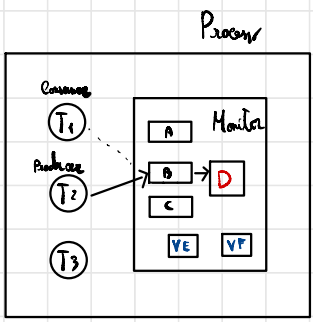
Con i thread a livello utente riusciamo ad implementare un'attesta passiva, nella quale il SO addormenta il processo e non fa busy waiting sprecando cicli di CPU. Questo scenario è da preferire all'attesa attiva. Con i thread a livello utente e l'istruzione TSL non si invocano chiamate di sistema e per tanto risulta più veloce ed efficiente rispetto a strumenti come sleep e wakeup che necessitano di system call ma non fa busy waiting (livello kernel).
Continuando a parlare di soluzioni per garantire mutua esclusione introduciamo il concetto di FUTEX

Fast user space mutex (solo LINUX) — pag 154

Implementazione "nascosta" di un meccanismo di mutex ibrida tra il modello a livello utente e kernel che cerca di tenere i vantaggi . Un futex è composto da due componenti , una in spazio kernel e una in user space (libreria).
Si usa la TSL (XCHG) per manipolare la variabile di lock su una struttura dati condivisa. La seconda parte a livello kernel blocca gli altri thread concorrenti utilizzando system call (solo se necessario). L'idea è che in uno scenario di scarsa contesa (percentuale di accessi concorrenti bassa) tutto viene gestito in modalità utente e tramite l'uso di TSL che valorizzano la variabile di LOCK e molto velocemente si avrà accesso alla struttura dati senza chiedere al kernel, eccetto quelle situazioni in cui si necessita di bloccare altri processi tramite l'utilizzo di system call (evento sporadico).
Questo strumento si adatta dinamicamente al contesto utilizzando uno strumento leggero (liberie user space) e invoca il kernel solo se necessario.

Monitor

I semafori sono delicati da utilizzare, ad esempio invertendo istruzioni si inseriscono chiamate bloccanti indesiderate nella zona critica. Vengono quindi introdotti uno strumento differente dai semafori che offrono dei vantaggi perchè legati a linguaggi di concezione più recente che permettono di sfruttare librerie più ad alto livello : i monitor.
Un monitor risolve gli stessi problemi del semaforo ma con metodologie differenti.
Un monitor è un'astrazione composta da metodi, variabili e strutture dati, supportata dai compilatori dei linguaggi di programmazione e non dal sistema operativo, che implementano quanto richiesto a livello di codice (il sistema operativo non offre monitor) utilizzando le strutture già viste fino ad ora. Un monitor garantisce la **mutua esclusione** nelle istruzioni eseguite all'interno del monitor stesso : qualunque thread può invocare i metodi presenti nel monitor ma una volta eseguito da un thread, qualunque altro thread che vorrà accedere alle stesse funzioni verrà bloccato.



Si inserisce una struttura dati all'interno del monitor e i thread del processo avranno accesso alla struttura condivisa solo attraverso i metodi presenti nel monitor, la cui esecuzione è garantita in modo esclusivo dal monitor stesso : in un istante di tempo in un monitor può essere attivo un solo processo. Lo strumento monitor è più comodo perchè il programmatore non scrive righe di codice per garantire mutua esclusione alla struttura dati, si passa solo dai metodi del monitor e si occuperà il compilatore ad implementare questo meccanismo attraverso semafori, mutex, futex ecc.. . Solo i linguaggi più recenti offrono questi strumenti (monitor), per esempio Java si mentre C , C#, Pascal non implementano questi strumenti.
Il problema della mutua esclusione viene risolto dai monitor ma per risolvere il problema di **sincronizzazione** vengono utilizzati degli strumenti di supporto dal Monitor.
Nel problema produttore-consumatore è facile che dei thread che fungono da producer e consumer, sfruttino delle procedure dei monitor che si scontrino con le problematiche di buffer pieno o vuoto. La soluzione risiede nell'utilizzo di variabili di condizione insieme a due operazioni su di esse : wait e signal. Quando il monitor si rende conto che è in una situazione di blocco, per esempio a causa dell'inserimento di un item in un buffer pieno, effettua una wait su una variabile full. Questa azione fa si che il chiamante si blocchi e consenta l'accesso ad un altro processo cui precedentemente era stato vietato l'accesso al monitor. Le variabili di condizione , a differenza dei semafori, non hanno un valore.

