DOMANDE ESAME SDC II

Anno 2015 - luglio:

Descrivere in pseudo codice C l'implementazione di un sistema produttore-consumatore con un buffer di N elementi.

```
const int sizeofbuffer=N;
semaphores s=1, n=0, e=sizeofbuffer;
void producer(){
        while true(){
                produce();
                semWait(n);
                semWait(s);
                append();
                semSignal(s);
                semSignal(e);
        }
}
void consumer(){
        while true(){
                semWait(e);
                semWait(s);
                take();
                semSignal(s);
                semSignal(n);
                consume();
        }
}
void main(){
        parbegin (producer, consumer);
}
```

Il semaforo s è usato per la mutua esclusione, quello n per il numero di elementi nel buffer, mentre il semaforo e è stato aggiunto per tener traccia del numero delle posizioni libere.

Descrivere i metodi di localizzazione del server in un sistema RPC.

La localizzazione del server in un sistema RPC avviene in 3 metodi fondamentali:

- **Metodo statico:** all'interno del client viene cambiato l'indirizzo IP del server.
- **Metodo dinamico:** lo stub client, mentre impacchetta i dati, invia un broadcast per richiedere l'indirizzo IP del server, quindi le macchine che implementano la procedura risponderanno al client.
- **Metodo server:** il client interroga il Name Server, il quale risponde con l'indirizzo IP del server giusto o con un errore. Alla ricezione del messaggio con l'indirizzo IP lo stub completa l'impacchettamento dei dati ed invia il messaggio.

Definire i tre concetti alla base della triade di sicurezza nei sistemi di computer.

I tre concetti alla base della triade di sicurezza nei sistemi di computer sono:

- **Confidenzialità (o riservatezza):** assicurare che dati privati o informazioni confidenziali non devono essere disponibili ad individui non autorizzati e che ognuno possa decidere quali informazioni possano essere inviare a qualcuno e quali no.
- **Integrità:** assicurare che le informazioni possano essere modificate solo in una specifica ed autorizzata maniera e che il sistema esegua le sue funzioni senza dover rendere conto a modifiche del sistema non autorizzate.
- **Disponibilità:** assicura che il sistema funzioni prontamente e che il servizio non venga negato ad utenti autorizzati.

Considerate il seguente *algoritmo del panettiere*: spiegare perché l'algoritmo non soddisfa la seguente proprietà: "se un processo p entra nella doorway prima di un processo q, allora p entra in sezione critica prima di q".

p.2

q.2

q.3 num[q]=1

p.3 num[p]=2

p.4

q.4

q.5

q.6 (cond. False) -> q esce dal 1° while

p.5

p.6 (cond. False) -> p esce dal 1° while

p.7 (num[q] < num[p]) -> p rimane nel 2° while

q.7 (num[p] > num[q]) -> q esce dal 2° while

q.8

q.9 CS

Come si vede p accede alla doorway prima di q dal momento che esegue l'istruzione 2 per prima, ma l'interleaving dei processi fa in modo che il numero di priorità di q sia assegnato prima di p. In questo modo anche se l'istruzione 7 è eseguita prima da p, quest'ultimo dovrà dare la precedenza a p per l'accesso alla CS.

Anno 2016 – luglio:

Spiegare che cosa sono le operazioni di "compare & swap" e come possono essere utilizzate per sincronizzazione processi per l'accesso ad una sezione critica. Elencare vantaggi e svantaggi di queste primitive.

Per garantire la mutua esclusione è possibile usare delle istruzioni macchina speciali che lavorano in modo atomico su una locazione di memoria. Possono pertanto effettuarvi lettura, scrittura e test senza interruzioni. La più comune di queste istruzioni è appunto la compare and swap. L'istruzione compare and swap (o compare and exchange) è eseguita tra un valore in memoria e un valore di test, se sono uguali allora bisogna eseguire una swap, cioè il valore in memoria è sostituito da un nuovo valore, altrimenti il valore in memoria rimane invariato. Tutto questo viene eseguito in un blocco di istruzioni atomiche, infatti inizia con ATOMIC() e finisce con END_ATOMIC().

