IMPORTANTI SC2

1. Processi e Thread

Un processo è composto dal codice del programma, un insieme di dati, e una serie di attributi che ne specificano l'evoluzione durante l'esecuzione: l'ID, lo stato, la priorità, il program counter, i puntatori di memoria, ecc.
All'avvio di un PC, o al riavvio, parte il bootstrap program il quale ha il compito di inizializzare i registri della CPU, fa un checkup della memoria e dei dispositivi, carica l'OS nella memoria e lo avvia. Una volta avviato l'OS, esso avvia il primo processo, chiamato "init", e aspetta di gestire qualche evento causato da interrupt o trap.

Un processo ha due caratteristiche fondamentali: è proprietario di uno spazio virtuale di memoria dove montare la propria immagine ed è trattato indipendentemente dall'OS, ossia potrebbe eseguire il proprio task in maniera concorrente con altri processi.
Un processo in sistemi multithread ha il compito di proteggere la propria immagine da altri processi, dalla CPU, files, ecc. Mentre i thread, che attenzione condividono tutti l'area di memoria del processo, svolgono un proprio task indipendente dagli altri thread. Essi hanno un proprio stato (contenuto nel TCB) , una stack di esecuzione, e anche una piccola area di memoria dove conservare le proprie variabili locali.
I benefici dei thread sono: il minor tempo di creazione rispetto a un processo, anche lo switching richiede meno tempo, condividendo la stessa area di memoria non richiedono l'intervento del kernel per comunicare.
Siccome l'OS gestisce i thread a livello dei processi, se il processo che contiene thread in questione venisse sospeso o chiuso, quest'azione si proietterebbe anche su tutti i thread. I thread avendo anch'essi uno stato (spawn, block, unblock, finish) possono essere sincronizzati.
Abbiamo due tipi di thread User Level Thread (ULT) oppure i Kerner Level Thread(KLT), anche chiamati processi leggeri. I primi vengono totalmente gestiti dall'applicazione, il kernel gestisce solo il processo che li crea. Dello scheduling dei secondi invece se ne occupa direttamente il kernel, windows è un esempio di questo approccio. I vantaggi dei primi sono che lo switch tra i ULTs avviene senza il passaggio alla kernel mode e funzionano su qualsiasi OS poiché implementati attraverso una libreria a livello utente. Lo svantaggio è che una chiamata da parte di un thread bloccherebbe tutti gli altri contemporaneamente. Perderemmo vantaggi delle architetture multicores. I vantaggi dei secondi sono che vari KLT possono essere schedulati su core differenti, se un solo thread effettua una chiamata bloccante non ci sono effetti sugli altri thread, gli stessi task del kernel possono essere multithread. Lo svantaggio è che lo switch tra thread richiede l'intervento del kernel.

Multiprocessing simmetrico
-Single Instruction Single Data (SISD): un singolo processore esegue un solo flusso di istruzioni che operano su dati memorizzati in una singola memoria.
-Single Instruction Multiple Data (SIMD): ogni istruzione è eseguita su un insieme diverso di dati da parte di diversi processori.
-Multiple Instruction Single Data (MISD): Un insieme di dati viene trasmessa a più processori che eseguono una sequenza differente di istruzioni.
-Multiple Instruction Multiple Data (MIMD): Una serie di processori eseguono simultaneamente diverse sequenze di istruzioni su differenti insiemi di dati.

Dei MIMD differenziamo quelli a memoria condivisa, di cui fanno parte i Symmetric Multiprocessors (SMP), e quelli a memoria distribuita, di cui fanno parte i Clusters.
I principali problemi di progettazioni degli OS multiprocessor sono: la concorrenza simultanea dei processi e dei thread, lo scheduling, la sincronizzazione, la gestione di memoria, gestione dei Fault.

2. Concorrenza

Il sistema operativo si occupa della gestione dei processi e dei thread: multiprogramming (gestione di più processi in sistemi solo-core) , multiprocessing (gestione di più processi in sistemi multicore) , distributed processing (gestione di più processi in sistemi distribuiti es. cluster).

DEFINIZIONI:

Operazione atomica: una funzione o un'azione implementata come una sequenza di una o più istruzioni che appaiono indivisibili; cioè nessun altro processo può vedere uno stato intermedio o interrompere l'operazione. È garantita un'esecuzione completa o nel caso evitata un'esecuzione parziale. L'atomicità garantisce isolamento dai processi concorrenti.

Critical section: Una sezione di codice in cui un processo ha bisogno di un permesso d'accesso affinché possa condividere informazioni. Non possono essere eseguiti più processi contemporaneamente su quel blocco di istruzioni.
Deadlock: Una situazione in cui due o più processi sono impossibilitati a procedere poiché tutti aspettano che sia fatto qualcosa da altri processi in deadlock.
Livelock: una situazione in cui due o più processi cambiano continuamente il loro stato in base ai cambiamenti degli altri processi coinvolti senza però effettuare lavoro utile.
Mutua esclusione: garantisce l'accesso ad un singolo processo della sezione critica.
Race condition: una situazione in cui più processi o thread leggono o scrivono un dato condiviso e il risultato finale dipende dal tempo di accesso a tale risorsa.
Il perdente sarà colui che per ultimo aggiorna il valore e da cui dipenderà il risultato finale.
Starvation: Situazione in cui un processo in stato RUNNABLE è bloccato per un tempo indefinito dallo scheduler; nonostante è pronto per procedere non viene mai scelto.

I principi della concorrenza sono la convivenza e la sovrapposizione. Purtroppo, la velocità d'esecuzione di un processo non può essere prevista a priori poiché dipende da più fattori come l'attività degli altri processi, la modalità di gestione degli INTERRUPT da parte dell'OS o la politica di scheduling dell'OS.
Tra due processi che non si vedono tra loro la relazione è indicata come **COMPETIZIONE**, i risultati di un processo non dipendono da risultati di altri processi però può cambiare il tempo di esecuzione di un processo.
I possibili problemi di controllo sono la mutua esclusione, lo stallo o lo starvation.
Tra processi che vedono altri processi indirettamente la relazione si chiama **COOPERAZIONE TRAMITE CONDIVISIONE**, i risultati di un processo possono dipendere da risultati di altri processi e anche qui il tempo di esecuzione può cambiare, da aggiungere ai problemi di prima è il controllo della coerenza dei dati.
Tra processi che vedono altri processi direttamente la relazione si chiama **COOPERAZIONE TRAMITE COMUNICAZIONE**, i risultati di un processo possono dipendere da risultati di altri processi e anche qui il tempo di esecuzione può cambiare, i problemi di controllo sono lo stallo(deadlock) o lo starvation.
In una **RESOURCE COMPETITION** più processi entrano in conflitto per l'uso di una stessa risorsa e vanno quindi affrontati questi problemi: mutua esclusione, deadlock, starvation
Requisiti per la mutua esclusione: deve essere forzata, un processo che si trova in una non critical section deve poter svolgere le proprie attività senza interferire con altri processi. Non si devono verificare deadlock o starvation. Non deve essere negato l'accesso a una cs se nessun altro è in quella cs. Non si devono effettuare assunzioni sulla velocità dei processi o il loro numero. Bisogna che un processo rimanga nella cs per un tempo finito.

HARDWARE SUPPORT FOR MUTUAL EXCLUSION

Possiamo disabilitare gli interrupt su sistemi single core in modo tale da poter garantire la mutua esclusione. Purtroppo però non si può effettuare lo stesso ragionamento su architetture multicore e inoltre l'efficienza di esecuzione potrebbe risentirne in negativo.
Oppure possiamo utilizzare le **COMPARE&SWAP INSTRUCTION**, ossia istruzioni che effettuano una compare tra un valore in memoria e un valore di test e se i valori sono gli stessi viene effettuato uno swap del valore in memoria.

```
int compare_and_swap(int* reg, int oldval, int newval){
        ATOMIC();
        int old_reg_val = *reg;
        if(old_reg_val == oldval)
                *reg = newval;
        END_ATOMIC();
        Return old_reg_val;
}
```

Es. uso in cs:

```
While(compare_and_swap(&reg, 0, 1) == 1) //do nothing
//se supero entro in cs
reg = 0 //permetto ad altri di entrare
```

Oppure usando una funzione che scambia i valore di due registri

```
While(true){
        Int keyi = 1;
        do exchange(&keyi, &reg)
        while(keyi != 0) //prova a scambiare, ma nel registro avremo 1 finchè altri processi sono in
            in cs, quando uno esce ci mette 0, io lo acchiappo e esco da while
        reg = 0  //permetto ad altri di entrare in cs
```

Vantaggi: Applicabile a un numero qualsiasi di processi sia su processori single-core sia su multi-core. Semplice e facile da verificare. Ogni cs può essere gestita da una variabile globale diversa.
Svantaggi: c'è un approccio busy-waiting, finchè un processo occupa la cs gli altri verificano la condizione nel while. E' possibile la starvation nel caso ci siano molti processi in attesa della flag di entrata. Deadlock è possibile.

DEFINIZIONI DI MECCANISMI PER LA CONCORRENZA

Semaforo: Un valore intero usato come flag per i processi. Su di esso si possono effettuare solo tre operazioni, tutte e tre atomiche: inizializzazione, decremento, incremento. Il decremento potrebbe bloccare un processo, l'incremento sbloccarlo.
Semaforo binario: Un semaforo che prende come valori solo 0 o 1.
Mutex: Simile a un semaforo binario ma con la differenza che il processo che blocca il semaforo (lo setta a 0) è lo stesso che lo sblocca (setta a 1).
Condition Variable: Un particolare dato che blocca un processo o un thread finchè una data condizione non si verifica.
Monitor: Un costrutto del linguaggio di programmazione che contiene variabili, procedure di accesso e codice di inizializzazione. Un solo processo alla volta può accedere al monitor e alle variabili all'interno di esso si può accedere solo tramite le procedure dettate dallo stesso monitor.
Quest'ultime sono considerate critical section. Il monitor potrebbe avere una coda d'attesa per l'accesso a dette variabili.
Event flags: Una parola di memoria usata come meccanismo di sincronizzazione. Applicazioni ad esempio possono associare a differenti eventi diversi bit di flags. Un thread può rimanere in attesa che si verifichi uno o più eventi osservando detti bit di flags. Il bloccaggio può avvenire in AND (ossia tutte le condizioni devono verificarsi) on in OR (ossia almeno una).
Mailboxes/Messages: Un modo per due o più processi per scambiarsi messaggi, anche usati per sincronizzarsi.
Spinlocks: Un meccanismo di mutua esclusione in cui un processo esegue infiniti loop di attesa aspettando che una variabile di accesso torni di nuovo disponibile.

Non c'è modo di sapere prima se una volta decrementato il semaforo quel processo verrà bloccato. Ne quale processo continuerà (su un processore single-core) l'esecuzione se ce ne sono due attivi contemporaneamente. Ne puoi sapere se un processo sta aspettando, quindi il numero di processi unblocked potrebbe essere 0 o 1.
La coda è utilizzata per mantenere un processo in attesa del semaforo.
Abbiamo due tipi di semaforo: Strong Semaphore, che basa la sua implementazione su una FIFO, oppure un Weak Semaphore, la cui rimozione dalla coda non è specificata.

PROBLEMA PRODUTTORE/CONSUMATORE

Uno o più processi generano dati e li conservano in un buffer. Un singolo consumatore può prelevare e utilizzare questi dati dal buffer. Un sono processo (PROD/CONS) può accedere al buffer condiviso. (Ipotesi generale, NB possiamo avere anche più consumatori).
Bisogna assicurarsi inoltre che produttori non inseriscano dati in un buffer pieno o che consumatori prelevino dati da buffer vuoti.
Analizziamo ora il caso di buffer infinito evitando il problema di buffer pieno:
Possiamo utilizzare o un binary semaphore in questo modo

```
binary_semaphore s = 1, delay = 0;

void producer(){
```

```
        while(true){
                produce();
                semWaitB(s);
                append();
                n++;
                if(n == 1) semSignalB(delay);
                semSignalB(s);
        }
}

void consumer{
        int m; //uso m  perché così salvo il valore da verificare per la wait su delay da modifiche da parte di producer…. Se producer schedula prima dell'if del consumer vado a consumare un elemento che non esiste.
        semWaitB(delay);
        while(true){
                semWaitB(s);
                take();
                n--;
                m = n;
                semSignalB(s);
                consume();
                if(m==0) semWaitB(delay);
        }
}
```

Oppure general solution:

```
semaphore n = 0, s = 1;

void producer(){
        while(true){
                produce();
                semWait(s);
                append();
                semSignal(s);
                semSignal(n);
        }
}

Void consumer(){
        While(true){
                semWait(n);
                semWait(s);
                take();
                semSignal(s);
                consume();
        }
}
```

O nel caso di bounded buffer bisogna stare attenti anche a buffer pieno.

```
semaphore n = 0, s = 1,e = sizeofbuffer;

void producer(){
        while(true){
                produce();
                semWait(e);
```

```
                semWait(s);
                append();
                semSignal(s);
                semSignal(n);
        }
}

Void consumer(){
        While(true){
                semWait(n);
                semWait(s);
                take();
                semSignal(s);
                semSignal(e);
                consume();
        }
}
```

I semafori possono essere implementati o attraverso software come l'algoritmo di Dekker o tramite un supporto hardware come visto in precedenza. La semWait e la semSignal DEVONO essere atomiche.
Es. basta inserire un while con un istruzione compare_and_swap tra una flag del semaforo e un valore di test, poi riportare la flag a 0 nel momento in cui finisco le operazioni sul semaforo.
Oppure inibire gli interrupts prima di effettuare le operazioni sui semafori, nel caso della wait se la decrementazione non porta al blocco del processo ristabilire gli interrupt, nel caso della signal sempre ristabilirli.

MONITORS

Sono costruzioni del linguaggio di programmazione che permettono di avere le stesse funzionalità dei semafori ma con un controllo più semplice. Su di esso sono definite procedure, sequenze di inizializzazione e variabili locali.
Le variabili locali sono accessibili solo tramite le procedure del monitor e non da primitive esterne.
Nel momento in cui un processo chiama una procedura del monitor entra in quest'ultimo.
Solo un processo può essere eseguito nel monitor alla volta.
Sulle variabili di condizione si può operare con due funzioni: cwait(c), che sospende il processo chiamante sulla condizione c, csignal(c), che riattiva l'esecuzione di qualche processo bloccato dopo una cwait(c) quando la condizione si verifica.
Es. bounded buffer prod/cons with monitor
NB nelle funzioni producer e consumer non avremo nessuna condizione di verifica poichè nel momento che un processo chiama append o take (procedure del monitor) automaticamente vengono gestiti i casi di buffer pieno e vuoto, inoltre la mutua esclusione è assicurata dall'implementazione stessa del monitor.

```
Monitor boundedbuffer;
char buffer[N]
int nextin, nextout;
int count;
cond notfull,notempty;

void append(char x){
        if(count == N) cwait(notfull);
        buffer[nextin] = x;
        nexttin = (nextin + 1) % N;
        count++;
        csignal(notempty);
}
void take(char x){
        if(count == 0) cwait(notempty);
        x = buffer[nextout];
        nextout = (nextout + 1) % N;
        count--;
```

```
        csignal(notfull);
}
```

MESSAGE PASSING

Il message passing è un approccio che garantisce sia sincronizzazione che comunicazione tra processi. Può essere impiegato sia su sistemi distribuiti o sistemi con architettura uniprocessor o multiprocessor.
Il tutto si basa su due primitive fondamentali: la send(destination, message) e la receive(source, message).
La sincronizzazione che si ottiene tramite questo approccio soddisfa moltissime specifica: possiamo avere delle primitive (send, receive) bloccanti o non bloccanti.
Il caso più sicuro è che entrambe siano bloccanti ossia nel momento in cui un processo A invia un messaggio al processo B, il processo A si blocca finche B non lo riceve e B si blocca finche A non invia qualcosa. Una stretta sincronizzazione che viene indicata con il nome di rendezvous.
Possiamo però optare per una non blocking send ossia viene bloccato solo il processo B ricevente, mentre il processo A una volta terminato l'invio del messaggio può continuare con il suo task.
Infine possiamo optare per un opzione non bloccante da parte di entrambe le primitive, nel caso la receive controllasse un buffer vuoto abbandonerebbe la richiesta.
Possiamo avere due tipi di addressing: diretto o indiretto.
Quello diretto identifica ogni processo con un id, da specificare ogni qual volta bisogna inviare un messaggio. Per la ricezione invece abbiamo due approcci: l'esplicito e l'implicito.
Il primo prevede la designazione di un processo sender in modo tale da poterne specificare l'id.
Il secondo invece nel parametro source della primitiva receive possiede un valore di ritorno quando è avvenuta la ricezione.
Quello indiretto invece prevede una struttura dati condivisa in cui i messaggi vengono inviati e accumulati in coda per poter essere poi prelevati dai ricevitori. La coda è definita mailboxe.
Questo approccio offre molta flessibilità poiché possiamo condividere la mailboxe tra N sender e M receiver. Ogni messaggio però conterrà una parte consistente di Header in cui verranno specificati molti parametri tra cui il tipo di messaggio, il destinatario effettivo, la lunghezza del messaggio, il codice di controllo e il messaggio vero e proprio.
Per garantire mutua esclusione in questo caso possiamo far condividere una mailboxe tra i processi in cui tutti sono receiver e tutti sono sender. Appena si ottiene un messaggio (token pass) si può entrare in critical section (receive bloccante) e poi si invia un messaggio contenente il pass per gli altri processi (send non bloccante). NB il primo messaggio lo invia il main per far partire il tutto.

