Appunti di Sistemi di Calcolo 2

Soykat Amin

Versione 1.1, Gennaio 2024

Con il contributo volontario di Gabriele Onorato

Indice

1 Processi e Thread

1.1 Processi

Ogni programma è un processo e ha una esecuzione, comprende codice, dati e attributi. Gli attributi di un processo sono: identificatore, stato, priorità, puntatori, stato dei registri della CPU, stato dell'I/O, indirizzo della prossima istruzione(Program Counter). Viene salvato tutto nel Process Control Block (PCB), creato e gestito dal Sistema Operativo che permette così di supportare più processi. Il sistema operativo inoltre permette all'utente di creare processi, a ciascuno garantisce tutte le risorse necessarie e permette a più processi di comunicare fra di loro.

Per la gestione dei processi si utilizzano le seguenti Unix System Calls: *fork()*, *wait()*, *exit()*. Ogni processo viene creato da un altro processo tramite una *fork()*, vengono chiamati processo padre e processo figlio. Sono quasi identici all'inizio ma si evolvono in maniera diversa perché nel codice è possibile decidere cosa dovrà fare il padre e cosa dovrà fare il figlio. Entrambi possono fare altre *fork()*. Si crea quindi un albero dei processi con una radice creata dal Sistema Operativo all'avvio. Ogni processo può decidere di attendere che il figlio termini il suo compito prima di proseguire tramite una *wait()*.

All'avvio il Bootstrap Program inizializza la memoria, i controller dei device e i registri della CPU. Poi carica il Sistema Operativo in memoria e lo fa partire. Questo creerà il primo processo (init) e si metterà in attesa di eventi quali hardware o software interrupt (trap).

Una *fork()* restituisce -1 in caso di errore, 0 nel figlio e l'identificatore del figlio nel padre. Il figlio eredita una copia identica della memoria, dei registri della CPU e dei file aperti dal padre. Anche il PCB del figlio sarà uguale a quello del padre, eccetto l'ID processo.

La struttura del codice è la seguente.

```
ret = fork();
switch(ret) {
    case -1:
        perror("fork");
        exit(1);
    case 0: // I am the child
        <code for child >
        exit(0);
    default: // I am parent ...
        <code for parent >
        wait(0);
}
```

Se la *fork()* è usata per creare un nuovo processo che deve eseguire un programma, è necessario utilizzare *exec()*. Quando viene eseguita, il Sistema Operativo sostituisce l'immagine di memoria con quella del programma da eseguire. Il nuovo programma deve avere un*main()*.

Alla fine dell'esecuzione di un processo, si utilizza la *exit(status)*, il cui parametro è accessibile dal padre e viene usato per indicare se il processo è terminato correttamente. Con la *exit()* vengono eseguite tutte le funzioni specificate con *atexit(fun)* e *on_exit(fun)*, vengono chiusi i file, le connessioni e viene deallocata la memoria. Se il processo ha figli, essi vengono assegnati ad *init*. Se il padre è ancora vivo, il processo diventa "zombie" per mantenere il risultato da restituire al padre qualora lo richiedesse con una *wait()*; in caso contrario, il figlio viene terminato. Il padre può attendere figli attraverso la *wait()*, che attende la terminazione del primo figlio e restituisce il suo PID (o -1 se non ci sono figli da attendere). Se si passa come parametro di una *wait()* un puntatore ad una variabile, in questa verrà messo come valore lo stato restituito dal figlio. La chiamata di *waitpid()* serve per attendere un figlio specifico.

1.2 Thread

Un processo ha due caratteristiche:

- Scheduling/Execution: Segue un percorso di esecuzione che potrebbe intrecciarsi con altri processi.
- Resource Ownership: Include uno spazio di indirizzamento virtuale che tiene l'immagine di memoria del processo.

Entrambe le caratteristiche sono trattate indipendentemente dal Sistema Operativo.

Un **thread** è una sequenza di istruzioni all'interno di un programma che può essere eseguita in modo indipendente dal resto del programma. Un Sistema Operativo può supportare un processo con un thread, più processi con un thread ciascuno, un processo con più thread, più processi con più thread ciascuno.

Il **Multithreading** è l'abilità di un Sistema Operativo di supportare più thread contemporaneamente, in questo caso ogni processo deve avere un indirizzo di memoria virtuale in cui mantenere la sua immagine e l'accesso deve essere protetto ai processori, ad altri processi, ai file e alle risorse I/O. Ogni thread ha uno stato di esecuzione(Running, Readym Blocked), un context data salvato quando non è Running, uno stack di esecuzione, spazio statico per variabili locali, accesso alla memoria e alle risorse del processo di cui fa parte. Un processo con singolo thread ha un Process Control Block, un User Address Space, un User Stack e un Kernel Stack. Un processo con più thread ha invece un PCB, un User Address Space e, per ciascun thread, un Thread Control Block, un User Stack e un un Kernel Stack.

Tra i benefici dei thread abbiamo che la creazione e la terminazione impiega meno tempo rispetto a un processo, la comunicazione fra thread non necessita l'invocazione del kernel, il passaggio da un thread all'altro necessita meno tempo.

Si usano i thread per lavorare sia in background che in foreground, per processamento asincrono, per programmi con struttura modulare e per la velocità di esecuzione (posso proseguire mentre un altro thread attende ad esempio risorse I/O).

Sospendere un processo significa sospenderne tutti i thread, così come terminarlo significa terminarne tutti i thread. Il Sistema Operativo può gestire contemporaneamente tutti i thread di un processo. I thread hanno uno stato di esecuzione e possono sincronizzarsi fra di loro. I possibili stati sono: Running, Ready e Blocked. Per cambiare lo stato di un thread posso fargliene creare un altro, bloccarlo, sbloccarlo o terminarlo.

Un esempio di utilizzo è per le **Remote Procedure Call** (chiamate di funzioni in esecuzione su un server remoto), se ho due richieste potranno essere fatte consequenzialmente (prima richiesta, attesa risultato, seconda richiesta, attesa risultato) o altrimenti tramite la creazione di un altro thread che si occuperà della seconda richiesta contemporaneamente alla prima.

I thread possono essere gestiti su due livelli: **User Level Thread** (ULT) o **Kernel Level Thread** (KLT).

Nel primo caso il Kernel non si accorge dell'esistenza dei thread che vengono gestiti interamente da una libreria, è limitante perché il processo avrà assegnato un solo core e non potrà quindi eseguire più thread contemporaneamente, ma si risparmia tempo non passando per il kernel e permette l'esecuzione su ogni sistema. Nel secondo caso il kernel manterrà il contesto dell'informazione fra il processo e i thread e lo scheduling sarà fatto a livello di thread, permette di avere in esecuzione contemporanea più thread su più core anche se causa una perdita di tempo dovuta al passaggio per il kernel. Esistono degli approcci combinati.

1.3 Caso di Studio: PThread

Nei sistemi UNIX si utilizza la libreria **PThread** (Posix Thread). Permette di organizzare il programma in diversi task indipendenti che possono essere eseguiti contemporaneamente, senza comportare la perdita di tempo dovuta ad una *fork()*.

Si può programmare con l'utilizzo di thread attraverso due tipi di schemi: **manager/worker** (un thread ne crea altri e aspetta i risultati) o **pipeline** (divido il task in più thread e li lancio tutti insieme).

Tutti i thread hanno accesso alla stessa memoria globale del processo oltre ad averne una privata.

Un codice si dice **thread-safe** se i thread possono essere eseguiti contemporaneamente senza produrre risultati inattesi e senza minare la correttezza dei dati in memoria condivisa.

Un thread si crea tramite una *pthread_create()* e sarà al pari del thread che lo ha creato. Si termina con una *pthread_exit()* al completamento del lavoro assegnato, questa permette inoltre di passare un valore che viene poi catturat dalla join (a differenza della *exit()*, non libera le risorse). Un thread può terminarne un altro tramite una *pthread_cancel()*. L'esecuzione di una *exit()* in un qualsiasi thread causa la terminazione di tutto il processo. Se il thread contenente il processo main termina, questo causa la terminazione di tutti i thread del processo. Per evitare questo comportamento bisogna chiamare *pthread_exit()* alla fine del main o alternativamente *pthread_detach()* sui thread creati.

1.4 Symmetric Multiprocessing

In passato si lavorava solo su macchine sequenziali mentre oggi abbiamo vari sistemi paralleli. Un approccio comune è il **Symmetric MultiProcessor** (SMP). La tassonomia di Flynn categorizza i sistemi in:

- SISD, singola istruzione su singolo dato;
- SIMD, singola istruzione su dati multipli;
- MISD, multiple istruzioni su un singolo dato;
- MIMD, multiple istruzioni su dati multipli.

Una tipica architettura SMP è composta da una memoria virtuale, un insieme di I/O adapter e un insieme di processori (ciascuno con cache L1 ed L2), tutto collegato tramite bus. Bisogna considerare come gestire pianificazione, sincronizzazione, memorie condivise.

2 Concorrenza

2.1 Generalità

Un Sistema Operativo può gestire multiprogrammazione (più processi su un core), multiprocessi (più processi su più core), gestione distribuita (più processi su più macchine).

La **concorrenza** può presentarsi in tre diversi contesti: applicazioni multiple (condivisione del tempo fra varie applicazioni), applicazioni strutturate (gestione applicazioni con più processi), struttura dei sistemi operativi (loro stessi sono composti da processi e thread).

Terminologia utile:

- Operazione atomica: una sequenza di istruzioni che non può essere interrotta. La sequenza viene eseguita interamente come un gruppo oppure non viene eseguita;
- Sezione critica: sezione di codice che richiede accesso a risorse condivise. Se acceduta da un thread o processo, tutti gli altri non potranno accedervi;
- Mutua esclusione: condizione che quando un processo in una sezione critica accede a risorse condivise, nessun altro processo sia in una sezione critica che accede alle stesse risorse condivise;
- Corsa alla risorsa (Race Condition): situazione in cui due o più thread o processi leggono o scrivono allo stesso tempo in una memoria condivisa, per cui il risultato dipende dal timing. Il dato finale sarà quello scritto dall'ultimo processo, che sovrascriverà il dato già scritto da altri processi;
- Deadlock: un processo va in deadlock se attende una situazione che non si verificherà mai, ad esempio due processi che attendono ciascuno la terminazione dell'altro;
- Livelock: due o più processi cambiano continuamente il loro stato in risposta ai cambiamenti di stato dell'altro, senza fare però lavoro utile;
- Starvation: situazione in cui un processo è pronto a ripartire ma non viene scelto dallo scheduler e rimane inattivo. Potrebbe bloccare una risorsa e non permettere agli altri processi di usarla.

Bisogna risolvere questi problemi stando attenti a far sì che il nostro processo vada a termine indipendentemente dagli altri.

Se ho due thread intervallati o in parallelo, si presentano in entrambi i casi gli stessi problemi. La velocità del nostro processo non è prevedibile perché dipende anche dalle attività di altri processi e da come il Sistema Operativo gestisce le interrupt e lo scheduling.

Le difficoltà della concorrenza riguardano:

- Condivisione di memorie globali;
- Gestione da parte del Sistema Operativo delle risorse;

- difficoltà nel trovare errori di programmazione poiché le esecuzioni sono non deterministiche e riproducibili.

Il Sistema Operativo deve:

- Conoscere tutti i processi in esecuzione;
- Allocare e deallocare le risorse per ogni processo attivo;
- Proteggere i dati e le risorse fisiche di ogni processo dalle interferenze di altri processi;
- Cercare di garantire che processi e output siano indipendenti dalla velocità di esecuzione.

Processi concorrenti sono in conflitto quando competono per l'utilizzo della stessa risorsa, in questo caso bisogna affrontare tre problemi:

- La necessità di mutua esclusione;
- Il deadlock;
- La starvation.

La **mutua esclusione** deve permettere l'accesso alla sezione critica se e solo se nessun altro processo è nella sezione critica, inoltre deve bloccare gli altri che tentano di accederci. Un processo deve rimanere nella sezione critica per un tempo finito. Bisogna sempre stare attenti a non causare deadlock e starvation.

È possibile applicare la mutua esclusione a livello hardware: disabilito i segnali di interrupt del sistema operativo così che nessuno fermi l'esecuzione della sezione critica, ma l'efficienza peggiora e questo metodo non funziona con sistemi multiprocessore. Servono istruzioni come la Compare&Swap: si confronta un valore in memoria con un valore di test, se sono uguali si scambia il valore in memoria con uno nuovo. Tutto ciò è fatto atomicamente ed è disponibile in X86, IA64, sparc, IBM.

```
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;
}
/*program mutual exclusion */
const int n = /* number of processes */;
int bolt;
void P (int i) {
    while (true) {
        while (compare_and_swap (&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main() {
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

Questo schema di programma prevede un insieme di thread che si fermano alla sezione critica. La variabile *bolt* è settata a 0 e quindi il primo thread che riesce a leggerla sostituisce il suo valore con 1 ed entra nella sezione critica. I thread che successivamente proveranno ad accedere a *bolt* leggeranno come valore 1, di conseguenza la compare fallirà e non verrà eseguito lo swap. Questi thread continueranno a provare a fare la *compare_and_swap()* fino a che il primo thread uscirà dalla sezione critica e rimetterà il valore di bolt a 0. A questo punto un altro dei thread entrerà in sezione critica.

Altrimenti si può usare l'istruzione *Exchange* che scambia il valore contenuto in un registro con un valore in memoria, è disponibile in Pentium and Itanium (IA64).

