



# ANALISI COMPARATIVA OS161 - MentOS

Pedone Matteo - s310171  
Pulvirenti Roberto - s317704  
Venezia Giuseppe - s309625



# MentOS

**MentOS** (**Mentoring Operating System**) è un sistema operativo open-source scritto in C utilizzato per scopi didattici. L'obiettivo di MentOS è di fornire un **ambiente** progettuale che sia abbastanza **realistico** da mostrare come funziona un vero sistema operativo ma **semplice** a sufficienza.

MentOS può essere scaricato dalla repository Github gratuita: [github.com/MentOS](https://github.com/MentOS) ed emulato tramite qemu.



# MentOS

Questo sistema operativo è strutturato ed impostato per sistemi **single core**, perciò non si occupa della gestione o dell'implementazione di funzioni adatte a sistemi multicore.

Sebbene sia nel complesso ben strutturato, essendo un sistema didattico, alcune **funzionalità** sono **incomplete** o non implementate.



# SYSTEM CALLS

MentOS



# System calls

Gli ingredienti sono:

- 1 funzione lato kernel
  - 1 funzione lato user
  - 1 numero unico  
assegnato alla system call
- *funzione lato kernel*
    - `int sys_open(const char *pathname, int flags, mode_t mode);`
  - *funzione lato user*
    - `int open(const char *pathname, int flags, mode_t mode);`
  - *unico numero associato alla system call*
    - `#define __NR_open 5`



# Struttura della cartella

## **inc/sys/unistd.h**

- Il file che definisce le system call user-side
- Nell'esempio precedente, il file contiene la funzione open()

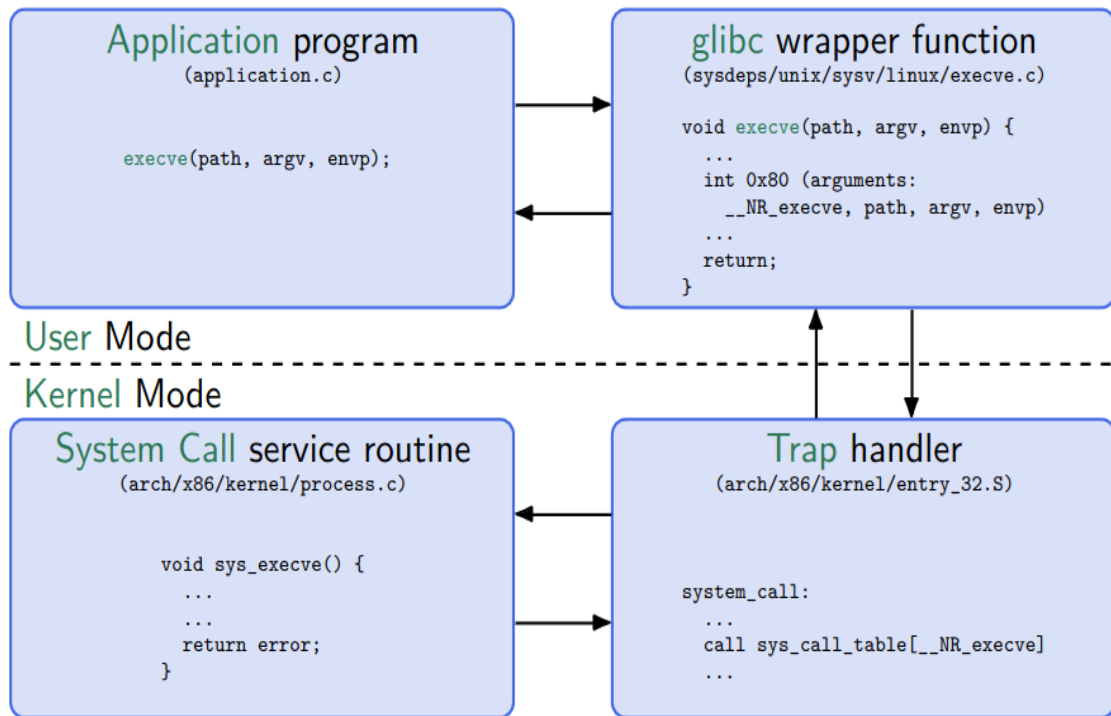
## **src/libc/unistd/\*.c**

- I files implementano le system call user-side
- In pratica, preparano gli argomenti per la system call di sistema e chiamano l'interrupt 80
- La open è implementata all'interno src/libc/unistd/open.c

## **inc/system/syscall\_types.h**

- contiene la lista dei numeri di system call
- nell'esempio: #define \_\_NR\_open 5

# System calls, esempio #1





# System calls, descrizione

1. L'applicazione effettua una chiamata di sistema chiamando una funzione di supporto nella libreria C.
2. La funzione di supporto copia gli argomenti della chiamata di sistema dallo stack a registri specifici della CPU e copia il numero della chiamata di sistema nel registro della CPU %eax. Successivamente, fa passare la CPU da *modalità utente* a *modalità kernel* (attraverso interruzione software int 0x80). Ogni chiamata di sistema è identificata da un nome nella libreria C e da un numero univoco nel kernel.
3. Il kernel esegue la routine `system_call()` che salva i valori dei registri nello stack del kernel, verifica la validità del numero di chiamata di sistema e invoca la routine di servizio della chiamata di sistema.