I vantaggi di queste istruzioni atomiche sono che possono essere applicate ad un numero qualsiasi di processi in uno o più processori che condividono memoria, sono facili da verificare e possono essere utilizzate sezioni critiche multiple definite ognuna dalla propria variabile.

Mentre gli svantaggi sono che mentre un processo è in waiting per l'accesso in sezione critica, continua a prendere tempo dal processore (busy waiting). Inoltre, si possono verificare sia Starvation (se un processo lascia la sezione critica e ce ne sono più di uno in wait) che Deadlock.

Si consideri il seguente pezzo di codice:

```
void func (char* str){
    char buf[126];
    strcpy (buf, str);
}
```

Spiegare perché può generare stack (buffer) overflow e come può essere sfruttato da un malintenzionato per lancaire un attack code (codice malevolo).

Supponiamo che un server contenga la funzione scritta sopra. Quando viene invocata, un nuovo frame (activation record) viene creato sullo stack. La memoria puntata da str viene copiata sullo stack, strcpy però non controlla che la stringa *str contenga meno di 126 caratteri. Se una stringa è più lunga di 126 viene copiata nel buffer, le locazioni in stack adiacenti verranno sovrascritte. Scrivere al di là dei limiti del buffer può alterare i

dati adiacenti e portare ad un comportamento indesiderato. Sovrascrivere dati sensibili con frammenti di codice, rende il programma inutilizzabile o ne causa la terminazione. Ugualmente pericolosa è la sovrascrittura di dati sensibili per alterare il control flow del programma. Gli attaccanti usano questo metodo per deviare l'esecuzione verso il loro codice malevolo.

Spiegare quali sono gli effetti collaterali legati al passaggio di parametri in una RPC. Illustrare il tutto con un esempio che indichi le problematiche legate all'implementazione dello stub client.

Il passaggio dei parametri nella RPC può avvenire in due modi:

- **Call by reference** (sconsigliata) è deprecated perché il riferimento ad una cella di memoria non ha senso in un altro elaboratore.
- **Call by copy/restore** copia una variabile, dallo stub del client, nel pacchetto dati e il nuovo valore della variabile (restituito dal server nei parametri di ritorno) sarà copiato, dallo stub del client, nella cella che contiene la variabile.

CLIENT SIDE	*SERVER SIDE*
begin	procedure doppioincr (var x, y: int)
…	begin
a=0;	…
doppioincr (a, a);	x = x+2;
writeln (a);	y = y+3;
end	end

Il risultato della call by reference sarà "a=5", perché il client chiama una procedura passando lo stesso parametro, quindi lo stesso riferimento. Mentre nel caso della call by copy/restore sarà "a=2 o a=3" in base all'implementazione dello stub del client (il server vedrà a ed a come due parametri differenti attraverso lo stub).

Considerare il seguente *algoritmo di* mutua esclusione di *Ricart-Agrawall* dove la variabile last_req:=0 invece di last_req:=maxint (linea 8) come nell'algoritmo originale:

…

Discutere se questa inizializzazione comporta problemi alla proprietà di "No-Starvation" o a quella di "mutua esclusione" o a entrambe. Motivare la risposta se possibile attraverso il disegno di specifiche esecuzioni dell'algoritmo (run).

i.6 CS

i.7 send REPLY

i.8 last_req=0

Il processo i esce dalla CS con num=1 e last_req viene settato a 0, invece che a maxint. Accede per la prima volta il processo j.

i.1 state=requesting

j.1 state=requesting

j.2 num=1; last_req=1

j.3 send Request(1) to p_i

i.9 num=max (1, 1)=1

i.10 (cond. True)

i.11 processo j inserito in coda

i.2 num=2; last_req=2

i.3 send Request(2) to p_j

j.9 num=max (2, 1)=2

j.10 (cond. True)

j.11 processo i inserito in coda

La riga i.10 verifica la condizione (last_req<t) dal momento che last_req=0, invece di maxint come nella versione originale. Anche nella riga j.10 la condizione è verificata, allora i processi i e j rimangono in coda in attesa di ricevere un REPLY dall'altro: DEADLOCK. A questo punto la proprietà di "No Starvation" non è più verificata. Quindi, in conclusione, questa inizializzazione comporta problemi alla proprietà di No starvation.

