Esame di Sistemi Operativi
AA 2017/18
26 Marzo 2018

Nome	Cognome	Matricola

Istruzioni

Scrivere il proprio nome e cognome su ogni foglio dell'elaborato. Usare questo testo come bella copia per le risposte, utilizzando l'apposito spazio in calce alla descrizione dell'esercizio.

Esercizio 1

Sia data la seguente tabella che descrive il comportamento di un insieme di processi

processo	**tempo di inizio**	**CPU burst 1**	**IO burst 1**	**CPU burst 2**	**IO burst 2**
P1	0	5	5	3	1
P2	1	2	5	2	5
P3	5	8	1	8	1
P4	7	1	9	1	9

Domanda Si assuma di disporre di uno scheduler preemptive *Shortest Remaining Job First* (SRJF). Si assuma inoltre che:

- l'operazione di avvio di un processo lo porti nella coda di ready, ma **non** necessariamente in esecuzione
- il termine di un I/O porti il processo che termina nella coda di ready, ma **non** in esecuzione.

. Si illustri il comportamento dello scheduler in questione nel periodo indicato, avvalendosi degli schemi di seguito riportati (vedi pagina seguente).

Soluzione Date queste premesse, in Figura 1 e' riportata la traccia di esecuzione dello lo scheduler specificato. I cicli di cpu burst sono indicati in verde ed in rosso l'I/O burst; le caselle azzurre indicano l'arrivo di un nuovo processo mentre in giallo viene indicata la fine dello stesso. E' bene notare che in

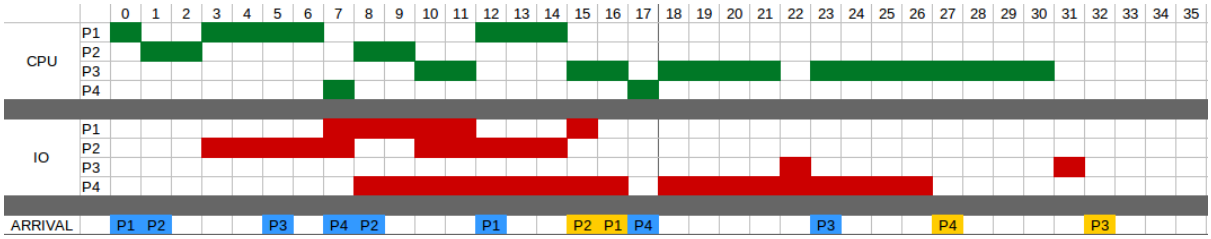


Figure 1: SRJF Scheduler

questo algoritmo di scheduling si suppone che ogni volta che un processo si presenta per la prima volta esso indichi quanto duri il suo CPU burst.

Nome	Cognome	Matricola

Esercizio 2

Sia dato un sottosistema di memoria con paginazione e segmentazione, caratterizzato dalle seguenti dimensioni:

- frame 4KB
- memoria fisica indirizzabile 16GB

Inoltre, ogni indirizzo logico necessita di 48 bit.

Domande

- Quanti bit sono necessari per l'address bus?
- Quanti bit sono necessari per indicizzare una pagina?
- Quanti segmenti ci saranno al massimo in questo sistema?

Soluzione

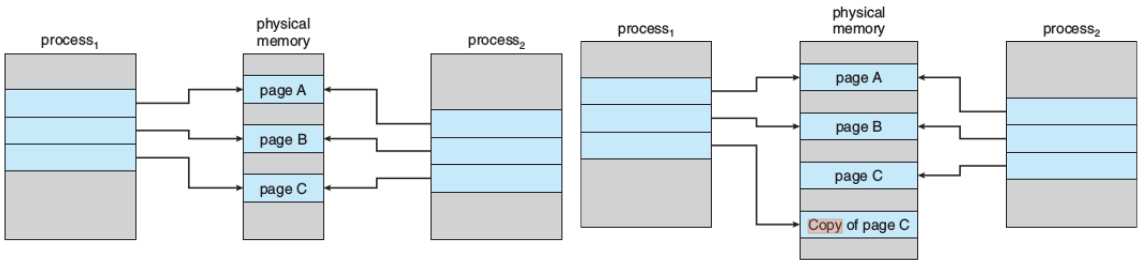
- L'address bus dovra' essere in grado di indirizzare almeno 16GB di memoria, quindi saranno necessari 34 bit (come minimo).
- $34 - 12 = 22$.
- $48 - 34 = 14$ quindi 2^{14}.

