1. Processi e Thread

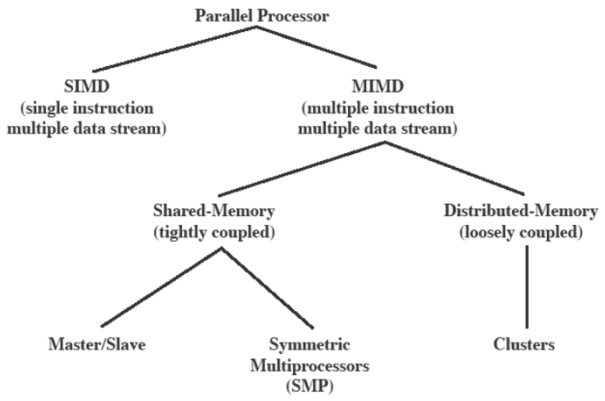
Ogni programma è un processo e ha una esecuzione, comprende codice, dati e attributi. Gli attributi di un processo sono: identificatore, stato priorità, puntatori, stato dei registri della CPU, stato dell'I/O, indirizzo della prossima istruzione. Viene salvato tutto nel **Process Control Block**, creato e gestito dal sistema operativo che può così 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 posso controllare chi è il padre e chi è il figlio. 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, altrimenti 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 Process Control Block del figlio sarà uguale a quello del padre, naturalmente eccetto l'id processo. La struttura del codice sarà quella in figura. Nel codice del figlio potrò usare una **exec(..)** per lanciare un nuovo programma che sovrascriva interamente il processo figlio. Alla fine dell'esecuzione di usa la **exit()** che prende come parametro uno status per indicare se si è terminati correttamente. Con la exit vengono eseguite varie funzioni per chiudere i file, le connessioni, deallocare memoria. Se il processo ha figli questi vengono assegnati ad init. Se il padre è ancora vivo si diventa zombie per mantenere il risultato da restituire alla wait del padre, altrimenti si muore. Il padre può aspettare 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 metto nei parametri di una wait il 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.

```
ret = fork();
switch (ret)
{       case -1: perror("fork"); exit(1);
        case 0: /codice del figlio/; exit(0);
        default: /codice del padre/; wait(0);
}
```

Un processo ha due caratteristiche: le risorse richieste (spaziale) e il cammino di esecuzione (temporale). Della parte temporale si occupano i **thread**. 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 accesso protetto al processore, ad altri processi, ai file e alle risorse I/O. Ogni thread ha uno stato di esecuzione, 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 Process control block, un user address space e, per ciascun thread, un Thread Control Block, un user stack e un un kernel stack. I **benefici** dei thread sono la rapidità di creazione e terminazione, la comunicazione fra thread non necessita l'invocazione del kernel, la rapidità nel passare da un thread all'altro. 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.

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 catturato dalla join (ricorda che non libera automaticamente le risorse come una exit, bisogna farlo a mano). 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.



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: mutua esclusione e sincronizzazione

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**:

- **Operazione atomica**: una sequenza di istruzioni che non può essere interrotta.
- **Sezione critica**: sezione di codice che richiede accesso a risorse condivise che non devono essere toccate da altri thread che lavorano contemporaneamente.
- **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. Magari inoltre blocca una risorsa e non permette agli altri 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 interrupts 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 irriproducibili. Il **sistema operativo** deve conoscere tutti i processi in esecuzione, il loro stato, allocare e deallocare risorse per ognuno, proteggere dati e risorse da 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 e 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. La sezione critica deve avere tempo finito. Bisogna sempre stare attenti a non causare deadlock e starvation. Si può rendere a **livello hardware**: disabilito 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. 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).

```
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;
}
```

```
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) {
                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));
}
```

```
/*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));
}
```

Questo schema di programma prevede un insieme di thread che si fermano alla critical section. 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. Questa implementazione si può 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 critical section semplicemente definendo più variabili di controllo. Fra gli **svantaggi** abbiamo il busy-waiting poiché il processo che attende continua a consumare tempo del processore, starvation quando un processo lascia la sezione critica e più di un processo sta attendendo di

entrarci, c'è possibilità che si verifichi il deadlock dovuto ad esempio a priorità fra processi, non è disponibile in tutte le architetture. 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: è un tipo di dato che 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 finchè 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.