Anno 2016 – febbraio:

Descrivere il meccanismo della system call Fork() con particolare riferimento al PCB del processo padre e del processo figlio.

Gli elementi del processo sono tutti contenuti nel suo PCB: l'identificatore, lo stato, la priorità, il PC, i puntatori, i context data, le funzioni I/O ed altro. Il Process Control Block è creato e gestito dal sistema operativo. Un processo (padre) invoca la fork() per creare un altro processo (figlio) che è esattamente il duplicato del primo. Il padre e il figlio eseguono in concorrenza e possono generare altri figli. La fork() restituisce: -1 in caso di errore, 0 nel caso del figlio, PID nel caso del padre (PID identificatore del figlio). Notare che il PCB per il figlio è la copia identica del PCB del padre al momento della chiamata. Il figlio prosegue poi con l'istruzione successiva della fork(). Il processo figlio termina l'esecuzione con la chiamata exit(number) che invia un segnale asincrono che sveglia il padre. Questo utilizza infatti wait() o waitpid() per aspettare rispettivamente la terminazione di qualche figlio o di un figlio in particolare, prima di proseguire a sua volta.

Descrivere le caratteristiche principali del costrutto programmativo MONITOR e le principali differenze con i semafori.

Il monitor è un costrutto di programmazione che fornisce funzionalità equivalenti a quelle dei semafori, ma più facili da controllare. Il monitor è un modulo software che consiste di una o più procedure, una sequenza di inizializzazione e dati locali.

Le principali caratteristiche sono:

- Le variabili locali sono accessibili solo tramite le procedure del monitor e non tramite procedure esterne.
- Il processo entra nel monitor invocando una delle sue procedure.
- Solo un processo alla volta può eseguire nel monitor (MUTUA ESCLUSIONE).

Il vantaggio principale rispetto ai semafori è che tutte le funzioni di sincronizzazione sono confinate al monitor. Se un monitor è programmato bene, l'accesso alla risorsa protetta è corretto per tutti i processi. Al contrario con i semafori anche i processi che accedono alla risorsa devono essere programmati correttamente.

Discutere le tecniche usate per la localizzazione del server nelle RPC.

Vedi ***Anno 2015 – luglio.***

Algoritmo di Dijkstra: sostituendo la riga 5 con "if y[k] then k:=i" quali sono le problematiche che si creano legate alle priorità di correttezza di Mutua Esclusione e No Deadlock? Se e quali vengono violate? Motivare la risposta.

ME (Mutua Esclusione):

Per contraddizione supponiamo che i due processi i e j siano in sezione critica:

i.8 < j.6

j.8 < j.6

i.8 < j.6 < j.8 < i.6 < i.8 **Contraddizione!!!**

Quindi assumendo che i e j siano entrambi in CS, si arriva ad una contraddizione.

Trovare, ora, un pattern per cui i e j rimangono bloccati nella tying section. Per dimostrare che c'è **deadlock** bisogna concentrarsi sul for perché un processo i!=k troverà y[k]=true e passerà il ciclo sentinella.

k=l ,l è in CS

i.3 x[i]=false

j.3 x[j]=false

i.4 cond. True

i.5 (cond. True) .> k=i

l.10 esce dalla CS

j.4 cond. True

j.5 (cond. true) -> k=j

j.6 k[j]=true

i.6 k[i]=true

i.7-8 ->goto 3 vengono rimandati alla linea 3, allora se questo blocco è

j.7-8 ->goto 3 eseguito all'infinito, allora si ha deadlock infinito.

Non è l'unico modo per creare deadlock, ci sono altri tipi di linearizzazione.

Ogni processo supererà il ciclo sentinella, tranne la prima volta in cui y[k] è inizializzato a false, perché troverà y[k]=true e potrà così impostare x[i]=true. Il processo rimarrà bloccato nel for se il blocco precedente è ripetuto all'infinito.

Se l'istruzione 6 è spostata dopo il for, allora la mutua esclusione non è garantita.

i.2

i.3

j.2

j.3

i.4 (cond. True)

i.5 (cond. True) -> k=i

j.4 (cond. True)

j.5 (cond. True) -> k=j

i.6

i.7 (cond. True)

j.6

j.7 (cond. True)

i.8 -> k[i]=true

i.9 -> CS

j.8 -> k[j]=true

j.9 -> CS

La run mostra chiaramente che se due processi i e j riescono ad eseguire le istruzioni 6 e 7 prima che l'altro ha eseguito la 8, allora entrano entrambi in CS.

