/* link testo https://www.cs.unibo.it/~renzo/so/compiti/2022.02.14.tot.pdf */

- Esercizio c1
- Esercizio c2 - Soluzione del prof
- Esercizio g1 - NOTE - *Perchè? - Metodi di implementazione*
  - Punti da seguire per risolvere l'esercizio:
- Esercizio g2
  - Risposte

Esercizio c1

Implementare un semaforo con monitor con 2 operazioni:

```
datatype dP(void);
void dV(datatype data);
```

Non è previsto assegmento di valore iniziale nel costruttore, l'invariante è lo stesso dei semafori (con init = 0): ndP <= ndV (dove ndP e ndV rappresentano rispettivamente il numero di operazioni dP e dV completate. I dati passati come parametro alla dV devono essere memorizzati in ordine LIFO. L'operazione nP restituisce il valore più recente fra quelli memorizzati (e lo cancella dalla struttura dati).

```
monitor semdata{
    datatype pila = stack();
    condition ok2p;

    procedure entry datatype dP(void){
        if pila.len() <= 0:
            ok2p.wait();
        return pila.pop();  // prende il dato più recente e lo cancella
    }

    procedure entry void dV(datatype data){ // data LIFO
        pila.push(data);        // inserisco data nella pila
        ok2p.signal();
    }
}
```

Esercizio c2

Dato un servizio di message passing asincrono implementare un servizio di message passing sincrono con selezione ordinata che ha la seguente interfaccia:

```
void snsend(msgtype msg, pid_t dest);
msgtype snrecv(pid_t sender, int n);
```

La funzione **snrecv** deve restituire l'n-esimo messaggio proveniente dal mittente specificato (che può essere any). - Se n == 0 restituisce l'ultimo messaggio. Esempi: - m = snrecv(tizio, 1): restituisce il primo messaggio da tizio (attende se non ve ne sono) - m = snrecv(ANY, 42): restituisce il 42-mo messaggio da chiunque (attende se ci sono meno di 42 messaggi in attesa di essere ricevuti) - m = snrecv(caio, 0): restituisce l'ultimo messaggio ricevuto da Caio (attende se non ci sono messaggi pendenti da Caio) - m = snrecv(ANY, 0): restituisce l'ultimo messaggio ricevuto, indipendentemente dal mittente.

```
#define ACK     // costante ovvia

void snsend (msgtype msg, pid_t dest){
    asend([getpid(), msg], dest);
    while (ACK != arecv(dest)){
        // non fai nulla, aspetti, sei bloccato!
    }
}

queue messages;
msgtype snrecv(pid_t sender, int n){
    messages.add(arecv(sender));    // aggiungo i messaggi ricevuti alla coda
    // cerco i messaggi di sender e conto fino a quando non trovo quello numero n
    if  (n == 0){
        msg = messages.last.get(sender);
    else {
        while (n != 0){     // scorro la lista fino a quando non arrivo a n
            msg = messages.get(sender);     // restituisco il valore del messaggio
            n--;
        }
    }
    asend(ACK, sender);
}
```

Soluzione del prof

```
void snsend(msgtype msg, pid_t dest){
 asend(<getpid(), msg>, dest);
 if(arecv(dest) == ACK)
  print("ERROR");
}
msgtype snrecv(pid_t sender, int n){
 while((<realsender, msg> = data.get(sender,n)) == NULL){
  <realsender, msg> = arecv(ANY);
  data.add(realsender, msg);
 }
 asend(ACK, realsender)
```

```
 return msg;
}
data.add(sender, msg)
 data.l.add((sender.msg))

data.get(sender, n)
 if(n==0):
  out=None;
  for(s,msg)in data.l:
   if(match(sender,s)) out = (sender,s)
  return out;

 else:
  return None
 for(s, msg) in data.l:
  if(match(sender,s):
   n--;
  if n == 0:
   return (s,msg)
 return
```

NOTE

- ha un meccanismo [cosa ho a disposizione] asincrono
- ha una politica [cosa devo implementare] sincrona

Esercizio g1

Considerare i seguenti processi gestiti mediante uno scheduler preemptive a priorità statica su una macchina biprocessore SMP:

```
P1: cpu 4 ms; I-O 4 ms; cpu 2 ms
P2: cpu 2 ms; I-O 4 ms; cpu 5 ms
P3: cpu 5 ms; I-O 3 ms; cpu 3 ms
P4: cpu 10 ms; I-O 1 ms
```

l'Input-Output avviene su un'unica unità. Per il processo P1 ha priorità minima seguito da P2, P3 e P4 in sequenza crescente, le richieste di I/O sono gestite in ordine FIFO. Calcolare il tempo necessario a completare l'esecuzione dei 4 processi. Indicare con chiarezza i punti dello schedule nei quali avviene la preemption.

