LEZIONE 7

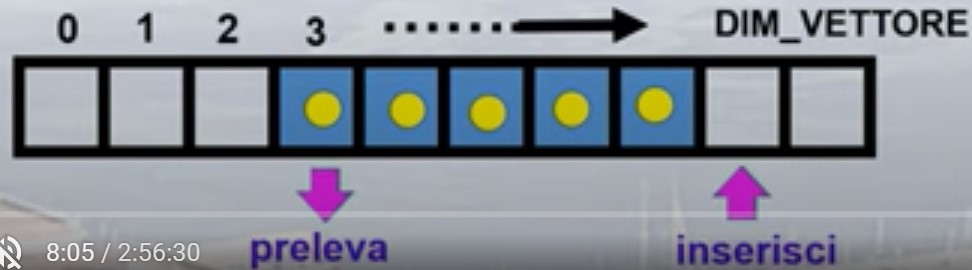
**Introduzione**

* Nei SO **multi-programmati** diversi processi o thread sono in **esecuzione asincrona** e possono condividere dati.
* I processi cooperanti possono condividere:

1. Uno spazio logico di indirizzi, cioè codice e dati (Thread)
2. Oppure solo dati

* **L’accesso concorrente a dati condivisi, se non si adottano politiche di sincronizzazione, può causare incoerenza degli stessi dati.**
* Mantenere la coerenza dei dati richiede meccanismi atti ad assicurare un’ordinata esecuzione dei processi cooperanti.

**Esempio: Memoria limitata- soluzione con memoria condivisa (es. lezione 4)**

Si suppone di usare una zona di memoria condivisa e di usare un vettore circolare di grandezza DIM\_VETTORE e che fa uso di 2 indici per accedere al vettore. -Con l’indice inserisci si punta alla prima posizione occupata del vettore. -Con l’indice preleva si punta alla prima posizione occupata del vettore. Al momento iniziale si pone a zero sia inserisci che preleva. Da tale definizione si ricava che: **-** **Se inserisci == preleva** ALLORA il vettore è vuoto. **-** **Se ((inserisci + 1) % DIM\_VETTORE) == preleva** ALLORA il vettore è pieno. **-Max elementi nel vettore = DIM\_VETTORE - 1**

**Soluzione memoria condivisa**  La soluzione del problema dei produttori e dei consumatori con memoria limitata consente la presenza contemporanea nel vettore di n-1 elementi.

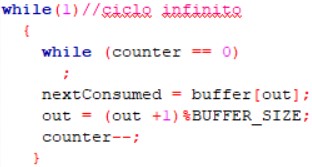
Si supponga di voler modificare il codice del produttore-consumatore aggiungendo una variabile intera **counter (contatore)** inizializzata a 0 e che si incrementa ogniqualvolta si preleva un elemento dal vettore.

**MEMORIA LIMITATA Dati condivisi: #define BUFFER\_SIZE 10** //supponiamo di avere un buffer di grandezza 10 **typedef struct {**  //poi una struttura nella quale la cosa importante è che definiamo un item  **… } item; item buffer [BUFFER\_SIZE];** //questo elemento è definito da un buffer di grandezza BUFFER\_SIZE//due puntatori in e out cioè di prendi e preleva e un contatore  **int in = 0; int out = 0; int counter = 0;**

**PROCESSO PRODUTTORE**

**Immagine che contiene screenshot

Descrizione generata automaticamente Spiegazione** Ci sta un while infinito al cui interno c’è un altro while fatto così: finché il counter == buffer\_size, ovvero il vettore è pieno, il produttore non può produrre; quindi l’unica cosa che può fare è attendere. Questo while non ha il corpo all’interno dato che è solo una condizione. Quando questa condizione non è vera il produttore produce; quindi con un indice puntatore al vettore buffer(buffer[in]=nextProduced) inserisce il prodotto appena creato nella prima locazione utile; dopo di che deve aggiornare il puntatore in, perché se arriava una nuova produzione deve sapere dove andarla a inserire (%buffer mi serve per implementare la circolarità del vettore, cioè se si arriva in fondo al vettore ricomincia il giro); in ultimo si incrementa il contatore.

