Sistemi Operativi

Memory Allocators

La memoria è vista dal kernel come un grande array lineare. Esistono due modi per allocare memoria e ognuno dei due deve far fronte a Frammentazione e Tempo.

Tipi di Allocazione

- **Slab Allocator - Memoria Fissa**

  Viene utilizzata de ci sono tanti elementi di dimensione fissa da allocare e la memoria viene affettata in tante fette quanti sono gli elementi da allocare.

  Per soddisfare richieste in *O(1)* potrei costruire uno SLAB formato da una lista di interi, ma a questa struttura è preferibile, poiché non posso utilizzare *malloc()*, l'*Array List*. In questo caso, se conosciamo la dimensione massima di una lista, mappo tale lista in array di *max-size* elementi. Memorizzo la lista nell'array e:

  - Nel blocco di posizione *i* metto l'indice dell'elemento successivo in lista
  - Negli altri blocchi metto *-1*
- **Buddy Allocator - Memoria Variabile**

  Ha un costo maggiore rispetto allo SLAB, ma posso utilizzarlo come una *malloc*. La memoria viene vista come un albero binario con vista in ampiezza e viene ricorsivamente divisa in due a seguito di una richiesta. Un "buddy" di un blocco di memoria è una regione ottenuta partizionando la regione genitore.

  Per gestire al meglio gli elementi dell'albero, utilizzo una bitmap (array contenente soli *0* e *1*) per far si che sappia facilmente se il nodo è libero oppure no.

Interrupts & Syscalls

Ogni interazione con il SIstema Operativo viene innescata da Interrupt che può sorgere a causa di eccezioni interne, chiamate esplicite (syscall) o da eventi esterni. Per monitorare una interruzione potrei fare:

- Polling: Interrogo continuamente lo stato tramite un *while(1)*
- Dormo la maggior parte del tempo e poi vengo svegliato solo al verificarsi di un evento.

Interrupt Vector e Tabella Syscall

- L'Interrupt vector è un vettore di puntatori a funzione che rappresentano *ISR* che a loro volta gestiranno i vari interrupt.
- La tabella delle Syscall invece contiene in ogni locazione il puntatore a funzione che gestisce quella determinata syscall. Alla tabella viene associato anche un vetto che contiene il numero di paramentri e l'ordine dei parametri che la determinata syscall prende in input.

Processi

La creazione di un processo necessita delle Syscall *fork*. Mediante questa vengono create due istanze dello stesso processo (il figlio eredita il PCB del padre). Un processo muore solo con la chiamata esplicita *wait*, ovvero solo quando qualcuno legge il suo valore di ritorno.

- *fork()*: La memoria del padre viene completamente copiata nel figlio. Se l'istruzione successiva è una *exec()* potremmo risparmiarci tale copia e utilizzare la syscall
- *vfork()*: Non copia la memoria del padre nella creazione del figlio. Se l'istruzione successiva NON è una *exec()* potrei riscontrare un comportamento anomalo.

PCB - Process Control Block

Contiene:

- PID
- UID
- Stato del programma (ready, running,...)
- Stato della CPU (registri)
- Informazioni di scheduling
- Informazioni sulla memoria
- Informazioni di I/O (Descrittori di file aperti)

Context Switch

È un particolare stato del SO durante il quale avviene il cambiamento del processo correntemente in esecuzione sulla CPU, permettendo a più processi di utilizzare la stessa CPU eseguendo più programmi contemporaneamente. Durante un cambio di contesto, il SO;

- Salva lo stato del processo P1 correntemente in esecuzione e carica il PCB2
- Salva lo stato del processo P2 e carica PCB1

CPU Scheduling

L'esecuzione di un processo può essere suddivisa in due:

- Esecuzione di Istruzioni - CPU BURSTS
- Attesa di Eventi - I/O BURSTS

Lo scheduler è un modulo che ha lo scopo di decidere quale processo dovrà andare in esecuzione, vedendo la coda di ready. Viene invocato solamente quando vi è una richiesta di I/O da parte di un processo. Esistono due tipi di Scheduler:

1. Preemptive: Permette di interrompere il processo in esecuzione a favore di un altro processo. Lo seleziona e lo lascia in esecuzione per un certo periodo di tempo; se allo scadere del tempo esso è ancora in esecuzione, verrà prelazionato lasciando spazio ad un altro processo
2. Non Preemptive: Non da la possibilità di interrompere l'esecuzione del processo, lasciandolo in esecuzione finché non si blocca e viene ricevuto un I/O