Non serve contare (non avere un valore sulle variabili condivise) in quanto è garantita la mutua esclusione che rende la regione di codice atomica e utilizzabile da un solo thread , per tanto non serve tenere conto del numero di sveglie perse perchè non si pone il problema di interruzione dell'esecuzione di un thread . Un thread nella sua zona critica può addormentarsi o svegliare altri thread ma non verrà mai interrotto da altri thread. Si necessita di tenere conto dei processi bloccati e viene implementato tutto con l'uso di semafori che tengono conto di una coda dei processi bloccati.
Il meccanismo di risveglio e di addormentamento può creare problemi : ad esempio due thread in esecuzione su metodi differenti fanno accesso alla stessa variabile condivisa da un certo punto in poi del codice dei metodi e quindi viene violato il meccanismo del monitor perchè due thread si ritrovano ad avere accesso alla stessa variabile condivisa in quanto magari un thread fa da producer ed uno da consumer . La soluzione sta nel ridefinire l'implementazione delle wait e signal.

Monitor Hoare

Si usa un meccanismo che lega fortemente signal & wait. Quando un thread invoca la signal su un altro thread svegliandolo, allora esso si dovrà addormentare in modo da non rimanere anch'esso attivo e capace di concorrere all'accesso della variabile condivisa. Questo metodo funziona ma si complica il lavoro del programmatore in quanto la semantica di un metodo interferisce con quella di un altro.

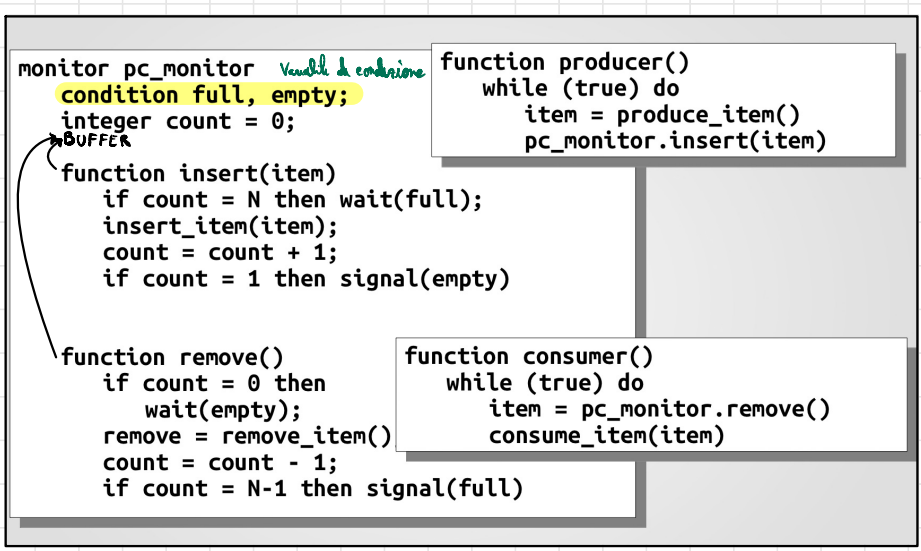
Monitor Mesa

Si utilizza un'implementazione migliore che non complica il lavoro del programmatore legando signal & continue.
Il risveglio del thread non viene applicato subito in quanto questo crea il problema di violazione alla mutua esclusione ma viene eseguita la signal quando il processo chiamante termina la propria esecuzione, in modo da non interlacciare le operazioni dei metodi.

Compromesso

Si vincola il metodo ad eseguire la signal sempre come ultima operazione, ovvero la signal corrisponde alla return in modo che chi la invoca non abbia codice da eseguire dopo la signal.

Questo approccio si può applicare solo sui thread e difficilmente si può implementare su modello multithread in quanto il funzionamento del monitor è una struttura astratta che vive all'interno dello spazio di indirizzamento condiviso e non è possibile implementarlo tra processi differenti in quanto non necessariamente i due processi utilizzano lo stesso linguaggio di programmazione e non ha senso utilizzare un monitor su due differenti linguaggi di programmazione.