**PROCESSO CONSUMATORE **

**Spiegazione** Ci sta un while(1) all’interno del quale si trova un altro while (counter ==0) fatto così: se non c’è niente nel vettore aspetta. Quando questa condizione non è verificata (ovvero ci sta almeno un elemento) lo si va a prendere e lo si mette in nextConsumed (nextConsumed = buffer[out] (buffer[out] indica la prima locazione occupata dall’elemento da prelevare)); in ultimo decremento il contatore.

**Immagine che contiene screenshot

Descrizione generata automaticamentePROBLEMATICHE**

Questi processi sono univoci. La variabile counter è condivisa, cioè P1, P2, P3 sono in attesa di usare la stessa risorsa. Il problema nasce: Abbiamo 2 istruzioni **counter++, counter- -** che devono essere eseguite in modo **ATOMICO, cioè che quella istruzione non venga interrotta nella sua esecuzione.**

L’istruzione “**counter++”** si può codificare il linguaggio macchina come:

**register1=counter register1=register1 +1 counter=register1**

L’istruzione “**counter- -”** si può essere realizzata come:

**register2=counter register2=register2 -1 counter=register2**

Se vogliamo aggiornare il contatore, dato che questa è una variabile che per esempio di trova in memoria. Dato che se un calcolatore deve incrementare una variabile, la deve prendere e portare all’interno della CPU, nel caso specifico in un registro della CPU (register1 in questo caso); quindi prende la variabile la copia ne registro1 e si incrementa, dopo di che si prende questa variabile e la si riporta indietro nella memoria. Questa cosa vale anche per counter - -

Il problema nasce qui. Nel caso specifico produttore-consumatore, bisogna subito ipotizzare che il produttore e il consumatore cercheranno di aggiornare il registro in maniera concorrente. Le istruzioni vengono ad intercalarsi, ovvero viene fatto un poco una e un poco l’altra, ciò può creare delle situazioni per la quale sfugge il controllo.

**Esempio** Si ipotizza che il valore della variabile counter sia inizialmente 5. Una sequenza è: **producer: register1=counter (register1 =5) producer: register1=register1+1 (register1 =6) consumer: register2=counter (register1 =5) consumer: register2=register2-1 (register1 =4) producer: counter=register1 (counter =6) consumer: counter=register2 (counter =4)** **Spiegazione** Producer prende la variabile counter e la deposita nel registro, dopo di che il producer incrementa il register1. Il contatore prende la variabile counter e la mette in un altro registro (register2), poi lo decrementa. A questo punto sia il producer che il consumer cercheranno di aggiornare il contatore; in questo caso prima lo fa il producer e poi il consumer. Il risultato del counter dovrebbe essere in tutti i casi 5 (sia se si produce prima o se si consuma prima); ma per questo intercalare il risultato che ci ritroviamo è errato (cioè 4 oppure 6) e secondo delle logiche di tempistiche avrò una **RACE CONDITION** (l’ultimo ad eseguire è il “vincitore”), cioè l’ultima istruzione annulla tutte le altre.

**Race Condition:** Situazione in cui più processi accedono e modificano gli stessi dati in modo concorrente, e i risultati dipendono dall’ordine degli accessi. **-Per prevenire questa situazione, i processi concorrenti devono essere sincronizzati.**

Problema della sezione critica

* Si considerino n processi concorrenti.
* Ciascun processo ha un segmento di codice chiamato **sezione critica ( critical section)** nel quale il processo può modificare variabili comuni.

**Sezione critica:** regione nella quale quando un processo A si trova all’interno bisogna essere certi che nessun altro processo possa avere accesso alle risorse che il processo A sta utilizzando in quell’area.

* Problema: quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica.