Esistono varie politiche di Scheduling:

- FCFS (First Come First Served): I processi vengono schedulati nell'ordine in cui giungono al sistema.
- SJF (Shortest Job First): Seleziona il processo che utilizzerà per minor tempo la CPU. Impossibile da implementare poiché dovrei conoscere a priori il tempo di arrivo dei processi ed ha un problema di *starvation* in quanto un processo potrebbe rimanere in attesa per troppo tempo prima che venga schedulato.
    - Esiste una variante dello SJF che però è *preemptive*: in questo caso se arriva un processo con CPU Burst < della CPU Burst di quello correntemente in esecuzione, quest'ultimo verrà interrotto lasciando il controllo della CPU a quello con lavoro minore.
- Priority Scheduler: Ad ogni processo è associato un numeretto che indica la priorità con cui esso dovrà essere schedulato. Più piccolo è il numeretto e maggiore sarà la sua priorità.
- RR (Round Robin): Ad ogni processo viene assegnato un quanto di tempo, durante il quale al processo è assegnato l'uso della CPU. Per scandire i quanti, alla fine di ognuno di essi viene generato un timer interrupt. Lo scheduler manterrà una coda di processi ready e selezionerà il primo processo in coda; quando scade il quanto, il processo viene messo in fondo alla coda.
- Real Time Scheduling: Devono essere rispettate delle scadenze, altresì chiamate deadline. Suddivisi in Soft/Hard Real Time.
- Multilivello: Questo tipo di scheduler prevede la creazione di *classi* di processi, che verranno categorizzati in base alle loro caratteristiche simili. Per ognuna di queste classi, verrà assegnata una priorità e potrei utilizzare una diversa politica di scheduling a seconda della classe che sto analizzando.
    - Cerco prima la classe con priorità massima con almeno un processo in ready
    - Continuo a schedulare seguendo la politica della classe di riferimento

    - La coda con priorità più bassa avrà il quanto più grande
- Multiple Processor Scheduling: Se ho a disposizione più CPU, lo scheduler dovrà seguire una di queste politiche
    - Homogeneous Processors: Tutti i core eseguono lo stesso set di istruzioni
    - Asymmetric Multiprocessing: Ho solo una CPU che esegue codice kernel e le interruzioni sono gestite da un solo core
    - Symmetric Multiprocessing: Tutti i processi sono in una coda di ready comune e questi sonno self-scheduling
    - Processor Affinity: L'esecuzione di un programma viene forzata su un particolare core della macchina

Valutazione Dello Scheduler Migliore - Queuing Models

Simulo i processi in base ad una distribuzione probabilistica esponenziale e faccio simulazioni attraverso la **Legge di Little**:

$$n = \lambda \cdot W$$

n = Lunghezza della coda; λ = Tempo medio di arrivo dei processi; W = Tempo medio di attesa dei processi in coda.

Main Memory

La CPU può accedere direttamente solo a memoria e a registri e la cache è posizionata tra i due con lo scopo di garantire un accesso più veloce quando si fa riferimento ad una istruzione o a un dato.

Allocazione Statica e Dinamica della Memoria - Address Binding

Il Dynamic Linker si occuperà di rimpiazzare gli indirizzi simbolici (le wildcard del file '.o') con indirizzi fisici (indirizzi del file sorgente).

- Early Binding = Avviene prima dell'esecuzione
- Late Binding = Avviene durante l'esecuzione

Il kernel memorizza le informazioni relative alle aree di memoria allocate al processo *P* in una tabella e le rende disponibili alla *MMU* (circuito situato tra CPU e memoria), la quale si occuperà dell'effettiva traduzione di indirizzi da logici a fisici.

Metodi Per fare MMU

- Reallocation and Limit: Basta prendere l'indirizzo logico, sommarlo al mio indirizzo fisico e se va oltre il comparatore (<) avviene una trap.

  Indirizzo Fisico = Reallocation + Logico

- Contigous Allocation: Ciascun processo è contenuto in una singola sezione contigua della memoria, la quale può anche essere ricompattata copiandola nella parte alta.
- Multiple Partition Allocation: Divisione della memoria in tanti pezzi e ne do un pezzo a ciascun processo

Allocazione Memoria di un Processo

Un processo ha bisogno di memoria contigua e dovrà fare i conti con problemi quali frammentazione, se non vi è abbastanza spazio contiguo, e protezione, facendo in modo che la memoria tra due processi non si sovrapponga.