```
void exchange (int *register, int *memory) {
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
/*program mutual exclusion */
const int n = /* number of processes */;
int bolt;
void P (int i) {
    while (true) {
        int keyi = 1;
        do exchange (&key, &bolt) while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main() {
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

Queste implementazioni si possono applicare su un qualsiasi numero di processi su uno o più processori che condividono la stessa memoria principale. Inoltre è molto semplice e può supportare diverse sezioni critiche semplicemente definendo più variabili di controllo.

Fra gli svantaggi abbiamo:

- Il Busy-Waiting poiché il processo che attende continua a consumare tempo del processore;
- La starvation quando un processo lascia la sezione critica e più di un processo sta attendendo di entrarci;
- La possibilità che si verifichi un Deadlock;
- Non è disponibile in ogni architettura.

Possiamo gestire la concorrenza anche grazie a meccanismi software:

- Semaforo: prevede un valore intero usato per segnalazioni fra i processi. Si possono effettuare solo tre operazioni: intialize, decrement e increment. L'operazione di decremento può bloccare un processo, l'operazione di incremento può sbloccare un processo. Nella pratica decremento quando voglio entrare in sezione critica ed incremento quando ne esco;
- Semaforo binario: semaforo che prevede l'utilizzo unicamente di valori 0 o 1;
- Mutex: simile ad un semaforo binario ma il processo che blocca il mutex (ovvero setta il valore a 0) deve essere quello che lo sblocca (ovvero setta il valore ad 1);
- Condition variable: una variabile blocca il processo fino a che una condizione non si avvera;
- Monitor: contiene dei metodi, se un metodo è chiamato da un processo allora nessun metodo potrà essere chiamato finché non termina il primo. Può prevedere una coda di processi che attendono.
- Event flag: un processo è bloccato fino a che certi bit non sono settati come vogliamo;
- Mailbox/Messaggi: i processi si scambiano messaggi come meccanismo di blocco e sincronizzazione;
- Spinlocks: si eseguono cicli infiniti aspettando la modifica di una variabile che indica disponibilità.

2.2 Semaforo

Si definisce una variabile intera in cui è possibile modificarla solamente con tre operazioni:

- Inizializzazione a valore intero non negativo;
- *semWait()* decrementa il valore;
- *semSignal()* incrementa il valore.

Non posso sapere se un processo si bloccherà o no fino a che non decremento il semaforo, né quale verrà bloccato fra due processi concorrenti.

In un sistema uniprocessore, se due processi si stanno eseguendo concorrentemente, non è possibile sapere quale processo continuerà dopo una semWait.

Non è possibile sapere se ci sono altri processi in waiting.

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait (semaphore s) {
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */
        /* block this process */
    }
}
void semSignal (semaphore s) {
    s.count++;
    if (s.count <= 0) {
        /* remove process P from s.queue */
        /* place process P on ready list */
    }
}
```

In un semaforo forte il processo che è stato bloccato per più tempo è il primo rilasciato dalla coda (FIFO), in un semaforo debole l'ordine in cui i processi escono dalla coda non è specificato.

```
/*program mutual exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P (int i) {
    while (true) {
        semWait (s);
        /* critical section */;
        semSignal (s);
        /* remainder */;
    }
}
void main() {
    parbegin (P(1), P(2), ..., P(n));
}
```

Le operazioni di *semWait* e *semSignal* devono essere necessariamente implementate come atomiche poiché la manipolazione di un semaforo è a sua volta un problema di mutua esclusione. Possono essere implementate in hardware o firmware con l'utilizzo di algoritmi come quello di Dekker o di Peterson.

Per *semWait* e *semSignal* è possibile utilizzare la struttura di programma con mutua esclusione attraverso *compare&swap* (utilizzando come flag un nuovo intero nella struttura del semaforo) o altrimenti posso inserire "inhibit interrupts" all'inizio della funzione e "else allow interrupts" alla fine della funzione.

Implementazione con la *compare&swap*:

```
semWait(s) {
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
```

```
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0) */;
    }
    s.flag = 0;
}
semSignal(s) {
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

Implementazione con le interrupt:

```
semWait(s) {
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else allow interrupts;
}
semSignal(s) {
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

In C, lo standard Posix mette a disposizione quattro metodi per il semaforo.

```
int sem_init(sem_t * sem, int pshared, unsigned value); //initialization
int sem_wait(sem_t *sem); //wait
int sem_post(sem_t *sem); //signal
int sem_destroy(sem_t *sem); //destruction
```

Parametri:

- sem: il semaforo con cui intendiamo lavorare;
- pshared: 0 se il semaforo viene usato da thread, 1 se usato da processi;
- value: il valore di partenza (quante risorse si possono condividere).

I metodi ritornano -1 in caso di errore, 0 altrimenti.

2.3 Monitor

I **Monitor** rendono più semplice l'utilizzo dei semafori. In C non c'è una libreria ufficiale che li implementi. Può contenere delle funzioni e delle variabili locali visibili solo dalle funzioni del monitor. Quando una funzione viene chiamata, nessun'altra potrà essere chiamata finché non termina la prima. Per sincronizzare tutto si usano delle **condition variables**, contenute nel monitor e accessibili solo dentro a questo:

- cwait(c), che sospende l'esecuzione del processo chiamante in base alla condizione c;
- csignal(c), che riprende l'esecuzione del processo bloccato da una *cwait* sulla stessa condizione.

2.4 Message Passing

Il **Message Passing** è uno degli approcci utile nella gestione di sistemi distribuiti che quindi non hanno memoria in comune. Permette la sincronizzazione e lo scambio di informazioni fra i processi.

Le primitive fondamentali sono *send(destination, message)* e *receive(source, message)*.

La comunicazione fra due processi implica la sincronizzazione; il ricevitore non può ricevere un messaggio finché viene mandato da un altro processo.

Quando chiamo una *receive* ho due possibilità, se un messaggio è in attesa di essere consegnato allora proseguo l'esecuzione, se nessun messaggio è stato inviato allora attenderò di riceverne uno o alternativamente abbandonerò il tentativo di *receive* e proseguirò.

Quando chiamo una *send* posso attendere una ricevuta di consegna o proseguire con l'esecuzione. Con la struttura BlockingSend-BlockingReceive sia il *sender* che il *receiver* sono bloccati fino a che il messaggio non è consegnato. La struttura NonblockingSend-BlockingReceive è la combinazione più utilizzata e permette di inviare più messaggi a diverse destinazioni rapidamente. Con la struttura NonblockingSend-NonblockingReceive nessuno dei due deve attendere.

Per la consegna di messaggi posso usare indirizzamento diretto o indiretto. Nel caso di indirizzamento diretto viene specificato il destinatario nella *send*, mentre nella *receive* è facoltativo il mittente. Nel caso di indirizzamento indiretto il messaggio è mandato ad una *mailbox* e il destinatario lo prende da lì. La mailbox può essere una semplice coda che mantiene i messaggi fino a che il *receiver* non li ha presi.

Ogni messaggio deve avere una struttura definita contenente tipo del messaggio, source e destination ID, lunghezza del messaggio, informazioni di controllo e contenuto del messaggio.

Implementazione di programma con mutua esclusione che sfrutta il Message Passing.

```
/*program mutual exclusion */
const int n = /* number of processes */;
void P (int i) {
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */;
        send (box, msg);
        /* remainder */;
    }
}
void main() {
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), ..., P(n));
}
```

2.5 Problema Produttore / Consumatore

Ho uno o più produttori che generano dati e li inseriscono in un buffer. Uno o più consumatori che prendono dal buffer un dato alla volta. Solo un consumatore o produttore alla volta può accedere al buffer. Bisogna garantire che un produttore non inserisca dati in un buffer pieno e che un consumatore non rimuova dati da un buffer vuoto. Supponiamo di avere un buffer infinito per cui i produttori non termineranno mai lo spazio e troviamo un modo di gestire i consumatori affinché non leggano dati se non ce ne sono disponibili. Nel caso in cui il buffer sia limitato ho bisogno di un semaforo che impedisca di scrivere in un buffer pieno. Inoltre potrei evitare di usare lo stesso semaforo s per accedere al buffer per produttori e consumatori, utilizzando due semafori s1 ed s2. Nel caso in cui io abbia un solo produttore o un solo consumatore avrò solo il semaforo s1 o solo s2.

Soluzione per buffer infinito:

```
/* program producer-consumer */
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();
    }
}
```

Soluzione per buffer finito:

```
/* program producer-consumer */
const int sizeofbuffer = /* buffer size */;
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();
    }
 }
```

Questo problema si può risolvere anche con i monitor. Infatti aiuta molto il fatto che solo un processo alla volta può eseguire il monitor e anche il fatto che ci sono dati locali che solo il processo che esegue il monitor può accedere.

2.6 Problema Readers/Writers

Il problema **Readers/Writers** prevede un'area di memoria condivisa fra più processi, alcuni che possono solo leggere e altri che possono solo scrivere. Vogliamo che un solo scrittore alla volta operi sull'area di memoria mentre non diamo limiti al numero di lettori che operano contemporaneamente. Se uno scrittore sta operando sulla memoria, non vogliamo che sia consentito leggere.

Soluzione:

```
/* program Readers/Writers */
int readcount = 0;
semaphore x=1, wsem=1;
writer: while (true) {
    semWait (wsem);
    WRITEUNIT();
    semSignal (wsem);
}
reader: while (true) {
    semWait (x);
    readcount++;
```

```
    if (readcount == 1) semWait (wsem);
    semSignal (x);
    READUNIT();
    semWait (x);
    readcount--;
    if (readcount == 0) semSignal (wsem);
    semsignal (x);
}
```

---

3 Deadlock

Il **Deadlock** è una situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa. È permanente e non ha soluzione.
Le risorse si dividono in:

- Riutilizzabili, possono essere utilizzate in sicurezza solo da un processo alla volta e non sono eliminate dopo l'uso (es. I/O, processore, database, file, semafori);
- Consumabili, possono essere create e distrutte (es. interrupt, segnali, messaggi).

Le condizioni per il deadlock sono:

- Mutua esclusione: Ogni risorsa può essere utilizzata da un solo processo alla volta;
- Hold-and-wait: Un processo può detenere risorse e richiedere altre risorse;
- No preemption: Un processo non può essere forzato a rilasciare una risorsa che detiene;
- Circular wait: Ciascuno dei processi coinvolti nel deadlock sta aspettando una risorsa che è detenuta da un altro processo coinvolto nel deadlock.

Le prime tre condizioni sono necessarie ma non sufficienti.
Possiamo avere 3 approcci risolutivi:

- Prevengo: Elimino le condizioni che lo generano. Posso attuare prevenzione indiretta (evito le tre condizioni necessarie) o diretta (evito circular wait). Nel caso della mutua esclusione, questa deve essere supportata dal Sistema Operativo e la possiamo impostare meno stringente in alcuni casi. Per risolvere la hold-and-wait il processo può prendere le risorse di cui ha bisogno solo quando sono tutte disponibili ma questo genera lunghe attese e non sempre so in anticipo di cosa avrò bisogno. Inoltre il processo potrebbe tenere le risorse inutilizzate per molto tempo. Per risolvere la no preemption il sistema operativo deve essere in grado di poter levare una risorsa ad un processo per garantirla ad un altro, bisogna quindi decidere un concetto di priorità e avere la capacità di salvare lo stato di un processo per farlo poi ripartire. Per evitare la circular wait devo definire una linea d'ordine per le risorse, per cui un processo che ha una risorsa potrà richiedere solo le risorse che presentano dopo nell'ordine, presenta lunghe attese e non sempre so in anticipo di quali risorse avrò bisogno.
- Evito: Faccio scelte dinamiche sullo stato attuale delle risorse. Devo decidere se la richiesta di allocazione di una risorsa fatta da un processo porterà ad un deadlock, per farlo ho bisogno di sapere anche quali saranno le richieste successive. Ho due approcci:
  - Resource Allocation Denial: Vieto l'accesso a una risorsa se rischio il deadlock;
  - Process Initiation Denial: Non faccio partire un processo se le future richieste porteranno a deadlock, ovvero se le risorse richieste più quelle già allocate superano quelle disponibili.

  Definisco safe state uno stato in cui l'allocazione di risorse ulteriori non porta a deadlock. La Resource Allocation Denial è meno restrittiva della prevenzione del deadlock ed è anche migliore della Process Initiation Denial perché evita di bloccare un intero processo. Gli svantaggi di questo approccio sono la necessità di conoscere in anticipo tutte le richieste di risorse di un processo, può considerare solo processi indipendenti e senza sincronizzazione, necessita di un numero fisso di risorse allocabili e l'impossibilità per un processo di terminare se ha risorse allocate.

- Individuo: Le strategie di individuazione hanno come idea fondamentale la concessione di risorse quando possibile e il controllo periodico volto a trovare situazioni di deadlock. Posso ad esempio controllare se sono in una situazione di deadlock ogni volta che alloco nuove risorse; In questo metodo individuo subito eventuali problemi ma è molto oneroso sulla CPU. Un classico protocollo prevede il segnare iterativamente i processi che possono essere sospesi o terminati. Se alla fine sono tutti segnati allora non abbiamo deadlock. Per recuperare un deadlock posso:
  - Terminare i processi in deadlock (soluzione comune);
  - Ripristinare il sistema ad uno stato precedente senza deadlock. Non garantisce che non si ripresenti lo stesso problema;
  - Terminare uno alla volta i processi in deadlock. Si cerca di non terminare tutti i processi in deadlock;
  - Usare la preemption per sottrarre le risorse finché non avviene il deadlock.

La mutua esclusione può essere garantita anche tramite approcci software. Supponiamo di avere una macchina a singolo/multi-processore, in cui i processi comunicano con una memoria centrale e richiedono mutua esclusione. Assumiamo che la mutua esclusione avvenga a livello di accesso della memoria. Vale a dire, gli accessi simultanei (lettura e/o scrittura) alla stessa locazione della memoria principale sono serializzati da una sorta di arbitro, anche se l'ordine di concessione dell'accesso non è specificato in anticipo. Al di là di questo, non si presume alcun supporto nell'hardware, nel sistema operativo o nel linguaggio di programmazione.

Si può utilizzare l'algoritmo di Dekker che funziona per mutua esclusione fra due processi. Potrei inserire una variabile locale che indica di chi è il turno e prima di entrare nella sezione critica controllo se è il mio turno, ma ho busy waiting. Potrei usare un flag per indicare se sono in sezione critica e se è occupato allora non entro, ma ho busy waiting e rischio che il flag venga settato a busy da entrambi i processi contemporaneamente. Potrei prima indicare che sono entrato in sezione critica e poi controllare se l'altro non c'è e posso proseguire, garantirei mutua esclusione ma rischierei deadlock nel busy-waiting fra il controllo e il cambio del flag. Potrei avere un flag per ogni processo, se il mio è su true e quello dell'altro processo è true mi rimetto a false, permetto all'altro di procedere e dopo un po' riprovo (rischio livelock in cui entrambi si settano a true, poi a false, poi a true e così via). Queste soluzioni sono tutte errate, nell'algoritmo di Dekker prendo ad esempio l'ultima ma in più mi baso anche su un turno definito in base a chi non è entrato in sezione critica da più tempo.

Di seguito la soluzione corretta.

