Prova scritta di sistemi operativi dell'1 giugno 2022

c.1

La soluzione proposta fa uso del modello a riferimento per i tipi non incorporati (à-la-Java). `auto` indica un tipo da inferire banalmente dal contesto.

```
monitor delay {

  // Per ogni processo attualmente bloccato, contiene il numero di tick
  // rimanenti e la condizione sul quale è bloccato.
  auto blocked = new List<Pair<int, Condition>>();

  // Contiene il numero di processi che erano bloccati che erano bloccati
  // all'inizio dell'ultimo tick: in altre parole, è una copia di blocked.size()
  // non sempre aggiornata. Funge da valore di ritorno per wait_ticks.
  int blockedLengthBackup = 0;

  int wait_tick(int nticks) {
    if (nticks < 1)
      return -1; // errore: nticks non valido
    auto c = new Condition();
    blocked.add(new Pair<int, Condition>(nticks, c));
    c.wait();
    return blockedLengthBackup;
  }

  void tick(void) {
    blockedLengthBackup = blocked.size();
    foreach (auto p in blocked)
      if (--p.first <= 0) { // prima aggiorno, poi controllo
        p.second.signal();
        // si suppone che remove() non faccia danni in un foreach
        blocked.remove(p);
      }
  }

}
```

c.2

La soluzione proposta fa uso del modello a riferimento per i tipi non incorporati (à-la-Java). `auto` indica un tipo da inferire banalmente dal contesto. Vengono usate variabili locali `static` anziché globali per enfatizzare la separazione fra strutture dati di `asend` e di `arecv`: quale che sia la propria scelta stilistica,

però, si tenga sempre a mente che il paradigma del passaggio di messaggi è a memoria privata, e che ogni processo ha quindi accesso solo alla propria copia della struttura dati. A un `msg_t` possiamo sempre prependere una intestazione usando `prepend_token()`. Essa può essere recuperata usando `first_token()`, mentre si può accedere al messaggio originale usando `tail()`.

```
// Il tipo di un identificatore incrementale di messaggio: a parità di mittente
// e destinatario, questi identificatori sono univoci e permettono quindi di
// ricostruire l'ordine FIFO.
typedef int mid_t;

void asend(msg_t msg, pid_t dest) {
  // A ogni destinatario, associa il prossimo mid_t da usare.
  static auto ids = new Map<pid_t, mid_t>();
  if (!ids.containsKey(dest)) // nuovo destinatario: partiamo da mid_t 0
    ids[dest] = 0;
  msg.prepend_token(ids[dest]++); // prima uso ids[dest], poi lo aggiorno
  nfasend(msg, sender);
}

msg_t arecv(pid_t sender) {
  // A ogni mittente, associa una coppia contenente:
  // 1. una coda con priorità dei messaggi. La relazione d'ordine implicita su
  //    essi definita considera il token più significativo (di fatto, il mid_t);
  // 2. il mid_t del prossimo messaggio di quel mittente.
  static auto queues = new Map<pid_t, Pair<PriorityQueue<msg_t>, mid_t>>();
  if (!queues.containsKey(sender)) // nuovo mittente: coda vuota e mid_t 0
    queues[sender] =
      new Pair<PriorityQueue<msg_t>, mid_t>(new PriorityQueue<msg_t>(), 0);
  // attendo di avere il messaggio con mid_t desiderato
  while (queues[sender].first.isEmpty() ||
         queues[sender].first.minKey() > queues[sender].second)
    queue[sender].add(nfarecv(sender));
  auto msg = queues[sender].first.removeMin(); // prendo il messaggio
  ++queues[sender].second; // prossima volta, messaggio con mid_t successivo
  return msg.tail(); // scarto il mid_t (informazione inutile per l'utente)
}
```

g.1

Dobbiamo risolvere nell'incognita IC specificandone i valori minimi. A prescindere dal fatto che lo stato sia sicuro o meno, deve sempre valere l'invariante del capitale iniziale:

$$\forall i \in \{1, 2, 3\}\ c_i \leq IC$$

Questo ci fornisce un limite inferiore per qualunque capitale iniziale, e dunque anche quelli degli stati sicuri:

$$(4, 4, 4) \leq IC$$

Da qui, possiamo derivare un limite inferiore anche per COH:

$$(2, 2, 0) \leq COH$$

Per dimostrare che un qualunque stato sia sicuro, dobbiamo individuare una permutazione dei processi che ci permetta di soddisfarli tutti. Procediamo per casi sulla scelta del primo processo della permutazione:

Cominciare da p1

Oltre al vincolo di cui sopra, per soddisfare p1 deve valere:

$$n_{(1)} = n_1 = c_1 - p_1 = (4, 2, 4) - (2, 0, 0) = (2, 2, 4) \leq avail[1] = COH$$

I due vincoli possono essere riassunti nella equivalente forma:

$$(2, 2, 4) \leq COH$$

Convenientemente, questo vincolo ci permette di soddisfare anche i p2 e p3 in quest'ordine: scegliendo questa permutazione, infatti, valgono sia $n_{(2)} \leq avail[2]$ che $n_{(3)} \leq avail[3]$.