Protezione della Memoria: Gestita da due registri della CPU...

- Base: Indirizzo di partenza della memoria allocata
- Limit: Size della memoria allocata

... con lo scopo di assicurarsi che ogni processo abbia uno spazio di memoria separato che non interferisca con quello di un altro processo.

Per garantire ciò:

1. Verifica che l'indirizzo sia >= di base
2. Verifica che l'indirizzo sia < di (base + limit)
3. Se tutto è rispettato avviene il caricamento in memoria

Compattazione della Memoria

Viene utilizzata per limitare o evitare la frammentazione esterna, spostando tutte le aree di memoria libere verso una estremità della memoria a tempo di compilazione.

Allocazione Non Contigua di Memoria - Approcci

- **Segmentazione**: Un processo viene suddiviso in un insieme di segmenti di differenti dimensioni sparsi in memoria; tale memoria conterrà una Tabella Dei Segmenti contente:
    1. Base: Indirizzo fisico in memoria
    2. Limit: Lunghezza del segmento
    3. Bit di Validità: 0 == !Valit; 1 == Valid
    4. Permessi *rwx*
- **Paginazione**: L'OS suddivide il processo in parti di dimensione fissa da lui stabilite (sono potenze di 2) chiamate pagine e la memoria è partizionata in aree, chiamate frames, di grandezza uguale a quella di una pagina. La CPU genera indirizzi suddivisi in:
    - Numero di Pagina: Contiene indirizzi di base del programma

- Offset di Pagina: Spiazzamento pagina

Base + Offset vengono inviati all'*MMU*, la quale si occuperà di generare l'indirizzo fisico. Viene utilizzato un TLB = Array contenente indirizzo a cui accedo più spesso; rappresenta un dizionario con chiave numero di pagina e valore il valore della pagina.

Differenze tra Segmentazione e Paginazione

Per la corretta implementazione, entrambe gli approcci hanno bisogno di strutture hardware per farli funzionare efficientemente.

- Ogni segmento ha dimensione variabile e perciò è identificato da indirizzo di base e indirizzo di limite. Tali informazioni sono contenute nella tabella dei segmenti e il sistema può decidere di non rendere accessibili alcuni segmenti tramite l'aggiunta di un bit di controllo. La frammentazione però porta a segmentazione esterna e ha la necessita di ricompattare i segmenti.
- Architetture moderne usano la paginazione per virtualizzare l'*address space* suddividendo la memoria in frames (pagine), di dimensione uguale. La parte alta dell'indirizzo verrà usata come "*chiave*" per la tabella delle pagine, evidenziando la pagina corrente, e la parte restante dell'indirizzo verrà utilizzato come offset all'interno della pagina. Può essere implementata anche in maniera gerarchica e anch'essa permette di proteggere zone di memoria attraverso l'utilizzo di bit di validità.

TLB e MMU

Rappresenta un Array presente che risiede nell'MMU contenente gli elementi più utilizzati di recente dalla tabella delle pagine. Memorizza alcune informazioni principali, come numero di pagina, numero di frame, bit di modifica e il campo per la protezione. Per quanto riguarda il funzionamento, l'MMU cerca l'indirizzo arrivatogli da tradurre nel TLB, se presente lo usa, altrimenti legge dalla tabella delle pagine e lo inserisce nel TLB.

Tempo di Accesso Effettivo a Memoria = p_hit(T_TLB + T_RAM)+ p_fault(2T_TLB+2T_RAM)

L'MMU gestisce anche il Page Protection tramite l'utilizzo di bit che indicano se l'area di memoria è accessibile in lettura o sia lettura che scrittura. In più contiene bit di validità atti a determinare se la pagina associata è nello spazio di indirizzamento logico del processo oppure no (Valid/Invalid).

Condivisione di Memoria tra Processi

Posso fare in modo che due processi condividano memoria, mappando lo stesso frame su due tabelle diverse (Shared Pages).

- Shared Code: Copia in lettura
- Private Code and Data: Ogni processo mantiene una copia separata di codice e dati.