```
boolean flag [2];
int turn;
void P0() {
    while (true) {
        flag [0] = true;
        while (flag [1]) {
            if (turn == 1) {
                flag [0] = false;
                while (turn == 1) /* do nothing */;
                flag [0] = true;
            }
        }
        /* critical section */;
        turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}
void P1( ) {
    while (true) {
        flag [1] = true;
        while (flag [0]) {
            if (turn == 0) {
                flag [1] = false;
                while (turn == 0) /* do nothing */;
                flag [1] = true;
            }
```

```
        }
        /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}
void main (){
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

L'algoritmo utilizza due variabili condivise: flag e turn. La variabile flag è un array di due booleani, uno per ciascun processo. La variabile turn è un intero che indica quale processo ha il diritto di accedere alla sezione critica (0 se turno di P0 1 altrimenti).

Ogni processo inizia impostando il suo flag a true. Quindi, il processo verifica il flag dell'altro processo. Se l'altro processo ha il flag a false o se turn indica che è il suo turno, il processo può accedere alla sezione critica. Altrimenti, il processo deve attendere fino a quando l'altro processo non ha terminato di eseguire la sezione critica.

Una volta che un processo ha terminato di eseguire la sezione critica, imposta il suo flag a false e aggiorna turn per l'altro processo.

Dijkstra ha generalizzato l'algoritmo di Dekker per poter funzionare con un numero di processi maggiore di 2.

```
boolean interested[N] = {false, ..., false} //Global
boolean passed[N] = {false, ..., false}
int k = <any> //k fra 0 e N-1
int i = <entity ID>//i fra 0 e N-1      Local
while (true) {
    /*non critical section*/
    interested[i] = true;    //i vuole entrare nella CS
    //k sceglie tra i processi che vogliono entrare in CS
    while (k != i) {
        passed[i] = false;
        if (!interested[k]) then k=i;
    }
    passed[i] = true;
    for j in 1...N except i do
        if (passed[j]) then goto "while (k != i)" //ricomincia se piu' di un processo
            vuole entrare
    /*critical section*/
    passed[i] = false; interested[i] = false;
}
```

Si rischia sempre che dueprocessi escano dal ciclo while contemporaneamente e quello che non è in k sia più veloce ed entri in critical section prima che quello che sta in k si setti come passed, comunque quello non in critical section ritornerà al ciclo while sopra. Se invece abbiamo 3 processi può essere che uno sia in k e quindi abbia diritto a proseguire ma per il sistema operativo abbia una priorità minore degli altri due, per cui rischiamo busy-waiting e starvation perché il processo che va in sezione critica potrebbe rimanere fermo (non in CPU) mentre i due che continuano a stare nel ciclo while utilizzano la CPU.

L'algoritmo di Dijkstra quindi garantisce la mutua esclusione ed evita il deadlock ma non garantisce la no starvation e inoltre necessita di read/write atomiche e di memoria condivisa per k.

4 Soluzioni software per la sincronizzazione

4.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra:

- Mutua Esclusione;

- Evita il deadlock;
- Non garantisce la no starvation;
- Ha bisogno di istruzioni atomiche per leggere/scrivere;
- Ha bisogno della memoria condivisa per k.

```
/* global storage */
boolean interested[N] = {false, ..., false}
boolean passed[N] = {false, ..., false}
int k = <any> // k in {0, 1, ..., N-1}
/* local info */
int i = <entity ID> // i in {0, 1, ..., N-1}
interested[i] = true
while (k != i) {
    passed[i] = false
    if (!interested[k]) then k = i
}
passed[i] = true
for j in 1 ... N except i do
    if (passed[j]) then goto 2
<critical section>
passed[i] = false; interested[i] = false
```

4.2 Algoritmo del Panettiere (Bakery)

L'algoritmo del Panettiere è un algoritmo di mutua esclusione che garantisce che un solo thread alla volta possa accedere alla sezione critica. È un algoritmo molto semplice da implementare e può essere utilizzato in ambienti con un numero qualsiasi di thread. L'algoritmo funziona sulla base di un contatore, che viene inizializzato a 0. Ogni processo che desidera accedere alla risorsa deve prima acquisire il contatore. Il thread che ha il numero più basso maggiore di 0 accede alla sezione critica. Dopo aver terminato si rimette il contatore del thread a 0.

Sia n il numero di processi e i il processo corrente. Allora per ogni processo i abbiamo la fase di assegnazione in cui verrà assegnato un numero tra 1 e un massimo prestabilito. Poi per ogni processo diverso da i verifichiamo se esiste uno con un numero minore. Se esiste i rimane in attesa altrimenti accede alla sezione critica. Se due processi i e j hanno lo stesso numero, allora accede il processo con numero più basso (se i=5 e j=9, allora accede i).

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

Caratteristiche dell'algoritmo:

- I processi comunicano scrivendo/leggendo da variabili condivise;
- Lettura/scrittura non sono operazioni atomiche: Il lettore può leggere mentre lo scrittore scrive;
- Quasiasi variabile condivisa è posseduta da un processo che può scriverci sopra, tutti gli altri posso leggervi;
- Nessun processo può scrivere contemporaneamente nelle variabili condivise;

- I tempi di esecuzione non sono correlati. Ciò significa che un processo non può prevedere quanto tempo impiegherà un altro processo per completare la propria esecuzione.

5 Comunicazione InterProcess

5.1 Meccanismi di Comunicazione InterProcess

Come comunicano i processi se non hanno memoria in comune? Si potrebbe utilizzare un file ma è una soluzione lenta e poco pratica. Un processo può accedere solo alla sua area riservata ma il kernel può accedere a tutto.

I processi potrebbero essere indipendenti (l'esecuzione di uno non influisce sull'altro) o cooperativi (il risultato di uno dipende dall'altro). Le ragioni per cui due processi potrebbero essere cooperativi riguardano la modularità del programma, la convenienza (lavorare su più processi potrebbe essere meglio), la velocità (se un pezzo di programma attende qualcosa il resto può proseguire), lo scambio di informazioni.

I meccanismi con cui si permette la comunicazione InterProcess sono di due tipi:

- Memoria condivisa: La memoria condivisa consente ai processi di scambiare informazioni posizionandole in una regione di memoria che viene condivisa tra i processi. Poiché la comunicazione non richiede chiamate di sistema o trasferimento di dati tra la memoria utente e la memoria del kernel, la memoria condivisa può fornire una comunicazione molto veloce. I processi potranno leggere e scrivere in quest'area di memoria, per questo avremo bisogno di un meccanismo che impedisca ai due processi di accedere contemporaneamente (ad esempio un semaforo). Per la memoria condivisa utilizziamo una serie di funzioni della Posix Shared Memory API.
- Message passing: Il fattore chiave che distingue questa funzionalità è la nozione di scrittura e lettura. Per comunicare, un processo scrive i dati nella struttura di comunicazione InterProcess (IPC) e un altro processo legge i dati. Queste funzionalità richiedono due trasferimenti di dati tra la memoria utente e la memoria del kernel: un trasferimento dalla memoria utente alla memoria del kernel durante la scrittura e un altro trasferimento dalla memoria del kernel alla memoria utente durante la lettura.

Tra le funzioni dello standard Posix per la memoria condivisa abbiamo:

- shm_open(), crea o apre una pagina di memoria condivisa. Ritorna il file descriptor di una pagina condivisa. É possibile creare una memoria condivisa privata;
- ltrunc() o ftruncate(), pone un limite alla grandezza della pagina di memoria condivisa;
- mmap(), mappa un file nella memoria di un processo;
- close(), chiude un file descriptor;
- shm_unlink(), rimuove una pagina condivisa.

```c
/* Program to write some data in shared memory */
int main() {
    const int SIZE = 4096; /* size of the shared page */

    /* name of the shared page */
    const char *name = "MY_PAGE";
    const char *msg = "Hello World!";
    int shm_fd;
    char * ptr;

    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SIZE);
    ptr = (char *) mmap(0, SIZE, PROT_WRITE,

    MAP_SHARED, shm_fd, 0);

    sprintf(ptr, "%s", msg);
    close(shm_fd);
    return 0;
}
```

```c
/* Program to read some data from shared memory */
int main() {
    const int SIZE = 4096; /* size of the shared page */

    /* name of the shared page */
    const char * name = "MY_PAGE";
    int shm_fd;
    char * ptr;

    shm_fd = shm_open(name, O_RDONLY, 0666);
    ptr = (char *) mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    printf("%s\n", ptr);
    close(shm_fd);
    shm_unlink(shm_fd);
    return 0;
}
```

5.2 Message Passing: pipes

Nel message passing abbiamo almeno due primitive, la send e la receive che prendono come parametro obbligatorio il messaggio e come parametri facoltativi rispettivamente il destinatario e il mittente. Il formato dei messaggi è fisso o variabile. Ciascuno contiene:

- Header: Tipo messaggio, ID destinatario e mittente, lunghezza del messaggio, informazioni di controllo;
- Body: Contenuto del messaggio

Nei sistemi Unix non abbiamo ID destinatario e ID mittente nella Header del messaggio. Generalmente si utilizza una FIFO per gestire i messaggi ma possiamo anche avere a che fare con priorità. Il message passing è realizzato tramite Pipes o Named-Pipes(FIFOs).

In sistemi Unix una Pipe attiva un collegamento monodirezionale fra due processi padre e figlio, quando il processo termina la pipe viene rimossa automaticamente, si basa sulla system call pipe(). Le pipe permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali, le informazioni lette vengono eliminate. A livello di sistema operativo una pipe è un buffer di dimensione stabilita. Le pipe sono utilizzare da processi relazionati e quindi uno dei due deve essere stato generato con una fork dall'altro. Le Named Pipe (FIFO) permettono la comunicazione anche tra processi non relazionati, non si chiudono autonomamente e sono bidirezionali.

Nei sistemi Unix l'uso delle pipe avviene attraverso la nozione di descrittore. La creazione di una pipe avviene con la funzione pipe(int fd[2]) che ha come argomento un buffer di due interi che saranno fd[0] descrittore di lettura e fd[1] descrittore di scrittura. Questi descrittori possono essere utilizzati attraverso le funzioni read() e write(). Se provo a leggere da una pipe vuota il processo si blocca in attesa che si riempia di dati da leggere, se provo a scrivere in una pipe piena il processo si blocca in attesa che si liberi spazio conseguentemente alla lettura di dati. Una pipe ha una dimensione massima pari a 16 pagine di memoria, ogni pagina occupa 4096 byte, la lettura/scrittura di pagine è atomica e non può essere interrotta, (ciò significa che se devo leggere 7000 byte, lo scheduler potrà interrompermi solo dopo che avrò finito di leggere la prima pagina, o solo dopo che ho completato la lettura) ovvero formalmente: "sono da considerarsi atomiche le operazioni effettuate su pipe di grandezza inferiore a una pagina", negli altri casi è possibile venir interrotti dallo scheduler e bisognerà agire considerando questa casualità. È possibile rendere la lettura e la scrittura non bloccanti, ma non lo trattiamo. Le pipe sono un dispositivo logico e vengono considerate finite quando nessun descrittore in scrittura è più aperto, in questo caso una eventuale chiamata read() restituirà 0. Allo stesso modo se tutti i descrittori in lettura sono chiusi e provo a scrivere riceverò un errore SIGPIPE. Per evitare di trovarci in situazioni di deadlock devo chiudere i descrittori appena finisco di utilizzarli, ogni processo lettore deve subito chiudere il descrittore in scrittura e ogni processo scrittore deve subito chiudere il descrittore in lettura. Se non accadesse potrebbe succedere che l'evento "tutti gli scrittori hanno terminato" non potrebbe mai avvenire e si potrebbe avere un deadlock.