4. Socket

Una socket è individuata dalla concatenazione di una porta (TCP layer) e un indirizzo IP (Network layer), quest'ultimo identifica un sistema host. Le stream sockets utilizzano TCP protocol e offrono un servizio affidabile di spedizione, le datagram sockets usano UDP protocol e la spedizione non è garantita, le raw sockets consentono l'accesso diretto ai protocolli di livello inferiore.
E' come detto un endopoint per una connessione basata su TCP/IP, il livello d'applicazione ci si collega, il programmatore si occupa delle API.
Molte porte sono riservate come quelle dalla 0 alla 1023 compresa, l'OS assegna porte a client connessi con poca durata di vita dalla 1024 alla 5000 compresa.
La Socket Address Structure è una struttura in C di tipo **sockaddr_in** in cui all'interno ci sono 5 proprietà fondamentali: **sin_len** che indica la lunghezza della struttura, **sin_family** che bisogna impostare ad **AF_INET** affinchè si possa utilizzare un IPV4, la **sin_port** numero di porta, un array **sin_zero** non utilizzato, e una struttura di tipo **in_addr** di nome **sin_addr** che contiene al suo interno l'indirizzo IP a 32 bit.
Le primitive per una connessione TCP per le sockets sono: **socket()** che crea un nuovo endpoint di comunicazione, **bind()** che associa alla socket un indirizzo locale, **listen()** che data una coda di client possibili in attesa di accettazione annuncia di essere pronta ad accettare connessioni, **accept()** che stabilisce passivamente una connessione, **connect()** che tenta di stabilire attivamente una connessione, **send()** invia dati, **receive()** riceve dati, **close()** rilascia la connessione.

Socket(): primo parametro è un intero che indica la famiglia dell'indirizzo (AF_INET = IPV4, AF_INET6 = IPV6, AF_LOCAL = local Unix), il secondo parametro è un intero che indica il tipo di socket (SOCK_STREAM, SOCK_DGRAM, SOCK_RAW), e un terzo intero che indica qualche protocollo usato nelle raw socket quindi lo piazziamo a 0. Ritorna -1 se fallisce senno l'fd della socket.

Bind(): prende un intero che è l'fd della socket, prende un puntatore a struct sockaddr (da castare) costruita in precedenza la lunghezza della struttura. Ritorna 0 se tutto apposto o -1, se errore è EADDRINUSE significa che l'indirizzo è gia usato.
Listen(): prende l'fd della socket e un intero che indica quanti client possono rimanere in attesa.
Accept(): prende l'fd della socket, , prende un puntatore a struct sockaddr (da castare) dove il client inserirà le informazioni del protocollo da lui usato e la lunghezza della struct. Ritorna un descritptor di riferimento al client o -1 se c'è stato un errore, avvia la connessione con il client.
Close(): prende solo l'fd della socket e la chiude. -1 in errore.
Connect(): prende l'fd della socket, , prende un puntatore a struct sockaddr (da castare) dove inserisce i dati del server e la sua lunghezza. Si usa nel client che non ha bisogno della bind perché è l'OS ad assegnarli la porta, ritorna l'fd della socket se va tutto bene sennò -1.
Recv(): prende fd socket, puntatore a buffer da riempire, la sua grandezza e delle flag che non ci interessano.
Send(): prende fd socket, puntatore a buffer da copiare, quanto è pieno, e delle flag che non ci interessano.

5. Deadlock

E' una situazione permanente e non ci sono soluzioni efficienti. Va evitato.
Una risorsa si dice riusabile se può essere usata da un processo alla volta in sicurezza e non è consumata dallo stesso. Si dice consumabile se può essere creata e distrutta.

CONDIZIONI PER DEADLOCK

3 condizioni necessarie ma non sufficienti:

1. Mutua esclusione: un solo processo alla volta può accedere ad una data risorsa.
2. Possesso e attesa: un processo può mantenere il possesso di risorse allocate mentre aspetta altre risorse.
3. Assenza di prelascio: Un processo non è forzato a rilasciare una risorsa in suo possesso

Un'altra condizione che risulta essere conseguenza delle prime 3 è l'Attesa circolare, ossia una catena chiusa di processi in cui tutti hanno almeno una risorsa bloccata richiesta da un altro processo in attesa.
Ci sono 3 tipi diversi di approcci verso il deadlock: uno preventivo, dove si cerca di eliminare una delle condizioni necessarie per il deadlock, uno d'esclusione, in cui dinamicamente di effettuano delle scelte in base al fatto che una richiesta potrebbe causare deadlock, e uno di detenzione, ossia captare la creazione di deadlock e cercare di ripararlo.

DEADLOCK PREVENTION STRATEGY

Creare un sistema in cui la possibilità di deadlock sia esclusa. Due differenti metodi: quello indiretto, dove si cerca di prevenire il verificarsi di una delle prime 3 condizioni necessarie, e uno diretto in cui si cerca direttamente di evitare che si formi un'attesa circolare (quarta condizione).
Nel caso in cui l'accesso ad una risorsa prevede la mutua esclusione, quest'ultima deve essere supportata dall'OS.
Per quanto riguarda il possesso e l'attesa, si può obbligare ogni processo a richiedere tutte le risorse all'inizio e nel caso le richieste non possano essere soddisfatte lo si blocca. Davvero una brutta soluzione per vari motivi: tra cui il fatto che molte risorse verrebbero bloccate in partenza per poi magari essere utilizzate alla fine, si viene anche meno alla programmazione modulare.
Per quanto riguarda la non liberazione delle risorse, ad un processo potrebbe essere rifiutate delle altre nuove richieste obbligando quel processo a rilasciare le precedenti risorse richiedendole di nuovo con l'aggiunta delle nuove risorse, oppure, l'OS potrebbe anticipare un secondo processo forzandolo a rilasciare una determinata risorsa. Si può fare questo discorso solo su risorse il cui stato è facilmente salvabile e ripristinabile.
Per l'attesa circolare si può ipotizzare di assegnare a ogni risorsa un indice naturale crescente con la condizione che si è in possesso di una risorsa i-esima si può solo richiedere un'altra risorsa j-esima con $j > i$.
Questa soluzione però rallenta moltissimo i processi impedendo l'uso di alcune risorse.

DEADLOCK AVOIDANCE STRATEGY

L'approccio prevede di prendere dinamicamente decisioni su richieste di allocazione risorse che potenzialmente potrebbero portare ad un deadlock. Questo richiede di conoscere le future richieste del processo. Questo approccio migliora la concorrenza rispetto alla deadlock prevention strategy.
Abbiamo due modalità:

1. Rifiuto del permesso di esecuzione: ossia non permettere l'avvio di un processo se la sua richiesta potrebbe portare ad un deadlock
2. Rifiuto di allocare una risorsa: non concedere un ulteriore richiesta di risorsa ad un processo se questa allocazione potrebbe portare ad un deadlock.

Negare l'avvio di un processo non è del tutto ottimale mentre il blocco dell'allocazione delle risorse potrebbe avere dei vantaggi nei confronti della deadlock detection strategy (sotto) ed è meno restrittivo della deadlock prevention strategy.
C'è però da dire che il numero massimo di risorse deve essere espresso a priori.
I processi devono essere indipendenti e senza requisiti di sincronizzazione.
Ci deve essere un numero fisso di risorse da allocare.
Nessun processo può terminare conservando le risorse precedentemente ottenute.

DEADLOCK DETECTION STRATEGY
A differenza della deadlock detection strategy la detection asseconda tutte le richieste di risorse dei processi finchè è possibile.
Possiamo decidere di effettuare un check per il deadlock molto frequentemente cioè a ogni richiesta di risorsa o meno frequentemente… dipende da come il problema potrebbe presentare un deadlock.
Un check a ogni richiesta significherebbe consumare decisamente il tempo della CPU ma attraverso algoritmi semplici è comunque molto semplice farlo.
Ci sono più metodi di recovery per una situazione di deadlock: si possono terminare tutti i processi coinvolti (il più comune approccio), salvare lo stato dei processi coinvolti in punti precedenti del loro task e riavviare tutti i processi, oppure terminare i processi in deadlock finche la situazione di deadlock non si verifichi più, oppure prevenire l'uso di alcune risorse finchè il deadlock non esista più.

SOFTWARE APPROACHES TO MUTUAL EXCLUSION
La mutua esclusione può essere implementata anche a livello software. Esempio concreto più processi comunicano tra loro tramite un'area di memoria condivisa e l'accesso a tale area deve essere in mutua esclusione… vediamo i vari tentativi e i loro problemi.

Primo tentativo

```
/* global */
int turn = 0;

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (turn != me) /* busy wait */ ;
/* CS */
turn = other;
```

Questo tentative garantisce mutual esclusione tra due processi… il problema però è che se un processo non effettua per nessun motive il proprio task settando infine il token pass con la var other, l'altro processo resterebbe in busy waiting per secoli.

Secondo tentativo

```
/* global */
boolean flag[2] = {false, false};

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (flag[other]) /* busy wait */ ;

flag[me] = true;
/* CS */
flag[me] = false;
```

Questo tentativo non garantisce mutua esclusione poiché se entrambi i processi verificassero la condizione di flag del while prima che qualcuno setti a true una delle due entrambi avrebbero accesso alla critical section.

Terzo tentativo

```
/* global */
boolean flag[2] = {false, false};

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;
flag[me] = true;
while (flag[other])  /* busy wait */ ;
/* CS */
flag[me] = false;
```

Questo tentativo invece porta ad una condizione di deadlock nel caso entrambi i processi settassero la propria flag a true. Entrambi aspetterebbero un tempo indefinito nel while.

Quarto tentativo

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

flag[me] = true;
while (flag[other]) {
flag[me] = false;
/* delay */
flag[me] = true;
}
/* CS */
flag[me] = false;
```

Questo tentativo evita il deadlock ma non il livelock.

Una corretta soluzione è l'algoritmo di Dekker.

```
int me = 0, other = 1; // P0 (flip for P1)
int turn = (uno qualsiasi all'inizio)

while (true) {
flag[me] = true;
while (flag[other]) {
if (turn == other) {
flag[me] = false;
while (turn == other)  /* busy wait */ ;
flag[me] = true;
}
}
/* CS */
turn = other;
flag[me] = false;
}
```

Il senso è: finchè ho bisogno di entrare in cs setto la mia flag a true. Poi mi chiedo ma anche l'altro processo ha bisogno di entrare o è entrato? Se la risposta è no entro in sezione critica altrimenti prendo delle precauzioni. Se il turno è il mio rifarò il controllo del ciclio while interno sulla flag dell'altro finchè non acrà settato la sua flag a false. Se non è il mio turno semplicemente mi faccio da parte e setto la mia flag a false e aspetto in busy waiting finchè l'altro non setta il turno a me, dopodichè mi rimetto interessato e riparte l'algoritmo di verifica.

L'algoritmo di Dijkstra generalizza l'algoritmo di Dekker per N processi.
Su altra pagina l'algoritmo.