Anno 2017 – giugno:

Componenti della RPC: Protocol Stack (Blast, Chan e Select).

BLAST:

Frammenta e riassembla messaggi e dipende tutto dalla grandezza data ai timer e non garantisce lo scambio di messaggi completi:

- **Lato sender:**
  - Mette i frammenti in memoria e li invia, mettendo il timer a DONE
  - Se riceve un SRR manda i frammenti mancanti e rimette il timer a DONE
  - Se riceve un SRR "Tutti i frammenti sono arrivati" allora libera i frammenti
  - Se il timer DONE è scaduto, libera i frammenti.
- **Lato receiver:**
  - Quando arriva il primo frammento mette il timer a LAST_FRAG
  - Quandi sono arrivati tutti i frammenti invia un SRR:
    - ✓ Ultimo frammento arrivato ma messaggi non completo (invia SRR e setta il timer a RETRY)
    - ✓ Scade il timer LAST_FRAG (invia SRR r setta il timer a RETRY)
    - ✓ Scade il timer RETRY per la prima o seconda volta (invia SRR e setta il timer a RETRY)
    - ✓ Scade il timer RETRY per una terza volta (finisce e libera il messaggio parziale)

CHAN:

Sincronizza il client con il server garantendo il trasporto dei messaggi (tiene conto della perdita di messaggi in quanto ognuno è memorizzato fino a che non arriva l'ACK relativo):

- Ogni messaggio (REQUEST, REPLY) rimane in memoria finche non riceve l'ACK corrispondente
- Mette il timer a RETRANSMIT rinvia il messaggio ogni volta che scade
- La ritrasmissione implica una duplicazione del messaggio

SELECT:

Si occupa di assegnare alla giusta RPC i dati che riceve. Per distinguere le diverse chiamate remote esistono due modi:

- **Flat:** identificatore per ogni procedura.
- **Gerarchico:** identificatore del programma + identificatore della procedura.

Meccanismi per la concorrenza:

Esistono due meccanismi per la concorrenza:

- **Semafori:** un valore intero utilizzato per la segnalazione tra processi. Può eseguire solo 3 operazioni: inizializzazione, incremento e decremento, il decremento per bloccare un processo (semWait) e l'incremento per sbloccarlo (semSignal).
- **Monitor:** un costrutto che incapsula variabili, procedure e inizializzazione in un tipo di dato astratto. Le sue variabili sono accessibili solo dalle sue procedure e solo un processo alla volta può accedere ad un monitor (il costrutto Monitor è inoltre disponibile su più linguaggi di programmazione rispetto al costrutto dei semafori).

Problema di lettori e scrittori (readers/writers) con scambio di messaggi (Messagge Passinf) tramite Mailbox.

Un'area di dati è condivisa tra molti processi, alcuni sui quali leggo soltanto (readers) o scrivo (writers). Devono essere soddisfatte:

- Un qualunque numero di readers può leggere contemporaneamente i file
- Un solo writer alla volta può scrivere nel file
- Se un writer scrive nessun reader può leggere nello stesso tempo

```
void reader(int i) {
        message rmsg;
        while (true) {
                 rmsg = i;
                send (readrequest, rmsg);
                receive (mbox[i], rmsg);
                READUNIT ();
                rmsg = i;
                send (finished, rmsg);
```

```
        }
}
void writer(int j) {
        message rmsg;
        while(true) {
                rmsg = j;
                send (writerequest, rmsg);
                receive (mbox[j], rmsg);
                WRITEUNIT ();
                rmsg = j;
                send (finished, rmsg);
        }
}
void controller() {
        while (true) {
                if (count > 0) {
                        if (! empty (finished)) {
                                receive (finished, msg);
                                count++;
                        }else if ( !empty (writerequest)) {
                                receive (writerequest, msg);
                                writer_id = msg.id;
                                count = count - 100;
                        } else if (!empty (readrequest)) {
                                receive (readrequest, msg);
                                count--;
                                send (msg.id, "OK");
                         }
                }
                if (count == 0) {
                        send (writer_id, "OK");
                        receive (finished, msg); count = 100;
                } while (count < 0) {
                        receive (finished, msg);
                        count++;
                }
        }
}
```

Algoritmo del panettiere (Lamport) modifica su righe 2/4 spostate nella CS).

