
- Due thread *nello stesso processo*

- Se il SO prevede opportuni meccanismi di *virtualizzazione della CPU*, ora il thread `PrintSomething()` viene eseguito e la lista effettivamente stampata

Immagine di memoria con due thread

- Se esaminiamo la memoria con un debugger:
  - Due set di registri della CPU
  - Due *stack*
- Problemi:
  - Posizionamento relativo delle stack nello spazio di indirizzamento
  - Dimensione massima da attribuire alle stack
  - Possibilità di violazione dei limiti da parte dei thread
  - Rilevazione delle violazioni

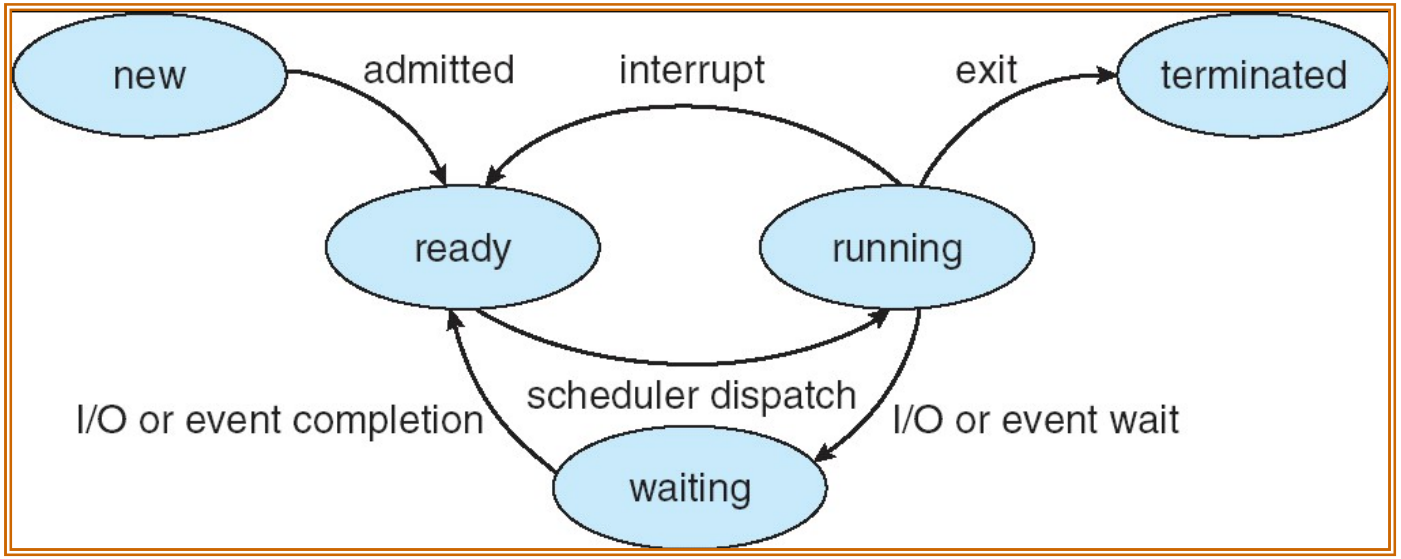


Stato di un thread

- Ogni thread ha un *Thread Control Block* (TCB):
  - Stato di esecuzione: registri CPU, program counter, stack pointer
  - Informazioni di *scheduling*: stato, priorità, tempo di CPU
  - Informazioni di *accounting*
  - Puntatori (per realizzare le code di scheduling)
  - Puntatore al processo a cui appartiene (PCB)
  - Eventuali altre informazioni (specifiche per ogni SO/libreria di MT)

- Il SO tiene traccia dei TCB (se kernel-level) in memoria protetta
  - Soluzione: Vettore, Lista, …

Il ciclo di vita di un thread (o di un processo)