I **Semafori**: si inizializza ad un valore non negativo, con la semWait decremento, con la semSignal incremento. Non posso manipolare il valore in altro modo se non attraverso queste tre funzioni. 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 figura una implementazione di primitive di un semaforo. In uno **strong semaphore** il processo che è stato bloccato per più tempo è il primo rilasciato dalla coda (FIFO), in un **weak semaphore** l'ordine in cui i processi escono dalla coda non è specificato.

```
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 */
        }
}
```

```
/*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 posso 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.

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. La sincronizzazione viene gestita tramite variabili di condizione locali del monitor che vengono modificate da cwait(c) o csignal(c) (operano sulla specifica condizione c).

Il **Message Passing** è utile nella gestione di sistemi distribuiti che quindi non hanno memoria in comune. Permette sincronizzazione e 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. 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 prosseguire 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

```
/*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));
}
```

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. In figura una implementazione di programma con mutua esclusione che sfrutta il message passing.

```
/* program producerconsumer */
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();
        }
}
```

```
/* program producerconsumer */
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();
        }
}
```

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.

```
/* program readersandwriters */
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);
         }
```

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.

3. Deadlock e Starvation

Il deadlock si verifica quando un processo bloccato attende un segnale di sblocco da un altro processo, questo a sua volta bloccato dal primo. È permanente e non ha soluzione, bisogna quindi evitarlo. Le **risorse** si dividono in **riutilizzabili** (I/O, processore, database, file, semafori), possono essere utilizzate in sicurezza solo da un processo alla volta e non sono eliminate dopo l'uso, o **consumabili** (interrupt, segnali, messaggi), possono essere create e distrutte. Le **condizioni** per il deadlock sono: mutua esclusione (se non ho mutua esclusione non potrò avere deadlock perché più processi accederanno alla risorsa senza attendere), hold-and-wait (un processo mantiene bloccate delle risorse mentre attende che gliene vengano assegnate delle altre), no preemption (non è possibile rimuovere forzatamente delle risorse ad un processo). Queste condizioni sono **necessarie ma non sufficienti**. Ho una **quarta condizione**, la **circular wait,** quando ho una catena di processi in cui ognuno mantiene una risorsa e ha bisogno di un'altra risorsa tenuta a sua volta da un altro processo (non necessariamente avrò deadlock). 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 l'hold-and-wait il processo può prendere le risorse solo quando sono tutte disponibili ma questo genera lunghe attese e non sempre so in anticipo di cosa avrò bisogno. 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 sono in ordine "dopo" quelle che già ha, 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) o **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 le richieste di risorse di un processo, può considerare solo processi indipendenti e senza sincronizzazione, necessita di un numero fisso di risorse allocabili e nessun processo può terminare se ha risorse allocate.

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}

Algoritmo di allocazione risorse:
if (alloc [i,*] + request [*] > claim [i,*]) <error>;
/le risorse richieste superano quelle disponibili/
else if (request [*] > available [*]) <suspend process>;
/non ho risorse disponibili/
else {  <define newstate by:
        alloc [i,*] = alloc [i,*] + request [*];
        available [*] = available [*] – request [*] >;
} /simulo l'allocazione delle risorse/
if (safe (newstate)) <carry out allocation>;
else {  <restore original state>;
        <suspend process>;
}
```

```
Algoritmo per il test for safety (banker's algorithm)
boolean safe (state S) {
        int currentavail[m];
        process rest[<number of processes>];
        currentavail = available;   /risorse disponibili/
        rest = {all processes};
        possible = true;
        while (possible) { /molto oneroso, controlla tutti i processi/
                <find a process Pk in rest such that
                claim [k,*] – alloc [k,*] <= currentavail;>
                if (found) {        /simula esecuzione di Pk/
                        currentavail = currentavail + alloc [k,*];
                        rest=rest – {Pk};
                }
                else possible = false;
        }
        return (rest == null);      /se false lo stato non è sicuro/
}
```

- **Individuo**: **e recupero** il deadlock. Le strategie di individuazione hanno come idea fondamentale la concessione di risorse solo se 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, questo metodo individue 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. Nel caso in esempio P4 viene segnato poiché ancora non ha risorse allocate, P3 viene segnato perché se gli diamo le risorse richieste (che sono disponibili) termina e rilascia tutte quelle che ha. A questo punto però né P1 né P2 possono avere allocate le risorse che richiedono e quindi siamo in una situazione di deadlock. Per **recuperare** un deadlock posso uccidere i processi in deadlock (soluzione comune), ripristinare il sistema ad uno stato precedente senza deadlock (non mi garantisce che non si ripresenti lo stesso problema), uccidere uno alla volta i processi in deadlock cercando di risolverlo senza doverli uccidere tutti, usare la preemption (sottraggo risorse) per eliminare forzatamente il deadlock.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

Allocation vector

R1	R2	R3	R4	R5
0	0	0	0	1

La **mutua esclusione** può essere garantita anche tramite **approcci software**. Supponiamo di avere processi che comunicano con una memoria centrale e richiedono mutua esclusione. Ho bisogno di arbitro che gestisca read/write dei processi senza l'intervento di sistema operativo, hardware o linguaggio di programmazione. Si può utilizzare **l'algoritmo di Dekker** che funziona per mutua esclusione fra due processi. Di seguito indichiamo vari tentativi prima di arrivare all'algoritmo corretto. 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.

