Sistemi di calcolo 2

Processo composto da: codice del programma, un insieme di dati, 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.
bootstrap program: parte all'avvio del pc, inizializza i registri della CPU, fa un checkup della memoria e dei dispositivi, carica l'OS nella memoria e lo avvia.
Caratteristiche fondamentali: è proprietario di uno spazio virtuale di memoria dove montare la propria immagine e 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.
I thread condividono tutti l'area di memoria del processo, svolgono un proprio task indipendente dagli altri, 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.
Se il processo che contiene thread in questione venisse sospeso o chiuso, quest'azione si proietterebbe anche su tutti i thread.
Due tipi di thread:

- **ULT (user level thread):** gestiti dall'applicazione,kernel gestisce solo il processo che li crea, funzionano su qualsiasi OS poiché implementati attraverso una libreria a livello utente, una chiamata a un thread bloccherebbe anche tutti gli altri, swiching senza bisogno del kernel
- **KLT (kernel level thread):** possono essere schedulati su core differenti, una chiamata bloccante non ha effetto sugli altri thread, lo switch richiedere l'intervento del kernel

Multiprocessing simmetrico:

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

SMP (Symmetric Multiprocessor): MIMD a memoria condivisa
Cluster: MIMD a memoria distribuita

Concorrenza

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).
Operazione Atomica: una funzione o un'azione implementata come una sequenza di una o più istruzioni che appaiono indivisibili; cioè nessun altro processo può 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***

Competizione: Due processi che non si vedono tra loro, possibili problemi di controllo sono la mutua esclusione, deadlock o starvation.
Cooperazione tramite condivisione: Processi che vedono altri processi indirettamente, possibili problemi di controllo sono la mutua esclusione, deadlock, starvation e coerenza dei dati.
Cooperazione tramite comunicazione: Processi che si vedono tra di loro, problemi di controllo sono deadlock o starvation

Resource Competition: più processi entrano in conflitto per l'uso di una stessa risorsa e vanno quindi affrontati questi problemi: mutua esclusione, deadlock, starvation

Hardware support mutual esclusion:

- Compare&Swap instruction (approccio busy waiting/starvation)

```
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 valori 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.
È 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 finché una data condizione non si verifica.
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.

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

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.
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.

- Due primitive fondamentali
  - Send(destination, message)
  - Receive(source, message)

Primitive bloccanti o non bloccanti

Rendevous: primitive bloccanti, ossia nel momento in cui un processo A invia un messaggio al processo B, il processo A si blocca finché B non lo riceve e B si blocca finché A non invia qualcosa

Non Blocking Send: viene bloccato solo il processo B ricevente, mentre il processo A una volta terminato l'invio del messaggio può continuare con il suo task.

Indirizzamento:

- **Diretto**: identifica ogni processo con un id, da specificare ogni qual volta bisogna inviare un messaggio
- **Indiretto**: 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.

Ricezione:

- **Esplicita**: designazione di un processo sender in modo tale da poterne specificare l'id
- **Implicita**: il parametro source della primitiva receive possiede un valore di ritorno quando è avvenuta la ricezione.

Socket

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.
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.

Edpoint per una connessione basata su TCP/IP di tipo sockaddr_in con 5 proprietà:

- **SIN_LEN**: lunghezza della struttura
- **SIN_FAMILY**: in questo corso AF_INET
- **SIN_PORT**: numero di porta
- **SIN_ZERO**: un array non utilizzato
- SIN_ADDR: struttura di tipo in_addr che contiene l'indirizzo IP a 32bit

Funzioni delle socket:

- **Socket():** crea un nuovo endpoint di comunicazione
- **Bind():** associa alla socket un indirizzo locale
- **Listen():** annuncia di essere pronta ad accettare connessioni
- **Accept():** stabilisce passivamente una connessione
- **Connect():** che tenta di stabilire attivamente una connessione
- **Send():** invia i dati
- **Receive():** riceve i dati
- **Close():** rilascia la connessione

Deadlock

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.
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 del prelascio: Un processo non è forzato a rilasciare una risorsa in suo possesso
4. Attesa circolare(conseguenza delle prime 3): 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

Deadlock prevention strategy: si cerca di eliminare una delle condizioni necessarie per il deadlock

- **Indiretto**: prevenire il verificarsi di una delle prime 3 condizioni necessarie
- **Diretto**: 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.
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.
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 $j > i$.

Deadlock avoidance strategy: dinamicamente si effettuano delle scelte in base al fatto che una richiesta potrebbe causare deadlock
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.

Deadlock detection strategy: provare a identificare la presenza di un Deadlock e prendere azioni per eliminarlo
Possiamo decidere di effettuare un check per il deadlock molto frequentemente cioè a ogni richiesta di risorsa o meno frequentemente.
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 finché la situazione di deadlock non si verifichi più, oppure prevenire l'uso di alcune risorse affinché il deadlock non esista più.

Algoritmo di Dekker(mutua esclusione 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

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 tentativo garantisce mutua 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.

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: finché 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 ciclo while interno sulla flag dell'altro finché non avrà 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 finché l'altro non setta il turno a me, dopodiché mi rimetto interessato e riparte l'algoritmo di verifica.

Algoritmo di Dijkstra(generalizza l'algoritmo di Dekker per N processi)