```
/* global storage */
boolean interested[N] = {false, ..., false} //vettori in cui ogni posizione i corrisponde a un processo
boolean passed[N] = {false, ..., false}
```

int k = <any> // k ∈ {0, 1, ..., N-1} //sostituto della var turn alg di dekker

/* local info */
int i = <entity ID> // i ∈ {0, 1, ..., N-1} //var me dell'alg di dekker

1. interested[i] = true
2. while (k != i) {
3. passed[i] = false
4. if (!interested[k]) then k = i
}
5. passed[i] = true
6. for j in 1 ... N except i do
7. if (passed[j]) then goto 2
8. <critical section>
9. passed[i] = false; interested[i] = false

Sono interessato e piazzo la mia flag a true. Poi verifico se è il mio turno, se non fosse così dico che non sono passato e mi chiedo se quello a cui tocca è interessato, se non lo è allora mi prendo il turno e successivamente dirò di essere passato. Poi controllo se qualcun altro è passato, nel caso di esito positivo torno al punto 2.

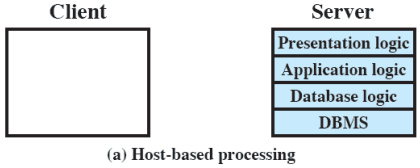
6. Client Computing

Il server abilita molti client all'accesso a database condivisi e abilita l'uso si sistemi di computer ad alte prestazioni per gestire questo database di informazioni.
Inoltre nel client ritroviamo la logica dell'applicazione (cioè come è fatta) e ovviamente abbiamo una grande cura della presentazione, ma la maggior parte del software dell'applicazione viene svolta sui server.
La chiave di una applicazione client/server è che i task dell'applicazione vengono svolti da entrambi i sistemi, mentre il loro sistema operativo o l'hardware potrebbe essere differente, ma poco importa poiché a livello di comunicazione usano lo stesso protocollo.
Le funzione effettive di una applicazione possono essere quindi divise tra client e server in modo tale da ottimizzare l'uso di risorse.

COMPONENTI DI APPLICAZIONI DISTRIBUITE
Le applicazioni orientate al business sono composte di 4 principali componenti:
- Logica di presentazione: GUI.
- Logica di I/O: inserimento di dati da parte dell'utente.
- Logica di business: servizi e calcoli improntati a un dato business
- Logica di storage di dati: conservazione dati, richiesta dati, integrità dei dati.

CLASSI DI CLIENT/SERVER APPLICATIONS
Abbiamo 4 classi generali:
1. Elaborazione basata su host
2. Elaborazione basata su server
3. Elaborazione basata sulla cooperazione
4. Elaborazione basata sul client

Le prime non sono vere e proprie applicazioni client/server, hanno più un aspetto tradizionale



(a) Host-based processing

Le seconde prevedono l'elaborazione totale dei dati da parte del server, mentre il client offre solo una logica di presentazione.

(b) Server-based processing

Le terze offrono un'elaborazione ottimizzata tra client e server, ma molto più complessa da implementare e mantere

(c) Cooperative processing

Le ultime invece prevedono l'elaborazione dei dati da parte del client, mentre le convalide e lo storage dei dati viene effettuato dal server.

(d) Client-based processing

L'architettura client/server è a tre livelli:

- User machine ossia il client.
- Un middle-tier server ossia un server intermedio che converte protocolli, fa il merge di differenti dati e altro.
- Backend server ossia lo storage vero e proprio

DEFINIZIONE DI SISTEMI DISTRIBUITI

Un sistema distribuito è un insieme di entità spazialmente separate, ognuna delle quali con un certo potere computazionale, capaci di comunicare e sincronizzarsi tra loro in funzione di un comune obiettivo sembrando ai loro utenti di essere un unico sistema coerente.

Vantaggi rispetto a sistemi centralizzati:

- **Economici**: una serie di microprocessori offre un miglior rapporto qualità/prezzo dei mainframes. Prezzi minori potenza di calcolo maggiore.
- **Velocità**: un sistema distribuito può avere una potenza maggiore di calcolo dei mainframes poiché più microprocessori ad una velocità bassa concorrono ad uno stesso risultato. Molto più ergonomico di un unico processore la cui potenza di calcolo equivalga a quella totale.
- **Distribuzione intrinseca**: molte applicazioni hanno intrinsecamente una organizzazione distribuita. Es. Catena di supermercati.
- **Affidabilità**: Se una macchina crasha, il sistema sopravvive.
- **Crescita incrementale**: Il sistema è modularmente espandibile potendo così aumentare la potenza di calcolo con piccole modifiche.
- **Un altro vantaggio**: l'esistenza di un grande numero di PC, la necessità delle persone di collaborare e scambiare informazioni

Vantaggi rispetto a PC indipendenti:

- **Condivisione di dati**: è permesso a molti utenti di accedere a dati condivisi.
- **Condivisione di risorse**: Es. costose periferiche come stampanti.
- **Comunicazione**: accresce la comunicazione tra uomini con mail, chat.
- **Flessibilità**: Distribuisce il lavoro da fare sulle macchine disponibili.

Svantaggi dei sistemi distribuiti:

- **Software**: è più difficile sviluppare software per sistemi distribuiti.
- **Network**: problemi come saturazione o perdita di pacchetti.
- **Sicurezza**: è più facile accedere a dati protetti.

Il primo obiettivo è la condivisione di dati e risorse. Ci sono due problemi da affrontare: la sincronizzazione e la coordinazione. Purtroppo ci sono delle caratteristiche che si differenziano da un sistema centralizzato che ci

limitano nel risolvere alcuni problemi di coordinazione come: concorrenza temporale e spaziale, mancanza di un orologio globale, fallimenti, latenze imprevedibili.

Problemi di progettazione dei sistemi distribuiti:

1. Trasparenza
2. Flessibilità
3. Affidabilità
4. Performance
5. Scalabilità

TRASPARENZA

Come far sembrare all'utente una serie di computer come un singolo computer. Questo obiettivo può essere raggiunto a due livelli:

1. Nascondendo la distribuzione all'utente.
2. O ad un livello più basso, fare si che il sistema sia trasparente ai programmi.

Ci sono diversi tipi di trasparenza:

- **Trasparenza della posizione**: gli utenti non possono sapere dove fisicamente si trovano le risorse.
- **Trasparenza della migrazione**: le risorse devono poter essere spostate in tranquillità senza che i loro nomi cambino.
- **Trasparenza di replica**: l'OS può effettuare copie aggiuntive di file e risorse senza che l'utente se ne accorga.
- **Trasparenza di concorrenza**: gli utenti non sono a conoscenza che altri utenti usino il sistema. E' quindi necessario gestire la mutua esclusione alle risorse.
- **Trasparenza del parallelismo**: uso automatico del parallelismo senza dover programmare esplicitamente in quella direzione. (IL SANTO GRAAL)

NB gli utenti non sempre vogliono completa trasparenza.

FLESSIBILITA'

Rende più facile fare modifiche. Il Kernel Monolitico esegue tutte le syscall che vengono catturate dal sistema. Il Microkernel invece fornisce servizi minori come la gestione della memoria, la gestione e lo scheduling di quale processo a basso livello e qualche dispositivo I/O di basso livello.

AFFIDABILITA'

I sistemi distribuiti devono essere più affidabili di sistemi singoli. Bisogna che ci sia sicurezza nell'utilizzo del sistema e soprattutto deve essere protetto da guasti o errori.

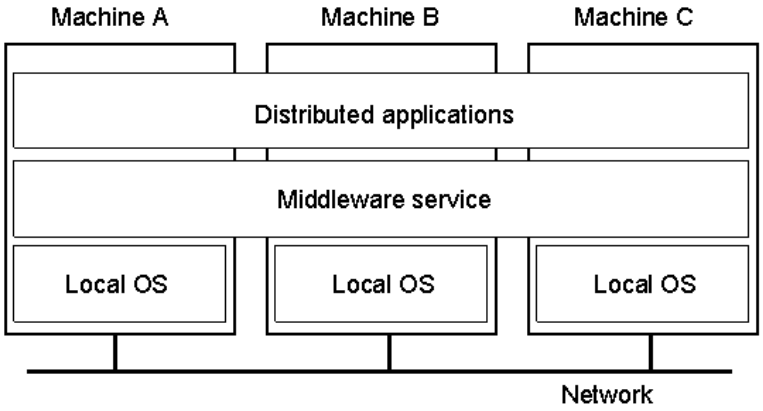
PERFORMANCE

Le performance perdono a causa della comunicazione che a volte potrebbe portare dei ritardi.
Inoltre rendere il sistema robusto ai guasti e agli errori ovviamente fa perdere di prestazioni.

SCALABILITA'

La scalabilità denota in genere la capacità di un sistema di aumentare o diminuire di scala in funzione delle necessità e disponibilità. In questo caso un aumento di potenza di calcolo deve poter migliorare tutto il sistema. Altrimenti se esiste uno o più colli di bottiglia quella miglioria potrebbe essere ininfluente.

MIDDLEWARE



Un sistema distribuito è organizzato come un middleware, il quale estende tutte le macchine che sono comprese nel sistema distribuito.

Problemi da affrontare:

- **Eterogeneità**: Os, velocità di clock, rappresentazione dei dati, memoria e HW.
- **Asincronia locale**: differente Hardware, interrupts.
- **Mancanza di conoscenza globale**: la conoscenza si propaga attraverso messaggi il cui tempo di propagazione è molto più lento dell'esecuzione di un evento interno.
- Errori di nodi o partizioni di rete.
- Mancanza di un globale ordine degli eventi.
- Coerenza vs disponibilità vs partizioni di rete.

BAKERY ALGORITHM

L'algoritmo di Dijkstra garantisce mutua esclusione, evita il deadlock, non garantisce però l'evitarsi dello starvation. Inoltre si ha bisogno di operazioni di read/write atomiche e una memoria condivisa per il parametro k (turno). Introduciamo ora l'algoritmo del panettiere.
Si pensi ad un bancone affollato di gente che vuol essere servita. Nel caso in cui non ci fosse nessuno in negozio il ticket è superfluo, altrimenti è il ticket a dettare l'ordine in cui la gente verrà servita.

Primo tentativo

```
while (1){
        /*NCS*/
        number[i] = 1 + max {number[j] | (1 <= j <= N) except i} //dorway
        for j in 1 .. N except i {
                while (number[j] != 0 && number[j] < number[i]); //bakery
        }
        /*CS*/
        number[i] = 0;
}
```

Qui il problema è che l'assegnazione del ticket a ogni processo non è descritta in maniera atomica. Questa condizione potrebbe assegnare un ticket uguali a due processi diversi.
In questo casi il risultato sarebbe che due processi entrano in CS, NO MUTUAL EXCLUSION.
Es P1 ha 1 e P2 ha 2, quando vanno a verificare la condizione del while verrà false poiché 1 < 1 false.
Se sostituisco al < il <= otterei una condizione probabile di deadlock perché entrambi aspetterebbero l'altro poiché 1 <= 1 true.

Secondo tentativo

```
while (1){
        /*NCS*/
        number[i] = 1 + max {number[j] | (1 <= j <= N) except i}
        for j in 1 .. N except i {
                while (number[j] != 0 && (number[j],j) < (number[i],i));
        }
        /*CS*/
        number[i] = 0;
}
```

La scrittura (number[j],j) < (number[i],i) significa
(number[j] < number[i] || (number[j] == number[i] && j < i))

Anche qui è possibile che due processi abbiano lo stesso ticket, la condizione nel while tratta quel caso ma questo tentativo non offre la mutua esclusione poiché potrei effettuare il for di controllo prima che ogni processo abbia preso il ticket. Inserirò quindi un array booleano per evitare ciò.

Quello finale

```
while (1){
        /*NCS*/
        choosing[i] = true;
        number[i] = 1 + max {number[j] | (1 <= j <= N) except i}
        choosing[i] = false;
        for j in 1 .. N except i {
                while (choosing[i] == true);
                while (number[j] != 0 && (number[j],j) < (number[i],i));
        }
        /*CS*/
        number[i] = 0;
}
```

Questo è l'algoritmo definitivo dove l'array choosing permette a ogni processo di entrare nella dorway in modo tale che gli sia assegnato un numero valido (anche non univoco). Un processo aspetterà finchè il valore dell'array choosing sarà false.

Caratteristiche dell'algoritmo:

- I processi comunicano attraverso variabili condivise come nell'algoritmo di Dijkstra.
- Read/write non sono operazioni atomiche.
- Tutte le variabili condivise sono di proprietà di un processo che può scriverci, gli altri possono leggerla.
- Nessun processo può eseguire due scritture simultanee
- I tempi di esecuzione non sono correlati

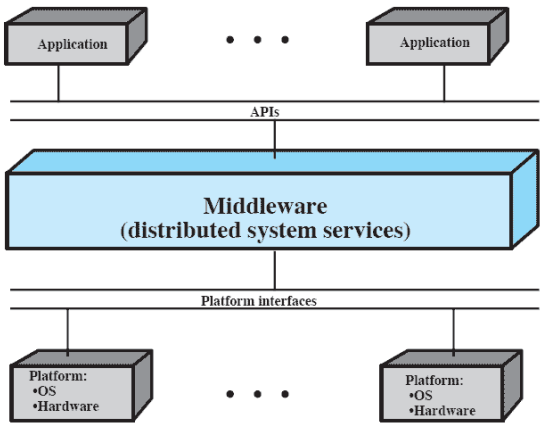
BAKERY ALGORITHM IN CLIENT/SERVER APP

```
while (1){ //client thread
        /*NCS*/
        choosing = true; //doorway
        for j in 1 .. N except i {
                send(Pj,num);
                receive(Pj,v);
                num = max(num,v);
        }
        num = num+1;
        choosing = false;
        for j in 1 .. N except i {  //backery
        do{
                send(Pj,choosing);
                receive(Pj,v);
        }while (v == true);
        do{
                send(Pj,v);
                receive(Pj,v);
        }while (v != 0 || (v,j) < (num,i));
        }
        /*CS*/
        num = 0;
}