Per creare una Named PIPE (FIFO) nei sistemi UNIX si usa la funzione mkfifo(char *name, int mode), dove *name è il nome della FIFO da creare e mode specifica i permessi di accesso alla

FIFO. La funzione restituisce -1 in caso di fallimento e 0 altrimenti. La rimozione di una FIFO avviene tramite la chiamata di sistema unlink().

Normalmente, l'apertura di una FIFO è bloccante, nel senso che il processo che tenti di aprirla in lettura (scrittura) viene bloccato fino a quando un altro processo non la apre in scrittura (lettura). Se si vuole inibire questo comportamento è possibile aggiungere il flag O_NONBLOCK al valore del parametro mode passato alla system call open() su di una FIFO. Ogni FIFO deve avere sia un lettore che uno scrittore; se un processo tenta di scrivere su una FIFO che non ha un lettore esso riceve il segnale "SIGPIPE" da parte del sistema operativo

5.3 Esempio: client/server tramite FIFO

Lato server:

```c
#include <stdio.h>
#include <fcntl.h>

typedef struct {
    long type;
    char fifo_response[20];
} request;

int main(int argc, char *argv[]){
    char *response = "fatto";
    int pid, fd, fdc, ret;
    request r;
    ret = mkfifo("/serv", 0666);
    if ( ret == -1 ) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
    }
    fd = open("/serv",O_RDONLY);
    while(1) {
        ret = read(fd, &r, sizeof(request));
        if (ret != 0) {
            printf("Richiesto un servizio (fifo di restituzione = %s)\n", r.fifo_response);
            /* switch sul tipo di servizio */
            sleep(10); /* emulazione di ritardo per il servizio */
            fdc = open(r.fifo_response,O_WRONLY);
            write(fdc, response, 20);
            close(fdc);
            exit(0);
        } /* end if (ret != 0) */
    }
}
```

Lato client:

```c
#include <stdio.h>
#include <fcntl.h>
typedef struct {
    long type;
    char fifo_response[20];
} request;
int main(int argc, char *argv[]) {
    int pid, fd, fdc, ret; request r; char response[20];
    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);
    if (r.fifo_response[0] > 'z' || r.fifo_response[0] < 'a' ) {
        printf("carattere selezionato non valido, ricominciare operazione\n");
        exit(1);
    }
    r.fifo_response[1] = '\0';
    ret = mkfifo(r.fifo_response, 0666);
    if ( ret == -1 ) {
        printf("\n servente sovraccarico - riprovare \n");
```

```
        exit(1);
    }
    fd = open("/serv",O_WRONLY);
    if ( fd == -1 ) {
        printf("\n servizio non disponibile \n");
        ret = unlink(r.fifo_response);
        exit(1);
    }
    write(fd, &r, sizeof(request));
    close(fd);
    fdc = open(r.fifo_response,O_RDONLY);
    read(fdc, response, 20);
    printf("risposta = %s\n", response);
    close(fdc);
    unlink(r.fifo_response);
}
```

5.4 Confronto tra pipe e FIFO

L'apertura di una pipe comporta ottenere sia il descrittore per la lettura che quello per la scrittura; bisogna poi chiudere quello che non viene usato. Invece l'apertura di una FIFO avviene in lettura o in scrittura.

Nella gestione si differenziano solo nella fase di apertura e chiusura, lettura e scrittura avvengono nello stesso modo.

Le pipe vengono usate tra processi imparentati, poiché al momento della fork() (che va fatta dopo aver costruito la pipe) condividono i file descriptor aperti e i relativi permessi. invece le FIFO possono essere usate tra qualunque tipo di processi.

6 Socket

Lo scambio di dati fra entità (contenute in sistemi) può essere molto complesso. Per questo si necessita di un protocollo che lo regoli, definito da semantica, sintassi e timing. Il mittente deve informare la rete riguardo l'identità del destinatario o attivare un collegamento diretto, inoltre deve assicurarsi che il destinatario sia pronto a ricevere e salvare i dati. Per la comunicazione si utilizzano vari moduli che collaborano fra di loro, i livelli di rete.

I socket sono un'astrazione di un canale di comunicazione tra due processi. Il canale di comunicazione è rappresentato da un indirizzo IP e da una porta. L'indirizzo IP identifica il computer su cui si trova il processo, mentre la porta identifica il processo specifico che si desidera contattare.

Esistono tre tipi di socket:

- Stream socket: Utilizza il protocollo TCP, cioè fornisce una connessione affidabile;
- Datagram socket: Utilizza il protocollo UDP, cioè è senza connessione e non affidabile;
- Raw socket: Permette l'accesso diretto al livello di rete. Cioè è responsabile della formattazione e dell'invio dei dati sulla rete.

Nel funzionamento il client manda una request con un numero di porta conosciuto (generalmente è standard per alcuni tipi di servizi) e il server risponderà non necessariamente sulla porta iniziale perché potrebbe avere altre comunicazioni in ingresso.

Le porte 0-1023 sono riservate, le porte 1024- 5000 sono effimere (assegnate dal sistema operativo e con durata breve), le porte 5001-65535 sono libere.

Le primitive necessarie per la comunicazione sono:

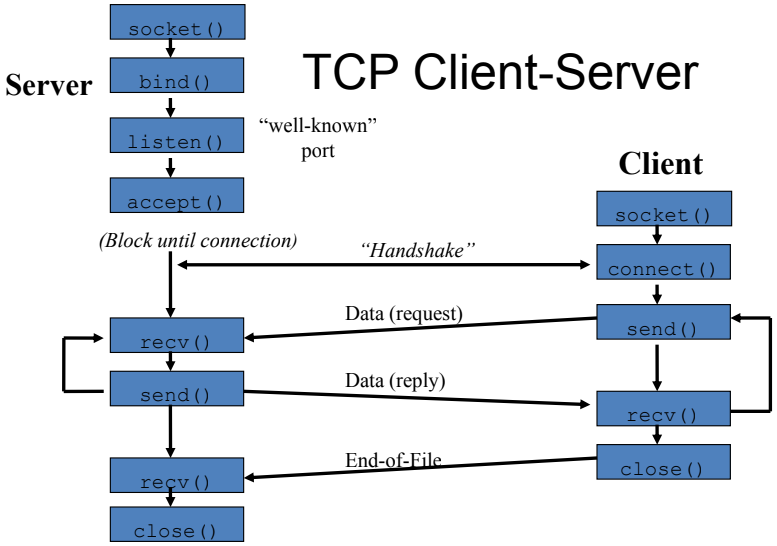
- socket(), restituisce un descrittore, cioè un intero che identifica il socket;
- listen(), la socket è pronta a ricevere richieste di connessione da altri client;
- connect(), utilizzata per stabilire una connessione tra una socket locale e una socket remota. La socket locale è la socket che il client sta utilizzando per stabilire la connessione. La socket remota è la socket che il server sta utilizzando per ascoltare le richieste di connessione;
- send(), per inviare i dati su una socket;

- receive(), per ricevere dati da una socket;
- disconnect(), per chiudere una connessione su una socket.

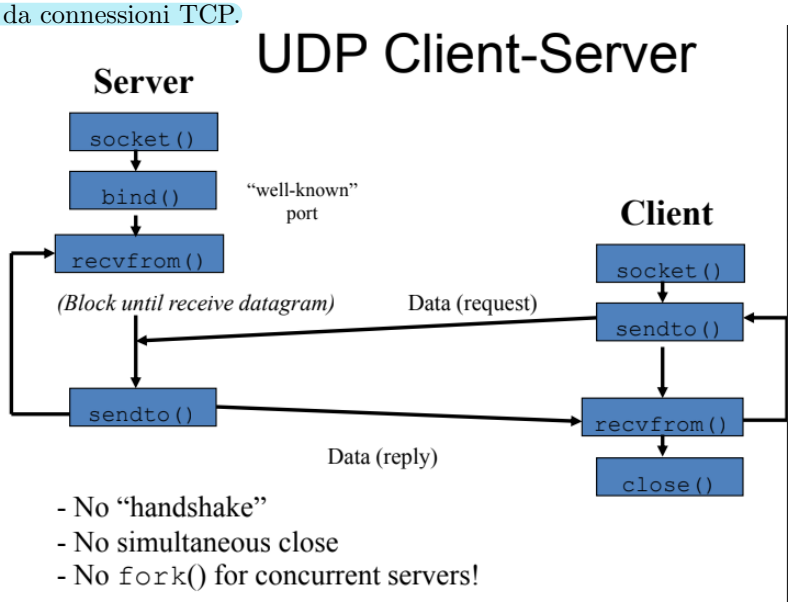
Le primitive per le socket TCP di Berkeley (Unix) sono:

- socket(), per creare una comunicazione;
- bind(), per assegnare il proprio indirizzo alla socket;
- listen(), per prepararsi ad accettare connessioni;
- accept(), per accettare una richiesta di connessione;
- connect(), stabilisce una connessione tra una socket locale e una socket remota;
- send(), per mandare dati;
- recv(), per ricevere dati;
- close(), per chiudere la connessione.

Quindi il client tenta di stabilire una connessione con il server tramite connect(), mentre il server si mette in attesa di connessione tramite listen(). Con accept() stabilisce la connessione.



Nel caso di UDP avrò le funzioni sendto() con l'indirizzo del destinatario per inviare dati e recvfrom() per ricevere dati, non viene usata la bind. Perciò connect() e accept() sono utilizzate solo da connessioni TCP.



La funzione int gethostname(char* hostname, int bufferLength) scrive su hostname il nome dell'host su cui il programma sta girando. Il secondo argomento pone un limite ai bytes che la funzione può scrivere su hostname. La funzione unsigned long inet_addr(char* address) converte un indirizzo IP dalla notazione puntata in un valore numerico a 32 bit(es. "128.173.41.41"). La funzione char* inet_ntoa(struct in_addr address) converte un indirizzo IP da struct in_addr in notazione puntata. La funzione struct hostent* gethostbyname(const char* hostname) restituisce le informazioni riguardanti un host name (es. www.google.it) passato in input. La funzione struct hostent* gesthostbyaddr(const void* addr, int len, int type) restituisce il nome di un host dato il suo indirizzo.

6.1 WinSock

Un'altra API utilizzabile per le socket è WinSock. È un'API di basso livello che fornisce un accesso diretto alle funzionalità di rete del sistema operativo Windows.

Differenze tra Berkeley Sockets e WinSock:

Berkeley	WinSock
bzero()	memset()
close()	closesocket()
read()	Non necessario
write()	Non necessario
ioctl()	ioctlsocket()

Le WinSock 1.1 supportano 3 diverse modalità:

- Blocking mode: l'applicazione verrà bloccata fino al completamento dell'operazione;
- Non-blocking mode: le funzioni come accept() non bloccano ma ritornano semplicemente uno stato;
- Asynchronous mode: utilizza i messaggi di Windows come FD_ACCEPT e FD_CONNECT. Sono entrambe funzioni bloccanti che rispettivamente attende l'accettazione della connessione da parte del server e l'altra tenta la connessione al server.

Le WinSock 2, oltre ad avere le stesse funzionalità della versione 1.1, supporta altri protocolli oltre al TCP/IP. Utilizza inoltre file diversi (winsock2.h) e cambia il nome di alcune funzioni.

6.2 Scheda di Rete

Una scheda di rete è l'interfaccia tra un host e la rete. La scheda è costituita da due parti separate che interagiscono tramite una FIFO. Una parte interagisce con la CPU, la seconda parte interagisce con la rete e implementa i livelli fisico e di collegamento.

Tutto il sistema è comandato da una SCO (sottosistema di controllo della scheda) che comunica con la CPU attraverso il CSR (Control Status Register), un registro in cui è possibile settare e leggere dei flag. L'host può comunicare cosa accade nel CSR in due possibili modi:

- Busy waiting, la CPU legge continuamente il CSR finché non trova una modifica che indica l'operazione da eseguire, ragionevole per dispositivi che non fanno altro come i router;
- Interrupt, la scheda invia un interrupt all'host che va quindi a leggere il CSR per capire cosa fare.

Il trasferimento dei dati dalla scheda alla memoria e viceversa può essere:

- Direct Memory Access: non coinvolge la CPU, l'host ha un'area di memoria assegnata dal SO in cui vengono immediatamente inviati i frame, bastano pochi bytes sulla scheda;
- Programmed I/O: lo scambio dati tra memoria e scheda passa per la CPU, bisogna bufferizzare almeno un frame sulla scheda, la memoria deve essere dual port per cui sia processore che scheda devono poter leggere e scrivere.