```
/* 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.

Client Computing

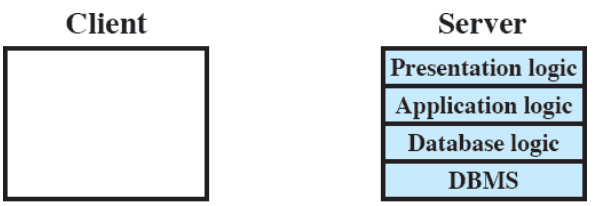
La chiave di una applicazione client/server è che i task dell'applicazione vengono svolti da entrambi i sistemi, mentre il loro sistema operativo o l'hardware potrebbe essere differente, ma poco importa poiché a livello di comunicazione usano lo stesso protocollo.

Componenti di applicazioni distribuite:

- 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

1. **Elaborazione basata su host**: non sono vere e proprie applicazioni client/server, hanno più un aspetto tradizionale
2. **Elaborazione basata su server**: elaborazione totale dei dati da parte del server, mentre il client offre solo una logica di

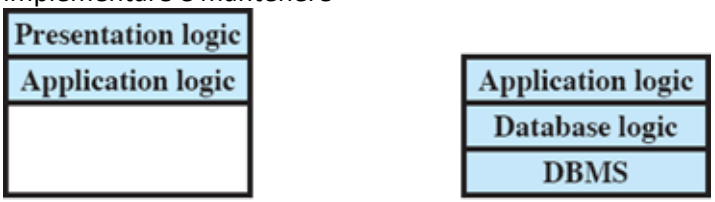


(a) Host-based processing

presentazione.

(b) Server-based processing

3. **Elaborazione basata sulla cooperazione**: elaborazione ottimizzata tra client e server, ma molto più complessa da implementare e mantenere



(c) Cooperative processing

4. **Elaborazione basata sul client**: elaborazione dei dati da parte del client, mentre le convalide e lo storage dei dati viene effettuato dal server.

(d) Client-based processing

Sistemi distribuiti:

- **Vantaggi**:
  - **Economici**: una serie di microprocessori offre un miglior rapporto qualità/prezzo dei mainframes
  - **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
  - **Distribuzione intrinseca**: Es. Catena di supermercati
  - **Affidabilità**: Se una macchina crasha, il sistema sopravvive
  - **Crescita incrementale**: Il sistema è modularmente espandibile
- **Vantaggi rispetto a pc indipendenti**: condivisione di dati, condivisione di risorse, comunicazione
- **Svantaggi**:
  - **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.

Problemi di progettazione dei sistemi distribuiti:

1. **Trasparenza**(Come far sembrare all'utente una serie di computer come un singolo computer):
   a. **Della posizione**: gli utenti non possono sapere dove fisicamente si trovano le risorse
   b. **Della migrazione**: le risorse devono poter essere spostate in tranquillità senza che i loro nomi cambino.
   c. **Di replica**: l'OS può effettuare copie aggiuntive di file e risorse senza che l'utente se ne accorga
   d. **Di concorrenza**: gli utenti non sono a conoscenza che altri utenti usino il sistema
   e. **Del parallelismo**: uso automatico del parallelismo senza dover programmare esplicitamente in quella direzione. (IL SANTO GRAAL)
2. **Flessibilità**: Rende più facile fare modifiche
3. **Affidabilità**: I sistemi distribuiti devono essere più affidabili di sistemi singoli e deve essere protetto da guasti o errori
4. **Performance**: rendere il sistema robusto ai guasti e agli errori ovviamente fa perdere di prestazioni
5. **Scalabilità**: 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.

Algoritmo di Backery(panettiere)
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).