while (1){ //server thread
        receive(Pj,message);
         if (message is a number) send(Pj,num);
         else send(Pj,choosing);
 }
```

MIDDLEWARE

Per ottenere i veri benifici dell'approccio client/server su sistemi distribuiti gli sviluppatori devono avere una serie di strumenti che permettono loro di avere un mezzo e uno stile di accesso uniforme alle risorse di sistema su tutte le piattaforme.
Ciò consentirebbe ai programmatori di costruire applicazioni basate su un sistema che sembra lo stesso per tutti.
Permette ad esempio di accedere ai dati dovunque essi siano localizzati con un metodo univoco.
Per fare ciò bisogna utilizzare interfacce di programmazione standardizzate e protocolli che si interpongono tra l'applicazione e il sistema operativo. L'insieme delle interfacce e dei protocolli prende il nome di middleware.



SERVICE-ORIENTED ARCHITECTURE (SOA)
Una forma di architettura client/server utilizzata nei sistemi aziendali.
Organizza le funzioni aziendali in una struttura modulare invece che applicazioni monolitiche per ciascun dipartimento. Di conseguenza le funzioni comuni possono essere utilizzate da diversi reparti interni e anche da partner commerciali esterni. E' costituito da un insieme di servizi e un insieme di app client che ne fanno uso. Le interfacce standardizzate sono usate per permettere la comunicazione tra i moduli di servizio e tra questi ultimi e le applicazioni.
Ogni dispositivo ha la propria applicazione che può accedere a determinati servizi, anche in comune utilizzando un interfaccia standardizzata e attraverso un Service Broker (mediatore).

MESSAGGE PASSING NEI SISTEMI DISTRIBUITI
Una comunicazione molto semplice: solo due primitive sono richieste, Send e Receive. Il processo che invia usa il modulo del message-passing standardizzato che sarà quello che tradurrà la richiesta al processo ricevente. Attraverso quindi il servizio messo a disposizioni del sistema intero dal modulo del message passing si ha un implementazione del protocollo di comunicazione.

AFFIDABILITA' VS INAFFIDABILITA'
La struttura del message-passing può essere "affidabile" ossia utilizza protocolli che prevedono il check dell'errore, ACK, la ritrasmissione e il riordino di messaggi spaiati. Siccome la spedizione è garantita non è necessario far sapere al processo inviante che la spedizione è andata a buon fine, comunque sia non è affatto sbagliato far tornare un ACK di conferma. Nel caso in cui, network down per una serie di motivi, la spedizione non dovesse andare a buon fine il processo inviante sarebbe avvisato di tutto.
All'altro estremo invece potremmo avere una struttura "inaffidabile" ossia il messaggio potrebbe essere spedito tranquillamente senza interesse nel sapere una condizione di successo o meno. Questa alternativa riduce enormemente sia la complessità che il processamento della spedizione e in più farebbe diminuire l'overhead del messaggio. Per quelle applicazioni che necessitano di conferma di spedizione, si potrebbe inviare una richiesta e rispondere ad essa con stesso un messaggio.

BLOCKING VS NONBLOCKING
Con un approccio nonblocking, un processo non è bloccato nel momento in cui fa uso di primitive di comunicazione (Send e receive). I processi fanno un uso efficiente e flessibile della struttura del message passing. E' più difficile testare e debuggare programmi che usano primitive nonblocking poiché le sequenze di invio possono creare problemi difficili e sottili.
Con un approccio blocking invece un processo che invia viene bloccato finchè il messaggio non è stato inviato ed è stata ricevuta la conferma di spedizione e in ricezione il processo è bloccato finchè il messaggio non è stato conservato in un buffer.

REMOTE PROCEDURE CALLS
Una variazione del modello basato sul message passing è il moeddo RPC. E' ormai ampiamente accettato e comunemente utilizzato per la comunicazione nei sistemi distribuiti. L'essenza è di permettere ai programmi su differenti macchine di interagire tra loro utilizzando semplici chiamate e ritorni come se stessero sulla stessa macchina.
Vantaggi dell'RPC:

1. E' ampiamente accettato, usato, ed è coerente con l'astrazione dei sistemi distribuiti.
2. L'uso del RPC abilita la specifica delle interfacce remote come un insieme di operazioni nominate con tipo designati. Si può quindi documentare facilmente le interfacce e sui programmi distribuiti è facile ora fare debugging su compatibilità del tipo.
3. Siccome le interfacce definite e standardizzate sono ora specificate, il codice di comunicazione per l'applicazione può essere generato automaticamente.
4. Siccome le interfacce definite e standardizzate sono ora specificate, gli sviluppatori possono scrivere moduli client e server che possono essere trasferiti attraverso pc e sistemi operativi con piccole modifiche e puntualizzazioni.

Il meccanismo dell'RPC può essere visto come un riferimento a un approccio affidabile di un modulo message passing bloccante.

Un sistema client può effettuare chiamate a funzioni definite localmente oppure affidarsi a chiamate di funzioni che invocano il meccanismo di RPC (può essere o non essere trasparente questa intenzione nei confronti dell'utente). Una volta che quest'ultimo è stato invocato viene contattato il server attraverso la creazione di un messaggio che contiene il nome del servizio e i parametri in dotazione. Il server, una volta computata la risposta, sempre attraverso il meccanismo di RPC ossia creando un messaggio, consegnerà il risultato al Client chiamante che lo conserverà in una variabile di ritorno.
NB la chiamata che il server fa al metodo è equivalente ad una chiamata locale, è giusto quindi affermare che i parametri verranno recuperati in stack.

PARAMETER PASSING
Passare dei parametri per valore è molto semplice, il valore stesso è copiato nel messaggio da inviare.
Molto più complicato è passare un parametro per riferimento poiché è necessario un unico puntatore ad un unico ampio sistema per ogni oggetto. Inoltre il peso dell'overhead potrebbe non valerne la pena.

PARAMETER REPRESENTATION
Un altro problema è come rappresentare parametri e valori di ritorno nei messaggi. Nel caso in cui il programma chiamante e quello chiamato siano scritti nello stesso linguaggio di programmazione e siano sullo stesso tipo di macchina con lo stesso OS non ci sono problemi. Ma se differiscono in qualcosa potrebbero esserci incompatibilità. Una soluzione potrebbe essere quella di standardizzare alcuni formati primitivi di comuni oggetti come interi e caratteri così da poter convertire tutti i tipi di parametri specifici in tipi standardizzati.

CLIENT/SERVER BINDING
Il binding specifica come la relazione tra la RPC e il programma chiamante sarà stabilita.
Abbiamo il **NONPERSISTENT BINDING**, ossia una connessione logica è stabilita tra due processi nel momento in cui si ha una chiamata ad una procedura remota ed è subito smontata nel momento in cui il valore di ritorno è stato consegnato. Siccome l'overhead coinvolto nello stabilire la connessione è importante, questo rende inappropriato l'uso di un nonblocking binding per RPC chiamate frequentemente dallo stesso chiamante.
Il **PERSISTENT BINDING** invece consiste in una connessione stabilita al momento della chiamata e mantenuta al momento della consegna del valore di ritorno. Potrà così essere riutilizzata per chiamate future. Se poi per un periodo di tempo specifico non vengono rilevate attività la connessione è smontata.
E' consigliata per applicazioni che fanno più volte chiamate a RPC, risparmiando sull'overhead di stabilimento di connessione.

SYNCHRONOUS VS ASYNCHRONOUS
Il concetto di sincrono e asincrono sulle RPC è simili a quello di blocking e nonblocking sui message passing.
L'approccio tradizionale è quello blocking cioè sincrono che prevede che il processo chiamante aspetti il valroe di ritorno della RPC. Il comportamento della RPC sincrona è prevedibile. Tuttavia non si riesce a sfruttare a pieno il parallelismo offerto dal sistema distribuito, di conseguenza avremo prestazioni inferiori dovute appunto a limiti di interazioni.
In alternativa abbiamo l'approccio nonblocking o asincrono che a sua volta non blocca il chiamante potendo quindi ricevere le risposte solo quando se ne ha il bisogno. Si ha quindi la possibilità di usare il client localmente in completo parallelismo con l'invocazione del server.

MECCANISMO ORIENTATO AGLI OGGETTI
Client e server si inviano messaggi attraverso oggetti.
Un client che ha bisogno di un servizio invia una richiesta ad un object broker (intermediario).
Il broker chiama l'appropriato oggetto e passa qualsiasi dato rilevante.
L'oggetto remoto risolve la richiesta e risponde al broker che ritornerà la risposta al client.
Il successo di questo approccio è dipeso dalla standardizzazione del meccanismo degli oggetti.

SEMANTICA DELLA CHIAMATA RPC
Il normale funzionamento di una RPC potrebbe essere interrotto:
- Il messaggio di chiamata o di risposta è perso.
- Il nodo chiamante si arresta in modo anomalo e viene riavviato
- Il nodo chiamato si arresta in modo anomalo e viene riavviato

La semantica della chiamata determina quanto spesso la procedura remota può essere eseguita sotto condizioni di errore.

At Least Once: Questa semantica garantisce che la chiamata è eseguita una o più volte ma non specifica quali risultati sono tornati al chiamante. Ha poco overhead ed è molto semplice da implementare: usando un timeout basato sulla ritrasmissione senza considerare le chiamate che sono crashsate sul server, il client continua ad inviare una richiesta al server finché non riceve un ACK. Se uno o più ACK vanno persi il server ptorebbe eseguire la chiamata più volte. Funziona per oeprazioni idempotenti. (non so che vuol dire)

At Most Once: Questa semantica garantisce che la chiamata RPC è eseguita al massimo una volta. O viene eseguita una sola volta o non viene eseguita affatto, dipende dal server se crasha o meno. A differenza della semantica precedente questa semantica richiede il rilevamento di pacchetti duplicati e funziona per operazioni non idempotenti.

Exactly once: questa semantica garantisce che la chiamata RPC avvenga, supponendo che il server se crashato venga riavviato prima o poi. Tiene conto delle chiamate orfane (quelle in cui il server è crashato) e consente loro di essere ripetute, un nuovo server le adotterà. Richiede però un'implementazione molto complessa.

CLUSTERS

E' un'alternativa al symmetric multiprocessing (SMP), possiamo definire un cluster come un gruppo di computer interconnessi che lavorano insieme come se fossero un'unica unità.
Ogni computer può eseguire operazioni in proprio e non solo in riferimento al cluster. Infatti ci si riferisce a ogni pc come nodo.
I benefici sono:

- **Assoluta scalabilità**: è possibile creare cluster così grande da superare la potenza della più grande macchina singola.
- **Scalabilità incrementale**: piccoli incrementi dipende dall'aggiunta di nuovi sistemi al cluster.
- **Alta disponibilità**: il fallimento di un nodo non è una condizione critica del sistema.
- **Alto rapporto prezzo/performance**: usando componenti a costo inferiore è possibile costruire un sistema la cui potenza in una macchina singola costerebbe molto di più.

METODI DI CLUSTERING: BENEFICI E LIMITAZIONI

Passive Standby: prevede un server secondario che si attiva in casi di fallimento del principale, facile da implementare ma ha un costo maggiore perché non è usato nel processamento di altri task.
Active Secondary: il server secondario è anche usato per processare task. Riduce i costi rispetto al precedente metodo ma aumenta la complessità.
Server separati: Server separati hanno i propri dischi. I dati sono continuamente copiati dal primo al secondo server. Altà disponibilità ma alto anche l'overhead nelle operazioni di copia.
Server connessi a dischi: i server sono collegati agli stessi dischi ma ognuno ha i propri, nel caso uno fallisse i suoi dischi verrebbero presi da un altro server. Riduce l'overhead di copia ma necessita di tecnlogia RAID per evitare fallimenti dovuti a errori nei dischi.
Server che condividono dischi: molti server condividono l'accesso ai dischi. Richiede un software che gestisca l'accesso ed è spesso usato con tecnlogia RAID.

PROBLEMI DI PROGETTAZIONE DEL SISTEMA OPERATIVO

Per quanto riguarda la GESTIONE DEGLI ERRORI esistono prevalentemente due approcci:
Utilizzo di cluster ad alta disponibilità:

- Offre un'alta probabilità che tutte le risorse saranno in servizio.
- Ogni richiesta persa, se riprovata, sarà gestita da un altro pc nel cluster.
- L'OS del cluster non da garanzie riguardo lo stato di transizioni eseguite parzialmente.
- Se si verifica un errore, le richieste in corso vengono perse

Cluster che gestiscono gli errori:

- Assicura che tutte le risorse sono sempre disponibili grazie all'uso di dischi condivisi ridondanti e meccanismi per il ritorno di transazioni non richieste e transazioni richieste completate.

La funzione di cambiare sistema ad un applicazione o alle risorse nel momento in cui quello in uso fallisce si chiama **FALLOVER**.
Il ripristino delle applicazioni e delle risorse sul sistema originale una volta riparato si chiama **FALLBACK**.

Quest'ultimo approccio può essere automatizzato ma lo si consiglia solo se realmente il problema può essere riparato ed è improbabile che si ripeta.
Il fallback automatico può scatenare una sequenza di errori che porterebbero le risorse a rimbalzare da un pc ad un altro rendendo difficile il recupero e diminuendo drasticamente le prestazioni.

Per quanto riguarda il **LOAD BALANCING**, un cluster richiede una capacità efficace di bilanciare il carico del lavoro tra il PC disponibili. Ciò ovviamente richiede il fatto che sia incrementalmente scalabile. Quando un nuovo PC viene aggiunto al cluster, l'impianto di bilanciamento del carico dovrebbe automaticamente comprendere la nuova unità. Il middleware deve riconoscere che i servizi possono apparire su diversi nodi del cluster e che possono migrare da un nodo ad un altro.

Per quando riguarda il **PARALLELIZING COMPUTATION**, esistono tre approcci:

1. **Parallelizing compiler**: determina, in tempo di compilazione, quali parti dell'applicazioni possono essere eseguite in parallelo. Queste poi sono divise su diversi pc nel cluster. Le prestazioni dipendono dalla natura del problema e se il compilatore è ben progettato.
2. **Parallelized application**: il programmatore scrive l'applicazione dall'inizio con l'intento di runnarla su un cluster, utilizzando ad esempio message passing per lo scambio di dati tra i nodi del cluster. E' l'approccio più rognoso per il programmatore ma quello che sfrutta al meglio la potenza e la struttura del cluster.
3. **Parametric computing**: si può utilizzare se l'essenza dell'applicazione è un algoritmo o un programma che deve essere eseguito un grande numero di volte, ogni volta con parametri e condizioni differenti. Esempio un modello di simulazione che effettua un test con differente parametri in ingresso. Sono importanti però tool di gestione dei test che una volta distribuito il carico su vari pc, raccolgano i risultati e li confrontino.

Ogni pc è connesso all'altro tramite una rete LAN, ogni pc ha unità di calcolo indipendente e il middleware permette di nascondere all'utente finale le modalità del lavoro effettuato dal sistema distribuito. Su di esso possono essere eseguite sia applicazioni sequenziali che parallele. Quest'ultime con prestazioni decisamente più alte sfruttando la struttura del cluster.

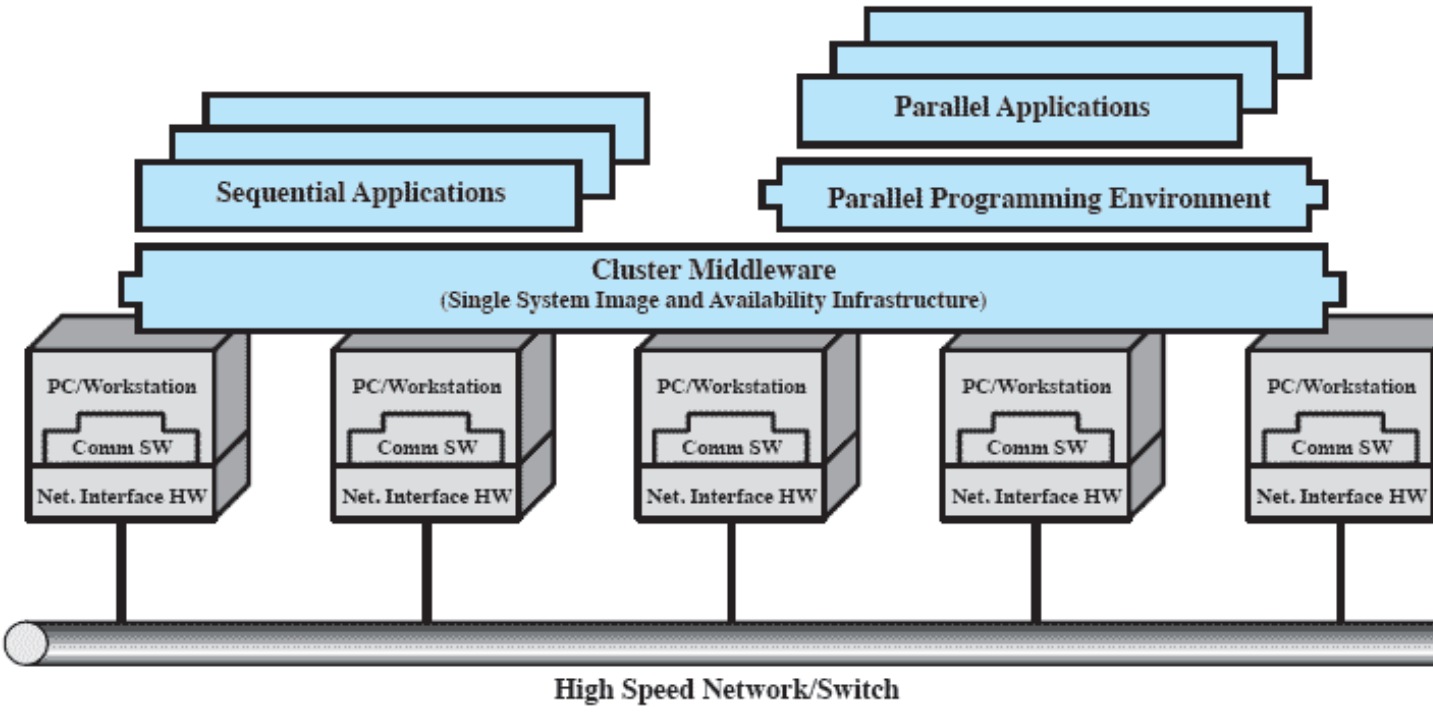


Figure 16.14 Cluster Computer Architecture [BUYY99a]

COMPARAZIONE TRA CLUSTER E SYSTEM MULTIPROCESSOR

- Entrambi forniscono una configurazione multi-proccessor per supportare applicazioni che richiedono prestazioni elevate.
- Entrambe le soluzioni sono commercialmente disponibili.
- SMP "ha più esperienza" (has been around longer than cluster").
- SMP è più facile gestirlo e configurarlo.
- I prodotti SMP sono ben consolidati e stabili.
- I cluster sono migliori per migliorie e scalabilità
- I cluster sono superiori in termini di disponibilità.

7. Pipe e FIFO

La comunicazione avviene in modo unidirezionale.
Le PIPE a livello di OS non sono altro che buffer di dimensione solitamente introno ai 4kb.
Le PIPE vengono utilizzate nella comunicazione di processi relazionati, cioè conseguenti ad una fork, altrimenti si usano le FIFO (named PIPE).
Per scriverci si usano i descrittori a mo' di semplici file.
Le PIPE sono dispositivi logici e pertanto la "chiusura del file" per convenzione avviene quando tutti coloro che condividevano il descrittore di scrittura fd[1] lo hanno chiuso.
Allo stesso modo uno scrittore che tenti di scrivere su una PIPE quando tutte le copie di fd[0] sono state chiuse riceverà un segnale SIGPIPE (broken PIPE).
Per fare in modo che tutto funzioni e non si verifichino deadlock, ogni processo DEVE chiudere il descrittore che non gli compete assicurando quindi una modularità e una correttezza di debug da parte della read e della write, altrimenti si andrebbe come detto in deadlock, ossia un lettore o uno scrittore aspetterebbe invano una risorsa da un partner che semplicemente non c'è.

Fifo

Come detto le FIFO si utilizzano per comunicazione tra processi non relazionati. Una FIFO (file speciale) viene creato con mkfifo() passando il nome e la modalità di creazione, ritorna -1 in caso di fallimento, altrimenti 0. RICORDA di fare l'unlink() per eliminare presenza della FIFO nell'OS.
Una volta creata va aperta con la open() specificando se in lettura o scrittura. L'apertura però è bloccante, se la si tenta di aprire in lettura si rimane bloccati finchè qualcuno non la apre il scrittura. Utilizzando O_NONBLOCK questo comportamento viene inibito. Scrivere su una FIFO che non ha lettori si riceve SIGPIPE dall'OS.

8. Tempo logico

In un sistema distribuito è impossibile avere un unico clock fisico condiviso da tutti i processo.
Soluzione 1:

- Si può decidere di sincronizzare con una certa approssimazione i clock fisici locali attraverso opportuni algoritmi.
- Ogni processo allega un timestamp contenente il valore del proprio clock fisico ad ogni messaggio che invia.
- I messaggi vengono così ordinati in funzione di un timestamp crescente.

Ma l'approssimazione dei clock non sempre si può mantenere limitata: ad esempio in un modello asincrono NO (si dice che un sistema distribuito è asincrono quando non è possibile stabilire un upper bound al tempo di esecuzione di ciascun step di un processo, al tempo di propagazione di un messaggio di rete, alla velocità di deriva di un clock).
Soluzione 2:

- Il timestamping avviene etichettando gli eventi con il valore corrente di una variabile opportunamente aggiornata durante la computazione.
- Questa variabile, poiché completamente scollegata dal comportamento del clock fisico locale, è chiamata clock logico.

Questa soluzione, non basandosi sul concetto di tempo reale, è adatta ad un uso in sistemi asincroni.

CLOCK FISICO

All'istante di tempo reale t, il sistema operativo legge il tempo dal clock hardware $H_i(t)$ del computer, quindi produce il software clock **: $C_i(t) = \alpha H_i(t) + \beta$** che approssimativamente misura l'istante di tempo fisico t per il processo P_i.
Il clock hardware $H_i(t)$ è un componente fisico (oscillatore al quarzo) caratterizzato da un parametro detto **DRIFT RATE**:

- L'oscillatore è caratterizzato da una sua frequenza che fa divergere il clock hardware dal tempo reale.
- Il drift rate misura il disallineamento del clock hardware da un orologio ideale per unità di tempo.

Un clock hw $H_i(t)$ è corretto se il suo drift rate si mantiene all'interno di un limite $\rho > 0$ finito. (es. 10^{-6} secs/sec).
Se il clock $H_i(t)$ è corretto allora l'errore che si commette nel misurare un intervallo di istanti reali [t,t'] è limitato: $(1 - \rho)(t' - t) \leq H_i(t') - H_i(t) \leq (1 + \rho)(t' - t)$ supposto $(t<t')$

Per il clock software Ci(t) spesso basta una condizione di monotonicità: $t' > t$ implica $C_i(t') > C_i(t)$.
Si potrebbe garantire monotonicità con un clock hw non corretto scegliendo opportunamente i valori α e β.
La risoluzione del clock è il periodo che intercorre tra gli aggiornamenti del valore del clock.
Ci chiediamo quanto deve essere la risoluzione per distinguere due differenti eventi.
NB il tempo di risoluzione < intervallo di tempo che intercorre tra due eventi rilevanti.

Clock guasto: se non rispetta le condizioni di correttezza. NB clock corretto ≠ clock accurato.
L'UTC (Coordinated Universal Time) è uno standard internazionale per mantenere il tempo.
E' basato su orologi atomici ma occasionalmente aggiustato utilizzando il tempo astronomico.
L'output dell'orologio astronomico è inviato broadcast da stazioni radio su terra e satelliti.
Computer con ricevitori possono sincronizzare i loro clock con questi segnali.

SINCRONIZZAZIONE DEI CLOCK FISICI

Possiamo sincronizzare i clock di processi appartenenti a sistemi distribuiti attraverso due differenti strategie:

- Sincronizzazione esterna: i clock sono sincronizzati con una sorgente di tempo S, in modo che, dato un intervallo I di tempo reale: $|S(t) - C_i(t)| < D$ per $i = 1,\ldots,N$ e per tutti gli istanti t in I, cioè i clock Ci hanno un'accuratezza compresa nell'intervallo D.
- Sincronizzazione interna: i clock di due computer sono sincronizzati l'uno con l'altro in modo che $|S(t) - C_i(t)| < D$ per $i = 1,\ldots,N$ nell'intervallo I. In questo caso i due clock Ci e Cj si accordano all'interno dell'intervallo D.

I clock sincronizzati internamente non sono necessariamente esternamente sincronizzati. Tutti i clock possono deviare collettivamente da una sorgente esterna sebbene rimangano sincronizzati tra loro entro l'intervallo D.
Se l'insieme dei processi è sincronizzato esternamente entro un intervallo D allora segue che è anche internamente sincronizzato entro un intervallo 2D.

TIME SERVICES

Il gruppo di processi che deve sincronizzarsi fa uso di un Time Service. Il Time Service può essere a sua volta implementato da un solo processo (server) oppure può essere implementato in modo decentralizzato da più processi:

Time Service centralizzato:

- **Request-driven (algoritmo di Cristian) – sync esterna**
- **Broadcast-based (Berkeley Unix algorithm) – sync interna**

Time Service decentralizzato:

- **Network Time Protocol – sync esterna**

ALGORITMO DI CRISTIAN

Time server centralizzato e passivo:

- Il time server S riceve il segnale da una sorgente UTC.
- Un processo P richiede il tempo con m_r e riceve t in m_t da S.
- P imposta il suo clock a t+ RTT/2.