Nel caso di Direct Memory Access l'allocazione dei frame nella memoria è organizzata attraverso una buffer descriptor list (una in lettura e una in scrittura) che è un vettore di puntatori ad aree di memoria (ovvero i buffer) dove è descritta anche la quantità di memoria disponibile in ciascuna

area. Per ethernet vengono tipicamente preallocati 64 buffer da 1500 byte. Per i frame che arrivano dalla scheda si utilizza la tecnica scatter read/gather write secondo la quale frame distinti sono allocati in buffer distinti, nel caso in cui un frame sia troppo grande può essere spezzato in più buffer.

Quando un messaggio viene inviato da un utente in una socket il sistema operativo lo copia in un buffer nella memoria in una zona di buffer descriptor(BD). Tale messaggio viene processato da tutti i livelli che aggiungono le intestazioni. Quando il messaggio è pronto, viene avvertita la SCO della scheda attraverso alcuni flag del CSR, la SCO invia il messaggio sulla linea e notifica alla CPU il termine dell'invio settando CSR e inviando un interrupt. La CPU a questo punto lancia un interrupt handler che resetta i flag del CSR e libera le opportune risorse.

Il device driver è una collezione di routine del sistema operativo che permettono la comunicazione fra il sistema operativo e l'hardware specifico della scheda.

Un interrupt handler disabilita le interruzioni, legge il CSR per capire cosa fare. Ci sono tre possibilità: c'è stato un errore, una trasmissione è stata completata, un frame è stato ricevuto. Nel secondo caso il bit LE_TINT viene messo a zero, ammette un nuovo processo nella BD dato che un frame è stato trasmesso e abilita le interruzioni.

7 Sistemi Distribuiti

Un sistema distribuito è un insieme di entità separate spazialmente, non più nello stesso computer. Con entità indichiamo processi e thread ad esempio. Devono poter comunicare e coordinarsi. I sistemi distribuiti sono già molto presenti in tutti gli ambiti (network di workstation, sistemi manufatturieri con catena di montaggi, MMOGs massive multiplayer online games) e spesso necessitano di velocità quasi real time.

I sistemi distribuiti sono preferibili rispetto a quelli centralizzati poiché sono:

- Economici: tanti sistemi sono più economici di uno più grande di pari potenza;
- Veloci: posso creare sistemi di una potenza altrimenti non ottenibile da un sistema singolo;
- Distribuiti: posso consultarlo da più punti fisici;
- Affidabili: il guasto di una piccola parte non impatta su tutto il sistema;
- Scalabili: permettono una crescita incrementale, posso fare piccoli miglioramenti quando voglio.

I vantaggi rispetto ai PC indipendenti sono:

- Condivisione di dati: permette a diversi utenti di acceder a database in comune;
- Condivisione di risorse: permette di condividere periferiche costose come le stampanti a colori;
- Comunicazione: migliora la comunicazione tra persone con l'utilizzo di email, chat ecc...;
- Flessibilità: distribuisce il lavoro con le macchine disponibili.

Lo sviluppo di software per sistemi distribuiti è più difficile rispetto ad altri sistemi. Inoltre la rete è facilmente saturabile. La sicurezza è un altro svantaggio dato che l'accesso facilitato è applicato anche ai dati sensibili.

L'obiettivo primario è la condivisione di dati e risorse che devono essere sincronizzate e coordinate. Questo comporta il dover gestire la concorrenza che non è più solo temporale ma anche spaziale. Inoltre internet non è completamente affidabile, non ho più un clock globale, devo poter gestire eventuali fallimenti di una macchina dalle altre, le latenze non sono più prevedibili. Esempi di sistemi distribuiti sono: LAN, DBMS, ATM, internet, ubiquitous computing, reti virtuali, peer to peer, cloud computing. Internet è il più grande sistema distribuito e il world wide web è la più grande applicazione distribuita al mondo.

I sistemi distribuiti comportano i seguenti problemi:

- Eterogeneità: le macchine, le reti, i sistemi operativi e i linguaggi di programmazione sono differenti;
- Apertura: deve essere pubblico, ad esempio per accedere ad internet posso utilizzare più browser. Il www ha quindi delle interfacce pubbliche;

- Sicurezza: riguarda la privacy, l'integrità intesa come protezione contro l'alterazione dei dati, la disponibilità ovvero evitare che ci sia un interferenza con l'accesso a una risorsa;
- Scalabilità: bisogna controllare i costi delle risorse fisiche, evitare perdita di prestazioni dovuta all'overhead, evitare colli di bottiglia nel sistema;
- Affidabilità: non devo avere problemi se fallisce una sola macchina, perciò devo trovare, tollerare e risolver,e eventuali fallimenti;
- Concorrenza: ora è spaziale e non solo temporale;
- Flessibilità: il sistema deve essere facilmente modificabile nonostante leghi sistemi operativi e kernel diversi;
- Performance: aumentando il numero di macchine aumenta l'overhead di gestione e aumentano i ritardi di comunicazione, per cui bisogna tenerne conto;
- Trasparenza: per l'utente l'esperienza non deve cambiare. Non deve potersi accorgere del fatto che il sistema è distribuito e non centralizzato.

L'inizializzazione di sistemi distribuiti può essere di tipo client-server o peer-to-peer. Bisogna inserire fra il sistema operativo e l'applicazione un middleware che permetta ai sistemi di comunicare e cooperare. Il middleware è il cuore del sistema distribuito e si collega ad ogni sistema operativo tramite diverse interfacce standard e a ogni applicazione grazie a specifiche API.

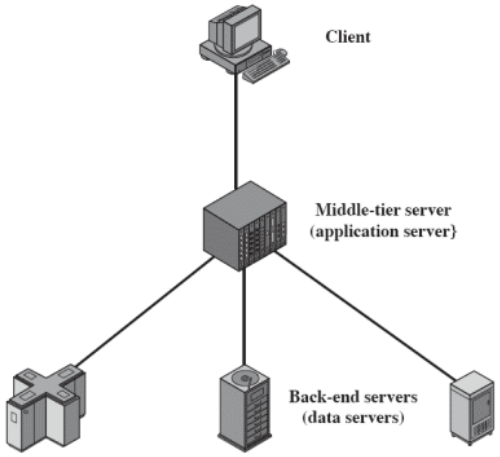
Anche il middleware presenta dei problemi, tra cui la mancanza di sincronizzazione locale e nel network (i tempi di propagazione dei messaggi può essere impredicibile), la mancanza di una conoscenza globale della rete e i fallimenti dei nodi o delle reti.

Nel caso di Client-Server computing, ho un client con basse prestazioni e un server che supplisce alle prestazioni del client. Sono connessi in una LAN o tramite internet. L'applicazione si svolge nel server mentre il client viene usato solo per mostrare il risultato finale. Il client e il server possono avere differenze sia hardware che software, l'importante è che seguono gli stessi protocolli di comunicazione e supportano le stesse applicazioni.

Supponiamo di avere 4 livelli: presentation logic, application logic, database logic, DBMS.
Esistono quattro tipi di applicazione client-server:

- Host based processing: Ha i quattro livelli sul server;
- Server based processing: Presentation è sul client, gli altri sul server. Cioè l'esecuzione viene fatta interamente dal server e il client fornisce la GUI;
- Client based processing: presentation, application, database sul client, database e DBMS sul server. Cioè il client si occupa dell'esecuzione e il server di controllare i dati e gestire i database;
- Cooperative processing: Presentation e application sul client, application, database e DBMS sul server. L'esecuzione viene fatta in maniera ottimizzata sia dal client che dal server. È difficile da implementare e mantenere.

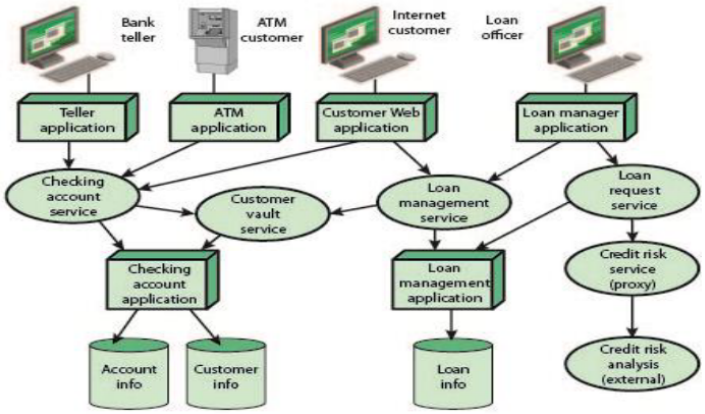
Esiste un'architettura Client/Server a 3 livelli in cui abbiamo una macchina utente, un server intermedio e dei server backend.



Un'altra architettura Client/Server è la SOA (Service Oriented Architecture). È spesso usata nelle grandi aziende e banche, essa organizza le funzioni in una struttura modulare interconnessa. Ho una serie di servizi e di client che sono collegati tramite interfacce che permettono la comunicazione.

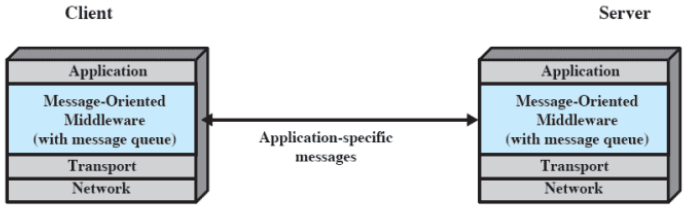
La struttura si suddivide in tre attori: Provider, Broker e Requester, Il ruolo di Requester viene svolto dal client, che contatta come middle-tier server il Broker che lo reindirizza al Provider che elaborerà il servizio richiesto.

I Provider notificano al Broker quali servizi svolgono, e ci possono essere più Provider che garantiscono lo stesso servizio, in collaborazione, con il contestuale sincronismo.



7.1 Message Passing

Per comunicare nei sistemi distribuiti è necessario il message passing che avviene fra i middleware.

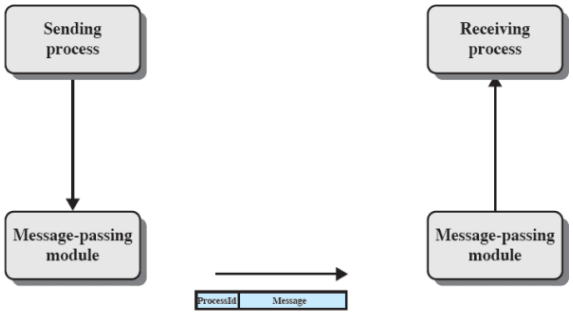


(a) Message-Oriented Middleware

Le primitive del message passing dei sistemi distribuiti sono le operazioni di base che consentono ai processi di comunicare tra loro. Queste primitive sono generalmente fornite dal sistema operativo o dalla libreria di comunicazione del sistema distribuito.

Le primitive di base del message passing sono:

- send(): permette a un processo di inviare un messaggio a un altro processo;
- receive(): permette a un processo di ricevere un messaggio da un altro processo.



Il processo mittente (Sending process) utilizza la primitiva send() per inviare il messaggio al processo destinatario (Receiving process). Il messaggio viene rappresentato dal blocco "Message". Il processo destinatario utilizza la primitiva receive() per ricevere il messaggio dal processo mittente. Il processo mittente e il processo destinatario possono essere eseguiti su macchine diverse. In questo caso, il canale di comunicazione è rappresentato da una rete di comunicazione.

7.2 Remote Procedure Call

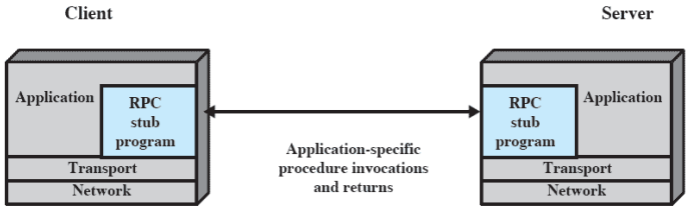
Una Remote Procedure Call (RPC) è un meccanismo che consente a un processo di chiamare una procedura in un altro processo che si trova su un computer diverso. La RPC fornisce un'interfaccia in modo che i programmatori possano scrivere codice che interagisce con processi remoti in modo trasparente.

La RPC è un componente essenziale dei sistemi distribuiti. Consente agli sviluppatori di creare applicazioni che richiedono l'interazione di processi che si trovano su diversi computer.

Dato l'utilizzo comune della RPC, abbiamo bisogno di una standarizzazione. Questo perché:

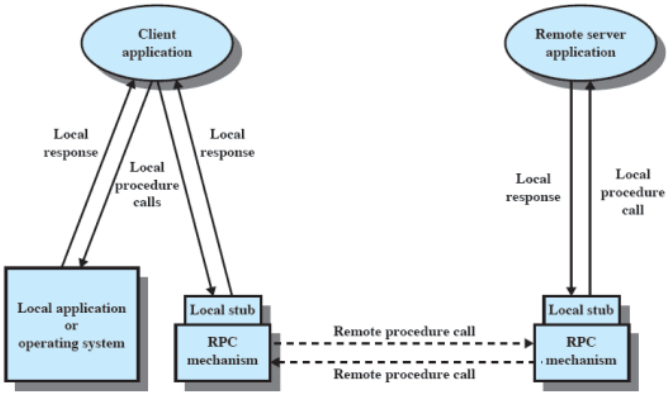
- Rende più facile per i programmatori creare applicazioni distribuite che utilizzano le RPC. Ciò è dovuto al fatto che i programmatori non devono preoccuparsi di implementare le funzionalità RPC da soli, ma possono invece utilizzare un'implementazione standard;
- Migliora la compatibilità tra applicazioni distribuite che utilizzano le RPC. Ciò è dovuto al fatto che le applicazioni possono utilizzare le stesse API RPC, indipendentemente dal sistema operativo o dall'ambiente di sviluppo utilizzato.