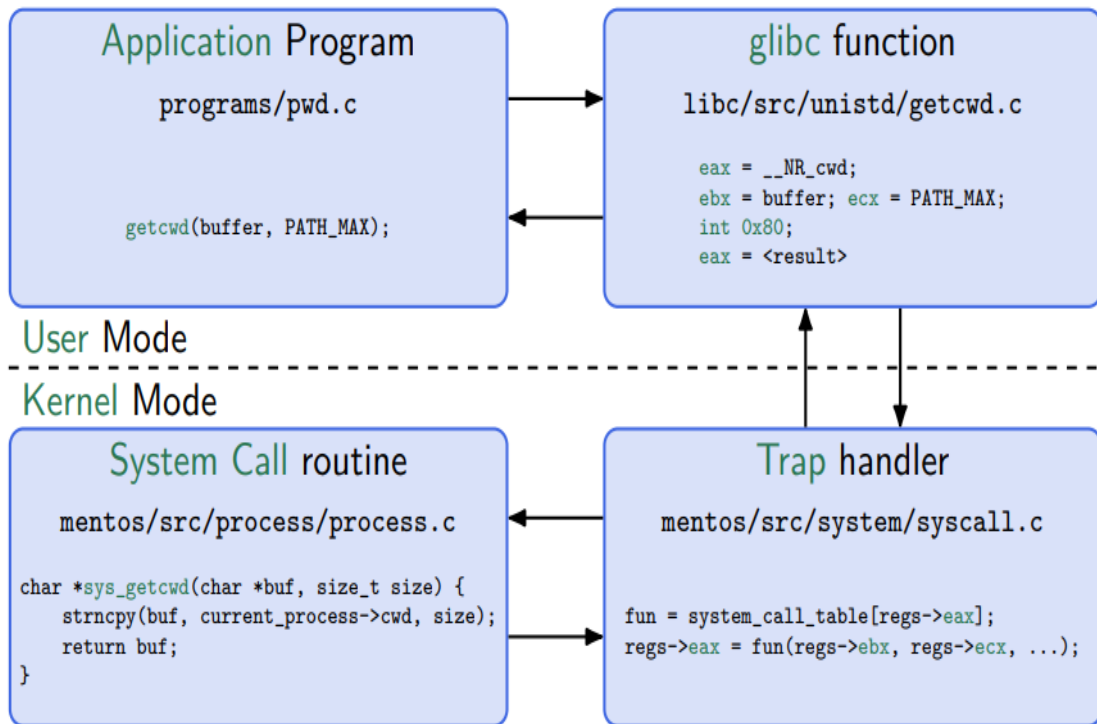




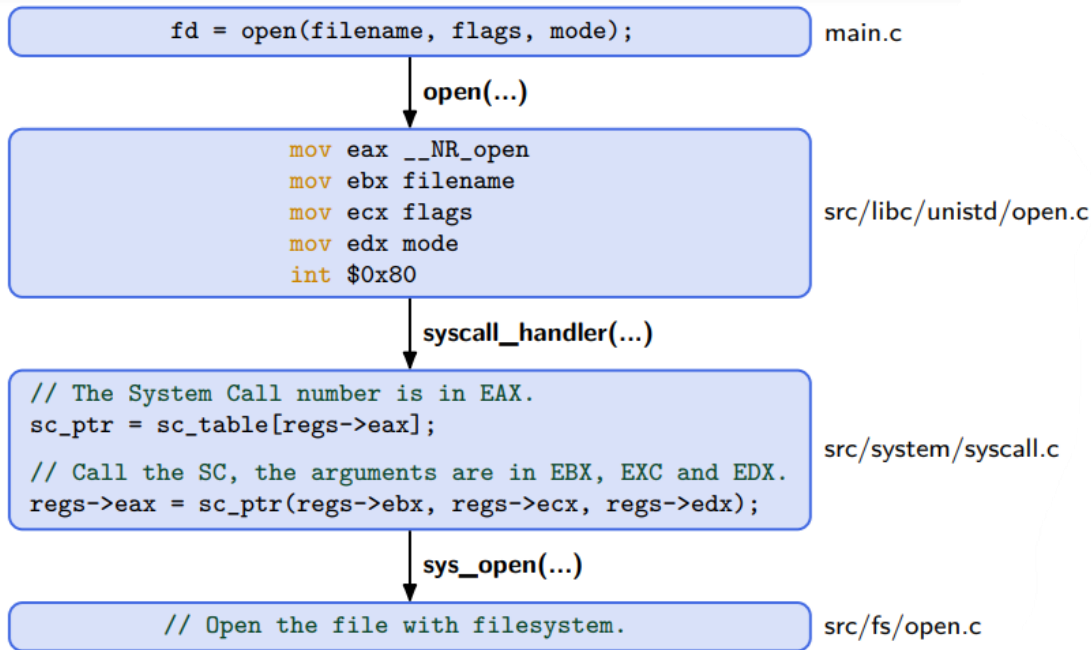
# System calls, descrizione

4. La routine di servizio viene eseguita e viene restituito uno stato a `system_call()`.
5. La routine `system_call()` ripristina i valori dei registri della CPU dallo stack del kernel e inserisce lo stato della routine di servizio eseguita nello stack. Allo stesso tempo, passa la CPU dalla *modalità kernel* alla *modalità utente* e ritorna alla funzione di supporto in C.
6. Se il valore di ritorno della routine di servizio della chiamata di sistema indica un errore, la funzione di supporto imposta la variabile globale `errno` utilizzando questo valore. Infine, la funzione di supporto restituisce al chiamante un valore intero che indica il successo o il fallimento della chiamata di sistema. Per convenzione, il numero negativo -1 (o un puntatore NULL) indica un errore al programma applicativo chiamante.

# System calls, esempio #2



# System calls





# ANALOGIE E DIFFERENZE

MentOS - OS161



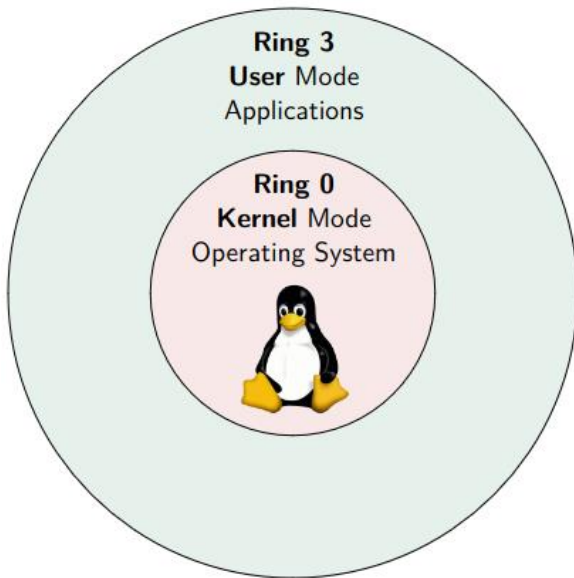
# System calls a confronto

OS161 gestisce le system calls in modo molto simile a MentOS:

- si scatena una trap.
- avviene il context switching (utilizzando il trap frame nel kernel stack).
- si esegue la system call.
- si ripristina lo user stack e si torna all'esecuzione in user mode.

Invece di scatenare una trap e salvare il contesto nel trap frame, in MentOS si passa dall'utente al kernel tramite l'interrupt 80 e non esiste un vero e proprio trap frame, ma ci sono dei registri speciali nello stack del kernel che vengono usati appositamente per il context switching e costituiscono i parametri della funzione.

# Livelli di privilegio



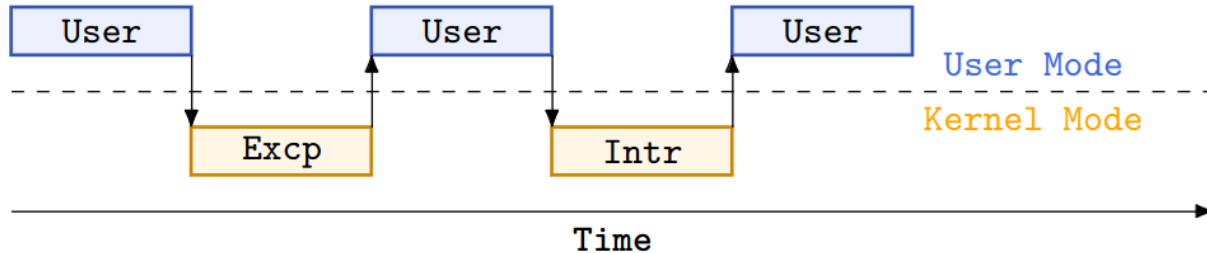
Un'altra differenza rispetto a OS161 è che in MentOS ci sono quattro livelli di privilegio diversi e non solo due.

In ogni istante la CPU sta eseguendo in un livello di privilegio specifico, che determina quali codici possono e non possono essere eseguiti.

Molti kernel moderni usano solo due livelli di privilegio, **Ring 0** (il più privilegiato) e **Ring 3** (il meno privilegiato).