Scambio di messaggi tra processi — pag 165

Questo è un modello ad alto livello, facile da usare per il programmatore, che si basa sul fatto che i processi che vogliono comunicare si trovano su macchine distinte e che , per tanto, non presentano un segmento di memoria condiviso.
Si parla ancora di interprocess comunication ma usando strumenti molto diversi da quelli visti finora . Si parla di scambio e sincronizzazione di messaggi tra macchine differenti e la necessità è quella di avere un modo per far comunicare processi sia sullo stesso server , sia su server differenti all'interno di un cluster (macchine separate). Un processo incapsula l'item prodotto all'interno di un messaggio e lo si invia ad un altro processo.

Vengono sfruttate delle primitive ad alto livello :
- send(destinazione, messaggio)
- receive(sorgente, messaggio)

Il ciclo di vita del consumatore è legato alla receive() e per tanto la funzione diventa una chiamata bloccante nei confronti del processo ricevente (consumer), ma anche la send diventa una chiamata bloccante dato che il buffer presenta una dimensione finita e se il consumer non consuma abbastanza velocemente gli item allora il producer viene anch'esso bloccato. Si necessita quindi di una sincronizzazione tra produttore e consumatore per lo scambio dati.

```
function producer()
    while (true) do
        item = produce_item()
        build_msg(m,item)
        send(consumer, msg)
```

```
function consumer()
    while (true) do
        receive(producer, msg)
        item=extract_msg(msg)
        consum_item(item)
```

Modello a mail-box

Il sistema operativo ha un buffer interno che usa per memorizzare gli item in uscita che non riescono ad essere consumati dal consumer in tempo. Quando si hanno più processi produttori e consumatori si utilizza un buffer "mail-box" al quale vengono inviati gli item e dal quale vengono prelevati, dato che si necessita di inviare degli item a processi che magari sono impegnati in quel momento e non riescono a consumare gli item. Il mail box si occupa di smistare gli item che contiene tra i vari consumer. Si copia l'item da inviare all'interno del mail-box attraverso una sistem call e se ne riusa un'altra per ricopiarlo dalla mail-box alla memoria del consumatore. Qualunque operazione implementata sulla mail-box è implementata con chiamate di sistema in quanto il mail box è implementato nel kernel ed in oltre si hanno diverse copie indesiderate degli item che introducono un importante overhead. Questo meccanismo è implementato solo su applicativi multiprocesso e non ha senso utilizzarlo in contesti multithread perchè questi condividono un segmento di memoria.

Problema dei cinque filosofi

Astrazione di un problema reale : ho più entità (nel nostro caso 5) che hanno bisogno di un certo numero di risorse da usare in modo esclusivo e per operare hanno bisogno di acquisirle. Operare in modo esclusivo implica che si prende possesso di una risorsa e poi rilasciarla. La regola dice che un filosofo per mangiare ha bisogno delle due forchette che stanno ai lati del piatto e una volta averle acquisite entrambe può mangiare, ma le forchette sono risorse condivise, in quanto se vengono usate da un filosofo non possono essere usate da altri. Si cerca una soluzione che permetta di massimizzare la capacità di mangiare dei filosofi (con 5 filosofi al massimo due se non sono adiacenti) e al contempo di prevedere un blocco di un processo se non è in grado di consumare.