NB RTT è il round trip time misurato tra P e S, si divide con il 2 perché a metà viaggio verrà inviato il valore di t, quindi il viaggio della richiesta viene ignorato.

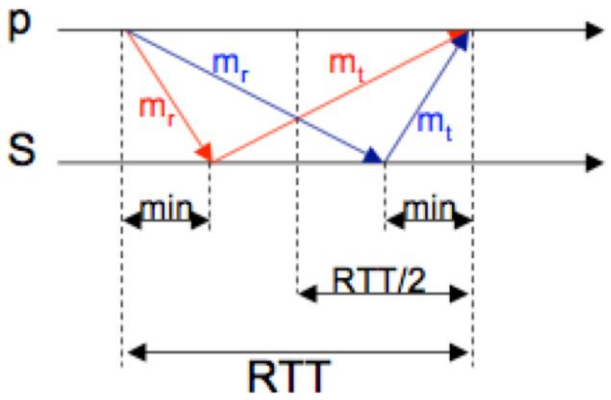
Accuratezza nell'algoritmo di Cristian:

- Caso 1: il tempo impiegato dal messaggio di ritorno è maggiore rispetto alla sima fatta utilizzando RTT/2 ed in particolare è uguale a RTT-min (dove min è il tratto di andata in questo caso, cioè quello più breve).

Δ = stima del tempo del messaggio di ritorno - tempo reale = (RTT/2) - (RTT - min) =
-RTT/2 +min = - (RTT/2 - min)

- Caso 2: il tempo impiegato dal messaggio di ritorno è minore rispetto alla stima fatta utilizzando RTT/2 ed in particolare è uguale a min (essendo il ritorno in questo caso il tratto più breve).

Δ = stima del tempo del messaggio di ritorno - tempo reale = (RTT/2) - min =
+ (RTT/2 - min)



Il tempo di S quando m_t arriva a P è compreso nell'intervallo [t+min, t+RTT-min].
L'ampiezza di tale intervallo è RTT – 2min.
Accuratezza ≤ ± (RTT/2 – min).

ALGORITMO DI BERKLEY

Algoritmo per la sincronizzazione interna di un gruppo di computer:

- Il time server è centralizzato e attivo (master).
- Il master invia a tutti gli altri processi (slaves) il proprio valore locale e richiede gli scostamenti dei loro clock da questo valore.
- Riceve tutti gli scostamenti e usa il RTT per stimare il valore del clock di ciascun slave.
- Calcola il valor medio.
- Invia a tutti gli slaves gli scostamenti necessari per sincronizzare i clock.

Guasti: se un master va in crash un'altra macchina viene eletta master, è tollerante a comportamenti arbitrari cioè se uno slave invia un valore di clock corrotto il master non lo considera nella somma poiché è impostato a considerare valori in un certo intervallo prefissato.
NB non si può pensare di imporre un valore di tempo passato ai processi slave con un valore di clock superiore a quello calcolato come valore di clock sincrono:

- Ciò provocherebbe un problema di ordinamento causa/effetto di eventi e verrebbe violata la condizione di monotonicità del tempo.
- La soluzione è quella di mascherare una serie di interrupt che fanno avanzare il clock locale in modo da rallentare l'avanzata del clock stesso.

NETWORK TIME PROTOCOL

Sincronizza client a UTC. Architettura: disponibile e scalabile poiché vengono usati server multipli e path ridondanti.
Le sorgenti di tempo sono autentiche.
La sottorete di sincronizzazione (tutti i livelli che includono i server) si riconfigura in caso di guasti:

- Un primary che perde la connessione alla sorgente UTC può diventare un server secondario.
- Un secondario che perde la connessione al suo primary (crash di questo) può usare un altro primary.

Modalità di sincronizzazione (sempre via UDP):

- **Multicast:** un server manda in multicast il suo tempo agli altri che impostano il tempo ricevuto assumendo un certo ritardo prefissato. Non molto accurato in mancanza di multicast hardware, poiché passa del tempo da un invio all'altro.

- **Procedure call**: un server accetta richieste da altri computer (come algoritmo di Cristian). Alta accuratezza. Utile se non si dispone di multicast hw.
- **Simmetrico**: coppie di server scambiano messaggi contenenti informazioni sul timing. Usata quando è necessaria un'accuratezza molto alta, ossia per gli alti livelli di gerarchia.

Con il metodo simmetrico bisogna tenere in considerazione i ritardi dei canali:
Per ogni coppia di messaggi scambiati tra i due server, NTP stima un offset O (o grande) tra i 2 clock ed un ritardo D (ritardo di trasmissione per i 2 msg).
Immaginiamo che o (o piccolo) sia il vero offset tra i due clock.
t2 = t1 + d1 + o e t4 = t3 + d2 – o dove o (o piccolo) è il vero offset, t1 il tempo di invio del primo messaggio, t3 quello del secondo, t2 il tempo di arrivo del primo messaggio, t4 del secondo, d1 e d2 i tempi di propagazione.
Da cui D = d1 + d2 = (t2-t1) + (t4-t3)
Se sottraggo t2 – t4 = (t1 + d1 + o) – (t3+d2-o) = t1 + d1 – t3 –d2 + 2o
Da cui o=((t2-t1) + (t3-t4)) / 2 + (d2-d1) / 2
Siccome il primo membro dell'addizione di o è proprio O posso dire che o = O + (d2 – d1)/2
L'offset stimato O ha quindi un errore massimo pari a ± (d2 – d1)/2.

TEMPO LOGICO
In alcune applicazione potrebbe interessare solo la sequenza degli eventi e non l'istante preciso. Per cui si fa riferimento al tempo logico.
Nel caso in cui appunto non si può sincronizzare i clock dei processi possiamo ordinare i messaggi sull'assunzione di due fatti:
- Due eventi che accadono sullo stesso processo possono sempre essere ordinati
- Un evento di ricezione di un messaggio segue sempre l'evento di invio del messaggio stesso

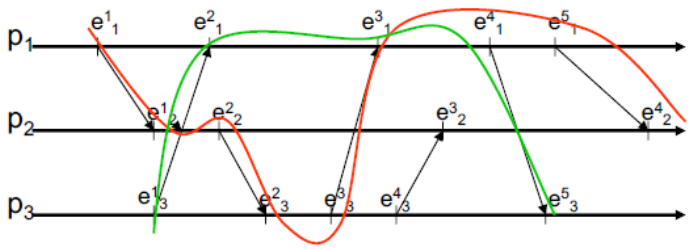
Gli eventi possono essere quindi ordinati secondo una nozione di causa-effetto.

Lamport introdusse la relazione che cattura le dipendenze causali tra gli eventi:
- Denotiamo con $->_i$ la relazione di ordinamento tra eventi in un processo Pi
- Denotiamo con -> la relazione "avvenuto-prima" tra eventi pari

NB Due eventi avvenuti in un processo Pi accadono nello stesso ordine in cui il processo li osserva, quando Pi invia un messaggio a Pj l'evento send avviene prime dell'evento receive.

DEFINIZIONE DELLA RELAZIONE AVVENUTO-PRIMA
Due eventi e ed e' sono in relazione attraverso una relazione avvenuto-prima (e -> e') se:
- Esiste un processo Pi tale che e ->i e'
- Comunque scegli un messaggio m $e_{send(m)}$ -> $e_{receive(m)}$ (send(m) è l'evento di invio del messaggio m e receive(m) è l'evento di ricezione dello stesso messaggio)
- Esiste e, e'', e''' tale che (e -> e'') Λ (e'' -> e''') (questa relazione è transitiva)

Usando queste tre regole è possibile definire una sequenza ordinata causalmente degli eventi e1, e2, …, en
NB
La sequenza e1, e2, …, en può non essere unica, potrebbe esistere una coppia <e1, e2> che non sono in relazione avvenuto-prima. Essi si dicono concomitanti (e1 || e2).
In un sistema distribuito per ogni coppia di eventi <e1, e2> deve poter essere vera una di queste condizioni:
1. e1 -> e2
2. e2 -> e1
3. e1 || e2



e^j_i is j-th event of process p_i

$S_1 = < e^1_1, e^1_2, e^2_2, e^2_3, e^3_3, e^3_1, e^4_1, e^5_1, e^4_2 >$

$S_2 = < e^1_3, e^2_1, e^3_1, e^4_1, e^5_3 >$

Note: e^1_3 and e^1_2 are concurrent

Questo esempio mostra come individuare una sequenza di relazioni happened-before.

LOGICAL CLOCK

Il clock logico, introdotto da Lamport, è un contatore software monotonicamente crescente, il cui valore non ha alcuna relazione con il clock fisico. Ogni processo Pi ha il proprio clock logico Li, usato per applicare i timestamp agli eventi.

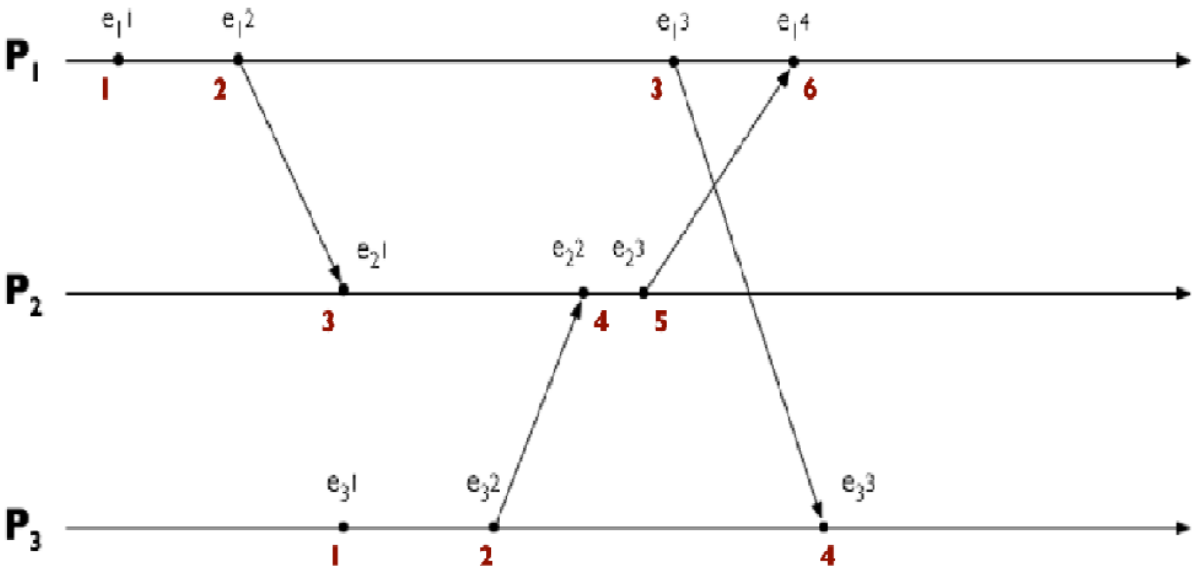
Denotiamo con Li(e) il timestamp, basato sul clock logico, applicato dal processo Pi all'evento e.

Proprietà: Se e -> e' allora L(e) < L(e').

Osservazione: si ottiene un ordinamento parziale, non è detto che guardando i timestamp di due eventi si riesca a capire in che relazione sono, cioè se L(e) < L(e') non è detto che e -> e'.

Un'implementazione:

- Ogni processo Pi inizializza il proprio clock logico Li a 0.
- Li è incrementato di 1 prima che il processo Pi esegua l'evento(interno o esterno che sia). Li = Li + 1.
- Quando Pi invia il messaggio m a Pj:
    1. Incrementa il valore di Li
    2. Allega al messaggio m il timestamp t = Li
    3. Esegue l'evento send(m)
- Quando Pj riceve il messaggio m con timestamp t
    1. Aggiorna il proprio clock logico Lj = max(t,Lj)
    2. Incrementa il valore di Lj
    3. Esegue l'evento receive(m)



NB e_i^j significa evento j-esimo del processo i-esimo.

Possiamo notare come e_1^4 e e_3^3 non sono in relazione happened-before, sono quindi concomitanti e hanno valore di timestamp diverso.
Mentre e_1^1 e e_3^1 sono concomitanti e hanno timestamp uguale.

Limiti del clock logico scalare:

- Garantisce questa proprietà: se e -> e' allora L(e) < L(e')
- Ma non è possibile garantire questa: if L(e) < L(e') allora e -> e'
- Di conseguenza non è possibile determinare, analizzando sono il clock scalare, se due eventi sono concorrenti o correlati da una relazione happened-before.

CLOCK VETTORIALE
Ad ogni evento e viene assegnato un vettore V(e) di dimensione pari al numero dei processi con la seguente proprietà: e -> e' ⇔ V(e) < V(e').

Che significato diamo al segno < tra due vettori?
V(e) < V(e') se e solo se:
comunque prendi x appartenente a {1, …, N} V(e')[x] >= V(e)[x] e esiste un x appartenente a {1,…,N} tale che V(e')[x] > V(e)[x]. (Cioè tutti gli elementi di V(e') devono essere maggiori o uguali ma comunque sia uno deve per forza maggiore!).

Comparare i valori di due clock vettoriali associati a due eventi distinti permette di capire la relazione che lega i due eventi:

1	1
2	2
0	2
V(e)	V(e')

$\Rightarrow \quad e \mapsto e'$

1	1
2	0
0	2
V(e)	V(e')

? $\quad e \,||\, e'$

Ogni processo Pi gestisce un vettore di interi Vi di n componenti Vi[1…n] ovvero una per ogni processo.
La componente Vi[x] indica la stima che il processo Pi fa sul numero di eventi eseguiti dal processo Px.
Il vettore è inizializzato a [-,-,…,0,…,-], 0 all'i-sema posizione, cioè quella di ogni processo.
Vi[i] rappresenta infatti il logical clock di Pi.

Il vettore viene aggiornato in base a queste regole:

- Quando Pi esegue un evento incrementa Vi[i] di un unità e poi associa un timestamp T all'evento il cui valore è pari al valore corrente di Vi.
- Quando Pi esegue un evento di invio messaggio, allega al messaggio il timestamp di quell'evento ottenuto dalla regola precedente.
- Quando arriva un messaggio a Pi con un timestamp T Pi esegue la seguente operazione:

comunque preso x appartenente a [1…n] Vi[x] = max(vi[x],Tx]) ossia prendo il massimo tra le due componenti dei due vettori.

- Quindi esegue l'evento di consegna ed esegue ovviamente la prima regola incrementando il proprio logical clock.

RICART-AGRAWALA ALGORITHM (mutual exclusion in distributed setting)
I clock scalari possono suggerire una soluzione per la mutua esclusione in un setting distribuito:
Se semplicemente adattiamo l'algoritmo di Lamport potremmo ottenere una soluzione inefficiente, poiché i processi agiscono indipendentemente (senza coordinamento) cercando di accedere alla sezione critica.

Intuizione dietro l'algoritmo RA:

- Ogni processo entra nella doorway suggerendo un numero.
- Poi il processo invia a tutti gli altri processi il numero aspettando che concedano l'accesso alla critical section.

RA Algorithm
Assunzioni: processi non falliscono, messaggi non si perdono, latenza del canale finita.
Variabili locali:

- #replies (inizialmente 0)
- State che può assumere {Requesting, CS, NCS} (inizialmente NCS)
- Q coda delle richieste in attesa (inizialmente vuota)
- Last_Req (inizialmente MAX_INT)
- Num (inizialmente 0)

Repeat

1. State = Requesting
2. Num = Num + 1; Last_Req = Num
3. for i from 1 to N send REQUEST(Last_Req) to Pi
4. Wait until #replies == N – 1
5. State = CS
6. CS
7. Comunque prendo elem in coda Q send REPLY to elem

Q = insieme vuoto
State = NCS
#replies = 0
Last_Req = MAX_INT

NB la linea 2 è eseguita atomicamente.

Se si riceve una REQUEST(t) from Pj

8. Num = max(t, Num)
9. If State == CS or (State == Requesting and (Last_Req, i) < (t, j))
10. Then insert (t,j) into Q
11. Else send REPLY to Pj

Se si riceve una REPLY from Pj

12. #replies = #replies + 1

Il senso dell'algoritmo è: se ho bisogno di entrare in CS mi metto in Requesting, aumento il mio num (token) e invio a tutti gli altri processi tranne me la richiesta d'accesso. Se ricevo N-1 REPLY semplicemente vuol dire che o i processi non sono interessati o ho priorità maggiore io dettata da un valore di Last_Req minore (o nel caso uguale, ho un indice minore).
In quel caso metto in coda le richieste di altri processi in modo tale che si blocchino non ricevendo il mio REPLY e entro in CS. Una volta uscito completo l'invio di tutti i REPLY ai processi in coda che aspettano per entrare in CS e mi resetto le var locali tranne NUM, serve come numeretto che si prende dal panettiere quindi va sempre incrementato. Se invece sono interessato ad entrare in CS ma aspettando i REPLY dagli processi mi arriva una richiesta da un altro processo che ha priorità maggiore rispetto alla mia allora sono costretto a inviargli un REPLY e a sbattere la testa in busy waiting sul while #replies != N-1.

9. Security

Obiettivi chiave della Computer Security:

- **Riservatezza**: la riservatezza dei dati assicura che le informazioni private o riservate non siano rese disponibili o divulgate a persone non autorizzate. La privacy garantisce che le persone controllino o influenzino a chi e da chi le proprie informazioni possano essere divulgate.
- **Integrità**: l'integrità dei dati assicura che le informazioni e i programmi siano modificati solo in un modo specificato e autorizzato. L'integrità del sistema assicura che un sistema esegua la funzione prevista in modo inalterato, libera da manipolazioni non autorizzate deliberate o involontarie del sistema.
- **Disponibilità**: assicura che i sistemi funzionino rapidamente e il servizio non sia negato agli utenti autorizzati.

NB:

- Una perdita di riservatezza è la divulgazione non autorizzata di informazioni.
- Una perdita di integrità è la modifica o la distruzione non autorizzata di informazioni.
- Una perdita di disponibilità è l'interruzione dell'accesso ad un sistema di informazione.

Altri due concetti importanti:

- **Autenticità**: la proprietà di essere un vero gruppo che può essere verificato e considerato affidabile, ossia la fiducia nella validità di una connessione, di un messaggio, di un mittente. Verificare che gli utenti siano chi dicono di essere e che ogni input in arrivo al sistema provenga da una fonte attendibile.
- **Responsabilità**: l'obiettivo di sicurezza che genera il requisito di assegnare univocamente delle azioni all'entitche le compie. Dobbiamo essere in grado di scovare una violazione della sicurezza. I sistemi devono tenere un registro delle proprie attività per consentire alle successive anali di tracciare le violazioni di sicurezza o aiutare nelle controversie sulle transazioni.

Tabella di vari attacchi e conseguenze.

Conseguenza : Divulgazione non autorizzata, una circostanza in cui un'entità ottiene l'accesso a dati per i quali non è autorizzata.
Attacchi:

- **Esposizione**: i dati sensibili vengono rilasciati direttamente a un'entità non autorizzata
- **Intercettazione**: un'entità non autorizzata accede direttamente a dati sensibili, attraverso fonti e destinazioni autorizzate.
- **Inferenza**: un attacco dovuto al fatto che un'entità non autorizzata accede indirettamente a dati sensibili sfruttando le caratteristiche o i mezzi di comunicazione. (Non necessariamente a dati contenuti nella comunicazione).
- **Intrusione**: un'entità non autorizzata ottiene l'accesso a dati sensibili eludendo la protezione di sicurezza di un sistema.

Conseguenza: Inganno, ossia una circostanza in cui un'entità autorizzata ottiene dati falsi ritenendoli veritieri.
Attacchi:

- **Mascheramento**: un'entità non autorizzata accede a un sistema o esegue un atto dannoso presentandosi come un'entità autorizzata.
- **Falsificazione**: dati falsi ingannano un'entità autorizzata.
- **Ripudio**: un'entità inganna un'altra negando falsamente la responsabilità di un atto.

Conseguenza: Rottura, una circostanza o evento che interrompe o impedisce il corretto funzionamento dei servizi e delle funzioni di sistema.
Attacchi:

- Incapacitazione (non esiste in italiano -> **Incapacitation**): impedisce o interrompe il funzionamento del sistema disabilitando un componente dello stesso.
- **Corruzione**: altera in modo indesiderato il funzionamento del sistema modificando negativamente funzioni o dati del sistema.
- **Ostruzione**: un attacco che interrompe la fornitura dei servizi di sistema ostacolando il funzionamento dello stesso.

Conseguenza: Usurpazione, un evento che determina il controllo di servizi o funzioni di sistema da parte di un'entità non autorizzata.
Attacchi:

- **Appropriazione indebita**: un'entità si assume il controllo logico o fisico non autorizzato di una risorsa di sistema.
- **Uso improprio**: fa si che un componente del sistema esegua una funzione o un servizio che è dannoso per la sicurezza dello stesso.

ATTACCHI PASSIVI
Si tenta di apprendere o utilizzare le informazioni dal sistema ma non si influisce sulle risorse dello stesso.
Rientrano tra questi le intercettazioni e il monitoraggio di trasmissioni.
L'obiettivo è ottenere informazioni dai dati trasmessi.
Difficile da rilevare poiché non lasciano traccia di modifiche su dati e risorse.
Per impedire questi attacchi si fa ricorso alla crittografia.
Si tratta quindi di un approccio di prevenzione più che di individuazione.

ATTACCHI ATTIVI
Si tratta di modificare flussi di dati o crearne di falsi.
Quattro categorie:

- **Riproduzione**: implica la cattura passiva di un'unità di dati e la sua successiva ritrasmissione per produrre un effetto non autorizzato.
- **Mascheramento**: ha luogo quando un'entità finge di esserne un'altra.
- **Modifica dei messaggi**: una parte di un messaggio legittimo viene alterata, oppure viene alterata la sequenza di invio dei messaggi o anche ritardato l'invio.
- **Negazione di servizio**: impedisce o inibisce il normale utilizzo o la gestione delle strutture di comunicazione, ad esempio si interrompe un'intera parte di rete.

MODELLI DI COMPORAMENTO DELL'INTRUSO
Hackers:

- Tradizionalmente coloro che lo fanno lo fanno per il brivido dell'azione.
- Gli attaccanti spesso cercano obiettivi di opportunità e quindi condividono informazioni con gli altri.
- Gli intrusi benigni consumano risorse e possono rallentare le prestazione per gli utenti legittimi.
- I sistemi di rilevamento delle intrusioni e i sistemi di prevenzione delle intrusioni sono progettati per contrastare questo tipo di minaccia da parte di hacker.
- I team di risposta alle emergenze informatiche sono iniziative cooperative che raccolgono informazioni sulle vulnerabilità del sistema e condividono ciò che scoprono con i responsabili del sistema (sgamano falle a pagamento).

Attacco hacker

1. Selezionano la vittima usando strumenti per l'individuazione dell'IP come NSLookup, Dig e altri
2. Mappano la rete per servizi accessibili usando strumenti come NMAP
3. Identificano potenziali servizi vulnerabili (in questo caso, pcAnywhere)
4. Brute force(guess) la password di pcAnywhere
5. Installano strumenti per l'amministrazione remota chiamati DameWare
6. Aspettano l'admin che logga e rubano la sua password
7. Usano la password per accedere al resto della rete

Società criminali:

- Gruppo organizzato di hackers.
- Si organizzano e scambiano consigli in forum nascosti.
- Un obiettivo comune sono file riguardanti carte di credito nei server degli e-commerce.
- Di solito hanno obiettivi specifici, o almeno classi di obiettivi, in mente.
- Attacchi rapidi e veloci.

Attacco

1. Agiscono velocemente e precisamente per rendere le loro attività difficili da individuare
2. Sfruttano il perimetro attraverso porte vulnerabili (exploit perimeter through vulnerable ports)
3. Usano cavalli di troia (software nascosti) per lasciare backdoor per poi rientrare
4. Usano sniffer per catturare le password
5. Non restano in giro finche' non vengono individuati (do not stick around until noticed)
6. Fanno pochi o nessun errore

Attaccanti interni:

- Tra i più difficili da individuare e prevenire.
- Può essere motivato dalla vendetta o semplicemente pensa di avere il diritto di farlo.
- I dipendenti hanno già accesso e conoscenza della struttura e del contenuto dei database aziendali.

Attacco

1. Creano account nella rete per loro e per i loro amici
2. Accedono a account e applicazioni che non userebbero normalmente per il loro lavoro di ogni giorno
3. E-mail former e potenziali datori di lavoro
4. Effettuano conversazioni furtive di messaggistica istantanea
5. Visitano siti Web che si rivolgono a dipendenti scontenti, come dcompany.com
6. Effettuano grandi download e copie di file
7. Accedono alla rete fuori orario

MALWARE

Termine generale per indicare un software dannoso.
Software ideato per causare danni o consumare risorse di un computer designato.
Spesso nascosto o mascherato come software legittimo.
In alcuni casi si diffonde ad altri computer tramite e-mail o dispositivi infetti.

Backdoor:

- Anche conosciuto come trapdoor.
- Un punto di accesso segreto in un programma che consente di accedere senza eseguire le consuete procedure di accesso di sicurezza.
- Un hook (aggrappo) di manutenzione è una backdoor che i programmatori usano per debuggare e testare programmi che richiedono una lunga configurazione.
- Diventa una minaccia quando la si usa per ottenere un accesso non autorizzato.
- E' difficile implementare i controlli del sistema operativo per le backdoor.

Logic Bomb:

- Uno dei più vecchi tipi di attacco.
- Codice incorporato in un programma legittimo pronto ad "esplodere" quando vengono soddisfatte determinate condizioni.
- Una volta innescata una bomba può alterare o cancellare dati o interi file, causare un arresto della macchina o fare altri danni.

Trojan Horse:

- Programma utile o apparentemente utile che contiene codice nascosto che, quando invocato, esegue alcune funzioni indesiderate e dannose.
- I Trojan Horse si adattano a uno di questi tre modelli:
    1. Continuano a svolgere la funzione del programma originale e contemporaneamente un'attività malevola.
    2. Continuano a svolgere la funzione del programma originale ma modificandone il funzionamento per eseguire l'attività malevola.
    3. Esecuzione diretta dell'attività malevola.

Platform Independent Code:

- A volte indicato come codice mobile.
- Programmi che possono essere spediti invariati a un insieme vasto di piattaforme ed eseguiti con semantica identica.
- Trasmesso da un sistema remoto a un sistema locale e quindi eseguito su quest'ultimo senza la volontà esplicita dell'utente.
- Spesso si comporta come un meccanismo per virus, un worm o un trojan horse da trasmettere all'utente.
- Sfrutta i vantaggi delle vulnerabilità.

Multiple-Threat Malware:

- Infetta in più modi.
- Un virus multipartito è in grado di infettare più tipi di file.
- Un attacco combinato utilizza più metodi di infezione o trasmissione per massimizzare la velocità del contagio e la gravità dell'attacco.
- Un esempio di questo tipo di approccio combinato è Stuxnet.

Virus:

- Software che infetta altri programmi modificandoli:
  - Porta con se codice per auto duplicarsi.
  - Viene incorporato in un programma su un computer
  - Quando il computer infetto entra in contatto con un pezzo di software non infetto, una copia del virus passa nel programma.