# Context switch overview

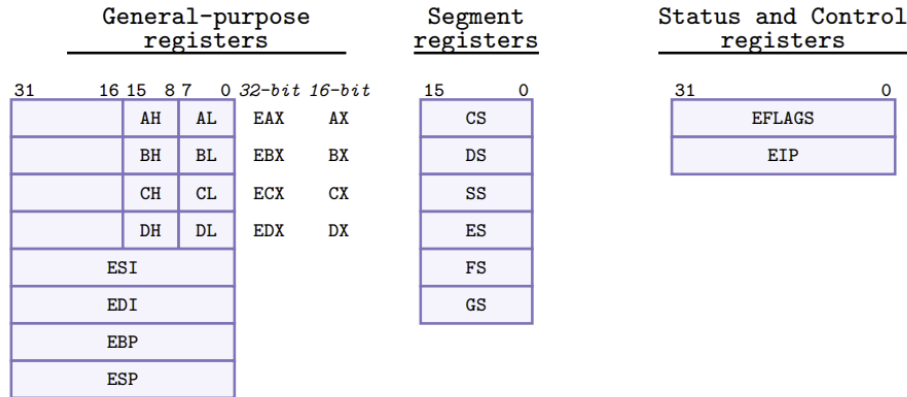
Ogni volta che la CPU cambia il livello di privilegio, si verifica uno switch di contesto. Esempi di eventi che fanno cambiare la modalità di esecuzione della CPU includono un clic del mouse, la digitazione di un carattere sulla tastiera, una chiamata di sistema



# Registri

In MentOS abbiamo a disposizione 16 registri di diversa dimensione divisi in:

- 8 General-purpose Data Registers (32 bit);
- 6 Segment Registers (16 bit);
- 2 Status and Control Registers (32 bit);



EFLAGS tiene traccia dei flags di controllo, di stato o di sistema mentre EIP rappresenta il program counter



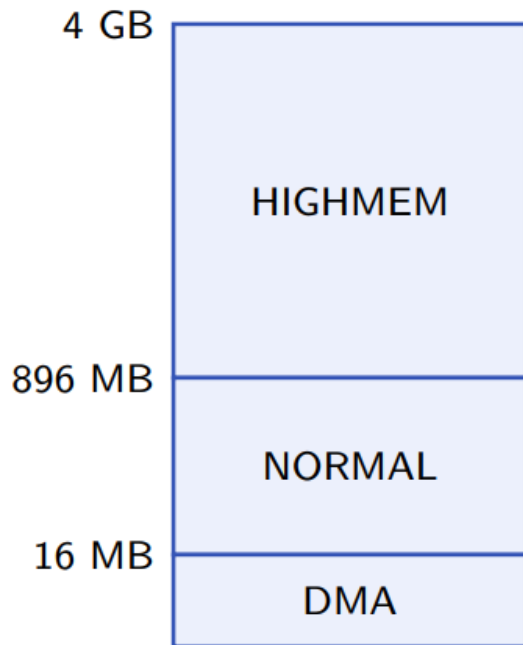


# GESTIONE MEMORIA FISICA

MentOS

# Partizionamento memoria fisica

La memoria fisica è suddivisa in 3 zone o intervalli:



- **ZONE\_HIGHMEM (>896 MB):** contiene i frame non mappati permanentemente in memoria kernel. È la memoria utilizzabile per i processi utente. Accessibile tramite Page Tables mapping;
- **ZONE\_NORMAL (16-896 MB):** contiene la memoria assegnata al kernel in maniera permanente. Viene inizializzata all'avvio con l'immagine del kernel (e spazio extra);
- **DMA (<16 MB):** contiene gli indirizzi di memoria utilizzabili per il *Direct Memory Access* (Nota: nella versione più recente di MentOS, questa zona è inclusa in NORMAL)



# Zone descriptor

La struct volta alla gestione delle informazioni di ogni zona è la struct *zone*:

```
struct zone {  
    unsigned long free_pages;           // Number of free pages in the zone.  
    free_area_t free_area[MAX_ORDER]; // buddy blocks (see next)  
    page_t * zone_mem_map;             // pointer to first page descriptor  
    uint32_t zone_start_pfn;           // Index of the first page frame  
    unsigned long size;                // Total size of zone in pages  
    char * name;                       // Name of the zone  
}
```

# Page descriptor

A differenza di quanto avviene in OS161, il kernel in MentOS tiene traccia dello stato corrente di ogni page frame in quanto l'allocazione di memoria può, già nella versione base, non essere contigua.

Tutte le informazioni necessarie sono contenute in una struttura dati denominata `page_t` (`inc/mem/zone_allocator.h`):

```
struct page_t {  
    int _count;  
    unsigned int private;  
    struct list_head lru;  
    //... continue  
}
```

**Un counter per ogni processo a cui è assegnato tale page frame (se -1 risulta libero)**

**flag per indicare se il page frame è libero**

**puntatore alla last recently used list di pagine**



# Zone page frame allocator

Il *zone page frame allocator* è il sottosistema del Kernel volto alla gestione delle richieste di allocazione/deallocazione dei gruppi di page frames contigui.

Tale sotto-sistema fornisce due importanti funzioni:

- ***alloc\_pages(zone, order)***: utilizzata per richiedere  $2^{\text{order}}$  contiguous page frames da una data zona. Ritorna il primo *page\_t* di un blocco che soddisfi la richiesta o NULL se l'allocazione è fallita. È il corrispettivo della *getppages* di OS161, tant'è vero che chiama la sua *ram\_stealmem* all'interno denominata *bb\_alloc\_pages*.
- ***free\_pages(page, order)***. Funzione utilizzata per rilasciare un ammontare di  $2^{\text{order}}$  page frame contigui da una data zona.



# Buddy system

Il buddy system è una strategia standard dei sistemi Linux che permette di allocare gruppi contigui di frame in potenza di due.

Abbiamo a disposizione una lista di liste ognuna delle quali colleziona i blocchi liberi di frame contigui in ordine di dimensione ascendente (l[3] ad esempio contiene la lista di blocchi di  $2^3$  frame contigui).

NOTA: Al minimo può essere richiesto un blocco di dimensione  $2^0$  page frame, vale a dire un singolo frame di dimensione 4KB



# Buddy system - alloc\_pages

**Step 1:** Ricerca di un blocco grande abbastanza da soddisfare la richiesta;

**Step 2:** Rimozione del blocco dalla lista;

**Step 3:** Divisione del blocco finché non è appena grande abbastanza per la richiesta (questo avviene perché c'è la possibilità che il blocco trovato sia molto più grande di quanto necessario e per evitare sprechi di memoria);

**Step 4:** La prima pagina del blocco è tornata come risultato.

NOTA: Nel caso di blocchi di dimensione maggiore rispetto a quanto richiesto lo step 3 permette di evitare sprechi. Successivamente tali frame contigui in eccesso vengono inseriti nella lista alla posizione che gli spetta (in base al numero di frame contigui)