Soluzione al problema della sezione critica Nella gestione della sezione critica, le garanzie principali da fornire sono tre:

1. **MUTUA ESCLUSIONE:** Se il processo Pì è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **PROGRESSO:** se nessun processo è in esecuzione nella sua sezione critica, solo i processi che desiderano entrare nella propria sezione critica possono partecipare alla decisione su chi sarà il prossimo ad entrare in sezione critica e tale decisione non può essere ritardata indefinitamente.
3. **ATTESA LIMITATA:** Se un processo ha già richiesto l’ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

**-** Si suppone che ogni processo sia eseguito a una velocità diversa da 0

**-** Non si può fare alcuna ipotesi sulla velocità relativa degli n processi

Primi tentativi di risolvere il problema

* Solo 2 processi P0 e P1
* Struttura generale del processo Pi (l’altro processo è Pj)

do {

Sezione critica

Sezione non critica

Sezione d’ingresso

Sezione d’uscita

}while (1);

* I processi possono condividere alcune variabili comuni per sincronizzare le loro azioni

Immagine che contiene screenshot

Descrizione generata automaticamenteAlgoritmo 1 Soddisfa la mutua esclusione ma non il requisito di progresso; questo perché nessuno garantisce che dopo Pi ci sia Pj, in quanto potrebbero arrivare altri processi o ritornare Pi in sezione critica.

Algoritmo 2

* **VARIABILI CONDIVISE**
* Inizialmente pronto[0]=pronto[1]=false.
* Pronto[i] = true => Pi pronto a entrare nella sua sezione critica

**Processo Pj do{**

**pronto[j] := true; while(pronto[i]);**



**pronto[j]=false;**

**sezione non critica**

**}while(1);**

Sezione critica

**Processo Pi do{**

**pronto[i] := true; while(pronto[j]);**

**pronto[i]=false;**

**sezione non critica**

**}while(1);**

* La stretta alternanza dell’algoritmo 1 è risolta con una variabile booleana pronto.
* Se while(pronto[j]) cioè se anche j è pronto, i non fa nulla; se però questa condizione non è verificata i può andare in sezione critica.
* Anche qui non è soddisfatto il requisito di progresso perché essendo concorrenti se entrambi (i e j) hanno pronto = true alla seconda istruzione (while) si bloccano entrambe.

Algoritmo 3

* Combina le variabili condivise segli algoritmi 1 e 2

**Processo Pj do{**

**pronto[j] = true; turno=i; while(pronto[i]&&turno=i);**

**pronto[i]=false;**

**sezione non critica**

**}while(1);**

**Processo Pi do{**

**pronto[i] = true; turno=j; while(pronto[j]&&turno=j);**

**pronto[i]=false;**

**sezione non critica**

**}while(1);**

Sezione critica

Sezione critica

* Soddisfa tutti i requisiti; risolve il problema della sezione critica; ciò perché oltre alla variabile pronto ci sta una turnazione; di conseguenza se sono entrambe pronte, in base a qualche criterio si sceglie chi entra prima in sezione critica.
* **Questo vale solo per due processi.**

Algoritmo del Fornaio

* **Sezione critica per n processi**
* Prima di entrare nella sua sezione critica, il processo riceve un numero. Chi detiene il numero più basso entra nella sezione critica.
* Se i processi Pi e Pj ricevono lo stesso numero se i < j allora Pi è servito per primo; altrimenti viene servito prima Pj. Questo può capitare in quanto, essendo Pi e Pj concorrenti potrebbe capitare che prendano lo stesso numero (ciò succede perché ogni processo viene battezzato dalla propria macchina con il proprio PID (process identify)).
* Lo schema di numerazione genera sempre numeri in ordine crescente; (es. 1,2,3,3,3,4,5)

**Funzionamento**

**do {**

**scelta[i]= true; numero[i]= max(numero[0],….,numero[n-1])+1; scelta[i]= false;**

**for (j = 0; j<n; j++){ while(scelta[j]); while((numero[j] !=0 ) && ( (numero[j] < numero[i]) || ((numero[j] == numero[i]) && (i< j)) ));**