  - L'infezione può essere diffusa scambiando dischi da computer a computer o attraverso una rete.
- Un virus informatico ha tre parti:
  - Un meccanismo di infezione
  - Un attivatore
  - Un payload
    - Può comportare danni
    - O attività benigne ma evidenti
- **Fasi del virus:**
  - **Fase dormiente:**
    - Il virus è inattivo
    - Sarà eventualmente attivato da qualche evento.
    - Non tutti i virus hanno questa fase
  - **Fase di propagazione**
    - Il virus inserisce una copia identica di se stesso in altri programmi o in certe aree del disco di sistema
  - **Fase di attivazione**
    - Il virus viene attivato per eseguire la funzione per la quale è stato progettato
    - La fase di attivazione può essere causata da una varietà di eventi di sistema
  - **Fase di esecuzione**
    - La funzione viene eseguita
    - La funzione potrebbe essere innocua (messaggio sullo schermo) o dannosa (distruzione di programmi e file di dati)
- **Classificazione dei virus**:
  - Non esiste uno schema di classificazione universalmente accettato
  - La classificazione per target include le seguenti categorie:
    - Boot sector infector ossia infetta un record di avvio principale e si diffonde quando un sistema viene avviato dal disco contenente il virus
    - File infector ossia infetta i file che l'OS o la shell considerano eseguibili
    - Macro Virus ossia infetta file con codice macro interpretato da un'applciazione
- **Strategia di occultamento**:
  - Una classificazione dei virus mediante la strategia di occultamento:
    - Virus crittografato ossia una chiave di crittografia casuale crittografa il virus
    - Virus furbo ossia si nasconde dal rilevamento dell'antivirus
    - Virus polimorfo ossia muta con ogni infezione. Le copie sono funzionalmente equivalenti ma hanno pattern di bit totalmente differenti.
    - Virus metamorfo ossia mutua con ogni infezione. Si riscrive completamente dopo ogni iterazione.

Macro Virus:

- A metà degli anni '90 i Macro Virus sono diventati i virus più diffusi
- Sono particolarmente minacciosi perché:
  - Sono indipendenti dalla piattaforma. Molti Macro Virus infettano documenti Word o altri documenti Office.
  - Infettano documenti, non porzioni di codici eseguibili

- Sono facilmente diffondibili, metodo comune è via e-mail
- I controlli di accesso al file system sono di uso limitato per impedirne la diffusione.

E-Mail Virus:

- I primi e-mail virus in rapida diffusione utilizzavano una macro di Microsoft Word incorporata in un allegato:
  - Se il destinatario apre l'allegato, la macro è attivata.
  - Il virus e-mail si inoltra a tutti gli utenti nella lista contatti del primo destinatario.
  - Il virus fa danni locali al sistema dell'utente.
- Nel 1999 il virus venne potenziato:
  - Può essere attivato soltanto aprendo una mail e non aprendo per forza l'allegato.
  - Il virus utilizza il linguaggio di scripting Visual Basic supportato dal pacchetto e-mail.

Worms:

- Un programma in grado di replicarsi e inviare copie da un computer all'altro attraverso la connessione di rete.
- Alla ricezione di un worm può essere attivato per replicarsi e propagarsi di nuovo.
- Oltre alla propagazione, il worm di solito svolge alcune funzione indesiderate.
- Cerca attivamente macchine da infettare e in ognuna di questa infettata funge da trampolino di lancio automatico per attacchi ad altre macchine.
- Per replicarsi e propagarsi utilizza mezzi di network:
  - Struttura mail elettronica: invia una copia di se stesso ad altri sistemi in modo che il suo codice venga eseguito quando l'e-mail o l'allegato viene ricevuto e visualizzato.
  - Funzionalità di esecuzione remota: esegue una copia di se stesso su un altro sistema usando una funzione di esecuzione remota esplicita o sfruttando un difetto di programma in un servizio di rete per sovvertire le sue operazioni.
  - Funzionalità di accesso remoto: accede a un sistema remoto come utente e quindi utilizza i comandi di quest'ultimo per copiare se stesso da un sistema all'altro.

Bots:

- Un programma che acquisisce segretamente il controllo su un computer collegato a internet e quindi utilizza tale computer per lanciare attacchi difficili da collegare al creatore del bot. Anche conosciuto come Zombie o Drone.
- Generalmente installato su tantissimi computer appartenenti a terze parti ignare.
- Una collezione di Bot che agiscono in maniera coordinata si chiama BOTNET
- Un **Botnet** ha tre caratteristiche:
  - La funzionalità di un bot
  - Controllato da remoto
  - Un metodo di propagazione dei bot per costruire il botnet
- Usi:
  - DDoS attack (Distributed denial-of-service) che blocca dei servizi all'utente
  - Spamming che invia messaggi in maniera massiva con una serie di mail
  - Sniffing traffic per ottenere informazioni sensibili come username o password
  - Spreading new malware ossia per diffondere malware
  - Manipulating online polls/games possibile poiché ogni bot ha un distinto IP e appare come una persona normale

Remote Control Facility:

- Si deve distinguere un bot da un worm: un worm si propaga e si attiva, mentre un bot è controllato da una struttura centrale.
- Un mezzo tipico per implementare un Remote Control Facility è un server IRC: tutti i bot si uniscono ad un canale specifico su questo server e trattano i messaggi in arrivo come comandi.
- Le botnet più recenti tendono a utilizzare canali di comunicazioni nascosti tramite protocolli come HTTP.
- Meccanismi di controllo distribuiti vengono anche utilizzati per evitare un singolo punto di errore.

Constructing the attack network:

- Il primo passo in un attacco botner da parte di un attaccante è di infettare un certo numero di macchine con i bot che verranno utilizzati per portare a termine l'attacco.
- Ingredienti essenziali:

    - Software che sia in grado di eseguire l'attacco
    - Una vulnerabilità in un ampio numero di sistei
    - Una strategia di localizzazione di macchine vulnerabili (un proccesso chiamato scanning)
- Nello scanning process l'attaccante deve per primo cercare macchine vulnerabili e infettarle.
- Il software del bot ripete nelle macchine infette lo stesso processo di scanning finchè un'ampia rete di macchine infette non è stata creata.

Rootkit:
- Insieme di programmi installati su un sistema per mantenere l'accesso di amministratore (root) a quel sistema.
- L'accesso root consente l'accesso a tutte le funzioni e i servizi del sistema operativo.
- Il rootkit modifica le funzionalità standard dell'host in modo malevolo e furtivo. Con l'accesso root un utente malintenzionato ha il controllo totale del sistema e può aggiungere, eliminare, modificare file, monitorare processi, lavorare attraverso rete e ottenere anche l'accesso backdoor.
- Un rootkit si nasconde sovvertendo i meccanismi che controllano e segnalano i processi, i file, i registri su un pc.
- Classificazione in base a come sopravvivono dopo il riavvio o alla modalità d'esecuzione:
    - Persistente: attivato ogni volta che il sistema si avvia.
    - Memory based: non ha codice persistente e non può sopravvivere a un riavvio.
    - User-mode: intercetta le chiamate alle API e modifica i risultati restituiti.
    - Kernel-mode: può intercettare le chiamate alle API native in modalità kernel. Può nascondere la presenza di unu processo malware rimuovendolo dall'elenco dei processi attivi dal kernel.
- Installazione:
    - I rootkit non si affidano direttamente a vulnerabilità per accede a un computer.
    - Un metodo di installazione è attravero un Trojan Horse.
    - Un altro mezzo di installazione può avvenire da parte di un hacker.

Attacchi con syscall:
- I programmi operativi a livello utente interagiscono con il kernel tramite syscall.
- In linux a ciascuna syscall viene assegnato un numero univoco.
- Tre tecniche possono essere utilizzate per modifica le chiamate di sistema:
    - Si può modificare la tabella delle chiamate di sistema in modo che punti al codice della rootkit.
    - Si può modificare gli obiettivi della tabella delle syscall.
    - Reindirizzare l'intera tabella delle syscall.

OPERATING SYSTEM DEFENSE

Autenticazione basata su password:
- Un metodo di difesa ampiamente utilizzato contro gli intrusi è il sistema delle password.
- La password serve pe autenticare l'ID del singolo accesso al sistema.
- L'ID fornisce sicurezza attraverso:
    - Determinando se l'utente è autorizzato ad ottenere l'accesso al sistema.
    - Determinando i privilegi d'accesso in funzione dell'utente.
    - Controllo d'accesso discrezionale.
- Il **sale** serve a tre cose:
    - Impedisce che password duplicate siano visibili nel file di password. (Anche se due utenti scelgono la stessa password verranno assegnati a valore di sale diversi).
    - Aumenta notevolmente la difficoltà di attacchi dizionario.
    - Diventa quasi impossibile scoprire se una persona ha utilizzato la stessa password su piu sistemi d'accesso.

Schema Password UNIX:
- Esistono due minacce per questo schema:
    - Un utente può guadagnare un account ospite su una macchina e poi utilizzare un programma cracker (un programma che indovina la password) sul sistema.

- Se l'attaccante è in grado di ottenere una copia del file delle password, il cracke program può essere utilizzato su un'altra macchina. Questo permette di scorrere milioni di password possibili in un tempo ragionevole.

Autenticazione basata sul Token:

- Gli oggetti che un utente possiede ai fini dell'autenticazione sono chiamati Token.
- Due tipi di token:
  - Memory cards: possono conservare dati ma non processarli. La più comune è la carta di credito con una banda magnetica sul retro. Una banda magnetica può memorizzare solo un semplice codice di sicurezza che può esser letto e riprogrammato da un lettore di schede economico. Ci sono anche schede che includono una memoria elettronica interna. Può essere utilizzata da sola o con qualche forma di password (PIN personal identification number).
  - Smart Cards: include un processore. Un oggetto intelligente che sembra una carta di credito è chiamato smart card. Possono essere calcolatrici, chiavi o altri piccoli oggetti portatili. L'interfaccia prevede una tastiera e un display per l'interazione uomo/token. Comunicano con un lettore specifico per le smart card. L'autenticazione può essere statica, dinamica generando una One Time Password o attraverso delle domande effettuate all'utente.

Autenticazione statica Biometrica:

- Si cerca di autenticare una persona attraverso le proprie uniche caratteristiche fisiche.
- Esempi di caratteristiche fisiche statiche:
  - Impronta digitale
  - Profilo della mano
  - Caratteristiche facciali
  - Scansione retina o iride
- Esempi di caratteristiche dinamiche:
  - Timbro della voce
  - Firma
- Caratteristiche fisiche:
  - Caratteristiche facciali: mezzi più comuni per l'identificazione tra persone. Esempio: occhi, sopracciglia, naso, labbra… Un approccio diverso è quello di usare una camera ad infrarossi per riprodurre una scansione termica facciale.
  - Impronte digitali: modello di creste e solchi sulla superficie del polpastrello. Unica per ogni uomo.
  - Geometria della mano: identifica caratteristiche della mano tra cui la forma la lunghezza e la grossezza delle dita.
  - Scansione retina: modello formato dalle vene sotto la superficie retinica è unico. Una scansione della retina ottiene questo modello attraverso l'uso di infrarossi.
  - Iride: i dettagli dell'iride sono unici.
  - Firma: ognuno di noi ha una scrittura personale.
  - Voce: il modello della voce dipende fortemente dalle caratteristiche fisiche e anatomiche dell'individuo.

Controllo dell'accesso:

- Una politica di controllo degli accessi stabilisce quali tipi di accesso sono consentiti, in quali circostanze e da chi.
- Le politiche d'accesso sono di solito raggruppate in queste categorie:
  - Controllo d'accesso discrezionale (DAC): controlla l'accesso in base all'identità del richiedente e alle regole di accesso che indicano quali sono (o non sono) i richiedenti autorizzati.
  - Controllo d'accesso vincolato (MAC) la m sta per Mandatory: controlla l'accesso basandosi sul tipo di utente di cui vengono controllati i permessi d'accesso da un sistema.
  - Controllo d'accesso basato sul ruolo (RBAC): controlla l'accesso in base al ruolo di ogni utente e alle regole che indicano quali privilegi ha ogni utente in base al ruolo ricoperto.

SISTEMA DI RILEVAMENTO INTRUSIONE BASATO SULL'HOST (IDS)

Monitora l'attività sul sistema in vari modi per rilevare comportamenti sospetti. Lo scopo principale è rilevare intrusioni, registrare eventi sospetti e inviare allarmi. Può rilevare sia intrusioni interne che esterne.

Rilevamenti anomali:

- Raccolta di dati relativi al comportamento degli utenti autorizzati nel tempo.
- Rilevamento soglia.
- Rilevamento basato sul profilo.

Rilevamento firma:

- Definisce un insieme di regole o di modelli di attacco che possono essere utilizzati per decidere se un dato comportamento è quello di un intruso.

Alcune informazioni sull'attività dell'utente vengono fornite in tempo reale all'IDS.

Se il sistema di controllo è nativo è comodo perché non abbiamo bisogno di software aggiuntivi per raccogliere i dati che ci interessa però purtroppo i registri di controllo nativi potrebbero non contenere le informazioni necessarie o in parte per scovare un'intrusione.

Se invece ci affidiamo ad un software specifico, ne usufruiamo grazie ad un venditore, potremmo comodamente trasferirlo su tantissimi sistemi. Lo svantaggio ovviamente è l'overhead che si crea nel momento in cui si devono fornire dati al software.

ANTIVIRUS APPROCHES

La soluzione ideale per minacce virus è la prevenzione. L'obiettivo di evitare infezioni sul proprio sistema non è del tutto raggiungibile, comunque sia la tattica della prevenzione riduce decisamente il numero di attacchi virali con successo. Se il rilevamento ha esito positivo, ma l'identificazione o la rimozione non è possibile, l'alternativa è scartare il programma infetto e ricaricare una versione pulita da una copia di backup.

Fasi:

- Rilevamento: una volta che l'infezione è avvenuta, l'antivirus determina l'attività del virus e lo localizza.
- Identificazione: una volta localizzato si cerca di identificarlo confrontandolo con le firme antivirali sopra descritte.
- Rimozione: Una volta identificato si rimuovono tutte le tracce del virus dal programma infetto e lo si ripristina allo stato originale. L'obiettivo è di rimuoverlo da tutti i sistemi affinché non si possa più diffondere.

GENERIC DECRYPTION (GD)

Consente al programma antivirus di rilevare facilmente anche i virus polimorfi più complessi mantenendo allo stesso tempo una velocità di scansione elevata. Quando viene eseguito un file contenente un virus polimorfo, quest'ultimo deve decrittografarsi per attivarsi. Di conseguenza all'apertura di questi file eseguibili viene effettuata una scansione tramite uno scanner GD.

Il problema di progettazione più complesso con uno scanner GD è determinare il tempo di esecuzione di ogni interpretazione.

Un GD scanner contiene:

- **Un modulo di controllo di emulazione**: controlla l'esecuzione del codice in analisi.
- **Scanner delle firme antivirali**: un modulo che analizza il codice in cerca di virus.
- **Emulatore della CPU**: un computer virtuale basato su software. I file eseguibili vengono interpretati dall'emulatore in modo che il processore sottostante non venga intaccato.

DIGITAL IMMUNE SYSTEM

Un approccio completo alla protezione dai virus sviluppato da IBM e perfezionato da Symantec. La necessità dello sviluppo nasce dalla sempre più alta presenza di virus che si propaga via internet.

Due tendenze in internet hanno avuto un impatto sempre crescente sul tasso di propagazione dei virus negli ultimi anni:

- Sistemi di posta integrati.

- Sistemi di programmi mobili (ossia, spostamento di codice malevolo su più piattaforme).

L'obiettivo del sistema è quello di fornire tempi di risposta rapidi in modo che i virus possano essere eliminati nel migliore dei casi appena vengono introdotti nei sistemi.

Appena un client viene infettato e l'analisi da esito positivo si notificano tutte le altre macchini amministratrici che diffonderanno l'avviso ai loro rispettivi client.

BEHAVIOR BLOCKING SOFTWARE

Si integra con il sistema operativo e monitora il comportamento del programma in tempo reale per scovare azioni dannose.

Potrebbe includere:

- Aprire o modificare determinati file.
- Formattazione dischi.
- Modifiche ai file eseguibili o macro.
- Modifica delle impostazioni di sistema critiche.
- Comunicazione network.

Operazioni:

1. L'amministratore setta politiche sui comportamenti accettabili del software e le carica su un server o sui desktop.
2. Il software dannoso riesce a superare il firewall.
3. Il software behavior-blocking segnala il codice sospetto al server e "insabbia" il software sospetto per impedirgli di avviarsi.
4. Il server avvisa l'amministratore che il codice sospetto è stato identificato e messo in quarantena e resta in attesa della decisione dell'amministratore sulla volontà di quest'ultimo di rimuoverlo o di autorizzarne l'esecuzione.

CONTROMISURE AI WORM

Una volta che il worm infetta una macchina è possibile rilevarlo con un software antivirus. La sua propagazione genera una considerevole attività di rete, quindi l'attività e il monitoraggio dell'utilizzo di quest'ultima possono essere un'arma di difesa.

Classi di difesa contro i worm:

A. Scansione worm basata sul filtro della firma virale: genera una firma del worm che viene poi utilizzata per impedire che le scansioni di worm entrino/escano da una rete/host.
B. Filtraggio dei worm in base al loro contenuto: simile alla classe A, ma si concentra sul contenuto del worm piuttosto che sulla firma.
C. Blocco del worm basato sulla classificazione del payload: le tecniche di rete esaminano i pacchetti per vedere se contengono worm.
D. Scansione di rilevamento del Threshold random walk (TRW): sfrutta la casualità nella scelta delle destinazioni da connettere come un modo per rilevare se uno scanner è in funzione.
E. Limitazione del rate: limita la velocità del traffico da un host infetto.
F. Blocco del rate: blocca immediatamente il traffico in uscita quando viene superata una soglia in termini di velocità di connessione.

CONTROMISURE AI BOT

Gli IDS e i Digital Immune System sono utili contro i bot. Una volta che i bot sono attivati e un attacco è in corso, queste contromisure possono essere utilizzate per rilevare l'attacco.

L'obiettivo principale è cercare di rilevare e disabilitare la botnet durante la sua fase di costruzione.

CONTROMISURE AI ROOTKIT

Può essere difficile da individuare e neutralizzare. Molti degli strumenti amministrativi possono essere compromessi e resi inutilizzabili. Per contrastarli si necessità di una varietà di strumenti di sicurezza a livello di rete e computer. I sistemi di rilevamento delle intrusioni basati sulla rete e su host possono cercare le firme di codice degli attacchi noti di rootkit nel traffico d'entrata. Anche il software antivirus può rilevare le firme conosciute del rootkit.

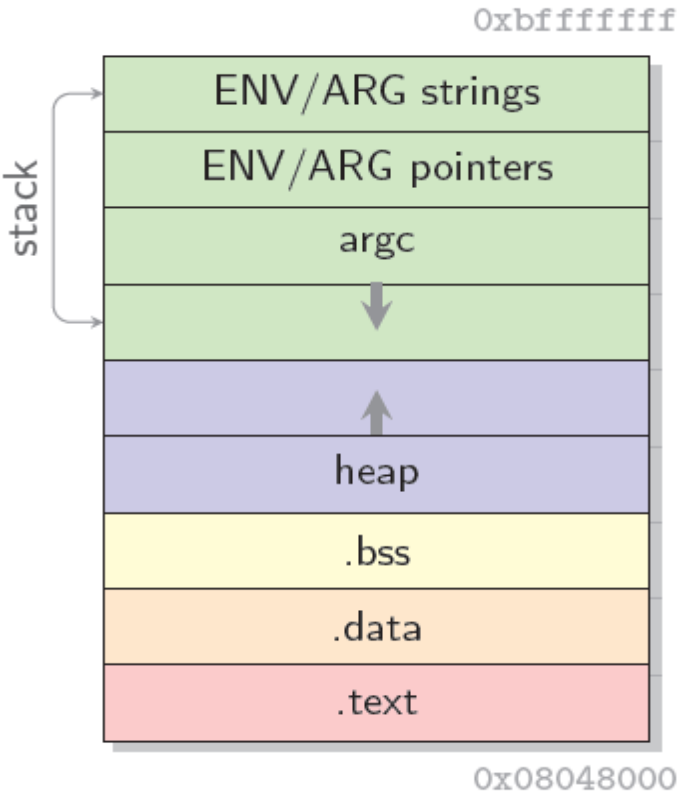
10. Memory Corruption

Le vulnerabilità di corruzione della memoria permettono ad un attaccante di far scrivere al programma attaccato aree di memoria o dati valori che il programmatore non intendeva.
Nei casi peggiori, questi possono portare all'esecuzione di comandi arbitrari sotto il controllo dell'attaccante.

Memory corruption deriva da:
- Buffer overflows (stack or heap)
- Out-by-one errors
- Type confusion errors
- Format strings
- Use-after-free

Ruolo dei linguaggi di programmazione:
- I linguaggi di basso livello gestiscono la memoria direttamente con vantaggio di efficienza e precisione, purtroppo però sono più comuni errori di programmazione che violano la sicurezza della memoria, dovendo utilizzare puntatori e l'aritmetica di questi e avendo un arbitrario accesso alla memoria.
- I linguaggi di più alto livello come Java ad esempio gestiscono la memoria autonomamente. Questo è un approccio sicuro contro la corruzione della memoria perché ad un accesso non consentito alla memoria viene lanciato un errore/eccezione. Si inficia però sulle prestazione e sulla liberta del programmatore.

LAYOUT DELLA MEMORIA DI UN PROCESSO



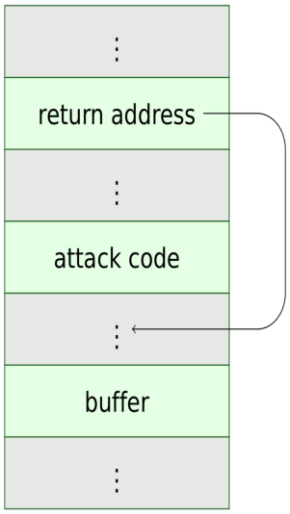
Heap e stack crescono in maniera contraria senza mai scavallarsi.

Come funziona la stack: lo stack di un programma (ossia stack di funzioni, runtime stack) contiene i frame dello stack (ossia i recordi di attivazione) per ogni funzione invocata. I meccanismi esatti dell'uso e della variazione della stack dipendono da molti fattori come CPU, OS, linguaggio, compilatore.
Ci sono però alcuni registri di supporto della CPU per gestirla correttamente:
- Stack pointer register: ESP
- Frame pointer register: EBP

NB Le operazioni di push e pop permettono il salvataggio o il restore di questi registri che se alterati cambiano il flusso "atteso" di un programma.

STACK OVERFLOW

Gli attacchi di stack overflow sono più rilevanti per programmi scritti con linguaggi a basso livello, in cui il programmatore potrebbe aver compiuto errori di accesso a memoria.



Il senso è quello di inserire un payload dannoso che sovrascrive tutto lo spazio allocato per il buffer richiesto dal programmatore fino alla cella che contiene il valore di ritorno.
L'indirizzo di ritorno viene modificato per tornare indietro (giù) da qualche parte in cui io ho inserito il mio codice di attacco. Tipicamente il codice d'attacco esegue una shell.

Guardare l'esempio sulle slide di stack overflow (more realistic):

- Ogni volta che si entra in un nuovo frame (funzione) va salvato ebp con una push e bisogna sovrascriverci esp così da tener conto l'inizio della nuova funzione. (SEMPRE)
- Prima della funzione getURL si salva nella posizione del return l'IP (istruction pointer) che in questo caso è la pop ebp della funzione IE.
- Dopo getURL ha inizio. Sistema ebp salvando il vecchio frame pointer e settando il suo nuovo.
- Nella funzione getURL dopo aver sistemato esp ed ebp si alloca buffer da 40b (10 x 4).
- Infine si preparano i parametri per la funzione read.
- Dopo aver letto e sovrascritto dannosamente anche l'area di memoria di getURL in cui era contenuto l'IP da cui doveva riprendere il flusso computazionale, il programma attaccato non farà altro che riprendere dal nuovo (nostro) IP che volontariamente inseriamo all'interno del buffer dove abbiamo scritto il nostro codice.

Questo è tutto. A volte però non serve neanche scrivere ed eseguire proprio codice dal buffer, bensì basta esclusivamente sovrascrivere dei valori per ottenere un accesso ad esempio.