# Buddy system - free\_pages

**Step 1:** Verifica se il blocco adiacente a quello dato è libero;

**Step 2:** Se sì, unisci i due blocchi;

**Step 3:** Ripeti lo step 1;

**Step 4:** Aggiungi il blocco alla free area list.





# IMPLEMENTAZIONE BUDDY SYSTEM

MentOS

# bb\_alloc\_pages

```
block_found:
// Get a block of pages from the found free_area_t. Here we have to manage pages. Recall, free_area_t
// collects the first page_t of each free block of 2^order contiguous page frames.
page = list_entry(area->free_list.next, bb_page_t, location.siblings);
// Remove the descriptor of its first page frame.
list_head_remove(&page->location.siblings);
// Set the page as allocated, thus, remove the flag FREE_PAGE.
__bb_clear_flag(page, FREE_PAGE);
// Check that the page is a ROOT_PAGE
assert(!_bb_test_flag(page, FREE_PAGE) && __bb_test_flag(page, ROOT_PAGE));
// Decrease the number of free block of the free_area_t.
area->nr_free -= 1;
// Found a block of 2^k page frames, to satisfy a request for 2^h page frames (h < k) the program allocates
// the first 2^h page frames and iteratively reassigns the last 2^k - 2^h page frames to the free_area lists
// that have indexes between h and k.
unsigned int size = 1UL << current_order;
while (current_order > order) {
    // At each loop, we have to set free the right half of the found block.
    // Refer to the lower free_area_t and order.
    current_order--;
    area = __get_area_of_order(instance, current_order);
    // Split the block and set free the second half.
    size = size/2;
    // Get the address of the page in the middle of the found block.
    bb_page_t *buddy = __get_page_from_base(instance, page, size);
    // Check that the buddy is free and not a root page.
    assert(!_bb_test_flag(buddy, FREE_PAGE) && !__bb_test_flag(buddy, ROOT_PAGE));
    // Insert buddy as first element in the list of available blocks (free_list).
    list_head_insert_after(&buddy->location.siblings, &area->free_list);
    // Increase the number of free block of the free_area_t.
    area->nr_free += 1;
    // Save the order of the buddy.
    buddy->order = current_order;
    // Set the buddy as a root page.
    __bb_set_flag(buddy, ROOT_PAGE);
}
// Set the order of the page we are returning.
page->order = order;
// Clear the free-page status from the page.
__bb_clear_flag(page, FREE_PAGE);
return page;
}
```

- La funzione richiede come parametri un'istanza del buddy system e l'ordine del blocco di pagine che vogliamo ottenere. Restituisce la prima pagina del blocco assegnato.
- Nell'immagine possiamo vedere l'implementazione della ricerca di un blocco libero sufficientemente grande. La `get_area_of_order` si limita a restituire la `bb_free_area` in posizione `current_order` nel vettore `free_area` dell'istanza.



# bb\_free\_pages

**Require:** free\_area array f, Block b, Order o

```
1: while o < MAX_ORDER - 1 do
2:   buddy = getBuddy(b, o)
3:   if !free(buddy) | order(buddy) ≠ o then
4:     break;
5:   end if
6:   removeBlock(f[o], buddy)
7:   if buddy < b then
8:     b = buddy
9:   end if
10:  o = o + 1
11: end while
12: addBlock(f[o], b)
```

- Punto fondamentale di tale algoritmo è il merge dei page frame contigui in un unico blocco di memoria;
- Il buddy block non è altro che il blocco contenente page frame contigui al blocco che si vuole liberare.



# Best Fit

L'implementazione di base per allocare page frame ai processi è Best Fit, in quanto il buddy system cerca il blocco di memoria di dimensione più piccola possibile per poter soddisfare la richiesta. Di seguito il pezzo di codice che permette di comprendere tale implementazione. È importante ricordare che in MentOS si utilizza una lista di liste ordinata in maniera scendente per dimensione dei blocchi:

```
//IMPLEMENTATION BEST FIT
unsigned int current_order;
for (current_order = order; current_order < MAX_BUDDYSYSTEM_GFP_ORDER; ++current_order) {
    // Get the free_area_t at index 'current_order'.
    area = __get_area_of_order(instance, order);
    // Check if area is not empty, which means that there is at least a block inside the list.
    if (!list_head_empty(&area->free_list)) {
        goto block_found;
    }
}
```



# Worst Fit

Il Worst fit da noi implementato muta semplicemente le condizioni del for loop:

```
//IMPLEMENTATION WORST FIT
unsigned int current_order;
for (current_order = MAX_BUDDYSYSTEM0 GFP_ORDER; current_order >= 0; --current_order){
    area = __get_area_of_order(instance, current_order);
    if(!list_head_empty(&area->free_list)){
        goto block_found;
    }
}
// No suitable free block has been found.
return NULL;
```



# GESTIONE MEMORIA VIRTUALE

MentOS



# Perché la memoria virtuale?

Il kernel applica la memoria virtuale per mappare gli indirizzi virtuali su indirizzi fisici. Vantaggi:

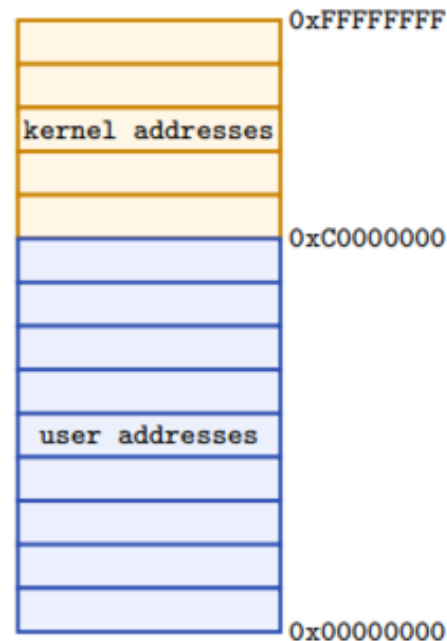
- La RAM può essere virtualmente suddivisa in kernel e spazio utente;
- ogni singolo frame di pagina può avere permessi di accesso diversi;
- ogni processo ha la sua mappatura della memoria;
- un processo può accedere solo a un sottoinsieme della memoria fisica disponibile;
- un processo può essere ricollocabile.

# Overview della memoria

Lo spazio di indirizzamento in MentOS è pari a 4GB e gli indirizzi hanno dimensione di 32 bit.

Tale spazio di indirizzamento risulta suddiviso in page frame ognuna delle dimensioni di 4 KB (per il Kernel il page frame rappresenta la più piccola unità di memoria indirizzabile).

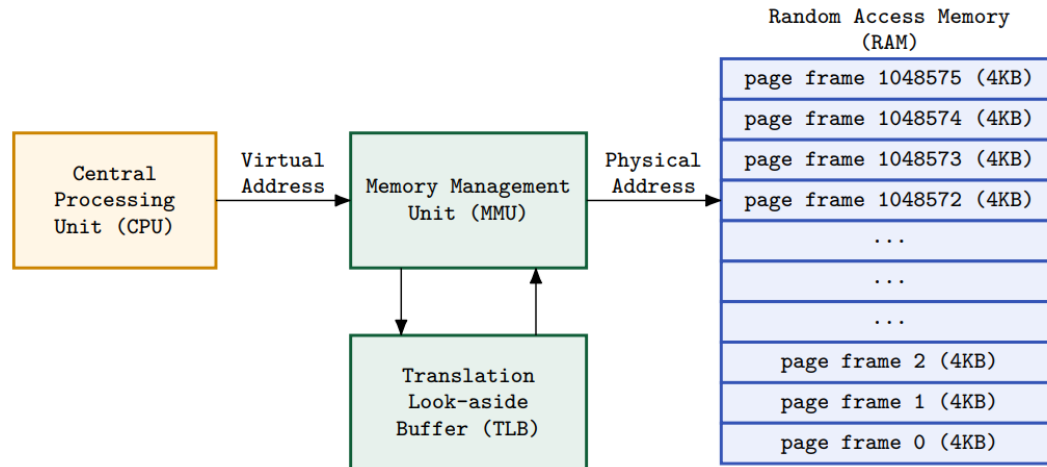
La RAM in MentOS è virtualmente suddivisa in Kernel space (1GB) e User space (3GB).