Anno 2017 – luglio:

Problema della sincronizzazione dei processi con Message Passing.

```
const int n=/*number of processes*/;
void P (int i){
        message msg;
        while(true){
                receive (box,msg);
                /*CS*/
                Send (box, msg);
        }
}
void main(){
        create_mailbox (box);
        send (box, null);
        parbegin (P(1), … , P(n));
}
```

La receive è bloccante, ma la send no. Un processo prima di entrare in CS cerca di ricevere un messaggio, se è vuoto viene bloccato. Appena arriva il messaggio accede in CS e al termine invia qualcosa. Il messaggio diventa un permesso di entrata.

Se più processi eseguono la recieve: se è presente un messaggio, solo uno eseguirà la receive e gli altri saranno bloccati; se la coda di messaggi è vuota, tutti i processi sono bloccati, appena arriva ne verrà attivato uno solo.

La non blocking send, blocking receive permette al processo di inviare uno o più messaggi ad una varietà di destinazioni il più velocemente possibile. Un processo che deve ricevere un messaggio prima di compiere lavoro utile deve essere bloccato finché quel messaggio non sarà arrivato.

Spiegare quali sono gli effetti collaterali legati al passaggio di parametri in una RPC. Illustrare il tutto con un esempio che indichi le problematiche legate all'implementazione dello stub client.

Vedi ***Anno 2015 – luglio.***

Formattazione dei dati.

Il passaggio di parametri deve essere regolamentato dal momento che chiamante e chiamato si trovano su macchine diverse con differenti architetture. La regolamentazione avviene in una codifica e decodifica comuni delle informazioni passate.

Le difficoltà principali sono legate alla rappresentazione di: interi e floating point. Per superare il problema dei primi si usano le forme canoniche. In pratica A codifica i dati secondo una forma concordata e B riceve i dati e li decodifica per adattarli alla sua organizzazione. Per superare il problema dei secondi si usa la tecnica receives makes right: non viene accordata precedentemente una forma, ma il ricevitore che si occupa di capire la forma utilizzata. Il problema è che potrebbe non riconoscere quella usata e quindi non si potrebbe comunicare.

Per riconoscere il tipo di dato osservo che, i dati all'interno di una forma possono essere: tagged (nel pacchetto sono specificati TYPE, LEN, VALUE) e untagged (non c'è una struttura prestabilita nel pacchetto).

Logical clock.

Il "Logical Clock", introdotto da Lamport, è un registro monotono crescente che incrementa il suo valore. Ogni processo p_i utilizza il proprio clock L_i per applicare il timestamp agli eventi e. Quindi se e è successo prima di e' allora L(e)<L(e').

L_i è il clock del processo i, mentre e_i^j è il j-esimo evento del processo i.

Per relazione "Happer-Before" si intende che se e->e' in un certo p_i allora e è accaduto prima di e', oppure la notazione e||e' se hanno lo stesso timestamp (concorrenti).

Buffer overflow

Vedi ***Anno 2016 – luglio.***

***Algoritmo di Dijkstra*: modifica su riga 8 (goto 4/goto6).**

Anno 2018 – aprile:

Descrivere il ruolo dei semafori nei meccanismi di sincronizzazione tra processi e thread. Discutere le operazioni fondamentali definite per un semaforo, e come i meccanismi di coda interni scelti dal SO per la loro implementazione determino due possibili tipologie di semafori.