Gerarchia Tabella delle Pagine

Le pagine vengono suddivise in strati, in modo tale che possa implementare una paginazione su due livelli. Utilizzo la tecnica dello **Swapping**, che consiste nel caricare interamente in memoria ogni processo, eseguirlo per un certo periodo di tempo al termine del quale verrà spostato sul disco in un'area chiamata Area di Swap. Le operazioni che verranno eseguite saranno dunque due:

- Swap-In: Da disco a memoria
- Swap-Out: Da memoria a disco

Virtual Memory

La virtual memory è una parte della gerarchia della memoria suddivisa in memoria + disco, infatti alcune componenti di un processo durante l'esecuzione si trovano in memoria, mentre altre risiederanno sul disco e saranno caricate solo quando necessario. In tal modo il kernel implementa l'illusione di una memoria più grande, combinando hardware con softare (MMU con Gestore della Memoria Virtuale). Virual memory può essere implementato tramite:

- **Paginazione Su Richiesta** (Demand Paging): Ogni porzione di indirizzamento prende il nome di pagina e tutte le pagine sono di egual dimensione (in potenza di 2). Viene in tal caso portata una pagina in memoria solo al momento del primo riferimento a una locazione appartenente alla pagina stessa.

  Ottimizzazione Demand Paging: **Copy on Write** (COW): Durante un *fork()* viene creato un processo figlio come copia della memoria del padre, ma se subito dopo tale istruzione il figlio esegue una *exec()* tale copia risulterebbe inutile poiché la sua memoria sarà rimpiazzata dai parametri di quest'ultima syscall. A tal proposito viene utilizzata la tecnica COW. Inizialmente entrambe i processi condivideranno le stesse pagine di memoria, marcate come *copy on write pages* e quando uno dei due processi vorrà modificare una delle pagine, quest'ultima verrà copiata.
- **Segmentazione su Richiesta**: Implementato da sistemi operativi in cui l'hardware disponibile non è sufficiente per implementare la paginazione. La tabella dei segmenti conterrà un bit di validità che serve a verificare se il segmento si trova nella memoria fisica oppure no.

Politiche di Sostituzione delle Pagine

Il gestore della memoria dovrà verificare quale pagina dovrà essere sostituita quando si verifica un *page fault* e non ci sono frames liberi in memoria e quanti frame allocare per ciascun processo tramite 3 algoritmi di sostituzione:

- Politica Ottimale: Sostituzione di tutte le pagine che non si useranno più nel periodo di tempo più lungo. Impossibile da implementare in quanto non possiamo conoscere a priori il comportamento futuro di un processo.
- Politica FIFO: Il sistema operativo mantiene una lista di tutte le pagine in memoria, dove la pagina di testa è la più vecchia e quella di coda è quella arrivata più di recente. La rimozione avviene dunque in testa.
- Politica LRU: Sostituzione della pagina utilizzata meno di recente con la pagina richiesta.

Working Set

L'insieme di lavoro, cioè l'insieme di tutte le pagine di un processo, viene detto working set. Ciò consiste nell'assicurarsi che il work di un processo sia caricato totalmente in memoria prima di consentire l'esecuzione di un altro processo.

Quando parecchi processi iniziano a spendere più tempo per la paginazione che per l'esecuzione, essi inizieranno ad andare in **trashing**. In tal caso il SO aumenterà il grado di multiprogrammazione avviando nuovi processi che cominceranno a loro volta ad andare in trashing a causa della mancanza di frames liberi.

File System

Risiede su uno storage secondario, sul disco, e fornisce sia un'interfaccia utente per la memorizzazione, mappatura logica a fisica che un'efficiente accesso al disco abilitando la conservazione dei dati.

Mount

Tramite l'operazione di mount, il FS viene informato che un nuovo FS è pronto per essere utilizzato. L'operazione, quindi, provvederà ad associarlo con un dato *mount-point*, ovvero la posizione all'interno della gerarchia del file system del sistema dove il nuovo file system verrà caricato. Prima di effettuare questa operazione di attach, ovviamente bisognerà controllare la tipologia e l'integrità del file system. Una volta fatto ciò, il nuovo file system sarà a disposizione del sistema.

