Prova del 2018 settembre 19

Esercizio p.1

```
#define aereporto_di_caricamento BLQ   #da cambiare per altre stazioni di caricamento

monitro dispathc:
    condition stazioni[airport_codes.length]
    lista valigie <owner> [airport_codes.length]
    int current_valigie = 0;
    condition cart;
    int posizione = 0;

    init():
        cart = new condition()
        for staz in stazioni:
            staz = new condition()


    cartat(code):
        posizione = code
        stazioni[code].signal()
        cart.wait()


    load(dstcode, owner):
        if (current_valigie < MAX):
            current_valigie++
        else:
            cart.signal()
            stazioni[aereoporto_di_caricamento].wait()

        valigie[dstcode].apppend(owner)


    type_owner unload(dstcode):
        if (posizione != dstcode):
            stazioni[dstcode].wait()

        if valiegie[dstcode].isEmpty():
             cart.signal()
             stazioni[dstcode].wait()

        current_valigie--
        owner = valigie[dstcode].pop()
        return owner
```

Esercizio p.2

Troppo difficile, ci devo ancora ragionare. E' comunque un po' strano perchè se è sincrono come fanno ad esserci messagi in LIFO per ogni processo? Forse il ricevente non specifica il processo dal quale vuole ricevere?

```
Stack <msg> pila_mess = new Stack()


void send(msg, p):
    asedn(msg, p)
    pila_mess.push( ( {messaggio": msg, "sender": p} )
    ack = arecv(p)

type recv(p):
    Stack <msg> tmp = new Stack;
    msg = res

    while (pila_mess.top().sender != p):
        mess_tmp = pila_mess.pop()
        tmp.push(mess_tmp)

    res = arecv(p)

    while (not pila_mess.isEmpty()):
        asend(pila_mess.pop(), p)

    asend(null, p)
    return res
```

Esercizio g.2

Diagramma di Gannt

Legenda: - il `-` indica l'esecuzione del processo che è scritto prima del `-` - il `+` indica che il processore non sta eseguendo niente - con `Px` indico che viene lanciato il processo `Px` - con `-Px` indico che il processo `Px` termina - con `X` indico che in quell'istante l'IO di un processo è terminato

```
     3  4  5   6   7   9  10  12  13  15  17  18  20  23  25
     |  |  |   |   |   |  X   |   |   |   |   X   |   X   |
P1---------P2----------P3-----P2--P1--P3--++++P2--++++P3---
        P2          P3                  -P1          -P2     -P3
```

- istante 3: coda ready è vuota, proseguo con p1
- istante 5: P1 va in waiting, eseguo P2
- istante 6: coda ready è vuota, proseguo con p2

- istante 9: P2 va in ready con 1 secondi mancanti a `long_compute()`, eseguo P3
- istante 10: P1 finisce IO, va in ready [P2, P1]
- istante 12: P3 va in ready con 2 secondi mancanti, eseguo P2 [P1, P3]
- istante 13: P2 va in waiting, eseguo P1 [P3]
- istante 15: P1 termina, eseguo P3
- istante 17: P3 va in waiting [P3]
- istante 18: P2 va in ready, eseguo P2
- istante 20: termino P2
- istante 23: P3 va in ready, eseguo P3
- istante 25: termino P3

Esercizio g.2

a) Perchè l'implementazione è inefficiente. O memorizzo dei contatori per ogni pagina, ma dovrei gestire l'overflow e dovrei scandire tutte le pagine in memoria per scegliere la vittima e tali contatori sarebbero in memoria quindi dovrei fare un sacco di accessi, o uso uno stack ma ogni accesso ad una pagina dovrei modificare lo stack per mettere tale pagina in cima, aggiornando così sei puntatori

b) Il journaling permette di ripristinare il filesystem ad uno stato coerente, ma ad esempio le transazioni che erano in coda (non confermate) non vengono in coda. Inoltre un il journaling non può salvare da guasti meccanici del disco.

c) No perchè serve il server

d) Si esistono, la conseguenza è che non si possono implementare meccanismi di protezione. Un processo utente ha gli stessi privilegi del kernel, quindi può implementare qualsiasi syscall nel suo codice. Ad esempio cui può modificare il filesystem (vogliamo evitare).