Client e server comunicano attraverso degli RPC stub program, presenti in entrambi.



(b) Remote Procedure Calls

Sia il client che il server, per comunicare, effettuano una Local Procedure Call (LPC), cioè un meccanismo interno che permette la comunicazione di processi nello stesso computer. Per effettuare una RPC, viene fatta una LPC allo stub che effettuerà una RPC.



La difficoltà maggiore delle RPC è il passaggio di puntatori. Questo perché abbiamo bisogno di un unico puntatore enorme. Inoltre, la rappresentazione/formattazione dei parametri e dei messaggi può essere difficile se i linguaggi di programmazione del client e del server differiscono.

Nel sistema Client/Server le connessioni possono essere:

- Persistenti: Vengono mantenute aperte per un periodo di tempo prolungato. Ciò consente alle applicazioni client e server di scambiare dati più volte senza dover stabilire e chiudere la connessione ogni volta. La connessione termina quando passa uno specifico tempo senza attività;
- Non persistenti: Vengono aperte e chiuse per ogni chiamata. Ciò significa che ogni chiamata deve stabilire una nuova connessione e chiudere la connessione quando la chiamata di procedura remota è terminata.

Le RPC possono essere:

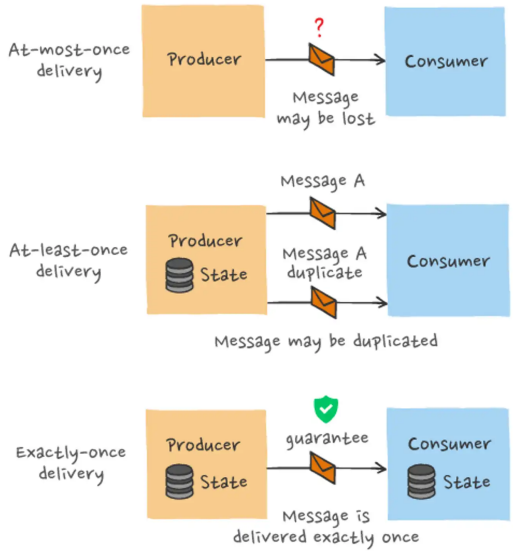
- sincrone, chi le chiama attende il risultato;
- asincrone, chi le chiama prosegue nell'elaborazione senza attendere il risultato che arriverà più avanti.

Le funzionamento normale delle RPC può essere interrotto per vari problemi (messaggio perso, chiamante o chiamato in crash). Per risolverli posso scegliere quale politica applicare.

At most once prevede che non si faccia nulla nel caso in cui non si riceva il risultato. Quindi il messaggio è consegnato una o zero volte.

At least once prevede che si faccia nuovamente la richiesta nel caso in cui non si riceva risposta, sorge un problema se alla fine ricevo due risultati che potrebbero essere diversi. Ha un overhead piccolo ed è facile da implementare.

Exactly one prevede che ogni messaggio è consegnato precisamente una volta. È la più difficile da implementare.



7.3 Cluster

Un cluster è un insieme di computer interconnessi tra loro in modo da poter essere visti all'esterno come un singolo sistema. I computer di un cluster sono chiamati nodi.

Tra i benefici dei cluster abbiamo:

- Scalabilità: È possibile aumentare o diminuire la grandezza del cluster a piacimento;
- Disponibilità: Il fallimento di un nodo non compromette l'intero sistema;
- Costo: Mettere insieme più computer ha un costo inferiore rispetto a una singola macchina con le stesse performance.

I nodi possono avere il disco condiviso o separato.

Metodo di Clustering	Descrizione	Benefici	Limitazioni
Standby Passivo	Se un server fallisce, un secondo prende il suo posto	Facile da implementare	Costo alto perché il server secondario entra in gioco solo nel caso di fallimento del primo
Secondario Attivo	Anche il secondo server viene usato per delle task	Riduce il costo perché il secondo server viene utilizzato	Incremento di complessità
Server Separati	I server separati hanno i propri dischi. I dati sono costantemente copiati dal disco del server primario a quello secondario	Disponibilità alta	Grande overhead di rete e del server per via di tutte le operazioni di copia
Server con dischi connesso	I server sono cablati a degli stessi dischi, ma ognuno ha i suoi dischi distinti. Se un server fallisce, l'altro prende il controllo dei suoi dischi	Riduce l'overhead del server e della rete per via della riduzione delle copie	In genere ha bisogno del mirroring del disco o della tecnologia RAID per compensare ai rischi di fallimento dei dischi
Server con dischi condivisi	Molteplici server condividono l'accesso dei dischi	Basso overhead di rete e del server. Riduzione del rischio di tempi di inattività causati da guasti al disco	Richiede il lock manager. Solitamente utilizzato con la tecnologia di mirroring del disco o RAID

Esistono due tipi di approcci che possono essere intrapresi in caso di fallimento:

- Cluster ad alta disponibilità: Hanno una probabilità alta che tutte le risorse sono in servizio. Tutte le query perse, se recuperate, verranno fatte da un computer diverso all'interno del cluster. In caso di fallimento, le query in esecuzioni vanno perse;
- Cluster tolleranti ai guasti: Ottenuto con l'ausilio di dischi condivisi ridondanti e meccanismi per annullare le transazioni non completate e confermare le transazioni completate. Si assicura che le risorse siano sempre disponibili.

La funzione di commutare un'applicazione e le risorse da un sistema guasto a un sistema alternativo nel cluster è chiamata fallover.

Il ripristino delle applicazioni e delle risorse sul sistema originale una volta risolto il guasto è chiamato fallback.

Il fallback può essere automatizzato ma conviene solamente quando il problema è risolto ed è improbabile che ricapiti. Altrimenti può succedere un continuo susseguirsi di fallover e fallback.

8 Internet of Things

L'IoT è qualsiasi cosa connessa con un po' di capacità di calcolo e capace di generare e ricevere dati (alexa, cellulare, tv, auto, mi band, lampade smart). Necessitano di sensori, attenuatori, microcontrollori, elementi di trasmissione e di identificazione univoca. Generalmente si considerano con ROM e RAM limitata, low-power, con performance e capacità di scambiare dati limitate. I sistemi operativi dedicati devono di conseguenza occupare poca memoria, supportare vari hardware e tipi di comunicazione, essere molto ottimizzati, efficienti energeticamente, con capacità realtime e sicuri. Poter gestire in ogni momento i propri dispositivi porta grandi risparmi. L'obiettivo finale è l'automazione della vita umana.

9 Sicurezza Informatica

La sicurezza è un processo che va gestito correttamente, nulla è completamente sicuro ma bisogna lavorare per avere un livello più alto possibile. Cercare di mettere in sicurezza un sistema equivale

a mettere in sicurezza le sue componenti meno sicure (regola dell'anello debole). Cercare sicurezza tenendo nascoste le implementazioni dei propri sistemi non è sempre la scelta giusta, rendendo pubblico il proprio codice si può infatti sperare che qualche utente trovi falle e le segnali. La crittografia è un ottimo strumento ma non basta.

Secondo la National Institute of Standards and Technology (NIST), la Computer Security è la protezione di un sistema di calcolo volta a preservare l'integrità, la disponibilità e la riservatezza di risorse del sistema informativo (hardware, software, firmware, informazioni/dati).

Esistono tre obiettivi della sicurezza (Triade CIA):

- Confidentiality: Assicura che le informazioni private o riservate non vengano diffuse a individui non autorizzati. Si assicura che gli individui abbiano il controllo delle proprie informazioni decidendo a chi diffonderle;
- Integrity: L'integrità dei dati assicura che le informazioni e i programmi subiscano cambiamenti solo in modo specifico e autorizzato. L'integrità del sistema assicura che un sistema svolga la funzione prevista in modo ineccepibile, senza manipolazioni non autorizzate, intenzionali o involontarie, del sistema stesso;
- Availability: Assicura che i sistemi funzionino prontamente e i servizi non siano negati agli utenti autorizzati.

Si ha in aggiunta due concetti fondamentali:

- Authenticity: Bisogna poter verificare la validità di una trasmissione, di un messaggio o del mittente di un messaggio. Tutti gli input devono arrivare da una fonte affidabile;
- Accountability: Bisogna essere in grado di rintracciare i responsabili delle violazioni di sicurezza. I sistemi devono tenere traccia delle loro attività per permettere una futura analisi forense per rintracciare le violazioni di sicurezza;

Le possibili minacce riguardano l'hardware (furto o danneggiamento), il software (programmi cancellati, copie non autorizzate di software, modifica non autorizzata dei programmi), i dati (file cancellati, lettura non autorizzata, file modificati senza autorizzazione) o le comunicazioni (messaggi cancellati, letti senza autorizzazione o modificati).

9.1 Attacchi Informatici

Gli attacchi si dividono in

- Passivi: L'hacker può solamente leggere informazioni ma non altera il sistema. Lo può fare in vari modi come il Tempest (estrae informazioni sfruttando l'emissione dal sistema di segnali radio, elettrici, termici, di vibrazioni ecc...) e il Packet Sniffing (intercettazione dei dati che vengono trasmessi su una rete);
- Attivi: L'hacker può leggere, modificare, generare, distruggere qualsiasi informazione. Esistono quattro tipi di attacchi attivi:
  - Reply: Cattura passiva di un'unità di dati e la sua successiva ritrasmissione per produrre un effetto non autorizzato;
  - Masquerade: Un entità finge di essere un'altra entità nella rete;
  - Modifica dei messaggi: Qualche porzione del messaggio viene alterata, o viene alterato l'ordine dei messaggi per produrre effetti non autorizzati;
  - Denial of Service (DoS): L'interruzione di un'intera rete, disabilitandola o sovraccaricandola di messaggi in modo tale da degradare le prestazioni.

Un hacker cerca di inserirsi in un sistema per sfida personale, per rivelare informazioni nascoste o per cercare vulnerabilità e segnalarle. L'Insider Attack prevede di farsi assumere in una compagnia per fare spionaggio e rubare dati. Una Criminal Enterprise è un gruppo organizzato di hacker. Un Advanced Persistent Threat è una Criminal Enterprise sponsorizzata da una nazione per avviare un attacco verso un'altra nazione. Hanno come target infrastrutture critiche in modo da sabotarle o diffondere informazioni sensibili.

Un malware è un software malevolo. È progettato per danneggiare o usare risorse di un computer. In alcuni casi si diffonde tramite e-mail o dischi infetti.

Queste sono le terminologie dei programmi pericolosi:

Name	Description
Virus	Malware that, when executed, tries to replicate itself into other executable code; when it succeeds the code is said to be infected. When the infected code is executed, the virus also executes.
Worm	A computer program that can run independently and can propagate a complete working version of itself onto other hosts on a network.
Logic bomb	A program inserted into software by an intruder. A logic bomb lies dormant until a predefined condition is met; the program then triggers an unauthorized act.
Trojan horse	A computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes the Trojan horse program.
Backdoor (trapdoor)	Any mechanisms that bypasses a normal security check; it may allow unauthorized access to functionality.
Platform independent code	Software (e.g., script, macro, or other portable instruction) that can be shipped unchanged to a heterogeneous collection of platforms and execute with identical semantics.
Exploits	Code specific to a single vulnerability or set of vulnerabilities.
Downloaders	Program that installs other items on a machine that is under attack. Usually, a downloader is sent in an e-mail.
Auto-rooter	Malicious hacker tools used to break into new machines remotely.
Kit (virus generator)	Set of tools for generating new viruses automatically.
Spammer programs	Used to send large volumes of unwanted e-mail.
Flooders	Used to attack networked computer systems with a large volume of traffic to carry out a denial-of-service (DoS) attack.
Keyloggers	Captures keystrokes on a compromised system.
Rootkit	Set of hacker tools used after attacker has broken into a computer system and gained root-level access.
Zombie, bot	Program activated on an infected machine that is activated to launch attacks on other machines.
Spyware	Software that collects information from a computer and transmits it to another system.
Adware	Advertising that is integrated into software. It can result in pop-up ads or redirection of a browser to a commercial site.

Una backdoor è una porta di servizio, spesso installata volontariamente per avere accesso secondario senza passare per la solita procedura di accesso sicura. Diventa una minaccia quando viene usata da un programmatore malizioso per avere un accesso non autorizzato.

Un trojan horse è un programma apparentemente innocuo che nasconde software malevolo. I trojan rientrano in uno dei tre modelli:

- continua a eseguire il programma originale e in più esegue attività dannose;
- esegue il programma originale modificandolo per svolgere attività dannose;
- modifica completamente il programma originale svolgendo attività dannose.