**}**

**Sezione critica**

**numero [i]=0;**

Sezione non critica

**} while(1)**

**Spiegazione** I primi 2 blocchi rappresentano una condizione d’ingresso, poi ci sta la sezione critica, dopo una sezione di uscita(numero[i]=0) e una sezione non critica. Il primo blocco dice: se il processo i-esimo vuole entrare in sezione critica deve prendere un numero in questo modo: si fa il massimo dei valori che sono già in giro e poi si fa +1; dopo di che si mette scelta =false perché la scelta già è stata fatta. Nel secondo blocco ci sta un ciclo for all’interno del quale ci sta un while(scelta[j]), se è vera non fa nulla, perché deve considerare solo chi ha già scelto non chi sta scegliendo. Poi ci sta un ulteriore while all’interno del quale ci sono altre condizioni che se sono verificate permettono di entrare in sezione critica.

Architetture di sincronizzazione

Controlla e modifica il contenuto di una parola di memoria in modo atomico.

**Funzione TestAndSet**

* TestAndSet è una funzione booleana che viene eseguita in modo atomico (come tutte le altre funzioni che servono per le architetture di sincronizzazione).
* In informatica Test significa recuperare il valore di una variabile e Set invece settare/impostare valore a vero.

Boolean **TestAndSet(**boolean&& obiettivo**)** // Essendo una funzione booleana, lavora su un dominio booleano prende una variabile obiettivo (che si trova in una specifica locazione di memoria).  **{ boolean valore = obiettivo**; //preleva il valore della variabile e la posiziona momentaneamente in valore **obiettivo = true;** //poi va ascrivere in quella locazione, mettendo obiettivo a true  **return valore;** //in fine restituisce il valore della variabile valore **}**

Mutua esclusione con TestAndSet

**Dati condivisi:** boolean = false;

* **Processo Pi do{**

**Blocco =false;**

**While(TestAndSet(blocco));**

**Sezione critica**

**Sezione non critica } while(1)**

* A differenza dell’algoritmo del fornaio è più snello **Spiegazione** Il processo i-esimo per entrare in sezione critica deve eseguire un test; cioè se TestAndSet(blocco) restituisce vero, allora il while rimane in loop bloccato; questo fin quando TestAndSet non restituisce false; e quando questo avviene il processo Pi va in sezione critica. Quando Pi finisce la sezione critica, imposta blocco a false così che qualche altro processo bloccato può entrare in sezione critica.

**Funzione SWAP (scambio)**

* L’istruzione swap agisce sul contenuto di due parole di memoria; come per TestAndSet è anch’essa eseguita in modo atomico.

**Spiegazione** partendo da 2 valori che si trovano nelle locazioni booleane a e b, si fa uni scambio, quindi quello che sta in a va in b e viceversa. Siccome la scrittura è distruttiva, si memorizza momentaneamente ciò che sta in a in una variabile temporanea, dopo di che si mette in a il valore di b e la variabile temp di prima, la si mette in b.

Void **swap(**boolean &a, boolean &b**) { boolean temp = a; a=b; b = temp; }**

Mutua esclusione con swap **Dati condivisi (inizializzati a false):** boolean blocco; boolean waiting[n]; //vettore a n posizioni

* **Processo Pi do {**

**chiave =true; while (chiave == true) swap (blocco, chiave);**

**blocco = false;**

**Sezione critica**

.

**Sezione non critica } while (1)**

Semafori

* Strumento di sincronizzazione che non richiede attesa attiva (evita l’attesa attiva)
* Un semaforo S è una variabile intera
* A questa variabile si può accedere solo tramite due operazioni atomiche predefinite:

1. **wait (S): while S <= 0 do no-op;** //finche s<=0 non fare nulla; quando questa condizione è falsa prosegui **S- -;** // e fai s –
2. **signal (S):**  **S++**; //fa un incremento su una stessa variabile

Sezione critica di n processi

**Dati condivisi: semaphore mutex;** //inizialmente mutex = 1

* **Processo Pi: do { wait(mutex);** //sezione d’ingresso che precede sezione criticasezione critica **signal(mutex);** //sezione di sbloccosezione non critica **} while(1);** Dato che il semaforo è 1 il primo processo che fa riesce ad entrare, però fa anche un decremento (con la wait, mettendo semaforo a 0), così che un secondo processo non possa accedere alla sezione critica; dopo di che lo incrementa con signal (quindi lo sblocca).