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.

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 finché 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

Middleware: Costituito da interfacce di programmazione standardizzati e protocolli che si interpongono tra l'applicazione e il sistema operativo
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.

SOA (Service oriented architecure): Organizza le funzioni aziendali in una struttura modulare invece che applicazioni monolitiche per ciascun dipartimento

MESSAGGE PASSING NEI SISTEMI DISTRIBUITI
Una comunicazione molto semplice: solo due primitive sono richieste, Send e Receive.

Affidabilità vs Inaffidabilità
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
- "**inaffidabile**" ossia il messaggio potrebbe essere spedito tranquillamente senza interesse nel sapere una condizione di successo o meno

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).
Con un approccio blocking invece un processo che invia viene bloccato finché il messaggio non è stato inviato ed è stata ricevuta la conferma di spedizione e in ricezione il processo è bloccato finché il messaggio non è stato conservato in un buffer.

RPC (remote procedure calls)
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.
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.
- Vantaggi: Ampiamente accettato, documentazione facile, generazione automatica, facilmente trasferibile

Parameter Passing
Passare dei parametri per valore è molto semplice, molto più complicato è passare un parametro per riferimento poiché è necessario un unico puntatore ad un unico ampio sistema per ogni oggetto

Parameter Representation: standardizzare alcuni formati primitivi di oggetti comuni come interi e caratteri così da poter convertire tutti i tipi di parametri specifici in tipi standardizzati

Client/Server binding
- **Non persistent binding**: una connessione logica viene 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
- **Persistent binding**: una connessione stabilita al momento della chiamata e mantenuta al momento della consegna del valore di ritorno

Sincrono vs Asincrono(simile al concetto di blocking o non blocking sui message passing)
L'approccio tradizionale è quello blocking cioè sincrono che prevede che il processo chiamante aspetti il valore di ritorno della RPC
L'approccio nonblocking o asincrono non blocca il chiamante potendo quindi ricevere le risposte solo quando se ne ha il bisogno

Meccanismo orientato agli oggetti:
Client e server si inviano messaggi attraverso oggetti

- **Object broker** come intermediario

Semantica della chiamata RPC(interruzione):
- 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

Quanto spesso si può eseguire sotto errore?
- **AT LEAST ONCE** (eseguita finché non riceve un ack)
- **AT MOST ONCE** (al massimo una)
- **EXACTLY ONCE**

Clusters (ogni pc viene detto nodo): un gruppo di computer interconnessi che lavorano insieme come se fossero un'unica unità.
- **Benefici**:
    - **Assoluta scalabilità**: è possibile creare cluster così grande da superare la potenza della più grande macchina singola.
    - **Scalabilità incrementale**
    - **Alta disponibilità**: il fallimento di un nodo non è una condizione critica del sistema
    - **Alto rapporto prezzo/performance**

Metodi di clustering:
- **Passive standby**: prevede un server secondario che si attiva in casi di fallimento del principale
- **Active secondary**: il server secondario è anche usato per processare task
- **Server separati** (alto overhead): I dati sono continuamente copiati dal primo al secondo server
- **Server connessi a dischi**: i server sono collegati agli stessi dischi ma ognuno ha i propri
- **Server che condividono dischi**: richiede un software che gestisca l'accesso (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 dà 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.

Fallover = la funzione di cambiare sistema ad un'applicazione o alle risorse nel momento in cui quello in uso fallisce
Fallback = rispristino delle applicazioni e delle risorse sul sistema originale una volta riparato
Load Balancing = un cluster richiede una capacità efficace di bilanciare il carico di lavoro tra i PC disponibili

Parallelizing computation

Parallelizing compiler: determina, in fase di compilazione quali parti possono essere eseguite in parallelo
Parallelized application: il programmatore scrive l'applicazione dall'inizio con l'intento di runnarla su un cluster
Parametric computing

Comparazione Cluster e System multiprocessor:
- Entrambi forniscono una configurazione multi-processor per supportare applicazioni che richiedono prestazioni elevate
- Entrambe le soluzioni sono commercialmente disponibili
- SMP ha più esperienza
- SMP è più facile da gestire e configurare
- I prodotti SMP sono ben consolidati e stabili
- I cluster sono migliori per scalabilità
- I cluster sono migliori in termini di disponibilità

PIPE

A livello di OS sono dei Buffer

Le PIPE vengono utilizzate nella comunicazione di processi relazionati, cioè conseguenti ad una fork, altrimenti si usano le FIFO (named PIPE).