Un semaforo è un meccanismo di concorrenza. Possiamo definire un semaforo come una variabile intera utilizzata per segnalazioni tra processi. Un semaforo può eseguire 3 operazioni atomiche, ovvero l'inizializzazione (a un intero non negativo), l'incremento (semSignal) e il decremento (semWait) di se stesso. Il decremento di un semaforo corrisponde al blocco di un processo, intuitivamente l'incremento del semaforo corrisponde allo sblocco. Un semaforo è implementato come una struct contenente due campi: il primo è una variabile intera non negativa, il secondo contiene una coda di tipo arbitrario, che viene riempita da un processo con la semWait e svuotata con la semSignal. Una coda è usata per bloccare un processo nello stato wait con il semaforo. Possiamo identificare due tipi di semafori: gli strong semaphores fanno in modo che il processo che è stato bloccato da più tempo sia rilasciato dalla coda per primo; i weal semaphores non hanno un ordine specificato che indichi come i processi vengano rimossi dalla coda.

Spiegare le problematiche legate al verificarsi di errori durante una chiamata RPC e quali sono i tre meccanismi implementativi più comuni (RPC semantics). Opzionalmente, illustrare graficamente come uno dei tre meccanismi si comporta in presenza di uno o più errori nella gestione di una richiesta RPC, mostrando le richieste inviate dal client ed eventuale/i risposta/i del server.

In un sistema distribuito non si ha la certezza che una chiamata di procedura avvenga un'unica volta.

La semantica RPC si basa su tre meccanismi:

- **At last once:** time out stub del client e ritrasmissione. Il messaggio di richiesta RPC inviato almeno una volta. Nel peggiore dei casi il server ritrasmette la stessa risposta n volte.
- **At most once:** richiesta rieseguita al più una volta, dopo di che torna un codice di errore se già è stata trasmessa.
- **Exactly once:** richiesta eseguita esattamente una volta. Nel server immagazzina i risultati della RPC in un disco (logger) e in caso di richiesta già effettuata viene ripresa dal log; invece nel client le richieste vengono tutte numerate con un SEQUENCE NUMBER, il client deve tener conto di un numero di reincarnazione, in caso di guasto e riaccenzione dl client, questo invia il numero di reincarnazione prima di iniziare ad eseguire le RPC. Ciò causerebbe una ricomputazione del server ma non è così grazie al confronto con il numero di reincarnazione memorizzato.

Spiegare che cosa esegue questo codice e quali possono essere le problematiche di corruzione dello stack a cui si può essere sottoposti:

```
int sleep (unsigned int seconds);
void bar (int arg){
        char buf[64];
        foo (buf, arg, &sleep);
        printf ("%s\n", buf);
}
int foo (char* buf, int arg, unsigned int (*funcp)(unsigned int)){
        char tmp[128];
        gets (tmp);
        strncpy (buf,tmp, 64);
        (*funcp)(arg);
        return 0;
}
```

Nella funzione foo, un function pointer (*funcp) è dichiarato direttamente dopo il buffer fissato tmp. Dal momento che gets non effettua controlli sul limite del buffer, sarà possibile causare un overflow. L'attaccante potrebbe sovrascrivere funcp con l'indirizzo di tmp: in questo modo alla prossima chiamata di funcp il controllo sarà trasferito a tmp. Quindi ci troviamo nel caso di functon pointer overflow.

Algoritmo di Lamport (PANETTIERE): quali modifiche sono state introdotte rispetto all'algoritmo originale? Quali sono le problematiche che si creano legate alle proprietà di correttezza e di No Starvation e Mutua Esclusione? (Se e quali vengono violate) Motivare la risposta.

Anno 2018 – giugno:

Definire Deadlock, Starvation e Livelock, esplicitare le differenze principali tra esse ed infine definire anche le loro implicazioni:

- **Deadlock:** è la situazione in cui due o più processi non sono in grado di proseguire perché ognuno aspetta che uno degli altri processi faccia qualcosa. ("situazione di stallo").
- **Starvation:** è la condizione in cui un processo ready non è mai scelto dallo scheduler. O meglio, alcuni processi sono in deadlock e non potranno più accedere a risorse allocate, altri invece non sono in deadlock e continuano.
- **Livelock:** è simile alla situazione di stallo in cui vari processi computazionali sono in continua evoluzione, ma non raggiungono mai un punto in cui nessuno di loro può procedere.

DEADLOCK => STARAVATION

STARVATION NO => DEADLOCK

NO STARVATION => NO DEADLOCK

Definizione di socket e funzioni con cui comunica una socket, spiegare inoltre la connessione client-server di una socket:

Una socket è un canale di comunicazione bidirezionale tra due end-point. Ogni end-point è identificato da una coppia indirizzo IP, porta.