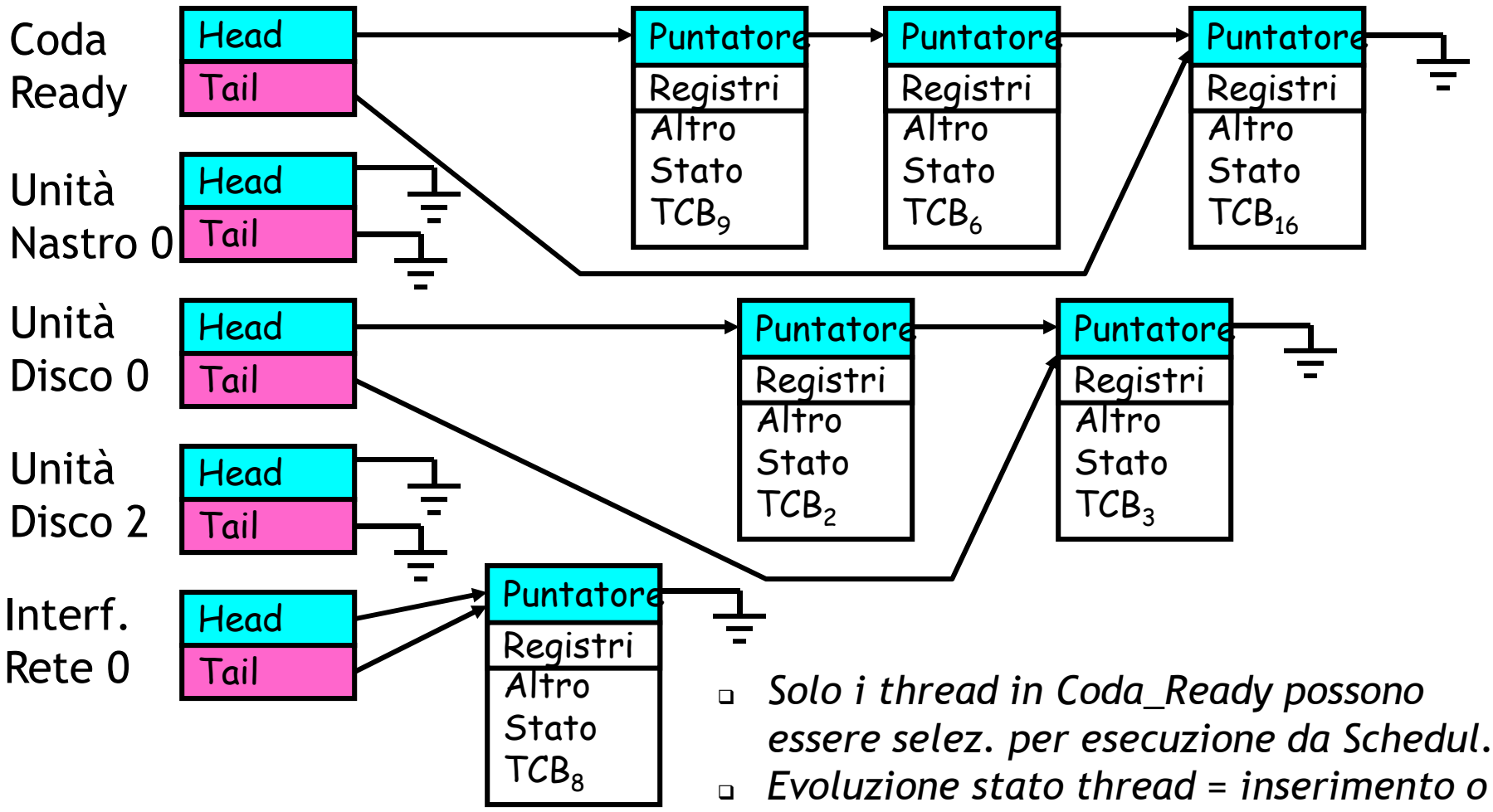


- Un thread esegue modificando il proprio stato:
  - new: in fase di creazione
  - ready: in attesa, pronto per l'esecuzione
  - running: il processore esegue le sue istruzioni
  - waiting: in attesa di qualche evento (I/O, signal, altro)
  - terminated: ha completato l'esecuzione
- I thread "attivi" sono rappresentati dai loro TCB, organizzati in code in base allo stato

Coda ready e code dei dispositivi di I/O

- Quando un thread *non è in esecuzione* il suo TCB è in una coda di attesa
  - Attesa di esecuzione (ready) o attesa di evento (waiting)
  - Esiste una coda separata per ogni dispositivo, segnale, condizione per cui un thread può attendere
  - Ogni coda può avere una diversa politica di scheduling