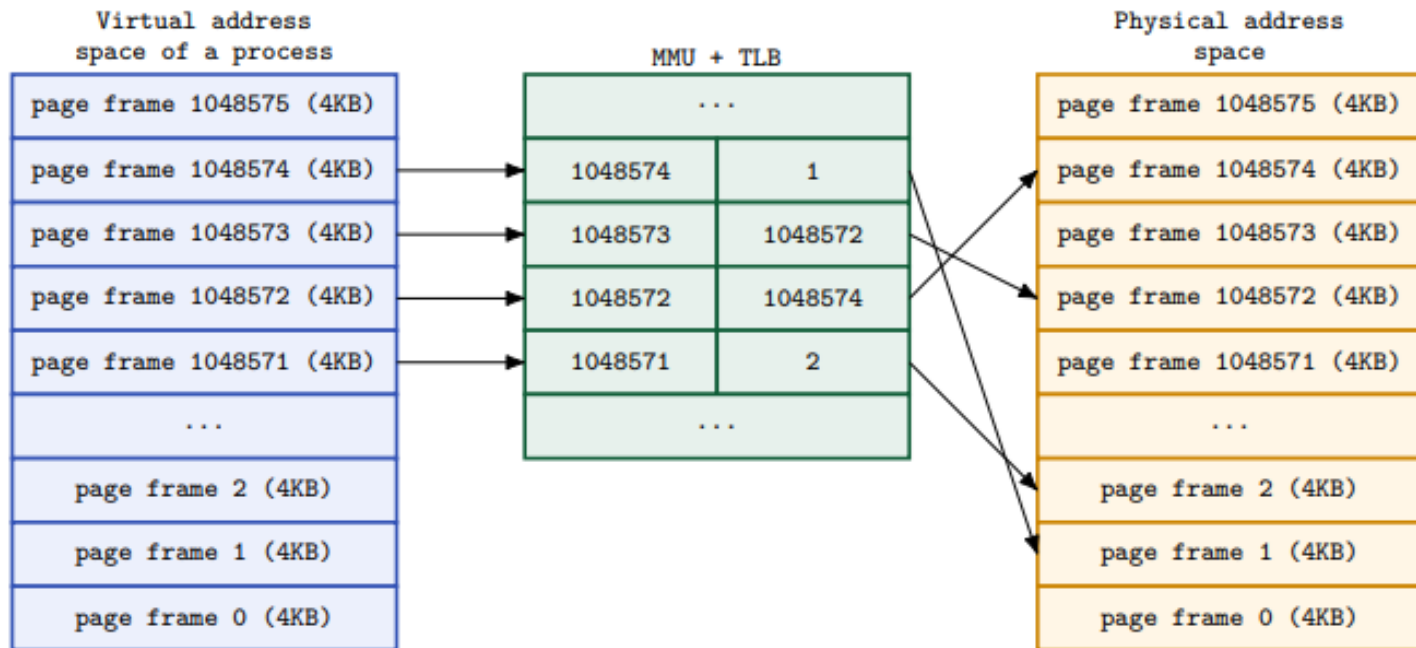


# MMU

Memory Management Unit (MMU) è il componente hardware che mappa gli indirizzi virtuali in indirizzi fisici. *Vantaggi:* la mappatura è eseguita nell'hardware, quindi non c'è nessuna penalizzazione delle prestazioni, stesse istruzioni della CPU utilizzate per accedere alla RAM e all'hardware mappato.



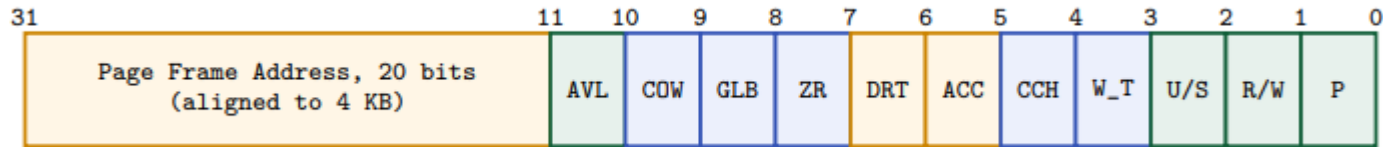
# MMU e TLB



# PTE

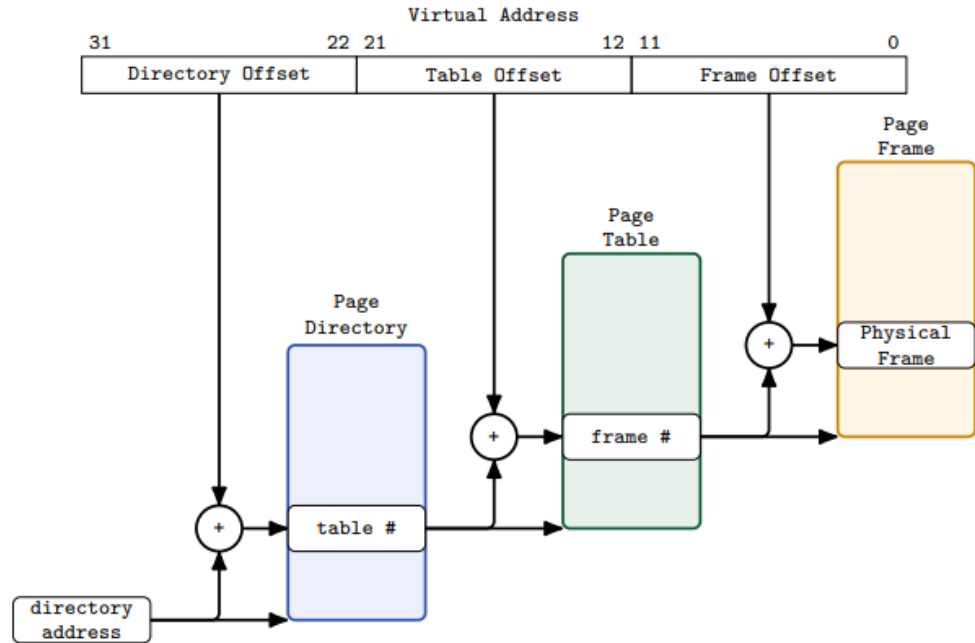
Per ogni pagina virtuale del processo, il Kernel mantiene una corrispondente Page Table Entry (PTE).

- P quando impostato, la pagina è nella memoria fisica.
- R/W quando impostato, la pagina è in modalità lettura/scrittura.
- U/S quando impostato, la pagina è accessibile a tutti.
- ACC quando impostato, è stato effettuato l'accesso alla pagina.
- DRT quando impostato, la pagina è sporca



# Traduzione indirizzo

Il Kernel organizza le sue PTEs attraverso una gerarchia a 2 livelli. Il primo livello è costituito da una **Page Directory** che è in grado di indirizzare 1024 page tables e il secondo livello rappresentato dalla **Page Table** che indirizza le 1024 entries.





# ANALOGIE E DIFFERENZE

MentOS - OS161



# Memoria virtuale a confronto

Anche in questo caso la gestione della memoria virtuale è molto simile nei 2 sistemi operativi le cui differenze sono:

- In MentOS la memoria riservata al kernel è inferiore a quella riservata allo spazio utente mentre in OS161 la memoria è divisa in due parti uguali.
- In MentOS la pagina (4KB in entrambi i OS) contiene 11 bit di controllo con 20 bits dedicati al page frame address, mentre in OS161 avevamo un numero inferiore di bits di controllo.
- In OS161 abbiamo la possibilità di gestire i livelli di virtualizzazione mentre in MentOS abbiamo una gerarchia a 2 livelli.