Nome	Cognome	Matricola

Esercizio 3

Cosa significa *Copy On Write* (COW) nella gestione della memoria? Illustrare con un semplice esempio il suo funzionamento.

Soluzione Creando un nuovo processo figlio a partire da un padre implica una copia della memoria usata da quest'ultimo. E' bene notare che molto probabilmente il figlio chiamera' una `exec()`, percio' tale copia puo' risultare inutile. A tal porposito e' possibile usare la tecnica di *Copy-On-Write*. In questo caso, inizialmente entrambi i processi - padre e figlio - condividono le stesse pagine di memoria - marcate come *copy-on-write pages* - e quando uno dei due vuole modificare una delle pagine questa verra' copiata. Il processo e' illustrato visivamente nella Figura 2.



(a) Prima che P1 modifichi qualcosa

(b) Dopo che P1 ha modificato la pagina C

Figure 2: Illustrazione del funzionamento del processo *Copy-On-Write*: i due processi condividono le stesse pagine di memoria finche' una delle due non necessita di modificarne qualcuna. In quel caso, la pagina interessata verra' copiata.

Nome	Cognome	Matricola

Esercizio 4

Come si puo' implementare l'algoritmo di sostituzione delle pagine "*Second Chance*"?

Soluzione L'algoritmo *Second Chance* - o altresi' noto come *clock algorithm* - e' un algoritmo FIFO per la sostituzione delle pagine che usa un reference bit (implementato in hardware) per capire quale pagine bisogna rimpiazzare. In particolare:

- se il reference bit di una pagina e' 0 allora essa viene rimpiazzata;
- se il reference bit e' 1 il bit viene settato a 0 e la pagina viene lasciata in memoria, quindi si passa alla pagina successiva - usando le stesse regole.

Un modo per implementare tale algoritmo e' attraverso una *coda circolare* con un puntatore alla pagina da rimpiazzare - che scorre nella coda finche' non trova una pagina con reference bit pari a 0. Trovata una pagina che soddisfa le richieste, viene eliminata dalla coda e rimpiazzata con la nuova. L'algoritmo e' illustrato in Figura 3.

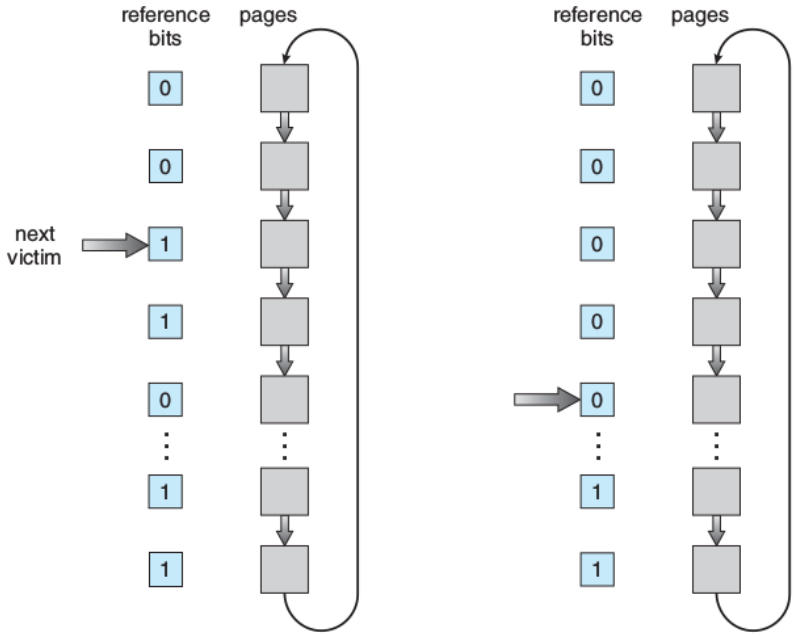


Figure 3: L'algoritmo di sostituzione delle pagine *Second Chance*. Il puntatore scorre finche' una pagina con reference bit 0 non viene trovata; quindi la nuova pagina prendera' il posto di quest'ultima nella coda circolare.