Coda ready e code dei dispositivi di I/O



- *Solo i thread in Coda_Ready possono essere selez. per esecuzione da Schedul.*
- *Evoluzione stato thread = inserimento o rimozione descrittore in code!*

Il ciclo di attivazione (dispatching)

- Per *porre in esecuzione i thread*, il SO esegue (in linea di principio) un ciclo del tipo:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- E' un ciclo *infinito* che incapsula tutte le funzioni del SO, salvo situazioni particolari e di emergenza
- Il SO deve svolgere tre azioni fondamentali:
  - mettere in esecuzione il codice utente con `RunThread()`
  - riprendere il controllo dal thread in esecuzione
  - scegliere il prossimo thread da eseguire

Esecuzione di un thread

- La prima operazione è `RunThread()`

- Per eseguire un thread il SO:
  - Carica il suo stato sulla CPU (registri, PC, stack pointer)
  - Carica l'ambiente (spazio di memoria virtuale, etc.), se lo switch è anche una commutazione di processo
  - Salta all'indirizzo presente nel PC

- Per riottenere il controllo il SO utilizza:
  - Eventi interni: il thread restituisce *volontariamente* il controllo o *richiede un servizio* al SO
  - Eventi esterni: il thread subisce una *preemption*

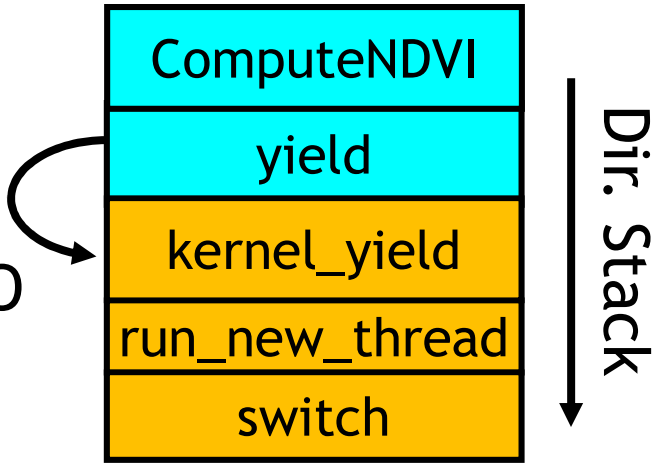
Eventi interni

- Tre tipi di eventi:
- Blocco per richiesta di I/O o attesa di evento di sistema
  - Una richiesta di I/O implicitamente cede la CPU
- Attesa di un “segnale” proveniente da un altro thread
  - Il thread chiede di attendere e pertanto cede la CPU
  - Le richieste di attesa di segnali, I/O, o eventi da parte del thread in esecuzione sono yield implicite
- Il thread esegue uno `yield()`
  - Il thread rinuncia volontariamente alla CPU, ad es. tra i vari passaggi di una sequenza di calcoli:

```
computeNDVI() { //large swath area 290 km^2
   while(TRUE) {
     ComputeNDVI_at_100m(); // mosaic cell
     yield();
   }
}
```

Stack per un thread che effettua yield

```
computeNDVI() {
    while(TRUE) {
      ComputeNDVIat100m();
      yield();
    }
}
```



- Per eseguire un nuovo thread:

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* altre attività */
}
```

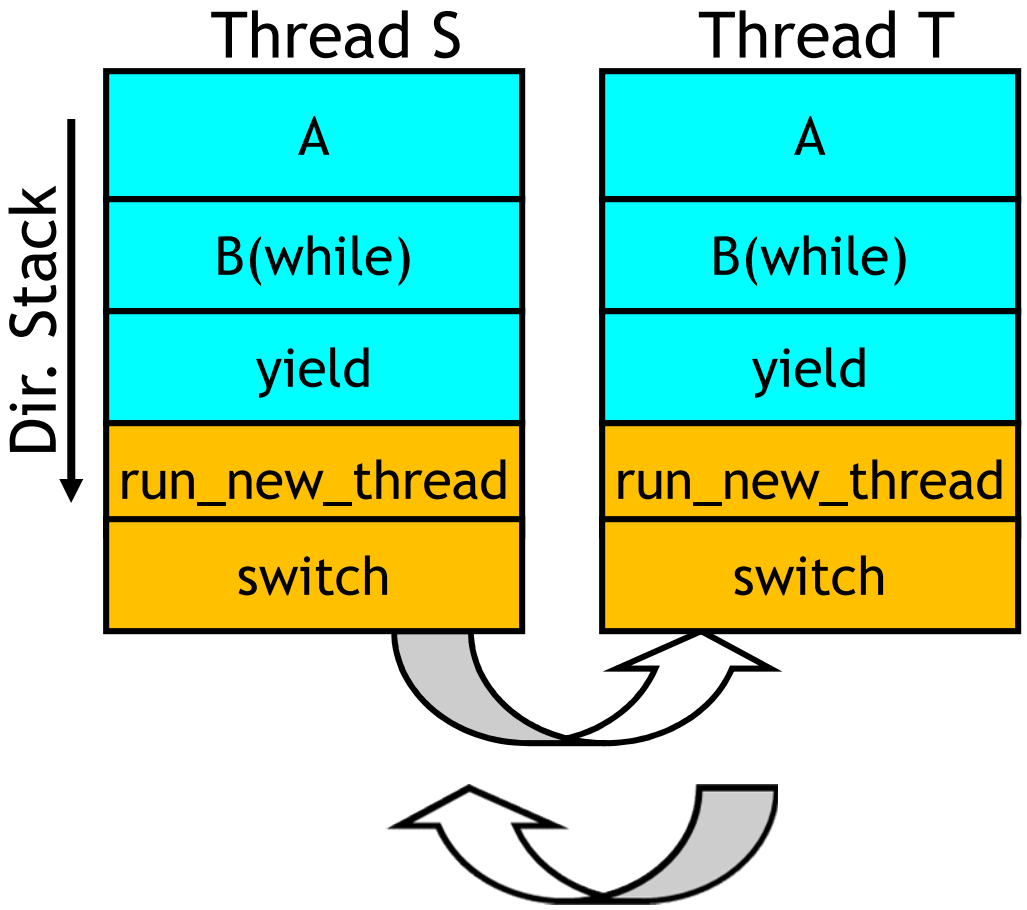
- Per commutare ad un nuovo thread il SO:
  - Salva ciò che il thread può invalidare: PC, registri, stack
  - Mantiene l'isolamento di ciascun thread

Stack per due thread che effettuano yield

- Due thread S e T eseguono il seguente codice (identico):

```
proc A() {
    B();

}
proc B() {
    while(TRUE) {
        yield();
    }
}
```



Salvataggio e ripristino dello stato: Thread switch

```
Switch(tCur,tNew) {
    /* Salva stato thread in esecuzione tCur*/
    TCB[tCur].regs.r7 = CPU.r7;
            …
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*ind ritorno*/