# ALGORITMI DI SCHEDULING

MentOS



# Algoritmi di scheduling

Lo **scheduler** viene chiamato dopo la gestione di un'interruzione/eccezione. Nel dettaglio, vengono eseguite le seguenti operazioni:

- aggiorna le variabili temporali del processo corrente;
- tenta di riattivare un processo in attesa. Se una condizione di attesa è soddisfatta, un processo viene riattivato impostando il suo stato su “in esecuzione” e inserendolo nella runqueue;
- esegue l'**algoritmo di scheduling** per scegliere il processo successivo da eseguire dalla CPU presente nella coda di esecuzione;
- esegue il cambio di contesto.





# Algoritmi di scheduling

MentOS supporta diversi tipi di algoritmi di pianificazione, che vengono selezionati durante la compilazione tramite **cmake** e chiamati dalla funzione **scheduler\_pick\_next\_task**.

Inoltre, supporta il **Real-Time scheduling**, cioè vengono eseguiti task o processi entro determinati vincoli di tempo. La coda di esecuzione potrebbe contenere sia *task periodici* che *task aperiodici*.

**scheduler\_pick\_next\_task** è una funzione centralizzata utilizzata dallo scheduler per eseguire il processo successivo e internamente questa funzione chiama l'algoritmo di pianificazione attualmente selezionato. In base all'algoritmo di pianificazione selezionato verrà scelto il processo successivo da eseguire.



# Algoritmi di scheduling

Gli algoritmi implementati per i task periodici sono:

- **Earliest Deadline First** (EDF)
- **Rate Monotonic** (RM)

Gli algoritmi per i task aperiodici invece:

- **Aperiodic EDF** (AEDF)
- **RR - Round Robin** (chiamato con la funzione `__scheduler_rr`)
- **Priority - Highest Priority First** (chiamato con la funzione `__scheduler_priority`)
- **CFS - Completely Fair Scheduler** (chiamato con la funzione `__scheduler_cfs`).

Le funzioni si possono trovare in `mentos/src/process/scheduler_algorithm.c`

# Round Robin

```
static inline task_struct *__scheduler_rr(runqueue_t *runqueue, bool_t skip_periodic)
{
    // If there is just one task, return it; no need to do anything.
    if (list_head_size(&runqueue->curr->run_list) <= 1) {
        return runqueue->curr;
    }
    // Search for the next task (we do not start from the head, so INSIDE, skip the head).
    list_for_each_decl(it, &runqueue->curr->run_list)
    {
        // Check if we reached the head of list_head, and skip it.
        if (it == &runqueue->queue)
            continue;
        // Get the current entry.
        task_struct *entry = list_entry(it, task_struct, run_list);
        // We consider only runnable processes
        if (entry->state != TASK_RUNNING)
            continue;
        // If entry is a periodic task, and we were asked to skip periodic tasks, skip it.
        if (__is_periodic_task(entry) && skip_periodic)
            continue;
        // We have our next entry.
        return entry;
    }
    return NULL;
}
```

Il codice dell'algoritmo Round Robin è già completo in MentOS come riportato a sinistra.

Il **Round Robin** è un algoritmo di scheduling dove ad ogni processo viene assegnato uno slice di tempo fisso in modo ciclico.

I suoi **vantaggi** sono che è l'algoritmo più facile da implementare, preventivo e soprattutto starvation-free, ovvero si evita che ci siano processi che non vengono mai eseguiti.

Il maggior **svantaggio** è che non è possibile dare priorità a delle task rispetto ad altre.



# Round Robin

Alla funzione si passeranno come parametri la runqueue e un flag di periodicità.

- Il primo check è se la lista contiene solo un elemento. In questo caso si ritorna la task corrente altrimenti si itera sulla lista di processi;
- Se l'iteratore ha raggiunto la testa della lista, passa alla successiva iterazione.
- Tramite la macro *list\_entry* si ottiene il puntatore alla struttura *task\_struct* a cui appartiene l'elemento corrente nella lista
- si verifica se il processo è eseguibile
- se il processo corrente è periodico e la variabile *skip\_periodic* è true, viene saltato e si passa al prossimo
- Se tutte le condizioni sono soddisfatte, il processo corrente è selezionato come prossimo processo da eseguire altrimenti si ritorna NULL



# ANALOGIE E DIFFERENZE

MentOS - OS161



# Scheduling a confronto


In **MentOS** è presente il codice di vari algoritmi di scheduling ed è possibile selezionarne uno in particolare tramite il comando:

```
cmake -DSCHEDULER_TYPE = [algoritmo_di_scheduling]
```

L'impostazione selezionata viene usata in *scheduler\_pick\_next\_task* per selezionare la funzione da invocare per la scelta della task successiva da eseguire.

In **OS161** invece l'unico algoritmo implementato è il RoundRobin, che viene gestito con la funzione *hardlock* presente in *kern/thread/thread.c*.

OS161 inoltre non fornisce la possibilità di poter switchare tra vari possibili algoritmi, dando modo di poter usufruire solo di uno.

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is light green. They are positioned diagonally, with the blue one partially covering the green one.

# IMPLEMENTAZIONE ALGORITMI DI SCHEDULING

MentOS



# High Priority First

L'algoritmo **High Priority First** è un algoritmo di scheduling per task aperiodici che risolve il problema evidenziato nel Round Robin, ovvero la presenza di task con priorità diversa.

A ogni processo viene assegnata una priorità tramite un numero statico: più risulta alta la priorità, più è basso il numero che viene assegnato al processo.

Il **vantaggio** è avere una esecuzione gerarchica, in quanto i processi si susseguono nella runqueue in base al valore della priorità.

Lo **svantaggio** è quello della starvation, ovvero l'attesa incondizionata di processi nella lista d'attesa per tempi troppo lunghi a causa di processi con priorità maggiore.



# High Priority First

Il codice dell'algoritmo **High Priority First** si presenta incompleto.

Le parti mancanti che sono state completate sono la scelta del task basandosi sulla priorità e la gestione del caso in cui due task presentino stessa priorità.

```
static inline task_struct *__scheduler_priority(runqueue_t *runqueue, bool_t skip_periodic)
{
#ifdef SCHEDULER_PRIORITY
    // Get the first element of the list.
    task_struct *next = list_entry(runqueue->queue.next, struct task_struct, run_list);

    // Get its static priority.
    time_t min = next->se.prio;

    // If there is just one task, return it; no need to do anything.
    if (list_head_size(&runqueue->curr->run_list) <= 1) {
        return runqueue->curr;
    }

    // Search for the task with the smallest static priority.
    list_for_each_decl(it, &runqueue->curr->run_list)
    {
        // Check if we reached the head of list_head, and skip it.
        if (it == &runqueue->queue)
            continue;
        // Get the current entry.
        task_struct *entry = list_entry(it, task_struct, run_list);
```

```
        // We consider only runnable processes
        if (entry->state != TASK_RUNNING)
            continue;
        // If entry is a periodic task, and we were asked to skip periodic tasks, skip it.
        if (__is_periodic_task(entry) && skip_periodic)
            continue;
        // Check if the entry has a lower priority.
        if (entry->se.prio < min) {
            next = entry;
            min = entry->se.prio;
        } else if (entry->se.prio == min) {
            // If the priorities are the same, check the arrival time ...
            // the first arrived one wins !
            if (entry->se.arrivaltime < next->se.arrivaltime) {
                next = entry;
                min = entry->se.prio;
            }
        }
    }
    return next;
}
```