Relazione di semafori Per la realizzazione di un semaforo si potrebbe ipotizzare di definire un valore e una struttura processo. Quindi in poche parole questo semaforo è composto da una lista di processi che possono entrare in coda per essere eseguiti.

* Definire il semaforo come una struttura del linguaggio C **typedef struct { int valore; struct processo \*L; } semaforo;**
* **Due semplici operazioni:**

1. **Block** sospende il processo che la invoca
2. **Wakeup(P)** riprende l’esecuzione di un processo P bloccato

Realizzazione Le operazioni dei semafori si possono definire ora come segue: **wait (S): S.valore - - ;** //semaforo //decrementa il campo valore del segnale  **if (S.valore < 0) {** //se questo valore è negativo allora fai ciò che sta scritto sottoaggiungi questo processo a **S.L;** //S.L sta per lista del segnale **block;** //effettua un blocco **}**

**Signal (S): S.valore++;** //incrementa il campo valore **if (S.valore <= 0) {** // se questo valore è negativo allora fai ciò che sta scritto sottotogli un processo **P** da **S.L; wakeup (P);**// significa svegliati; quindi riprende l’elaborazione **}**

Semaforo come strumento generale di sincronizzazione Fino ad ora abbiamo visto il semaforo per la realizzazione di zona critica. In realtà si può anche ipotizzare l’uso di questo strumento per un sincronismo molto più generale. Come si sa, non si può conoscere quale processo viene svolto prima; tuttavia ci sono situazioni in cui è necessario saperlo. In questo caso ad esempio:

* Esegue B in Pj solo dopo che A è stato eseguito in Pi
* Usa un **flag inizializzato a 0**
* Immagine che contiene orologio, giallo, segnale, schermo

  Descrizione generata automaticamenteCodice:

**Spiegazione** Se il flag è 0 e lo attraversa prima il primo blocco eseguendo il codice A, si arriva a effettuare signal flag, il quale fa l’incremento della variabile flag che da 0 passa a 1. A questo punto Pj vuole andare in esecuzione e fa la wait su flag, che vale 1(dato che è stata appena incrementata da A); quindi dato che wait vale solo se flag <=0, non è rispettata prosegue e viene decrementata (quindi flag passa a 0), ma va avanti ed esegue anche B.

Quindi avendo impostato flag a 0, A viene fatto prima di B. Nel caso opposto (cioè se parte prima Pj): se parte prima Pj, si entra nella wait, e dato che flag vale 0, si blocca; quindi rimane in loop senza poter andare avanti fin quando qualcuno dall’esterno noncambia il valore di flag (che è quello che dovrebbe fare Pi).

* **Quindi l’esecuzione prima di A e poi di B è forzata**

Stallo e attesa attiva

* **Stallo (deadlock): situazione in cui due o più processi attendono indefinitamente un evento che può essere causato solo da uno dei processi in attesa.**
* Siamo S e Q due semafori inizializzati a 1
* **Immagine che contiene segnale, pensile, persone, camion

  Descrizione generata automaticamente**Se si esegue prima P0, questo fa wait(S) e va avanti facendo anche wait (Q), andando avanti fino alla fine(fa anche le due signal)
* Se si esegue prima P1 arriva in fondo come prima
* Nel caso in cui vadano cin concorrenza invece succede: P0 fa la wait(S), e in contemporanea P1 fa la wait(Q); in questo modo sia S che Q sono state poratte a 0, di conseguenza nessuno dei due processi può andare avanti (rimangono bloccate alle seconde waiit). Ciò causa attesa indefinita dato che entrambe le condizioni di sblocco sono avanti e nessuno dei due processi riesce ad accedervi (aspettando entrambi che sia l’altro a sbloccali).
* **Attesa indefinita (starvation): situazione d’attesa indefinita nella coda di un semaforo.**