    /* Carica stato nuovo thread tNew ed esegui */
    CPU.r7 = TCB[tNew].regs.r7;
            …
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Ritorna a CPU.retpc -> a tNew !*/
}
```

Realizzazione del thread switch

- I processori moderni sono caratterizzati da decine o centinaia di registri (da circa 80 a 400 e oltre), da 32 bit fino a 128 bit
- La funzione di switch deve essere altamente *efficiente* ed è normalmente codificata in *assembly*
- `retpc` è l'indirizzo a cui salta il return: poichè `switch` ha caricato su SP il puntatore per `tNew`, il ritorno è effettuato al *nuovo thread*
- L'ottimizzazione in `switch` è fondamentale, ma la *correttezza* deve essere garantita in tutte le situazioni
- Ad es., se un registro non viene salvato si potranno avere malfunzionamenti intermittenti, in base all'uso del registro

Prestazioni di thread e context switch

- Almeno ogni 10-100 ms si verifica un thread- o context-switch!

- Alcuni numeri di overhead per Linux (i5, i7):
  - context switch time in Linux: 3-4 μs
  - thread switch time: 100 ns, circa il doppio senza affinity

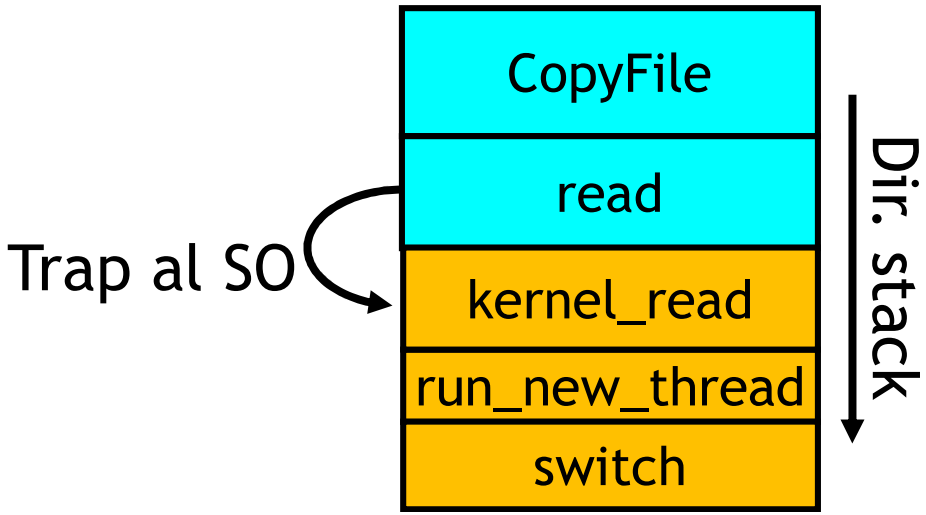
- L'overhead di *context switch* può aumentare moltissimo in base all'occupazione della cache da parte del processo (dimensione working set)

Cause di commutazione dei thread

- Occasioni di commutazioni *volontarie*
  - Yield #
  - Richiesta di I/O }
  - Attesa di evento o segnale } yield impliciti !

- Il thread che chiede un dato o attende un evento sa che presumibilmente cederà il controllo

Commutazione per operazione di I/O



- Quando un thread richiede un blocco di dati dal file system (o altra operazione di I/O):
  - Il codice utente invoca una system call
  - Il kernel *inizia* l'operazione di read
  - Il kernel esegue switch per porre in esecuzione un nuovo thread
- Il SO marca lo stato del thread che ha invocato read come *waiting* e inserisce il suo TCB nella coda della specifica operazione di I/O

Commutazione per comunicazione tra thread

- Esempi di comunicazione tra thread: attesa di evento, signal, join
- La commutazione è gestita dal SO in modo simile al caso determinato da una richiesta di I/O

Cause di commutazione dei thread

- Commutazioni *volontarie*
  - Yield
  - Richiesta di I/O
  - Attesa di evento o segnale

- Commutazioni *involontarie*
  - Tempo
  - Eventi che rendono pronti altri thread a maggiore priorità
  - Eccezioni sincrone generate dal thread in esecuzione

- Classificazioni: volontarie vs. involontarie, sincrone vs. asincrone, esterne vs. interne ?

Eventi esterni

- Un thread che non esegue mai I/O, non attende segnali e non effettua yield potrebbe mantenere il processore e tutte le risorse indefinitamente (programma compute-bound -- es. ComputePI ?)
- Per garantire che il ciclo di dispatching del SO ottenga nuovamente il controllo si devono utilizzare eventi *esterni*
  - Interrupt: segnali provenienti dall'hardware o generati in software che interrompono il thread in esecuzione e determinano un salto al kernel
  - Timer: segnale periodico generato da un orologio hw ogni N millisecondi
- Gli eventi esterni si devono verificare con una frequenza adeguata per l'attivazione del dispatcher

Meccanismo delle interruzioni

- Le CPU prevedono: una o più linee di *interruzione*, una o più linee di interruzione non mascherabile (NMI), un insieme di linee che forniscono l'identità dell'interrupt (interrupt vettorizzato)