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.

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.

Clock logico = Variabile opportunamente aggiornata durante la computazione che contiene il valore corrente di timestamping etichettati da eventi
Clock fisico = clock hardware (oscillatore al quarzo) caratterizzato da un parametro detto DRIFT RATE che 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).
Clock guasto: se non rispetta le condizioni di correttezza. NB clock corretto ≠ clock accurato.

Clock software: $\mathbf{Ci(t)= \alpha Hi(t) + \beta}$ che approssimativamente misura l'istante di tempo fisico t per il processo P_i.
UTC = coordinated universal time, è basato su orologi atomici ma occasionalmente aggiustato utilizzando il tempo astronomico.

SINCRONIZZAZIONE DEI CLOCK FISICI

- **Sincronizzazione esterna**: i clock sono sincronizzati con una sorgente di tempo S, in modo che, dato un intervallo I di tempo reale: $|S(t) - Ci(t)| < D$ 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 $|S(t) - Ci(t)| < D$ per $i = 1,\ldots,N$ nell'intervallo I. In questo caso i due clock Ci e Cj si accordano all'interno dell'intervallo D.

Time services:

- **Centralizzato:**
  - **Request-driven (sync esterna) algoritmo di cristian**
- **Broadcast based:**
  - **Berkeley unix algorithm (sync interna)**
- **Decentralizzato:**
  - **Network time protocol (sync esterna)**

Algoritmo di cristian

Algotirmo 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

Network time protocol sincronizza client a UTC

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.

Modalitá di sincronizzazione:

- **Multicast**: un server manda in multicast il suo tempo agli altri che impostano il tempo ricevuto assumendo un certo ritardo prefissato
- **Procedure Call**: un server accetta richieste da altri computer
- **Simmetrico**: coppie di server scambiano messaggi contenenti informazioni sul timing

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

DEFINIZIONE DELLA RELAZIONE AVVENUTO-PRIMA DI LAMPORT

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)

Clock logico: contatore software monotonicamente crescente
Clock vettoriale

RICART-AGRAWALA ALGORITHM (mutual exclusion in distributed setting)

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.

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**:

- **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'entità che 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.
- **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:

- **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 sì 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.
Attacchi attivi: Si tratta di modificare flussi di dati o crearne di falsi
- **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

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

Attacchi interni
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:

- 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

Logic Bomb: Codice incorporato in un programma legittimo pronto ad "esplodere" quando vengono soddisfatte determinate condizioni

Trojan Horse
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: Programmi che possono essere spediti invariati a un insieme vasto di piattaforme ed eseguiti con semantica identica.

Multiple-Threat malware: Un attacco combinato utilizza più metodi di infezione o trasmissione per massimizzare la velocità del contagio e la gravità dell'attacco

Virus:

- Software che infetta altri programmi modificandoli:
  - Porta con sé 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 sé 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 per target:

- **Boot Sector Infector**: infetta un record di avvio principale e si diffonde quando un sistema viene avviato dal disco contenente il virus

- **File Infector**: infetta i file che l'OS o la shell considerano eseguibili
- **Macro Virus**: infetta file con codice macro interpretato da un'applciazion

Classificazione per occultamento:
- **Virus crittografico**: una chiave di crittografia casuale crittografa il virus
- **Si nasconde dall'antivirus**
- **Virus polimorfo**: muta con ogni infezione. Le copie sono funzionalmente equivalenti ma hanno pattern di bit totalmente differenti
- **Virus metamorfo**: mutua con ogni infezione. Si riscrive completamente dopo ogni iterazione.

Macro virus:
- 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

Email 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 botnet 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 sistemi
    - Una strategia di localizzazione di macchine vulnerabili (un processo 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 finché 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:

- 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 più 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 cracker program può essere utilizzato su un'altra macchina. Questo permette di scorrere milioni di password possibili in un tempo ragionevole.

Autenticazione basata su token

- **Memory Cards**: possono conservare dati ma non processarli. La più comune è la carta di credito con una banda magnetica sul retro
- Smart Cards: : include un processore, l'interfaccia prevede una tastiera e un display per l'interazione uomo/token. L'autenticazione può essere statica, dinamica generando una One Time Password o attraverso delle domande effettuate all'utente.