I Platform Independent Code sono codici che possono essere eseguiti su più piattaforme hardware o software senza la necessità di ricompilarlo. Questo vantaggio di essere indipendente dalla piattaforma può essere uno svantaggio perché può essere un meccanismo con cui introdurre virus, worm o trojan facilmente.

Un virus è un malware che infetta programmi modificandoli e si riproduce. Può infettare la sezione di boot, i file e le macro. Le infezioni possono avvenire introducendo un disco infetto o tramite la rete. Un virus è composto da tre parti:

- Meccanismo di infezione: Questa è la parte del virus che rileva, infetta e si diffonde a nuovi file o sistemi. Cerca file o sistemi vulnerabili e inietta il suo codice in essi;
- Trigger: Questa è la parte del virus che determina quando il payload, il codice dannoso, deve essere eseguito Può essere attivato da vari trigger, come una data specifica, un evento particolare o un'azione specifica dell'utente;
- Payload: Questa è la parte principale distruttiva del virus. Contiene il codice dannoso che esegue le azioni dannose volute dal creatore del virus. Il payload può variare dalla cancellazione o dalla corruzione di file, al furto di dati sensibili, alla perturbazione delle operazioni di sistema, al blocco degli utenti dai propri sistemi.

Un multiple-threat malware è un tipo di malware che può eseguire più tipi di attacchi dannosi. Usa più metodi di infezione o trasmissione per massimizzare la velocità di contagio e i danni.

Un rootkit è un programma malevolo che fa ottenere permessi di amministratore del sistema. Possono essere:

- Persistenti: si attiva ogni volta che parte il sistema;
- Basati sulla memoria: non sopravvive al riavvio;
- User mode: intercetta le chiamate alle API e ne modifica i risultati;
- Kernel mode: intercetta chiamate ad API native in kernel mode e può nascondere la presenza di un malware rimuovendolo dalla lista dei processi attivi del kernel.

9.2 Buffer Overflow

Buffer Overflow è un meccanismo di attacco comune per cui ormai conosciamo varie tecniche di prevenzione. Sfrutta errori nella programmazione per cui diventa possibile scrivere più dati della capacità disponibile nel buffer, sovrascrivendo locazioni di memoria adiacenti. Questo può portare alla corruzione dei dati di un programma, trasferimento del controllo ed esecuzione di codice dell'attacker.

Per sfruttare il buffer overflow bisogna trovare vulnerabilità nel programma, capire come il buffer è salvato in memoria e determinare il potenziale di eventuali corruzioni di memoria.

Uno stack contiene degli stack frames (record di attivazione), ciascuno per ogni funzione chiamata. All'interno di questi si trovano le variabili locali, eventuali parametri passati e il return address. Si possono utilizzare dei frame pointers per indicare precise locazioni di memoria dello stack. Sfruttando il buffer overflow, nel modificare il contenuto di un array nello stack posso andare oltre e sovrascrivere lo spazio allocato fino al return address. Posso così fare in modo che punti alla porzione di codice che voglio venga eseguita, generalmente è shell code inserito nel buffer che mi ha permesso di modificare il return address.

Un linguaggio di programmazione di basso livello manipola la memoria direttamente per cui può essere più efficiente e preciso, ma si rischia di violare le astrazioni indicate. Un linguaggio di programmazione che si assicura che letture e scritture rimangano nei limiti del buffer si dice che impone memory safety.

Per fermare questo genere di attacchi bisogna programmare più attentamente e controllando i limiti del buffer, questo si può fare con specifici linguaggi di programmazione o attraverso librerie per linguaggi che non lo fanno di default. La protezione si può distinguere in due categorie: compile-time defense (rende il programma resistente ad attacchi) e stack protection mechanism (rivela e risolve eventuali attacchi in programmi esistenti). Per prevenire questi attacchi bisogna evitare di usare funzioni che non controllano la dimensione dell'input (es. strncpy al posto di strcpy in C).

Una tecnica di prevenzione compiler-level consiste nell'utilizzo di canarini: si inserisce una porzione di memoria nello stack fra il return address e le variabili locali di 32 bit e prima di proseguire dove indicato dal return address si controlla che il contenuto del canarino non sia stato modificato. Lo svantaggio è che richiede la recompilazione.

Un'altra tecnica system-level prevede di dividere le sezioni di memoria in cui è permesso eseguire da quelle in cui è permesso scrivere (W xor X). DEP (Data Execution Prevention).

Un Bot è un programma che si diffonde come un virus ma senza avere effetti indesiderati sul momento. Prende il controllo di centinaia o migliaia di dispositivi per creare una botnet fino a che non viene attivata per lanciare attacchi che non rendano tracciabile lo scrittore del bot (zombie o drone). Le caratteristiche necessarie di una botnet sono: capacità di eseguire un codice, facile controllo da remoto, facilità di propagazione. Sfruttano vulnerabilità comuni a molti sistemi. Attraverso ping e provando con una serie di indirizzi IP casuali tentano di entrare in reti locali e di propagarsi al loro interno. Con una botnet posso fare vari attacchi fra cui: DDoS, spamming, sniffing traffic, keylogging (cattura input tastiera), diffusione di malware, installazione di pubblicità per remunerazione e manipolare sondaggi o giochi online.

Un DoS (Denial of Service) è un attacco che prevede l'esaurimento delle risorse di un sistema per far sì che non sia disponibile a richieste lecite. L'attacco può riguardare la banda di rete, le risorse di sistema o le risorse di applicazioni. Un classico esempio è il flooding di ping, ovvero l'invio di dati da una rete che ha più capacità a una con meno capacità per intasarla e causare perdita di traffico. In questo tipo di attacco si rischia che il destinatario dell'attacco possa risalire alla sorgente da cui è partito. Altri tipi di attacchi flooding sono: ICMP flood (usa pacchetti di

tipo ECMP come le richieste echo), UDP flood (usa pacchetti UDP con alcune porte specifiche) o TCP SYN flood (mando dei SYN, cioè richieste di connessione, ricevo dei SYN ACK dal server che si mette in attesa del mio ACK finale ma non lo mando, così il server esaurirà la coda e richieste lecite saranno scartate). Posso inoltre utilizzare una botnet per rendere il mio attacco più efficace e meno tracciabile, in questo caso si parla di Distributed Denial of Service (DDoS). Potremmo bloccare il traffico se fosse inusualmente alto, ma potrebbe essere lecito e inoltre l'attaccante avrebbe comunque raggiunto il suo obiettivo. Si può risolvere limitando il numero di ping al secondo o inserendo dei captcha. Anche i service provider potrebbero essere attivi per evitare attacchi di questo tipo, bloccando pacchetti con indirizzo IP mittente che non corrisponde con quello reale o tenendo traccia dei percorsi che fanno i pacchetti, ma non è un costo che porterebbe beneficio al service provider stesso per cui è raro che implementi queste funzionalità.

9.3 Tecniche di protezione

9.3.1 Crittografia

Analizzeremo quattro tipi di elementi:

- Criptazione simmetrica;
- Criptazione asimmetrica (chiave pubblica);
- Firme digitali e gestione di chiavi;
- Funzioni hash sicure.

Alla base della crittografia abbiamo numeri casuali, che nei computer sono in realtà pseudocasuali. Sono importanti per via della loro imprevedibilità. I numeri pseudocasuali vengono generati da algoritmi, per questo sono più prevedibili di numeri generati da sorgenti non deterministiche.

Nella crittografia simmetrica si genera una chiave specifica per una comunicazione con cui si cifra e decifra il messaggio tramite un algoritmo. Un tipico attacco è il brute force in cui si tenta ogni possibile chiave per accedere al messaggio cifrato. Questo attacco non è realizzabile perché le chiavi sono di una lunghezza tale che si necessiterebbe di anni per trovare quella giusta.

Key Size (bits)	Number of Alternative Keys	Time Required at 1 Decryption/μs	Time Required at 10^6 Decryptions/μs
32	$2^{32} = 4.3 \times 10^9$	$2^{31}\ \mu s$ = 35.8 minutes	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	$2^{55}\ \mu s$ = 1142 years	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	$2^{127}\ \mu s = 5.4 \times 10^{24}$ years	5.4×10^{18} years
168	$2^{168} = 3.7 \times 10^{50}$	$2^{167}\ \mu s = 5.9 \times 10^{36}$ years	5.9×10^{30} years
26 characters (permutation)	$26! = 4 \times 10^{26}$	$2 \times 10^{26}\ \mu s = 6.4 \times 10^{12}$ years	6.4×10^{6} years

Per cifrare un insieme di dati li posso dividere in blocchi e cifrare ciascuno separatamente. Se invece ho uno stream di dati la cifratura deve essere continua. Nello stream faccio lo xor con la chiave per ogni parte. La chiave deve essere generata pseudo-casualmente(...)

Gli algoritmi di criptazione simmetrica noti sono tre: DES, Triple-DES e AES.

L'algoritmo DES (Data Encryption Standard) funziona dividendo il testo in chiaro in blocchi di 64 bit. Ogni blocco viene quindi cifrato utilizzando una chiave di 56 bit. Il processo di cifratura è costituito da 16 passaggi, ciascuno dei quali viene eseguito utilizzando una funzione di permutazione e una funzione di sostituzione. La funzione di permutazione è una funzione che riorganizza i bit del blocco. La funzione di sostituzione è una funzione che sostituisce i bit del blocco con altri bit.

L'algoritmo Triple-DES consiste nel ripetere DES tre volte usando due o tre chiavi uniche. È più sicuro ma più lento.

L'algoritmo AES (Advanced Encryption Standard) è nato per rimpiazzare il DES. Utilizza blocchi di 128 bit e chiavi da 128/192/256 bit.

Factors	AES	3DES	DES
Key Length	128, 192 or 256 bits	(k1, k2 and k3) 168 bits (k1 and k2 are same) 112 bits	56-bit
Cipher type	Symmetric block cipher	Symmetric block cipher	Symmetric block cipher
Block size	128,192, or 256 bits	64 bits	64 bits
Developed	2000	1978	1977
Cryptanalysis resistance	Strong against differential, truncated differential, linear, interpolation and square attacks	Vulnerable to differential; Brute Force Attacker could analyze plain text using differential cryptanalysis	Vulnerable to differential and linear cryptanalysis; weak substitution tables
Security	Considered Secure	One Only weakness, which exists in DES	Proven inadequate
Possible Keys	2^{128}, 2^{192}, or 2^{256}	2^{112} or 2^{168}	2^{56}
Possible ASCII printable character keys	95^{16}, 95^{24} or 95^{52}	95^{14} or 95^{21}	95^{7}
Time Required to check all possible keys at 50 billion keys per second**	For a 128-bit key: $5x10^{21}$ years	For a 112-bit key: 800 days	For a 56-bit key: 400 days

La crittografia asimmetrica, conosciuta anche come crittografia a chiave pubblica, è un tipo di crittografia che utilizza due chiavi diverse per criptare e decriptare i dati. Una delle chiavi, chiamata chiave pubblica, è condivisa con chiunque voglia comunicare con il proprietario della chiave. L'altra chiave, chiamata chiave privata, è segreta e solo il proprietario della chiave la conosce.

Per criptare i dati con la crittografia asimmetrica, il mittente utilizza la chiave pubblica del destinatario per criptare i dati. I dati criptati sono illeggibili a chiunque non abbia la chiave privata corrispondente. Per decriptare i dati, il destinatario utilizza la propria chiave privata per decriptarli. I dati decriptati sono leggibili solo al destinatario.

Se per esempio Alice e Bob vogliono intraprendere una comunicazione sicura, entrambi generano una coppia di chiavi pubbliche e private. Si scambiano le chiavi pubbliche in modo che Alice ha la chiave pubblica di Bob e la propria chiave privata, Bob ha la chiave pubblica di Anna e la propria chiave privata.

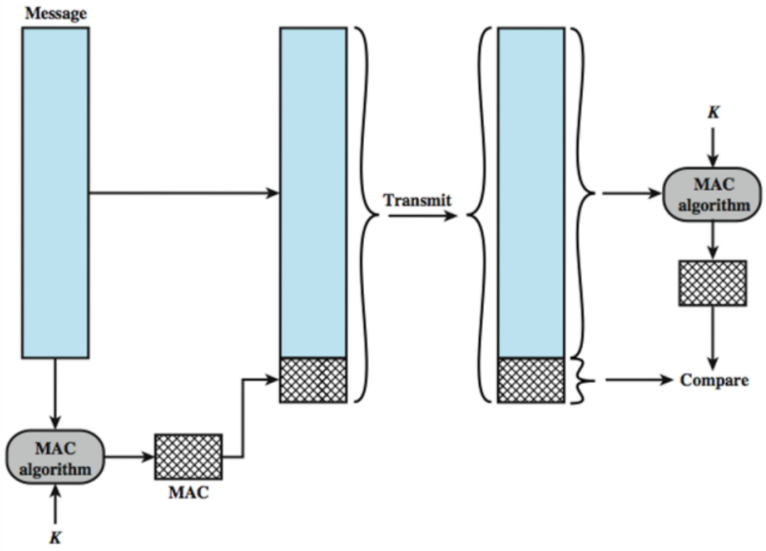
Deve essere semplice creare queste coppie di chiavi ma impossibile risalire alla privata partendo da quella pubblica.