Quest'ultimo vincolo su COH è quindi sufficiente e necessario per la sicurezza di uno stato che può soddisfare tutti i processi a partire da p1 (è quindi anche condizione sufficiente per la sicurezza di uno stato nel caso generale, ma non è affatto detto che sia necessaria anche nel caso generale). Riesprimiamo la nostra relazione in termini di capitale iniziale:

$$(4, 4, 8) \leq IC$$

$(4, 4, 8)$ è quindi un primo minimo per il capitale iniziale.

Cominciare da p2

Analogamente al primo caso:

$$n_{(1)} = n_2 = c_2 - p_2 = (4, 4, 2) - (0, 2, 0) = (4, 2, 2) \leq avail[1] = COH$$

$$(4, 2, 2) \leq COH$$

La permutazione (p2, p3, p1) è ora automaticamente soddisfatta.

Tornando a IC, notiamo che $(6, 4, 6)$ è il nostro secondo minimo.

Cominciare da p3

Analogamente agli altri casi:

$$n_{(1)} = n_3 = c_3 - p_3 = (2, 4, 4) - (0, 0, 4) = (2, 4, 0) \leq avail[1] = COH$$

$$(2, 4, 0) \leq COH$$

La permutazione (p3, p1, p2) è ora automaticamente soddisfatta.

Tornando a IC, notiamo che $(4, 6, 4)$ è il nostro terzo minimo.

g.2

1. La paginazione nasce come una tecnica di allocazione della memoria, tuttavia presenta buone qualità anche nel contesto della memoria virtuale. Possiamo elencarne alcune:
   - la suddivisione della memoria in pagine (o la loro controparte fisica, i frame) è in linea con le necessita' della memoria virtuale di una suddivisione della memoria in modo da poter fare swap-(in,out) dei dati non utili al momento;
   - la tenica è implementabile tramite hardware semplice (molto più di quello richiesto per la segmentazione) il che riduce notevolmente il costo dell'aggiunta della memoria virtuale ad un sistema;
   - la suddivisione in pagine riduce la frammentazione interna (vista la dimensione tipicamente picola dei frame) e elimina (quasi) completamente la frammentazione esterna (ne rimane tra l'area di memoria del kernel e l'inizio della prima pagina);
   - è una tecnica che risulta trasparente al programmatore e non richiede alcuno sforzo implementativo da chi scrive software di livello superiore al kernel. In questo risulta molto più comoda della segmentazione che necessita di cura da parte del programmatore o di chi implementa il compilatore.
2. Lo scheduler Round Robin e' pensato per programmi altamente interattivi, dove non si vuole far aspettare l'utente che interagisce continuamente con il programma. I sistemi batch si pongo all'esatto opposto dello spettro, in quanto sono macchine pensate per eseguire lavori in differita, senza l'interazione umana. In questo scenario i continui context-switch provocati dallo scheduler RR causerebbero solo un degrago delle performance e nessun beneficio.

3. Il file system FAT mantiene i blocchi del disco in cui sono contenuti i dati in una tabella centralizzata denominata appunto File Allocation Table. In questo modo quando si necessita di accedere ad un file basta leggere la tabella e seguire i puntatori ai blocchi contenuti nella tabella stessa. Così, ad esempio, un disco meccanico deve svolgere molti meno spostamenti della testina per completare una lseek. In un filesystem con allocazione concatenata invece si sfruttano gli ultimi word-length bit di un blocco per puntare al successivo in cui si trova il proseguimento dei dati. Questa tecnica costringe la testina del disco a muoversi continumanete da un settore all'altro del disco per una ipotetica lseek, saltando da un blocco all'altro. Un ulteriore vantaggio di FAT e' che lascia tutta la dimensione del blocco a disposizione per i dati da memorizzare, diminuendo la frammentazione interna in alcune casistiche.
4. La tecnica del sale consiste nell'appendere una quantità di byte random al termine di una password prima di cederla alla one way function che ne genererà l'hash. In seguito la hash dovrà essere memorizzata assieme al suo sale per poterne effettuare la verifica. In questo modo si prevengono attacchi di tipo dizionario in cui una serie di password comuni è già stata hashata e si controlla quali hash sul sistema bersaglio combaciano con quelle nel dizionario dell'attaccante. Con il sale si rende inutile un dizionario generato senza tenere conto del sale e si costringe l'attaccante a generare una quantità di hash di dimensioni proibitive (il sale infatti aumenta notevolmente il numero di combinazioni). L'unica speranza che resta all'attaccante è quella di computare le hash sul momento, neutralizzando dunque questo tipo di attacco. Si noti che affinché l'attaccante possa penetrare un sistema in questo modo egli dev'essere in possesso del file /etc/shadow, o su sistemi datati, /etc/passwd.