# High Priority First

```
static inline task_struct *__scheduler_priority(runqueue_t *runqueue, bool_t skip_periodic)
{
#ifdef SCHEDULER_PRIORITY
    // Get the first element of the list.
    task_struct *next = list_entry(runqueue->queue.next, struct task_struct, run_list);

    // Get its static priority.
    time_t min = next->se.prio;
```

La priorità del task viene letta nel campo **prio** della struct **se** (struttura *sched\_entity*, contenuta in *task\_struct*, che tiene tutte le informazioni riguardo le attività di scheduling)

```
    // Check if the entry has a lower priority.
    if (entry->se.prio < min) {
        next = entry;
        min = entry->se.prio;
    } else if (entry->se.prio == min) {
        //if the priorities are the same, check the arrival time ...
        //the first arrived one wins !
        if (entry->se.arrivaltime < next->se.arrivaltime) {
            next = entry;
            min = entry->se.prio;
        }
    }
}
return next;
```

Nel caso in cui il task corrente abbia priorità minore di quello precedentemente selezionato, vengono aggiornati *next* e *min*. Nel caso di priorità uguale si è scelto di basarsi sul tempo di arrivo di un task, selezionando quello con tempo di arrivo precedente all'altro.

# Rate Monotonic

```
static inline task_struct *__scheduler_rm(runqueue_t *runqueue)
{
    //Initializing pointer to the next task to schedule
    task_struct *next = NULL, *curr_entry;
    //The first nearest next period is initialized with the maximum time
    time_t nearest_Next_Period = UINT_MAX;

    //Loop over runqueue to find the task with the closest next period
    list_for_each_decl(it, &runqueue->queue){

        //Do we reach the head of the runqueue? If yes, skip it !
        if(it == &runqueue->queue)
            continue;

        //Gets the pointer to the task_struct of the current element of list
        curr_entry = list_entry(it, task_struct, run_list);

        //Is the current entry non periodic or periodic but under analysis? If yes, skip it !
        if(!curr_entry->se.is_periodic || curr_entry->se.is_under_analysis)
            continue;

        //Is this the executed task? Is the next period about to start? If both yes, update the fields !
        if(curr_entry->se.executed && (curr_entry->se.next_period <= timer_get_ticks())){
            curr_entry->se.executed = false;
            curr_entry->se.deadline += curr_entry->se.period;
            curr_entry->se.next_period += curr_entry->se.period;
        }

        //check the next period field of a non executed task
        else if(!curr_entry->se.executed && (curr_entry->se.next_period < nearest_Next_Period)){
            next = curr_entry;
            nearest_Next_Period = next->se.next_period;
        }
    }

    //No tasks found? Let's execute a default algorithm, like RR
    if(next == NULL)
        next = __scheduler_rr(runqueue, true); //true means skip periodic tasks
    return next;
}
```

Il **Rate Monotonic** è un algoritmo per task periodici che si basa su una priorità fissa.

La priorità è assegnata a ogni task nella sua fase di analisi di schedulabilità in modo tale che una priorità alta del task corrisponda ad un'alta frequenza di richiesta (ovvero minor periodo del task).

Il rate monotonic è un algoritmo **pre-emptive**, ovvero un task può essere interrotto per dare priorità ad un altro appena arrivato con un periodo minore.



# Rate Monotonic

Alla funzione si passa solamente la lista di processi e si procede secondo questi step:

- inizializzazione variabili
- si itera poi sulla lista di processi
- se ci troviamo all'inizio della lista si passa al task successivo
- si ottiene il task corrente
- se il task è aperiodico o periodico ma sotto analisi si passa al successivo
- se il task è stato appena eseguito e il prossimo periodo sta per iniziare si setta *executed* a false e si incrementano i valori di *deadline* e *next\_period* (che indicano la absolute deadline e l'inizio del nuovo periodo) con il valore *period* (periodo atteso del task)
- nel caso in cui si trovasse un task con periodo minimo viene cambiato il valore *next* e *nearest\_Next\_Period*
- se nessun task è stato selezionato si ricorre ad analizzare i task aperiodici tramite Round Robin (scelto da noi per semplicità)



# Earliest Deadline First

L'algoritmo **Earliest Deadline First** è un altro algoritmo per task periodici, molto simile al precedente ma si basa sul campo *deadline*.

Lo scopo è quindi di andare ad individuare il task con deadline più vicina.

Anche questo è un algoritmo **pre-emptive**, ovvero un task può essere interrotto per dare priorità ad un altro appena arrivato con un deadline più vicina.

Nella slide successiva viene riportato un esempio figurato di come agisce il EDF.

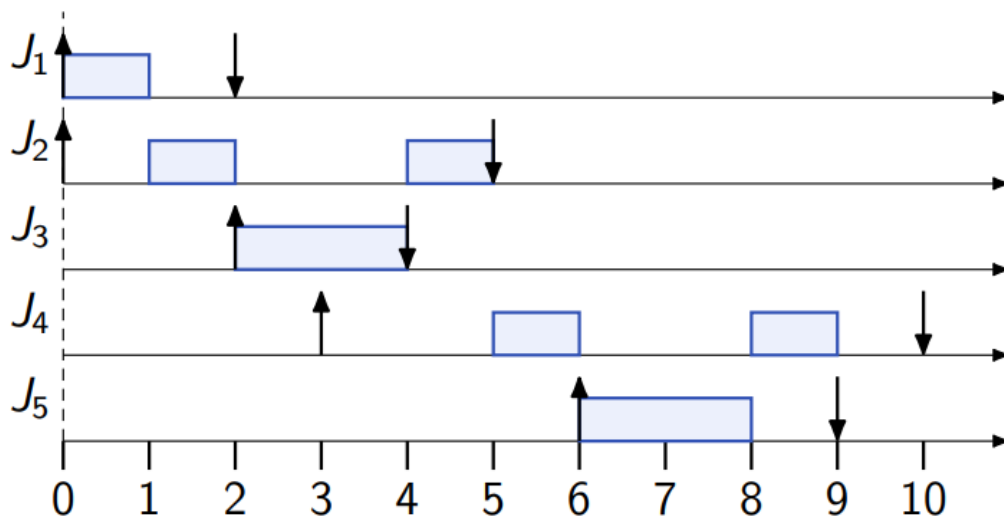
# Earliest Deadline First

	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$
$a_i$	0	0	2	3	6
$C_i$	1	2	2	2	2
$d_i$	2	5	4	10	9

$a$  = tempo di arrivo

$C$  = tempo di esecuzione

$d$  = deadline



Appena arriva un task con deadline minore, viene interrotto il task corrente dando priorità al nuovo arrivato. Al termine riprende il task con deadline minore in coda.

