- Segui decalogo (Aggiunto sotto)

Semafori
la P ha all'interno una cosa: coda, unica cosa da verificare è mettere semaforo e che la FIFO non si scassi

processo D
-> mutex.P
<codice critico>
-> mutex.V

Se il processo deve aspettare si blocca ma se rilascia la mutex un altro processo potrebbe entrare e non aspettare P

2 soluzioni
- passaggio testimone non rilascio mutex e vado avanti ad eseguire fino a che mi risveglio IF-THEN-ELSE
- creo una coda di processi FIFO
queue.enqueue(pid)
ram = queue.dequeue(pid)
sem.V(ram);

rilascio il mutex nel ramo else

monitor
metto le condition da controllare posso metterne anche per ogni funzione che aspetta NON all'inizio // semaforo deadlock
conviene differenziare per non svegliare tutto (questo non lo vogliamo)

> condition ok to attraverso
> condition ok to leave

message passing
Attieniti a cosa hai e cosa devi sviluppare
sincrono: send <-> rec
asincrono: send -> rec (-> aspetto che riceva)
Totalmente asincrono send rec -> può essere null se non riceve mess
ricorda che hai sempre le primitive
- ack è l'acknowledge in sostanza lo usi per non bloccare il send per passaggio

asincrono -> sincrono

COMANDI	MESSAGE PASSING	!!!!!!!!
sincrono:	ssend(msg_t m, pid_t dest) sreceive(pid_t snd)	= può essere ANY SOLO RECEIVE
asincrono:	asend(msg_t m, pid_t dest) arecv(pid_t snd)	
Totalmente asinc.	asend ne - receive (snd)	

FUNZIONAMENTO ACK

```
void ssend(...){
  asend(<getpid(), msg>, dst);
  msg_t ack = arecv(dst)
(NON void) msg_t srecv(...) {
  msg_t m;
  pid_t s;
  <s, m> = arecv(snd);
  asend(ack, s);
}
```

così la blocca e sblocca il send con ack recv è già bloccante

Da sincrono a asincrono
mi serve un SERVER che riceva il mes. e faccia da tramite

SSEND -> SERVER <-> arecv, arecv, arecv
risposta
CODA MESSAGGI

Il server va costruito tramite una coda e count che tenga salvato quanti proc. in attesa

MP asincrono dato quello sincrono (senza gestione ANY)

```
/* p is the calling process */
void asend(msg_t m, pid_t dst) {
  ssend("SND(m,getpid(),dst)", server);
}
msg_t areceive(pid_t snd) {
  ssend("RCV(getpid(),snd)", server);
  msg_t m = sreceive(server);
  return m;
}
process server {
  /* One element x process pair */
  int[][] waiting;
  Queue[][] queue;
  while (true) {
    handleMessage();
  }
}
```

```
void handleMessage() {
  msg = sreceive(0);
  if (msg == <SND(m,p,q)>) {
    if (waiting[p,q]>0) {
      ssend(m, q);
      waiting[p,q]--;
    } else {
      queue[p,q].add(m);
    }
  } else if (msg == <RCV(q,p)>) {
    if (queue[p,q].isEmpty()) {
      waiting[p,q]++;
    } else {
      m = queue[p,q].remove();
      ssend(m, q);
    }
  }
}
```

NB: gli indici sono invertiti ma è corretto

DECALOGO // REGOLE

Semafori

- Le operazioni sui semafori (in notazione ad oggetti) sono S.P() e S.V(). Non hanno alcun parametro ne' alcun valore di ritorno. Se invece volete usare la notazione procedurale le operazioni sono P(S) e V(S), unico parametro e' il semaforo e nessun valore di ritorno. in alcuni es. conviene non farlo scrittamente
- Quando si usano semafori tutti gli accessi ai dati condivisi devono avvenire in mutua esclusione e non si deve scassare la FIFO, usa uno dei due approcci SOPRA
- Le code dei semafori e il valore dei semafori sono strutture e

variabili private gestite dall'implementazione del paradigma e non sono accessibili dichiara le code fuori dalla fun.
- Se non altrimenti specificato i semafori sono generali e fair (FIFO)
- I semafori *Devono* avere un valore iniziale. mutex = 1, altri a 0
- E' sicuramente errato un programma che usa un semaforo solo con operazioni P e nessuna V o viceversa.

Monitor

- Le operazioni definite sulle variabili di condizione sono c.wait() e c.signal(). Non hanno parametri, ne' valore di ritorno.
- Bloccarsi (Busy Wait) all'interno di un monitor è deadlock = non fare while che aspettano una condizione
- Le code delle variabili di condizione, lo urgent stack sono strutture private gestite dall'implementazione del paradigma e non sono accessibili
- La routine di inizializzazione di un monitor non può contenere wait e signal
- se non altrimenti specificato la politica delle variabili di condizione è signal-urgent
- E' sicuramente errato un programma che su una variabile condizione effettua solo wait e mai signal. (viceversa la condizione e' inutile).
- E' sicuramente errato un programma che ha una wait come prima istruzione di ogni procedure entry DEADLOCK

Message Passing

- Esistono tre paradigmi: sincrono, asincrono, completamente asincrono. Si usa quello indicato dall'esercizio.
- Se il testo indica "message passing asincrono" allora NON è "completamente asincrono"
- E' sicuramente errato un programma che usa solo send e nessuna receive o che usa solo receive e nessuna send
- Se non e' indicato diversamente i servizi di message passing sono FIFO
- Message Passing e' un paradigma a memoria privata: non possono esistere variabili condivise fra i processi.
- Processi server vanno utilizzati se espressamente richiesti o se non è possibile implementare i servizi senza.

In generale

- Chiamate di fantasia quali block, wakeup, restart non devono mai comparire in soluzioni di esercizi ; wait(), signal()
- Semafori, monitor, message passing sono paradigmi indipendenti. Gli esercizi indicano quale usare. Non si possono *mischiare* primitive di paradigmi diversi. non fare monitor con i mutex
- E' sicuramente errato un programma che usa variabili prima di assegnare loro un valore iniziale
- Soluzioni con semafori, monitor o message passing non devono contenere busy-wait (sono stati creati per evitarlo)
- E' possibile definire strutture dati senza implementarle a patto che:
- non contengano primitive concorrenti (sincronizzazione o comunicazione)
- non siano miracolose, devono poter essere implementabili quindi i parametri devono contenere tutte le informazioni necessarie per fornire i servizi richiesti.