- Il controllore programmabile delle interruzioni (PIC) fornisce un Registro delle interruzioni (IRR), un Registro di maschera (IMR), un circuito di priorizzazione delle interruzioni, un FF di abilitazione generale (AG) delle interruzioni (IE/ID), un timer real-time o a frequenza del clock
- La CPU programma il PIC (in fase di boot) e interagisce, quando necessario, con le unità esterne che generano le interruzioni (es., disco, stampante, rete)

Processo delle interruzioni

- Si verifica un'interruzione esterna mentre è in esecuzione un thread
- In hw:
  - *Salvataggio PC*
  - *Disabilitazione interruzioni*
  - *Passaggio a modo Supervisore*
- In sw:
  - *Gestore delle interruzioni (handler)*
- In hw:
  - *Ripristino PC*
  - *Ritorno a modo Utente*

Gestore delle interruzioni

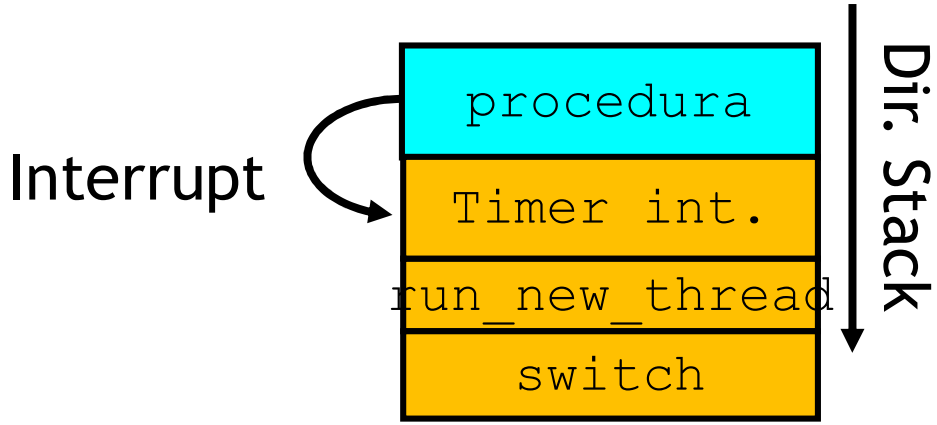
- Innalza la priorità
- Riabilita il FF di abilitazione generale
- Salva i registri
- Passa il controllo all'handler vero e proprio (servizio, ISR)

  < servizio dell'interrupt>
- Ripristina i registri
- Abbassa la richiesta dell'interrupt servito
- Disabilita il FF di abilitazione generale
- Ripristina il livello di priorità
- Esegue istruzione RTI

Interruzione

- Un'interruzione è un context switch iniziato dall'hw
- Non c'è una istruzione specifica del thread in esecuzione in cui viene deciso di cedere il controllo al SO o di eseguire un diverso thread
- L'interrupt handler è sempre eseguito immediatamente, in competizione eventualmente con altri interrupt a priorità diversa
- La presenza di segnali di interrupt pendenti e abilitati è verificata in un passo del *ciclo di fetch-execute*, quindi al livello più interno possibile

Commutazione determinata da timer



- Il timer assicura istanti di decisione di scheduling
- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- Risolve il problema di "controllare" l'esecuzione dei thread *(preemption)* → *preemptive multithreading*

Commutazione determinata da eventi esterni

- La gestione degli interrupt di I/O (eventi esterni) è analoga a quella del timer
  - Anziché `DoPeriodicHouseKeeping()` il SO esegue `ServiceIO()`

- ➔ A seguito degli eventi esterni il SO effettua uno switch analogo a quello determinato da yield
- In questo caso la cessione è involontaria e il thread in esecuzione subisce una *preemption*
- NB: la presenza di punti obbligati di preemption rende in pratica superfluo eseguire yield da parte del programmatore

Scelta del thread da porre in esecuzione

- Il SO deve *scegliere un thread* da affidare al Dispatcher per l'esecuzione
  - Nessun thread pronto – il dispatcher esegue un ciclo di attesa o un "idle thread" per attività di servizio (ad es. per modalità a basso consumo)
  - Un solo thread pronto – quello!
  - Più thread pronti - scelta in base a criteri di priorità
- Criteri di scelta (scheduling):
  - LIFO (last in, first out): lista ordinata in base al tempo in cui il thread è divenuto pronto -> il più recente
  - FIFO (first in, first out): ancora in base al tempo -> quello da più tempo in attesa -- più "fair" (?)
  - Scelta casual (si può fare di meglio?)
  - Coda con priorità: lista ordinata in base al campo "priorità" del TCB

Caveat

- Qualunque sia l'algoritmo di scheduling adottato dal SO, il programmatore non ha alcun controllo né garanzia sulla sequenza di esecuzione dei thread realizzata a basso livello
- (Qualche limitata eccezione più avanti, in alcune tipologie di sistemi RT)
- D'altro canto, è impossibile verificare («testare») un programma per tutte le possibili sequenze di microesecuzione dei thread
- Nonostante questo, il programmatore aspira a scrivere e ad eseguire programmi (multithread) che funzionano correttamente!