Per comunicare via socket come prima cosa dobbiamo aprirla con la funzione ***socket()***, questa funzione crea un edpoint di comunicazione e ne ritorna il descrittore.

Successivamente per accettare connessioni in ingresso (LATO SERVER) sono necessarie tre funzioni da chiamare in sequenza:

- ***bind():*** collega il descrittore della socket ad un indirizzo (endpoint).
- ***listen():*** segnala che una socket può essere usata per accettare connessioni.
- ***accept():*** attende l'arrivo di connessioni da accettare.

Per creare una connessione in uscita (LATO CLIENT) eseguo la funzione ***connect()***, la quale effettua un tentativo di connessione verso l'endpoint specificato, usando il descrittore della scoket.

Infine chiudo la socket con la funzione ***close()***.

Ci sono inoltre altre due funzioni che utilizza una socket:

- ***send():*** invia delle informazioni.
- ***receive():*** si mette in ascolto di alcuni dati.

Stati di esecuzione dei thread:

Un thread è la suddivisione di un processo in due o più sottoprocessi eseguiti in maniera concorrente. È detto anche "processo leggero", in quanto è un meccanismo più efficiente per gestire più flussi all'interno del SO.

I thread sono composti da:
- Stato di esecuzione (Ready, Running, Blocked)
- Thread Context
- Execution Stack
- Pre-Thread Static Storage
- Accessi a memoria e risorse del processo

Gli stati di esecuzione dei thread vengono cambiati con 4 operazioni:
- **Spawn:** il thread viene lanciato.
- **Block:** il thread viene bloccato ed è in attesa di un evento.
- **Unblock:** quando si verifica l'evento, il thread si sblocca e torna in Ready.
- **Finish:** quando un thread termina il suo compito, viene terminato.

Monitor: caratteristiche e pseudocodice

Vedi ***Anno 2016 - febbraio.***

```
monitor <nome_del_monitor> {
type <variabili_condivise>
```

```
procedure <nome_della_procedura> (type <parametri>) {…}
}
```

Spiegare che cosa esegue questo codice e quali possono essere le problematiche di corruzione dello stack a cui si può essere sottoposti:

```
char* secret = “My_D4rk3st_s3cr3t”;
int authenticate_user (){
    int authorized = 0;
    char buffer[128];
    printf (“Enter your password: ”);
    gets (buffer);
    authorized = strcmp (buffer, secret) ? 0 : 1;
    if (authorized) printf (“Hey welcome back!\n”);
    else printf (“Go away stranger!\n”);
    return authorized;
}
```

La funzione authenticate_user() compara la stinga passata con il buffer locale , il quale eseguendo la funzione unsafe gets() non ha controlli sul terminatore di stringa. Dal momento che authenticate_user() usa argomenti arbitrari dalla linea di comando, un attaccante potrebbe causare overflow del buffer per modificare lo spazio in memoria adiacente. Quindi ci troviamo nel caso di stack smashing dove lo scopo dell’attaccante è quello di modificare RIP cambiando così il flusso di esecuzione e dirigendolo in una locazione dove sono state inserite, in precedenza, delle istruzioni.

Algoritmo di Dijkstra.

ALGORITMI

Dijkstra:

```
Shared variables
        x[1,..n]: array of Boolean, initially all false
        y[1,..n]: array of Boolean, initially all false
        k: integer in range 1,..N, initially any value in its range
Local variables
        j: integer in range 1,..N
repeat
        1 NCS
        2 y[i]:= true                          % inizio trying protocol %
        3 x[i]:= false
        4 while k≠i do                         % ciclo della sentinella %
        5 if not y[k] then k:=i
        6 x[i]:= true
        7 for j:= 1 to n do
        8 if i≠j and x[j] then goto 3          % fine trying protocol %
        9 CS
        10 y[i]:=x[i]:= false;                 % exit protocol %
Forever
```

Quando il processo i vuole entrare in sezione critica prova prima ad impostare k al suo id i, passando quello che si chiama ciclo della sentinella. Il valore k ci dice l'ultimo processo che è uscito dal ciclo sentinella.