```
get_medical_info()
{
      boolean authorized = false;
      char name [10];
      authorized = check();
      read_from_network (name);
      if (authorized)
            show_medical_info (name);
      else
            printf ("sorry, not allowed");
}
```

Qui basta sovrascrivere la variabile authorized con true al momento dell'inserimento del buffer. Non serve iniettare codice.

Oppure possiamo inserire un programma nel buffer o nell' "ambiente" del sistema. Molto semplice perché è l'attaccante che decide cosa inserire nel buffer. Spesso si vuole inserire una shell per poter prendere il comando della macchina attaccata.

Il tipo di vettore da iniettare è composto da tre parti:

- NOP sled: una sequenza di istruzione che non fanno nulla. Vengono utilizzati per facilitare l'individuazione dell'inizio dello shellcode. Per evitare che per errori di calcolo si inserisca un indirizzo che punti ad una

sequenza mediana nello shellcode (rendendo inutilizzabile il programma), si fa puntare ad una parte che SICURO contiene istruzioni NOP. Con lo scorrere di queste istruzioni inutili si arriva SICURAMENTE all'inizio dello shellcode, mantenendo l'integrità di quest'utlimo.

- Shellcode: una sequenza di istruzioni macchina da eseguire.
- Shellcode address: l'indirizzo di memoria dove è contenuto lo shellcode (leggi su).

Le cose sono comunque un po' più complicate: vengono usati codici codificati per evitare che alcune funzioni o caratteristiche del programma attaccato possano interferire con il proprio codice.
Es. se si utilizza la funzione strcpy() che termina la funzione di copia al primo NULL byte e se il codice dell'attaccante contiene più NULL byte di suo si ha un codice copiato parzialmente. Ecco che la codifica priva il codice dell'attaccante dei NULL byte per poi reinserirli dinamicamente.

OFF-BY-ONE ATTACK (FRAME POINTER OVERWRITE)
In alcuni casi da parte dell'attaccante solo un byte di overflow può essere utilizzato per l'attacco. Puntiamo a sovrascrivere il byte meno significativo del frame pointer salvato di cui ogni funzione ne fa la push per salvarlo, così nel ritorno avremo un ebp corrotto.

Es.