Alcuni esempi di algoritmi a chiave pubblica sono: RSA, exchange algortithm, digital signature standard, crittografia a curva ellittica, homomorphic cryptography.

La chiave privata può essere usata per autenticarsi. Si cifra con la chiave privata e chiunque decifra con la chiave pubblica sa chi ha mandato il messaggio.

Il Message Authentication Code (MAC) è un piccolo blocco di dati che viene generato e messo in coda al messaggio Il MAC viene generato utilizzando un algoritmo di crittografia che utilizza una chiave segreta conosciuta solamente da mittente e destinatario.

L'obiettivo del MAC è garantire che il messaggio non sia stato modificato durante la trasmissione e che provenga da una fonte attendibile. Per verificare l'autenticità e l'integrità di un messaggio, il destinatario calcola il MAC del messaggio ricevuto e lo confronta con il MAC ricevuto dal mittente. Se i due MAC corrispondono, il destinatario può essere sicuro che il messaggio non è stato modificato e che proviene dalla fonte attendibile.



Si utilizzano funzioni di hash che permettono di avere come output un numero di lunghezza fissa indipendentemente dal messaggio in input. Come funzione hash in genere si usa l'algoritmo SHA.

Un certificato di chiave pubblica è un documento digitale che associa una chiave pubblica a un'entità, come una persona, un computer o un'organizzazione. Il certificato è firmato da un'autorità di certificazione (CA), che è un'entità fidata che garantisce l'autenticità della chiave pubblica.

Un modo di generare il certificato prendere il certificato senza firma digitale, generare il suo hash code, cifrarlo con la chiave privata del CA e aggiungerlo al certificato non firmato. Per verificare l'autenticità di un certificato di chiave pubblica, è necessario verificare la firma digitale dell'autorità di certificazione. Questo può essere fatto utilizzando la chiave pubblica dell'autorità di certificazione.

Le Digital Envelope, o buste digitali, sono un tipo di crittografia a chiave pubblica che consente di proteggere i dati inviati da un mittente a un destinatario.

Per creare la busta, il mittente genera una chiave simmetrica con cui cifrare il messaggio e la chiave pubblica del destinatario per cifrare la chiave simmetrica. L'unione del messaggio cifrato e la chiave cifrata formano la Digital Envelope. Per aprire la busta, il destinatario usa la propria chiave privata per decifrare la chiave simmetrica, e la usa per decifrare il messaggio.

9.3.2 Autenticazione

Ci sono vari metodi per autenticare un utente. Vediamo quello basato sulle password e quello basato sui token.

Nell'autenticazione basata su password, a ogni ID viene associata una password. L'ID determina i privilegi dell'utente. Per mantenere un certo grado di sicurezza alle password, non vengono salvate direttamente ma viene salvato l'hashcode. L'hashcode viene modificato con il "sale", che ne modifica alcuni bit.

Un utente, per autenticarsi, digita l'ID e la password. A questo punto il sistema, tramite l'ID, risale al "sale" e all'hashcode. Dalla password digitata e dal "sale", viene creato l'hashcode e fatto il confronto con quello salvato.

Il "sale" ha tre scopi:

1. Previene che password uguali siano visibili nella tabella (avendo "sale" diverso, hanno hashcode diversi);
2. Incrementa la difficoltà di attacchi offline alla tabella;
3. Diventa quasi impossibile scoprire se una persona abbia usato la stessa password su tutti i sistemi.

Nei sistemi UNIX, ci sono due minacce alla tabella delle password:

- Un utente può accedervi usando un guest account;
- Password cracker.

Se un attaccante ottiene una copia della tabella delle password, può usare un cracker che permette di provare milioni di possibilità in un tempo ragionevole.

Un token è un oggetto che un utente possiede e permette di autenticarlo. Esistono due tipi di token:

- Memory card: possono memorizzare dati ma non processarli. Un esempio è una vecchia carta di credito con una banda magnetica. una banda magnetica può memorizzare solo un semplice codice di sicurezza, che può essere letto e riprogrammato da un lettore specifico;
- Smart card: contengono un microprocessore integrato. Un esempio è una carta di credito. L'autenticazione può essere statica o dinamica generando password.

Possiamo anche utilizzare le nostre biometrie per autenticarci poiché sono uniche e le abbiamo sempre con noi, una caratteristica richiesta è che non varino nel tempo. Alcuni esempi sono le caratteristiche facciali, le impronte digitali, la forma della mano, la retina, l'iride, la firma o la voce. Il pericolo maggiore è che vengano "rubate" perché non si possono resettare. Sono utilizzate anche biometrie comportamentali.

9.3.3 Controllo di accesso

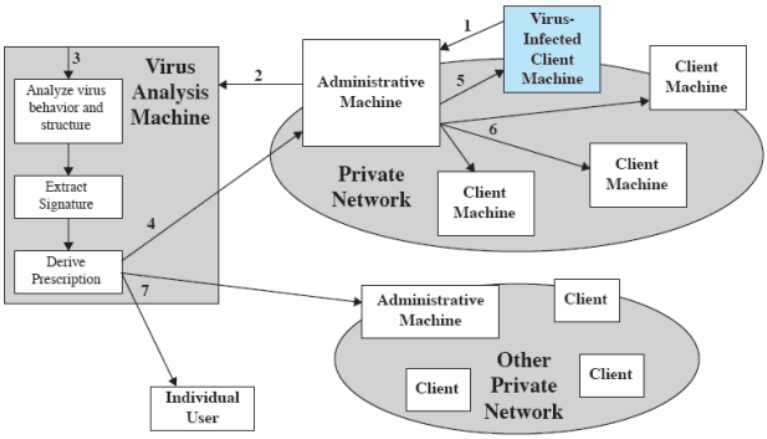
Nel controllo di accesso, si assegna il tipo di accesso in base alle circostanze o all'utente che lo richiede. Le politiche di access control sono di tre tipi:

- Discretionary Access Control (DAC): limita l'accesso alle risorse del sistema in base alle autorizzazioni del soggetto che sta tentando di accedervi;
- Mandatory Access Control (MAC): limita l'accesso alle risorse del sistema in base alla sensibilità delle informazioni che le risorse contengono e alle autorizzazioni del soggetto che sta tentando di accedervi;
- Role-Based Access Control (RBAC): limita l'accesso alle risorse del sistema in base ai ruoli dei soggetti che stanno tentando di accedervi. Si possono avere più ruoli.

9.3.4 Antivirus

L'antivirus è una soluzione efficace che controlla i file prima di memorizzarli, se malevoli li rimuove a meno che l'utente non dia indicazione diversa. Ogni antivirus può avere un suo decifratore che serve a rilevare complessi virus polimorfi (virus che nasconde la cosiddetta "impronta virale", ovvero la sequenza di byte che identifica in maniera univoca un virus, il quale crittografa il proprio codice e utilizza di infezione in infezione una chiave diversa).

Nel caso di sistemi più grandi si usa un Digital Immune System che identifica potenziali virus e li manda ad una macchina interna o esterna che analizza, ne crea una signature e la manda ai sistemi locali per poterlo riconoscere.



9.3.5 Firewall

Un firewall si occupa di bloccare pacchetti non legittimi provenienti dalla rete con delle regole. Queste regole possono essere basate su un'ampia gamma di criteri, tra cui indirizzi IP, porte di

rete, protocolli di rete, tipi di dati ecc. I firewall hanno delle limitazioni, come il fatto che esistono attacchi che lo raggirano, non protegge da minacce interne alla rete, non protegge bene le WLAN, i dispositivi si possono infettare fuori dalla rete e successivamente usati dentro la rete.

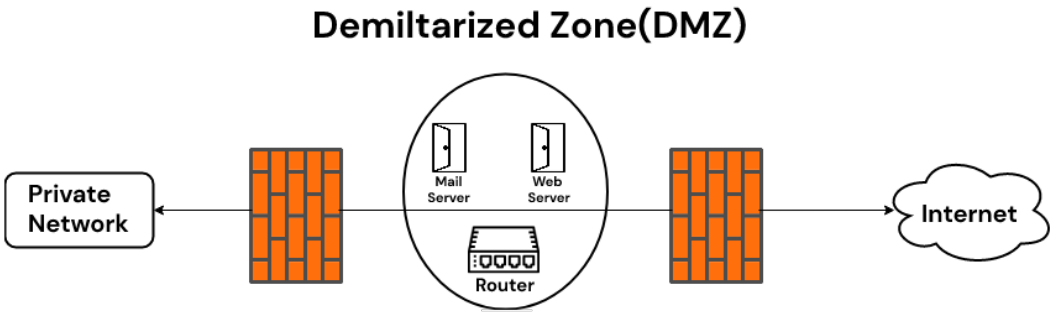
I firewall possono essere classificati in base alla loro posizione nella rete:

- Firewall perimetrali: si trovano all'interfaccia tra una rete interna e una rete esterna, come Internet;
- Firewall interni: si trovano all'interno di una rete per proteggere le risorse interne da attacchi provenienti da altre parti della rete;
- Firewall di host: si trovano su un singolo computer per proteggere quel computer da attacchi provenienti da altre parti della rete.

Esistono altre classificazioni di firewall, come quello basato sugli strati di rete:

- Packet filtering firewall: analizzano i pacchetti in base ai loro indirizzi IP di origine e destinazione, alle porte di origine e destinazione e ai protocolli di rete utilizzati. Ci sono due tipi di politiche, discard (fa passare i pacchetti solo se permessi) e forward (fa passare i pacchetti a meno che non sia espressamente negato). Non previene attacchi basati su bug di applicazioni, non supporta autenticazioni avanzate, vulnerabile agli attacchi su bug del protocollo TCP/IP, una configurazione impropria può portare a violazioni;
- Stateful inspection firewall: operano a livello di rete. Tuttavia, a differenza dei firewall di filtraggio dei pacchetti, i firewall di ispezione dello stato mantengono una tabella di stato che tiene traccia delle connessioni di rete attive. Questa tabella di stato include informazioni come gli indirizzi IP di origine e destinazione, le porte di origine e destinazione, i protocolli di rete utilizzati e lo stato della connessione. Quando un nuovo pacchetto di dati arriva a un firewall di ispezione dello stato, il firewall utilizza la tabella di stato per determinare se il pacchetto è parte di una connessione esistente. Se lo è, il firewall consente il traffico. Se non lo è, il firewall applica le regole di filtraggio dei pacchetti per determinare se consentire o bloccare il traffico;
- Application proxy firewall: operano a livello di applicazione. Possono bloccare o consentire il traffico di rete in base al tipo di applicazione, al protocollo di applicazione e ai dati specifici;
- Circuit-level proxy firewall: operano a livello di trasporto. Funzionano creando un tunnel virtuale tra i dispositivi client e server. Questo tunnel virtuale viene utilizzato per inoltrare il traffico di rete in modo sicuro tra i due dispositivi. Ad esempio, può essere configurato per bloccare il traffico di file sharing peer-to-peer (P2P) o per limitare la quantità di dati che possono essere trasferiti tramite una connessione. Non possono analizzare il contenuto dei dati che vengono trasferiti attraverso il tunnel virtuale.

Una zona demilitarizzata (DMZ) è una sottorete fisica o logica che separa una rete interna da una rete esterna, come Internet. La DMZ contiene server o altri servizi che devono essere accessibili dal pubblico, ma che devono essere protetti dalla rete interna.



9.3.6 Intrusion detection

La rilevazione degli intrusi si basa sull'assunzione che il comportamento dell'intruso differisce da quello di un utente legittimo, e analizza quindi il comportamento degli utenti nel tempo, se la

rilevazione avviene abbastanza in fretta si possono evitare possibili danni e compromissioni, un sistema di questo tipo funziona anche da deterrente.

Può essere nativo, quindi senza bisogno di software addizionale, ma potrebbero mancare delle informazioni utili all'identificazione o specifico, grazie a un venditore esterno, ma si crea un grosso overhead.

- Rilevamenti anomali: usato per Dos, scanning e worms;
  - Threshold detection: Controlla un eccessivo verificarsi di eventi nel tempo;
  - Profile based: Caratterizza il passato comportamento di utenti o gruppi e identifica deviazioni significative.
- Rilevamento firma: usato per app, tran, net layers e violazioni impreviste.
  - 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, grazie anche all'aiuto dell'intelligenza artificiale.

Una honeypot è un sistema (singolo o una rete) che simula la rete interna ed è posta generalmente prima del firewall verso l'esterno, il suo scopo è attirare su di sé gli attacchi per raccoglierne informazioni.

9.4 Conclusioni

La sicurezza è connessa alla qualità del software, infatti gli attacchi mirano a parti difettose di un codice, magari inserendo input malevoli.

La Difensive Programming è una forma difensiva che richiede attenzione in ogni aspetto della progettazione del software, controllando tutte le possibili vulnerabilità.

Bisogna stare attenti soprattutto ai possibili input che può ricevere un programma, per evitare "attacchi ad iniezione".

Per la sicurezza dell'OS è importante la pianificazione dell'istallazione, configurazione, aggiornamento e mantenimento dell'OS stesso e delle sue applicazioni principali.