Nel secondo ciclo (for) si controlla se ci sono stati passaggi concorrenti attraverso il primo ciclo ed in questo caso si deve eseguire ulteriore scrematura per permettere ad un solo processo di entrare in sezione critica (tra quelli passati per il ciclo della sentinella)

L'algoritmo soddisfa

- no deadlock
- mutual exclusion
- ma non soddisfa no starvation.

Modifiche al codice

Se istruzione 6 `x[i] = true` è spostata dopo il `for` allora la mutua esclusione non è più garantita

Se sostituisco riga 5 con `if y[k] then k = i` allora la mutua esclusione non è più garantita

Lamport/Panettiere:

Shared variable
num[1,..n]: array of integer, initially all 0
choosing[1,..n]: array of Boolean, initially all false
%process i owns num[i] and choosing[i]%
Local variable
j: integer in range 1,..N
repeat
1 NCS
2 choosing[i]:= true *%DOORWAY%*
3 num[i]:= 1+ max {num[j] : n >= j >=1}
4 choosing[i]:= false
5 **for** j:= 1 **to** n **do begin** *%BAKERY%*
6 **while** choosing[j] **do skip**
7 **while** num[j] ≠0 **and** {num[j], j } < {num[i],i} **do skip**
8 end
9 CS
10 num[i]:=0;
Forever
Questo algoritmo garantisce:

- mutual exclusion (deriva dal fatto che se un processo i è nella doorway ed un processo j è nella bakery section allora {num[j],j} < {num[i],i})

- no starvation, che implica no deadlock (nessun processo attende per sempre perchè prima o poi avrà numero di attesa più piccolo)

Doorway: il processo `i` segnala che sta scegliendo il numero, lo impost al `max+1` e mette `choosing[i] = false`. Non c'è garanzia che i numeri scelti siano diversi tra i processi (ci possono essere più processi contemporaneamente)

Bakery: ogni processo deve assicurarsi che lui è il prossimo a dover entrare in CS (ciclo `while` serve a questo). Il processo i cicla fino a quando non è sicuro che ogni altro processo j o non è nella doorway oppure ha un numero di coda maggiore del suo, oppure lo ha uguale e j>i. A questo punto i accede alla CS

First come first served: i.5 < j.2 allora `i` entra in CS prima di `j`. Infatti se i ha già scelto il suo numero j avrà priorità almeno `priorità(i)+1`. Questo permette no starvation

Domande

Processo p che entra prima di q non necessariamente entra prima di q in CS, perchè c'è controllo anche su identificatore del processo

Upper bound di num? È unbounded perchè anche con soli due processi alternanti, `num[i]` cresce. Il proprio num infatti non viene risettato a 0

Vantaggi rispetto a Dijkstra

1. Garantisce mutua esclusione in sistemi in cui non si fanno assunzioni su atomicità di lettura e scrittura
2. No starvation

Agrawal:

Local variables

#replies: integer initially set to zero

state: in set {requesting, CS, NCS} initially set to NCS

Q : queue of pending request {T, i}initially set to empty

last_req: integer initially set to maxint

num: integer initially set to zero

begin

1.state := requesting
2.num = num + 1; last_req = num;
3.send REQUEST (last_req) p1, … , pn
4.wait until #replies = (n – 1)
5.state := CS
6.CS
7.send REPLY to any request in Q; Q!=0; state :=NCS; #replies :=0
8.last_req = maxint

end

Upon the receipt of REQUEST(t) from process j

9.num := max(t, num)
10.if state = CS or (state = requesting and {last_req, i} < {t, j})
11.then insert Q({t, j})
12.else send REPLY to pj

Upon the receipt of REPLY from process j

12.#replies ++;

Si evita di violare mutual exclusion grazie al fatto che se un processo è in CS o se ha numero di attesa inferiore ad un altro processo che ha fatto una richiesta, il primo ritarda l'invio del messaggio di reply (richieste ritardate vengono inserite nella coda Q)

Questo algoritmo garantisce:

- mutual exclusion
- no starvation (che implica no deadlock)

Modifiche al codice

Se Last_Req=0 invece che Last_Req=max_int.