Directory

Rappresenta nella realtà un insieme di voci di file, che può contenere:

- Hard Link: Puntatori a file che se eliminati tutti, eliminano il file stesso
- Soft Link: Puntatori "soft", non intaccano il contenuto del file
- Altre Directory

File Descriptor

È un intero ottenuto come valore di ritorno a seguito della chiamata alla syscall *open()*. A seguito di tale chiamata, il sistema scandisce il FS in cerca del file e, una volta trovato, il **FCB** è copiato nella tabella globale dei file aperti. Per ogni singolo file aperto, anche se da più processi esiste una sola entry nella tabella globale dei file aperti. Viene, quindi, creata una entry all'interno della tabella dei file aperti detenuta dal processo, la quale punterà alla relativa entry nella tabella globale.

Devices e Implementazione Funzioni

Il filesystem */dev* fornisce al sistema un aggancio per dispositivi fisici esterni. In Unix ogni device è accessibile come un file e questi vengono categorizzati in:

- A Blocchi: Come ad esempio file di testo
- A caratteri: Come ad esempio le telecamere che rappresentano stream

La funzione ioctl permette di interagire con il driver generico e tramite essa è possibile settare e ricavare i parametri di tale device, come ad esempio la risoluzione della webcam. Per configurare devices seriali a caratteri è possibile utilizzare le *API* racchiuse nella interfaccia termios, che ci permette di avere accesso alle caratteristiche di tale device.

Dischi

Un disco è un dispositivo primitivo che può effettuare solo due operazioni basilari:

- **Lettura di un Blocco**
- **Scrittura di un Blocco**

Per dialogare direttamente con il disco, l'OS utilizza il driver del File System che implementa le primitive *open(), read(), write(), ...* per accedere ai blocchi del disco (visto come grande file binario).

> - *Per ogni file aperto, nel Kernel, viene creata un'istanza della struttura e viene memorizzato il descrittore tramite un'apposita struttura "OpenFileInfo". Per ogni descrittore aperto da un processo, viene memorizzata in una lista accessibile dal PCB un'apposita struttura "OpenFileRef". Ogni OpenFileRef punterà alla corrispondente OpenFileInfo e memorizza un indipendente puntatore a file per la gestione di seek/read/write*
> - *Sul Disco invece un file è grande almeno quanto un blocco, è caratterizzato da "FileControlBlock" che risiede all'inizio del blocco per gestire informazioni*

> *specifiche del file e ogni directory avrà una struct di intestazione che estenderà l'FCB.*

Allocazione di un File

Una delle problematiche a cui un FS deve sopperire è come allocare lo spazio necessario ad ogni file che andrà a "storare" sul disco.

- Nel caso di **Linked Allocation** ogni file sarà una lista concatenata di blocchi, dove ogni blocco conterrà il puntatore al successivo. Tale tipo di allocazione permetterà dunque di allocare i blocchi in maniera sparsa sul disco ed il file finirà quando incontreremo un puntatore a *NULL*. In questo caso localizzare un blocco richiederà numerosi cicli di I/O e i puntatori influiranno negativamente sullo spazio disponibile per contenere i dati.
- Anche per la **Indexed Allocation** i blocchi saranno disposti in maniera "scattered" sul disco (sparsi) ma in questo caso ogni file conterrà un *Index Block* contenente i puntatori a tutti gli altri blocchi componengti del file.
    - Quando un file viene creato, tutti i puntatori dell'index block saranno settati a *NULL*
    - Quando un blocco viene richiesto e scritto, il puntatore a tale blocco entrerà nell'index block

  Tale tipo di allocazione permette di guadagnare velocità rispetto ad una implementazione tramite linked list nel caso in cui si effettuino molti accessi scattered ai blocchi, poiché non occorrerà scandire tutta la lista ma fare una semplice ricerca. Il costo da pagare però è lo spazio necessario a contenere l'index block stesso.
- I primi file system usavano il modello di **Allocazione Contigua** allocando una singola area di memoria a ogni file al momento della creazione. Ogni file occuperà un insieme di blocchi contigui ma anche se per allocare basterà sapere solo il blocco iniziale e la sua lunghezza, genera frammentazione esterna e interna.

Gestione dello Spazio Libero

- Bitmap: Mantengo una bitmap all'inizio del disco in cui ogni bit corrisponderà ad un blocco. Utilizzerò tale bitmap per cercare un blocco libero più vicino
- Linked List: Identico allo *Slab Allocator*. Punterà a tutti i blocchi liberi sul disco.