Tipi di semafori

* **Semaforo contatore (counting semaphore):** il suo valore intero può variare in un dominio logicamente non limitato. (se si esegue un numero ripetuto di wait, in qualche modo si deve aver messo i processi a riposo, ma comunque aver decrementato la variabile. Quindi questa variabile S conta quanti sono i processi in coda al semaforo)
* **Semaforo binario (binary semaphore):** il suo valore intero può essere soltanto 0 oppure 1; può essere semplice da realizzare.
* È possibile implementare un semaforo contatore S in termini di un semaforo binario. (In qualche modo si potrebbe essere interessati a realizzare un semaforo contatore usando uno o più semafori binari).

Realizzazione di S come semaforo binario

* **Strutture dati:** semaforo\_binario S1, S2; int C;
* **Inizializzazione:** S1=1 S2=0 C=valore iniziale del semaforo contatore S **Spiegazione** Si prende una struttura dati con due semafori binari S1 e S2, e un intero C. S! e S2 sono inizializzati rispettivamente a 1, e C è il valore iniziale del semaforo contatore S. A questo punto si devono definire le funzioni con wait e signal.

**Realizzazione di S**

**Operazione signal wait (S1); C++; if (C <= 0) signal(S2); else signal (S1);**

**Operazione wait wait (S1); C--; if (C < 0) { signal(S1); wait (S2); } signal (S1);**

Problemi tipici di sincronizzazione

* **PROBLEMA DEI PRODUTTORI E DEI CONSUMATORI CON MEMORIA LIMITATA** In un sistema concorrente, i produttori vorrebbero produrre in modo concorrente, così come i consumatori vorrebbero consumare in maniera concorrente; però se i produttori producono in modo concorrente, può succedere un conflitto dato che il buffer è limitato, quindi bisogna controllare se è possibile produrre, sincronizzando il tutto per vedere se ci sta la possibilità di produrre. La stessa cosa vale per i consumatori.

**Dati condivisi**  semaforo piene, vuote, mutex; **inizialmente:** piene = 0, vuote = n, mutex = 1

**PROCESSO PRODUTTORE do { …** produce un elemento in **appena\_prodotto … wait (vuote); wait (mutex); …** inserisci **appena\_prodotto** in **vettore … signal (mutex); signal ( piene); } while (1);**

**PROCESSO CONSUMATORE do { wait (piene); wait (mutex); …** rimuovi un elemento da vettore e inseriscilo in **da\_consumare … signal (mutex); signal (vuote); …** consuma l’elemento contenuto in **da\_consumare … } while (1);**

* **PROBLEMA DEI LETTORI E DEGLI SCRITTORI** Ci sono dei lettori e degli scrittori (processi). Se c’è un solo scrittore e tanti lettori (che possono leggere solo quando lo scrittore ha finito di scrivere ovviamente), non si crea conflitto. Il problema nasce nel momento in cui ci sono più scrittori, in quanto a differenza della lettura che è passiva cioè possono esserci più lettori che leggono in contemporanea, gli scrittori possono scrivere uno alla volta, quindi devono in qualche modo sincronizzarsi tra loro. **Dati condivisi** semaforo mutex, scrittura; **inizialmente:** mutex =1, scrittura = 1, numlettori =0

**PROCESSO SCRITTORE wait (scrittura); …** esegui l’operazione di scrittura **… signal (scrittura);**

**PROCESSO LETTORE wait (mutex); numlettori++; if (numlettori == 1) wait (scrittura); signal (mutex); …** esegui l’operazione di lettura **… wait (mutex); numlettori --; if (numlettori == 0) signal (scrittura); signal (mutex);**