# Earliest Deadline First

```
static inline task_struct * __scheduler_edf(runqueue_t *runqueue)
{
    //Initializing pointer to the next task to schedule
    task_struct *next = NULL, *curr_entry;
    //The first next deadline is initialized with the maximum time
    time_t next_deadline = UINT_MAX;

    //Loop over runque to find the task with the closest next period
    list_for_each_decl(it, &runqueue->queue){

        //Do we reach the head of the runqueue? If yes, skip it !
        if(it == &runqueue->queue)
            continue;

        //Gets the pointer to the task_struct of the current element of list
        curr_entry = list_entry(it, task_struct, run_list);

        //Is the current entry non periodic or periodic but under analysis? If yes, skip it !
        if(!curr_entry->se.is_periodic || curr_entry->se.is_under_analysis)
            continue;

        //Is this the executed task? Is the next period about to start? If both yes, update the fields !
        if(curr_entry->se.executed && (curr_entry->se.next_period <= timer_get_ticks())){
            curr_entry->se.executed = false;
            curr_entry->se.deadline += curr_entry->se.period;
            curr_entry->se.next_period += curr_entry->se.period;
        } //check the deadline field of a non executed task
        else if(!curr_entry->se.executed && (curr_entry->se.deadline < next_deadline)){
            next = curr_entry;
            next_deadline = next->se.deadline;
        }
    }

    //No tasks found? Let's execute a default algorithm, like RR
    if(next == NULL)
        next = __scheduler_rr(runqueue, true); //true means skip periodic tasks
    return next;
}
```

Il codice è pressoché simile al Rate Monotonic.

L'unica differenza è che si deve andare a fare il controllo sulla **deadline** e non più sul periodo.

Anche qui nel caso in cui nessuna task è stata selezionata si ricorre al Round Robin per l'analisi dei task aperiodici



# Aperiodic Earliest Deadline First

L'algoritmo **Aperiodic Earliest Deadline First** è una versione semplificata dell'algoritmo EDF per task aperiodici, in quanto non vengono effettuati i controlli di periodicità e non vengono aggiornati i periodi di eventuali task che sono appena stati eseguiti o stanno per iniziare. Tutti i task vengono trattati come aperiodici.

Rimane solo il check sulla deadline per la selezione del task con deadline più vicina

Un'altra differenza è che questo algoritmo non necessita di analisi di schedulabilità. Ciò potrebbe portare alcune task a mancare la propria deadline.





# Aperiodic Earliest Deadline First

```
static inline task_struct *__scheduler_aedf(runqueue_t *runqueue)
{
    //Initializing pointer to the next task to schedule
    task_struct *next = NULL, *curr_entry;
    //The first next deadline is initialized with the maximum time
    time_t next_deadline = UINT_MAX;

    list_for_each_decl(it, &runqueue->queue){

        //Do we reach the head of the runqueue? If yes, skip it !
        if(it == &runqueue->queue)
            continue;

        curr_entry = list_entry(it, task_struct, run_list);

        //checks of deadline
        if(curr_entry->se.deadline != 0){
            if(curr_entry->se.deadline <= next_deadline){
                next = curr_entry;
                next_deadline = next->se.deadline;
            }
        }
    }

    //No tasks found? Let's execute a default algorithm, like RR
    if(next == NULL)
        next = __scheduler_rr(runqueue, true); //true means skip periodic tasks
    return next;
}
```



# Completely Fair Scheduler

Il **Completely Fair Scheduler (CFS)** è un algoritmo per task aperiodici. Ha come obiettivo quello di prevenire la starvation dei processi. Per farlo, la CPU verrà assegnata equamente ai vari processi a seconda di un valore che indica il tempo totale di esecuzione di un processo basato su un peso. Questo valore è calcolato tramite:

$$\text{vruntime}(p) = \text{vruntime} + \text{delta\_exec} * (\text{NICE\_0\_LOAD} / \text{weight}(p))$$

- `vruntime` è il tempo virtuale;
- `delta_exec` è l'ultimo tempo utilizzato dal processo `p` nella CPU;
- `NICE_0_LOAD` generalmente è il valore di default dei processi con priorità normale
- `weight(p)` è il peso di `p`.



# Completely Fair Scheduler

```
static inline task_struct *__scheduler_cfs(runqueue_t *runqueue, bool_t skip_periodic)
{
    // Get the first element of the list.
    task_struct *next = list_entry(runqueue->queue.next, struct task_struct, run_list);

    // Get its virtual runtime, setting it as the initial minimum value !
    time_t min = next->se.vruntime;

    // If there is just one task, return it; no need to do anything.
    if (list_head_size(&runqueue->curr->run_list) <= 1) {
        return runqueue->curr;
    }

    // Search for the task with the smallest vruntime value.
    list_for_each_decl(it, &runqueue->curr->run_list)
    {
        // Do we reach the head of the runqueue? If yes, skip it !
        if (it == &runqueue->queue)
            continue;
        // Get the pointer to the task struct of the current element of list
        task_struct *entry = list_entry(it, task_struct, run_list);
        // We consider only runnable processes
        if (entry->state != TASK_RUNNING)
            continue;
        // Is the task a periodic task and skip_periodic==true ? If yes, skip it!.
        if (__is_periodic_task(entry) && skip_periodic)
            continue;

        // Check if the element in the list has a smaller vruntime value.
        if (entry->se.vruntime < min) {
            next = entry;
            min = entry->se.vruntime;
        }
    }
    return next;
}
```

Il codice è pressoché simile a quelli precedenti.

L'unica differenza è che si deve andare a fare il controllo su **vruntime**, selezionando il task con valore minimo.

Maggior attenzione si può riporre sulla funzione di aggiornamento del vruntime...

# Completely Fair Scheduler

```
static void __update_task_statistics(task_struct *task)
{
    // See 'prio.h' for more support functions.
    assert(task && "Current task is not valid.");

    // While periodic task is under analysis is executed with aperiodic
    // scheduler and can be preempted by a "true" periodic task.
    // We need to sum all the execution spots to calculate the WCET even
    // if is a more pessimistic evaluation.
    // Update the delta exec.
    task->se.exec_runtime = timer_get_ticks() - task->se.exec_start;

    // Perform timer-related checks.
    update_process_profiling_timer(task);

    // Set the sum_exec_runtime.
    task->se.sum_exec_runtime += task->se.exec_runtime;

    // If the task is not a periodic task we have to update the virtual runtime.
    if (!task->se.is_periodic) {
        // Get the weight of the current task.
        time_t weight = GET_WEIGHT(task->se.prio);
        // If the weight is different from the default load, compute it.
        if (weight != NICE_0_LOAD) {
            // Get the multiplicative factor for its delta_exec.
            double factor = ((double)NICE_0_LOAD)/((double)weight);
            // Weight the delta_exec with the multiplicative factor.
            task->se.exec_runtime = (int)((double)task->se.exec_runtime*factor);
        }
        // Update vruntime of the current task.
        task->se.vruntime += task->se.exec_runtime;
    }
}
```

L'algoritmo CFS richiede una funzione `__update_task_statistics`, con il compito di aggiornare i pesi delle task (ovvero i `vruntime`), che prende come parametro solo la task corrente, e viene chiamata dalla `scheduler_pick_next_task` prima della funzione di scheduling.

I vari step sono:

- Calcolo di **delta\_exec** (salvato nel campo `exec_runtime` della `se`) come differenza tra istante di tempo attuale ed il tempo d'inizio dell'esecuzione;
- Calcolo del **peso della task** usando una tabella che correla priorità e peso, e nel caso in cui questo peso non corrispondesse al carico di default (`NICE_0_LOAD`) si scala il `delta_exec` della task;
- Aggiornamento di **vruntime** aggiungendo ad essa il `delta_exec`.



# GRAZIE PER L'ATTENZIONE

Pedone Matteo - s310171

Pulvirenti Roberto - s317704

Venezia Giuseppe - s309625