IPC

Per comunicare tra loro, i processi dovranno scambiarsi informazioni attraverso *comunicazione interprocesso* (IPC).

Message Passing - Memoria non Condivisa

È un meccanismo di IPC che prevede l'utilizzo di messaggi, gestiti tramite opportune syscall attraverso una coda di messaggi. Il processo P_1 consegna l'are di memoria al kernel, il quale la consegnerà a P_2. La comunicazione sarà basata sulle operazioni di send e receive che saranno sviluppate in base alle specifiche dell'utente (comunicazione sincrona/asincrona, diretta/indiretta, limitata/illimitata, ...). Il sistema operativo implementa una *mailbox* garantendo la creazione della coda di messaggi da parte dei processi, se non esiste la coda, oppure il link ad essa a partire da un identificatore della coda, se già esistente. Le operazioni fondamentali sono:

- Check dello status della coda
- Post di un messaggio
- Attesa di un messaggio (bloccante o non bloccante)

Shared Memory - Memoria Condivisa

In tal caso viene riservata una porzione di memoria condivisa tra i vari processi, i quali potranno scambiarsi informazioni semplicemente leggendo o scrivendo da/in tale porzione di memoria. I processi sceglieranno la locazione di memoria e la tipologia di dati e dovranno sincronizzarsi in modo da non operare contemporaneamente sugli stessi dati garantendo mutua esclusione. Tale implementazione è molto utile nel caso di producer/consumer o analogamente client/server. Sarà dunque necessario istanziare un buffer condiviso da entrambi i processi in modo che il produttore possa rendere disponibile ai consumatori ciò che ha prodotto.

Stack e Context Switch

Un contesto di un processo, cioè ciò che mi serve per eseguirlo o per far continuare la sua esecuzione, richiede:

- Registri della CPU
- Memoria del processo

Coroutines

Rappresentano un pezzo di programma che effettua salti da una funzione all'altra, cambiando contesto. In *C* è implementato dalla libreria *ucontext*.

- **getContext**: Inizializza la struttura puntata da *ucp* al contesto corrente.

```c
int getcontext(ucontext_t *ucp);
```

- **setContext**: Salva il contesto corrente in *ucp*, contesto che precedentemente era stato salvato (raramente utilizzato)

```c
int setcontext(const ucontext_t *ucp);
```

- **makeContext**: Crea il trampolino, lanciando una funzione come prima istruzione modificando il contenuto puntato da *ucp*.

```c
void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
```

- **swapContext**: Scrive il contesto di esecuzione in cui voglio andare a leggere dalla CPU dove voglio saltare

```c
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

Esempio:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <unistd.h>

static ucontext_t ctx[3];

static void f1 (void){
    printf("Start f1\n");
    sleep(1);
    swapcontext(&ctx[1], &ctx[2]);  // da qui salto alla funzione 2
    sleep(1);
    printf("Fine f1\n");
}

static void f2 (void){
    printf("Start f2\n");
    sleep(1);
    swapcontext(&ctx[2], &ctx[1]);
    sleep(1);
    printf("Fine f2\n");
}
int main(int argc, char *argv[]){
    char st1[8192];
    char st2[8192];
```

```
    getcontext(&ctx[1]);    // setto il contesto di ctx[1]

    ctx[1].uc_stack.ss_sp = st1;    // gli do la stack...
    ctx[1].uc_stack.ss_size = sizeof(st1);  //... e la sua size
    ctx[1].uc_link = &ctx[0]; // link a dove deve saltare

    makecontext(&ctx[1], f1, 0);    // gli dico cosa deve lanciare (f1)

    /* FACCIO LA STESSA COSA CON IL SECONDO: Setto il ctx[2], ... */

    getcontext(&ctx[2]); // ... a qui (*2)
    ctx[2].uc_stack.ss_sp = st2;
    ctx[2].uc_stack.ss_size = sizeof(st2);
    ctx[2].uc_link = &ctx[1];

    makecontext(&ctx[2], f2, 0);

    puts("Inizio il salto, dopo aver già settato tutta la merda!");
    swapcontext(&ctx[0], &ctx[2]);  // Salto da qui... (*1) +
    return 0;
}
```

Il programma stamperà dunque:

```
Inizio il salto, dopo aver già settato tutta la merda!
Start f2
Start f1
Fine f2
Fine f1
```