Nome	Cognome	Matricola

Esercizio 5

Descrivere con un breve esempio un File System con allocazione concatenata (*Linked Allocation*).

Soluzione Una delle problematiche a cui un File System deve sopperire e' come allocare lo spazio necessario ad ogni file che andra' a storare sul disco. Nel caso della *Linked Allocation*, ogni file sara' una lista concatenata di blocchi, dove ogni blocco contiene il puntatore al successivo. Cio' permette quindi di allocare i blocchi in maniera scattered sul disco (non devono essere per forza contigui). Il file finisce quindi con un puntatore a `NULL`. Ovviamente, in questo caso, localizzare un blocco puo' richiedere numerosi cicli di I/O. Inoltre i puntatori utilizzati influiranno negativamente sullo spazio disponibile per contenere i dati: supponendo che i blocchi siano da 512 byte e che un puntatore sia 4 byte, lo 0.78% del disco sara' occupato da puntatori.

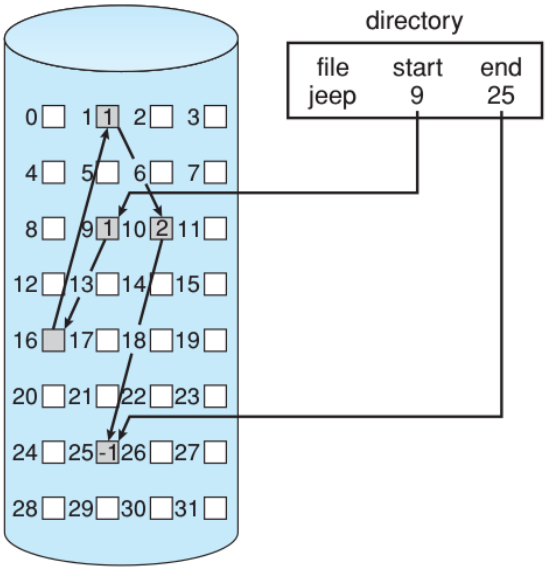


Figure 4: Esempio di File System con *linked allocation*. Il file *jeep* iniziera' nel blocco 9 ed avra' al suo interno il puntatore al successivo - il 16 - e cosi' via, finche' nel blocco 25 `next` → `NULL`.

Un altro problema di tale implementazione e' la sua affidabilita'. Infatti basta un solo puntatore corrotto per danneggiare uno o piu' files.

Un esempio visivo di tale implementazione e' riportato in Figura 4.

Nome	Cognome	Matricola

Esercizio 6

Cosa succede in questo programma se eseguito su un sistema con 4 CPU che implementa i thread a livello utente o al livello kernel?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 4

typedef struct {
  int id;
  int result;
} ThreadArgs;

int doVeryHeavyAndLongUselessComputation(const int src_);

void* runner(void* arg_) {
  ThreadArgs* t_args = (ThreadArgs*)arg_;
  printf("ID: %d \n", t_args->id);
  t_args->result = doVeryHeavyAndLongUselessComputation(t_args->id);
  printf("Exit thread %d \n", t_args->id);
  return 0;
}