- **soluzione 1**:

```
int N=5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```

Questa soluzione presenta un problema in quanto se tutti decidono di prendere la forchetta alla destra e mangiare nessuno riuscirà ad acquisire un'altra forchetta e quindi tutti i filosofi si bloccano.

Soluzione 2 :

Si tenta di acquisire la seconda forchetta e se non è possibile acquisirla allora si lascia anche la prima e si riprova dopo. La soluzione non funziona se tutti determinano una pausa fissa e quindi ricontrolleranno tutti allo stesso momento senza successo.

Soluzione 3 :

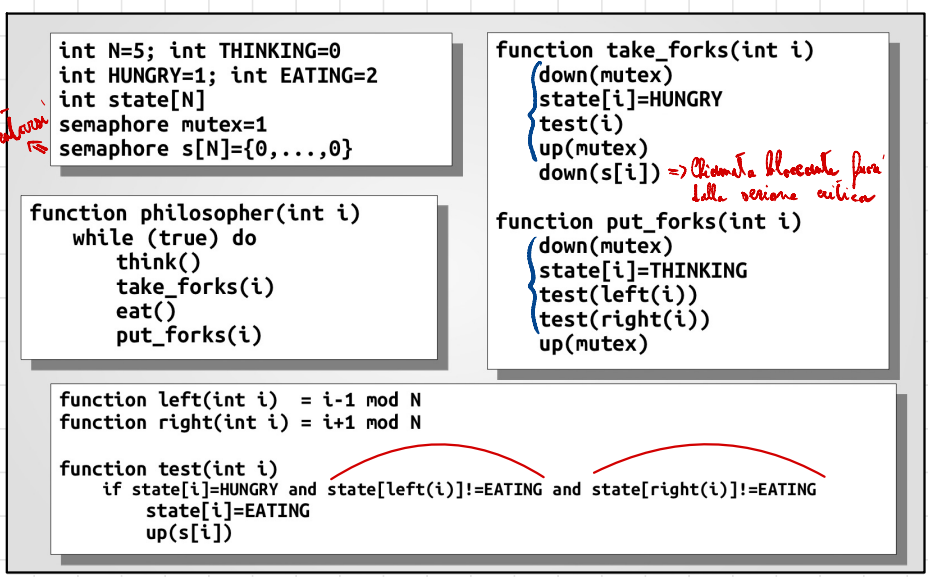
Si randomizza il tempo di attesa per rilasciare la forchetta e questo permette con buona probabilità di mangiare per più di un filosofo.

Soluzione 4 :

Si sfrutta l'utilizzo di un mutex per acquisire il lock sull'intero tavolo , obbligando un filosofo ad acquisire il lock per mangiare. Il problema di questa soluzione grossolana è che solo un filosofo riuscirà a mangiare.

Soluzione 5 (basata sui semafori) :

Non si bloccano l'intero tavolo ma si usano i semafori su ogni singolo filosofo per autoaddormentarsi. Viene stabilito un mutex sulla variabile condivisa "state[ ]" che è un vettore che indica lo stato dei filosofi (pensa, affamato, mangia).



La test serve a vedere se le forchette sono entrambe disponibili e la down(s[i]) fa effettuata al di fuori della test in quanto la test si trova dentro la sezione critica tra down(mutex) e up(mutex) ed è bloccante, quindi la sposto al di fuori della sezione critica, in modo che verrà eseguita solo dopo aver rilasciato il mutex. Le condizioni verificate dalla test sono una serie di predicati in and logico che controllano se i filosofi vicini sono in fase di eating, in quel caso sono in possesso delle forchette che mi servono e quindi non posso procedere.
L'operazione put_fork() serve per mettere il proprio stato a thinking ed esegue due test per svegliare i filosofi alla destra e alla sinistra, qualora sono stai bloccati mentre erano in uno stato di hungry.

Soluzione 6 (basata sui monitor) :

Si sfrutta la mutua esclusione implicita e le primitive di sleep e wakeup. Vengono utilizzate tante variabili condivise quanti sono i filosofi.

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2

monitor dp_monitor
    int state[N]
    condition self[N]

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING
            wait(self[i])

    function put_forks(int i)
        state[i] = THINKING;
        test(left(i));
        test(right(i));

    function test(int i)
        if ( state[left(i)] != EATING and state[i] = HUNGRY
        and state[right(i)] != EATING )
            state[i] = EATING
            signal(self[i])

function philosopher(int i)
    while (true) do
        think()
        dp_monitor.take_forks(i)
        eat()
        dp_monitor.put_forks(i)
```

Lanciare signal e wait su variabili di condizione (che non hanno un valore) non hanno effetto e l'unico effetto è quello del cambiamento del vettore state. L'unico cambiamento rispetto alla soluzione con semafori sta nella test : si controlla (in sezione critica) il vettore state per verificare se i filosofi adiacenti stiano mangiando e quindi impegnando la forchetta. Questa soluzione introduce una semplificazione dovuta al fatto che la mutua esclusione è garantita a priori dal meccanismo implicito dei monitor.