* **PROBLEMA DEI FILOSOFI A CENA** Si considerino cinque filosofi che passano la vita pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso; e la tavola è apparecchiata con un numero di bacchette pari al numero di filosofi. Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i comensi alla sua destra e alla sua sinistra. Un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare. **PROBLEMA: due filosofi vicini non possono mangiare assieme (quindi si devono sincronizzare) Possibili Soluzioni:**

1. **Solo 4 filosofi possono stare contemporaneamente a tavola.** [Così non si avrà mai il blocco totale, anche se qualcuno dovrà aspettare il proprio turno].
2. **Un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (questa operazione va eseguita in una sezione critica).** [Così si rompe la condizione di possesso e attesa]
3. **Si adotta una soluzione asimmetrica:** **un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra; invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra. SOLUZIONE CON STRUTTURA DI SINCRONIZZAZIONE Dati condivisi** semaforo bacchetta [5]; **inizialmente tutti i valori sono 1** //le prime 2 wait controllano se  il filosofo può prendere le due bacchette

**Filosofo i: do { wait (bacchetta[i]) wait (bacchetta[(i+1)%5]) …** mangia **… signal (bacchetta[i]); signal (bacchetta[(i+1)%5]); …** pensa **} while (1);**

Regioni critiche

* Costrutto di sincronizzazione ad alto livello
* Una variabile condivisa **v** di tipo **T** è dichiarata come:

**v: shared T**

* Alla variabile **v** si può accedere solo dall’interno di un’istruzione

**region v when B do S**

dove **B** è un’espressione booleana

* Mentre si esegue l’istruzione **S** nessun altro processo può accedere alla variabile
* **Regioni che si riferiscono alla stessa variabile condivisa si escludono vicendevolmente.**
* Quando un processo vuole accedere alla variabile condivisa **v** nella regione critica, si valuta l’espressione booleana **B,** se risulta vera si esegue l’istruzione **S;** altrimenti il processo rilascia la mutua esclusione ed è ritardato fino a che **B** diventa vera e nessun altro processo si trova nella regione associata a **v.**

Monitor Costrutto di sincronizzazione di alto livello, che consente la condivisione sicura di un tipo di dato astratto fra processi concorrenti.  **monitor nome\_monitor {** dichiarazioni di variabili condivise  **procedure body** P1 **(…) { … } procedure body** P2 **(…) { … } procedure body** Pn **(…) { … } {** codice d’inizializzazione **} }**

* Per consentire ad un processo di attendere all’interno di un monitor, occorre definire una variabile condizionale come **condition x, y;**
* Le uniche operazioni eseguibili su una varaibile **condition** sono **wait** e **signal**

1. L’operazione

**x.wait();**

implica che il processo che la invoca rimanga sospeso fino a che un altro processo non invoca l’operazione

**x.signal ();**

1. L’operazione **x.signal** risveglia esattamente un processo sospeso. Se non esistono processi sospesi, l’operazione **signal** non ha alcun effetto.

Sincronizzazione in Linux Nelle prime versioni di Linux non era gestita la prelazione. La tecnica di sincronizzazione più semplice nel kernel di Linux è l’intero atomico rappresentato mediante il tipo di dato opaco **atomic\_t**

Le operazioni matematiche che usano interi atomici vengono eseguite senza interruzione. Altre tecniche più sofisticate disponibili nel kernel:

* **mutex\_lock()** (per entrare nella sezione critica)
* **mutex\_unlock()** (in uscita dalla sezione critica)

Sincronizzazione in Windows

* Quando il nucleo di tale SO accede a una risorsa globale in sistema **a singola CPU,** disabilita temporaneamente le interruzioni aventi procedure di gestione che potrebbero accedere alla stessa risorsa globale
* In un sistema **con più unità d’elaborazione,** protegge l’accesso alle risorse globali con i semafori ad attesa attiva **(spinlocks)**
* Per la sincronizzazione fuori dal nucleo, il SO offre gli **oggetti dispatcher** (dispatcher objects) che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi mutex, semafori ed eventi.
* Gli eventi sono un meccanismo di sincronizzazione e si può usare in modo simile alle variabili condizionali