NOTE

Con *preemption* si intende la capacità di un SO di interrompere un processo in esecuzione e di passare alla CPU un altro processo. #### *Perchè?* - garantire

la corretta esecuzione di processi ad alta priorità - gestire la concorrenza tra processi - garantire la risposta del sistema per processi con utilizzo di troppe risosre.

Metodi di implementazione

1. con un timer che interrompe periodicamente l'esecuzione di un processo [Interval Timer]
2. interruzio generata da un evento esterno

Punti da seguire per risolvere l'esercizio:

1. Crea una tabella che mostri i tempi di esecuzione per ogni processo, indicando le fasi di CPU e I/O.
2. Assegna una priorità a ogni processo in base all'ordine indicato nell'esercizio.
3. Utilizza un algoritmo di schedulazione a priorità statica per determinare l'ordine di esecuzione dei processi.
4. Utilizza una coda FIFO per gestire le richieste di I/O.
5. Simula l'esecuzione dei processi, tenendo conto delle loro priorità e delle richieste di I/O.
6. Calcola il tempo totale necessario per completare l'esecuzione dei quattro processi.
7. Indica chiaramente nello schedule i punti in cui avviene la preemption, ovvero i punti in cui un processo con priorità più alta interrompe l'esecuzione di un processo con priorità più bassa.

Il tempo necessario per completare l'esecuzione dei processi è di 17ms. [Vedi tabella sul foglio]

Notiamo che la *preemption* si verifica ai tempi: - 4ms che alla fine dell'esecuzione di P1 parte P4 - 2ms che alla fine dell'esecuzione di P2 parte P3

Correzione 9/2/23

Il tempo necessario è di 19ms e c'è preemption solo al tempo 8ms con P1

Esercizio g2

Rispondere alle seguenti domande (motivando opportunamente le risposte):

a) Perché viene usata la paginazione per implementare la memoria virtuale?

b) L'algoritmo del banchiere dato uno stato di allocazione delle risorse restituisce un valore binario: safe o non safe. In quali casi il sistema operativo esegue l'algoritmo del banchiere? Cosa succede se il risultato è safe e cosa se il risultato è non-safe?

c) Fornire esempi di file system con allocazione contigua, e spiegare perché sarebbe inefficiente usare altri metodi di allocazione nei casi d'uso tipici di questi file system.

d) perché l'invenzione degli interrupt ha reso i sistemi operativi più efficienti?

Risposte

a. perchè la paginazione permette al sistema operativo di gestire efficacemente la memoria disponibile, di simulare l'esistenza di più memoria di quella effettivamente presente, e di utilizzare tecniche avanzate per aumentare la sicurezza e l'efficienza del sistema.

Infatti la paginazione permette di dividere la memoria fisica in più pagine di dimensioni fisse [frame], le quali vengono suddivise e i processi utilizzano solo quelle necessarie per l'esecuzione, rendendo la memoria più efficiente.

b. L'algoritmo del banchiere viene eseuito dal sistema operativo ogni volta che non si può soddisfare immediatamente una richiesta di assegnazione. Quando il risultato è "safe" si assegnano le risorse se è garantito che non ci sia la possibilità di deadlock e viene aggiornato lo stato di allocazione. Quando il risultato è "unsafe" c'è rischio di deadlock, dunque non si assegnano le risorse fino a quando non vengono rilasciate risorse sufficienti e il processo resta in attesa.

c. gli interrupt permettono un utilizzo più efficente delle risorse ed evitano il busy waiting da parte del processore per controllare periodicamente se un dispositivo ha terminato il suo compito.