```
void func(char *str)
{
        char buf[256];
        int i;

        for (i=0;i<=256;i++)
                buf[i]=str[i];
}
int main(int argc, char* argv[])
{
        func(argv[1]);
}
```

Cosa succede? Disegniamo la stack:

argv[1]
ret main
push ebp (salvo il vecchio frame pointer) Qui inizia il nuovo frame pointer
buf
buf
buf
buf
buf
buf
buf
buf
i Qui arriva lo stack pointer

E' facile vedere che grazie alla sequenza di dichiarazione delle variabili e al fatto che c'è = nel for possiamo accedere (dannosamente) all'area della stack dove è stato pushato il vecchio ebp sovrascrivendolo. Se scambiassimo le dichiarazione cioè mettessimo prima la i e poi il buf lo scavallamento corromperebbe l'intero i.

SOVRASCRIZIONE DI UN PUNTATORE A FUNZIONE
In C si usano puntatori a funzioni per poterle chiamare. L'indirizzo di memoria in cui sono salvate vengono salvate in variabili passate come parametri ad altre funzioni.

Es.

```
unsigned sleep(unsigned seconds); // <unistd.h>

void bar(int arg) {
      char buf[64];
      foo(buf, arg, &sleep);
      printf("%s\n", buf);
}

int foo(char* buf, int arg, unsigned (*funcp)(unsigned)) {
      char tmp[128];
      gets(tmp);
      strncpy(buf, tmp, 64);
      (*funcp)(arg);
      return 0;
}
```

Il problema ovviamente è nella funzione foo. Nel momento in cui dichiaro un array di 128 char in cui poi chiedo un inserimento da tastiera non controllato. Nonostante io ne copi solo 64 nell'array buf, con la chiamata precedente potrei sovrascrivere l'indirizzo della funzione sleep con l'indirizzo di una mia funzione.

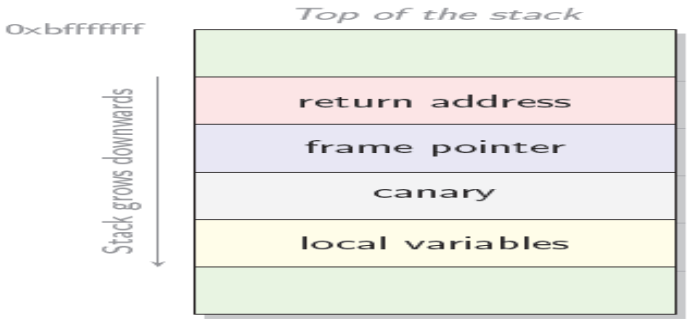
Disegno la stack da bar:

buf
buf
buf
buf
&sleep
arg
buf (l'indirizzo del primo elemento)
ret
push ebp (salvo vecchio frame pointer)
tmp
tmp
tmp
tmp
tmp
tmp
tmp
tmp

Ora se scrivo piu di 128 posso sovrascrivere fino a &sleep e nell'andare avanti verra chiamata la mia funzione.

COME FERMARE QUESTI ATTACCHI?
Bisogna scrivere codice in maniera preventiva, con un uso cautela della memoria. Spesso ci si dimentica anche delle free e non è del tutto automatizzato questo procedimento. Nessuna delle contromisure esistenti blocca completamente il problema del buffer overflow.
Possiamo però evitare alcune funzioni: molte funzioni in C non controllano la dimensione dell'input, infatti strcpy è una funzione sicura se e solo se la usa un programmatore attento ed oculato. Possiamo però far riferimento alle varianti in n, ossia ad esempio strncpy, la versione che invece copia esattamente n bytes. Attenzione però se la dimensione è più lunga di quella specificata da n nessuno null byte '\0' è inserito.
Ancora meglio sono strlcpy, strlcat, così come snprintf.
Ci sono poi combinazioni di trucchetti che possono evitare le situazioni sopra elencate.
CANARINI
L'obiettivo è di scovare un overflow dell'indirizzo di ritorno. Si inserisce un canarino nella stack all'inizio di una funzione (valore a 32 bit) tra l'indirizzo di ritorno e le variabili locali.
Tipi di canarino:

- Terminator
- Random
- Random XORL'epilogo controlla se il canarino è stato alterato. Lo svantaggio è che richiede la ricompilazione.



DEP / NX bit / W⊕X E ROP
Idea: si separano le aree di memoria eseguibili da quelle scriviibili. Una pagina di memoria non può essere eseguibile e scrivibile contemporaneamente. Da qui il nome "Data Execution Prevention (DEP).
Di conseguenza se si prova ad eseguire una parte di memoria non eseguibile (come la stack) l'attacco fallisce.

Per bypassare questa sicurezza dobbiamo chiarire alcuni punti:
- Possiamo manipolare un puntatore del codice (esempio return address).
- La stack è scrivibile.
- Conosciamo l'indirizzo di una funzione della LibC adatta.

L'idea su cui si basa il ROP (Return Oriented Programming) è quella di creare piccole catene di gadget (una serie di poche istruzioni che eseguono parte di codice che l'approccio DEP ci vieta di fare essendo la stack non eseguibile). Un gadget può essere estrapolato da una codice più ampio e stesso le funzioni della libC possono essere utilizzate. Se al valore di ritorno inseriamo un altro indirizzo di un gadget otteniamo questa catena che fa le cose che noi vogliamo. Un riuso del codice combinato.

ASLR (Address Space Layout Randomization)
Idea dietro ASLR:
- Si riorganizzano in maniera random le posizione delle aree di memoria dei dati chiave. (stack, data, text)
- Caso buffer overflow: l'attaccante così non saprà mai l'indirizzo dello shellcode.
- Caso return-into-libc: l'autore dell'attacco non conosce gli indirizzi delle funzioni della libC.

Problemi con questo approccio:
- Implementazioni a 32 bit usano pochi bit randomizzati.
- Di solito si applica esclusivamente al codice della libC.
- Così un attaccante può tentare l'exploit in area non randomizzata
- Perdite di informazioni:
    - I bug di divulgazione della memoria sconfiggono ASLR.
    - Nelle attuali implementazioni conoscere l'indirizzo di un puntatore a funzione è sufficiente per determinare tutti gli altri, poiché sono tutte ancore allo stesso offset tra loro rispetto all'implementazione non randomica. Saperne uno significa saperli tutti.