ThreadArgs args[NUM_THREADS];
pthread_t threads[NUM_THREADS];

int main(int argc, char** argv) {
  for (int i = 0; i < NUM_THREADS; ++i) {
    args[i].id = i;
    pthread_create(&threads[i], NULL, runner, (void*) &args[i]);
  }

  for (int i=0; i<NUM_THREADS; ++i){
    void* retval=0;
    pthread_join(threads[i], &retval);
  }
  return 0;
}
```

Soluzione Con un una implementazione dei thread a livello kernel tale programma girera' piu' velocemente - o al limite nello stesso tempo - dell'implementazione a livello user.

Cio' poiche' piu' user thread possono essere mappati su un singolo kernel thread, andando ad usare effettivamente una sola CPU - i.e. modello di multi-threading *many-to-one*. La libreria GNU Portable Thread segue questo modello di multi-threading.

Nome	Cognome	Matricola

Esercizio 7

Cos'e' un *File Control Block* (FCB)? Cosa contiene al suo interno?

Soluzione Il FCB e' una struttura dati che contiene tutte le informazioni relative al file a cui e' associato. Esempi di informazioni possono essere: permessi, dimensione, data di creazione, numero di inode (se esiste), ecc. . Inoltre il FCB contiene informazione su la locazione sul disco dei dati del file - ad esempio in un FS con allocazione concatenata il puntatore al primo blocco del file. In Figura 5 e' riportata una illustrazione di tale struttura.

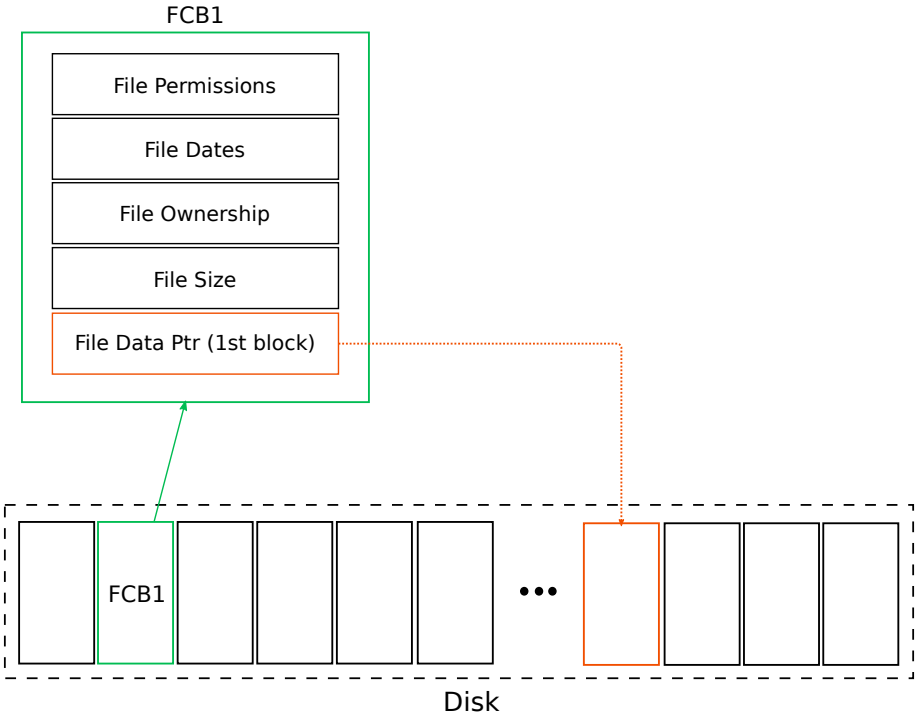


Figure 5: Esempio di FCB.La struttura contiene tutti gli attributi del file compresa la locazione dei dati - qui rappresentato dal puntatore al primo blocco della lista contente i dati, supponendo un FS con linked allocation.

Nome	Cognome	Matricola

Esercizio 8

Che relazione c'e' tra un File Descriptor ed una entry nella tabella globale dei file aperti del file system?

Soluzione Un File Descriptor consiste in un file handler che viene restituito ad un processo in seguito ad una chiamata alla syscall `open()`. In seguito a tale chiamata, il sistema scandisce il FS in cerca del file e, una volta trovato, il FCB e' copiato nella tabella globale dei file aperti. Per ogni singolo file aperto, anche se da piu' 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 puntera' alla relativa entry nella tabella globale, insieme ad altre informazione - e.g. permessi, locazione del cursore all'interno del file, ecc. La syscall `open()` restituisce per l'appunto l'entry all'interno della tabella del processo - il File Descriptor. Piu' `open()` su uno stesso file da parte di uno stesso processo generano descrittori diversi.

Nome	Cognome	Matricola

Esercizio 9

Si consideri un Sistema Operativo batch. Si assuma che esso debba essere in grado di gestire job che arrivano ogni 0.5s e durino in media 10s.

Domanda Specificare, quindi, il numero minimo di entries della tabella dei processi e motivare la risposta.

Soluzione Per dimensionare opportunamente la coda dei processi e' possibile usare la Legge di Little, descritta dalla formula:

$$n = \lambda \cdot W \tag{1}$$

dove n indica la dimensione della coda, λ la frequenza media di arrivo e W la durata media dei processi. Dato cio' la dimensione minima della tabella dei processi sara pari a 20 - per semplice sostituzione nella (1).

Nome	Cognome	Matricola

Esercizio 10

Dato un buffer illimitato gestito internamente tramite una lista concatenata come riportato di seguito:

```
// simple linked list element
typedef struct {
  struct ListItem* prev;
  struct ListItem* next;
  void* datum;
} ListItem;


// list structure
typedef struct {
  struct ListItem* first;
  struct ListItem* last;
  int size;
} ListHead;


// adds at the end of the list l a new record storing datum
void ListItem_pushBack(ListHead* l, void* datum);

// returns the first available element of the list
void* ListItem_popFront(ListHead* l);


// blocking linked list
typedef struct {
  sem_t semaphore; // number of elements in the list
  pthread_mutex_t mutex; // ward for the non-atomic operations of push and pop
  ListHead buffer;
} UnlimitedBuffer;

void UnlimitedBuffer_init(UnlimitedBuffer *b);
void UnlimitedBuffer_postDatum(UnlimitedBuffer *b, void* d);
void* UnlimitedBuffer_getDatum(UnlimitedBuffer *b);
```

Domanda Si fornisca una implementazione delle funzioni

- `void UnlimitedBuffer_init(UnlimitedBuffer *b)` la quale inizializza la struttura `UnlimitedBuffer`;
- `void UnlimitedBuffer_postDatum(UnlimitedBuffer *b, void* d)` responsabile di mettere in coda un dato e sbloccare eventuali chiamanti;
- `void* UnlimitedBuffer_getDatum(UnlimitedBuffer *b)` la quale prende dalla coda un dato.

Soluzione Una possible implementazione delle funzioni e' la seguente:

```
void UnlimitedBuffer_init(UnlimitedBuffer *b) {
  sem_init(&(b->semaphore), 0, 1); // initialize semaphore to 1
  pthread_mutex_init(&(b->mutex), NULL); // initialize the mutex
  // initialize the buffer - aka the linked list
  b->buffer.first=0;
```

```
  b->buffer.last=0;
  b->buffer.size=0;
}


void UnlimitedBuffer_postDatum(UnlimitedBuffer *b, void* d) {
  // lock in order to be the only one that modifies the buffer
  pthread_mutex_lock(&(b->mutex));
  // add the new guy
  ListItem_pushBack(&(b->buffer), d);
  // release the mutex
  pthread_mutex_unlock(&(b->mutex));
  // post on the semaphore (there is a new guy in the buffer)
  sem_post(&(b->semaphore));
}


void* UnlimitedBuffer_getDatum(UnlimitedBuffer *b) {
  void* d = 0;
  // wait until there is an available guy in the buffer
  sem_wait(&(b->semaphore));
  // lock in order to be the only one that modifies the buffer
  pthread_mutex_lock(&(b->mutex));
  // take the guy
  d = ListItem_popFront(&(b->buffer));
  // release the mutex
  pthread_mutex_unlock(&(b->mutex));
  return d;
}
```