- Fisiche statiche:
    - Impronta digitale
    - Profilo della mano
    - Caratteristiche facciali
    - Scansione retina o iride
- Dinamiche:
    - Timbro della voce
    - Firma

Controllo dell'accesso

Una politica di controllo degli accessi stabilisce quali tipi di accesso sono consentiti, in quali circostanze e da chi

- **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
- **Vincolato(MAC):** controlla l'accesso basandosi sul tipo di utente di cui vengono controllati i permessi d'accesso da un sistema
- **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 basato sull'host (IDS)

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.

Approccio Antivirus:

- **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

Generic Decryption (GD)
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

- Uno scanner GD 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 (sviluppato da IBM e perfezionato da Symnatec)
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

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
La sua propagazione genera una considerevole attività di rete

- **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
- **Filtraggio dei worm in base al loro contenuto**: simile alla prima, ma si concentra sul contenuto del worm piuttosto che sulla firma.
- **Blocco del worm basato sulla classificazione del payload**: le tecniche di rete esaminano i pacchetti per vedere se contengono worm.
- **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.
- **Limitazione del rate**: limita la velocità del traffico da un host infetto.
- **Blocco del rate**: blocca immediatamente il traffico in uscita quando viene superata una soglia in termini di velocità di connessione.

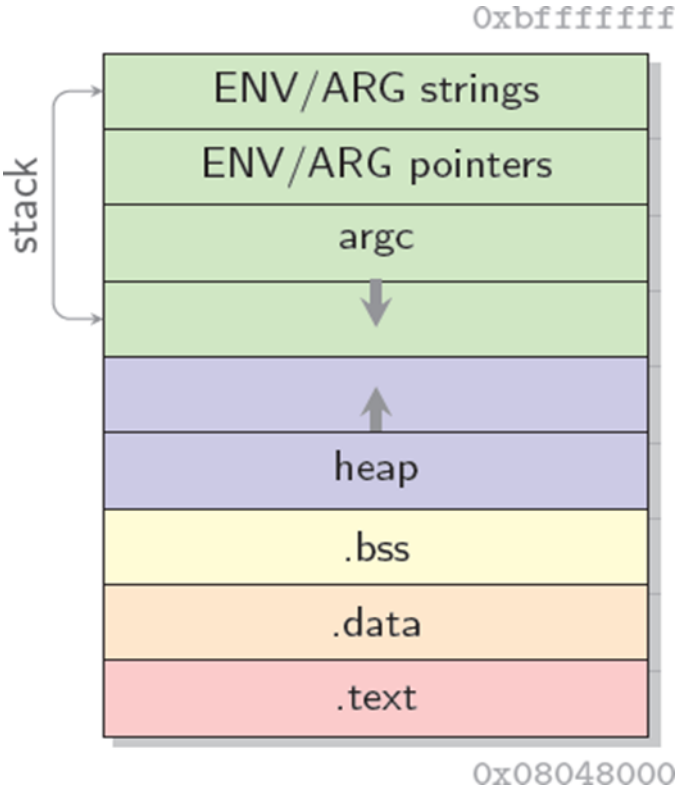
Contromisure ai bot
L'obiettivo principale è cercare di rilevare e disabilitare la botnet durante la sua fase di costruzione

- **IDS e i digital immune system**: rilevano l'attacco in corso

CONTROMISURE AI ROOTKIT
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

Memory corruption

- Buffer overflows
- Out-By-One errors
- Type confusion errors
- Use-After-Free



La stack di un programma contiene i frame dello stack (ossia i record di attivazione) per ogni funzione invocata
Ci sono però alcuni registri di supporto della CPU per gestirla correttamente:

- **Stack pointer register**: ESP
- **Frame pointer register**: EBP

STACK OVERFLOW

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

```
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, di solito una shell

Vettore da iniettare:

- **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

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

È 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 dichiarazioni 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 copiate 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 verrà chiamata la mia funzione

COME FERMARE QUESTI ATTACCHI?

Bisogna scrivere codice in maniera preventiva, con un uso cautela della memoria
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.

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.

- Terminator
- Random
- Random XORL

DEP (Data execution prevention): si separano le aree di memoria eseguibili da quelle scrivibili. Una pagina di memoria non può essere eseguibile e scrivibile contemporaneamente
Di conseguenza se si prova ad eseguire una parte di memoria non eseguibile (come la stack) l'attacco fallisce

ROP (Return oriented programming): 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 parte di codice più ampio e possono essere utilizzate le funzioni della libC. Se al valore di ritorno inseriamo un altro indirizzo di un gadget otteniamo questa catena che fa le cose che noi vogliamo

ASRL (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.