Dijkstra ha **generalizzato** l'algoritmo di **Dekker** per poter funzionare con un numero di processi maggiore di 2. Si rischia sempre che due processi 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.

```
Algoritmo di Dekker
/*GLOBAL*/ boolean flag[2] = {false, false}
int me = 0, other = 1; //P0 (flip for P1)
while (true) {
        /*non critical section*/
        flag[me] = true;
        while (flag[other]) {
                if (turn == other) {
                        flag[me] = false;
                        while (turn == other) /*busy wait*/;
                        flag[me] = true;
                }
        }
        /*critical section*/
        turn = other;
        flag[me] = false;
}
```

```
Algoritmo di Dijkstra
/*GLOBAL*/     boolean interested[N] = {false, …, false}
                boolean passed[N] = {false, …, false}
                int k = <any>       //k fra 0 e N-1
/*LOCAL*/      int i = <entity ID>//i fra 0 e N-1
while (true) {
        /*non critical section*/
        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 "while (k != i)"
        /*critical section*/
        passed[i] = false; interested[i] = false;
}
```

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. Nel 1975 Lamport sviluppa l'algoritmo del panettiere (**bakery algorithm**). Il concetto principale è che i processi prendono un numeretto e si mettono in fila per entrare nella sezione critica. Una prima implementazione basica prevede un array in cui ogni elemento contiene il numero di fila di un processo. Quando voglio entrare nella sezione critica controllo i numeri di fila degli altri processi e mi setto nell'array nel mio indice al massimo più uno. Quando arriva il mio turno, ovvero non c'è nessun processo con numero minore del mio, entro nella sezione critica e quando esco metto il mio numeretto a zero. Con questa implementazione rischio il deadlock se due processi si settano contemporaneamente allo stesso numero di fila, potrei controllare gli identificatori dei processi e dire che in questo caso entra quello con identificatore minore. Il problema è che quando cerco il massimo per settare il mio numeretto devo utilizzare una operazione non atomica che potrebbe causare errori e non garantire mutua esclusione. La soluzione è nel codice in figura, prevede che un processo attenda nel controllare il numero di fila di un processo che lo sta modificando, maximum serve per non arrivare a numeri di fila infiniti. Le **caratteristiche** di questo algoritmo sono che i processi comunicano leggendo e scrivendo su variabili condivise (come dijkstra), read e write non devono più essere operazioni atomiche, ogni variabile condivisa è posseduta da un processo che la può scrivere mentre gli altri la possono leggere, nessun processo può quindi fare scritture concorrenti, i tempi di esecuzione sono scollegati.

```
Ho N processi, ciascuno con identificatore i tra 1 ed N.
Ho due array: choosing contenente booleani e number contenente interi.
while (1) {
        /*non critical section*/
        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));
        }
        /*critical section*/
        number[i] = 0;
}

Questa riga "number[i] = 1 + max {number[j] | (1 <= j <= N)}" si implementa:
local1 = 0;
for local2 in 1..N {
        local3 = number[local2];
        if (local1 < local3) local1 = local3;
}
number[i] = 1 + local1
```

- **Comunicazione Inter Processo (IPC)**

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 inter processo sono di due tipi: **memoria condivisa** o **scambio di messaggi**. Nel primo caso il kernel assegna un'area di memoria condivisa che può essere acceduta da entrambi i processi, nel secondo caso i messaggi sono scambiati tramite il kernel.

Nel caso di **memoria condivisa**, una volta assegnata dal kernel, questa sarà di rapido accesso per i processi perché non avranno bisogno di ripassare dal kernel ogni volta. 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.

Nel caso di **message passing**, i messaggi sono salvati in una coda del kernel. Un processo scrive in questa coda e l'altro legge ma entrambi devono chiamare il kernel ogni volta e questo causa una notevole perdita di tempo. Abbiamo bisogno di 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 stabilito, ciascuno contiene: tipo messaggio, ID destinatario e mittente, lunghezza del

messaggio, informazioni di controllo, contenuto del messaggio. Nel caso di Posix 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).

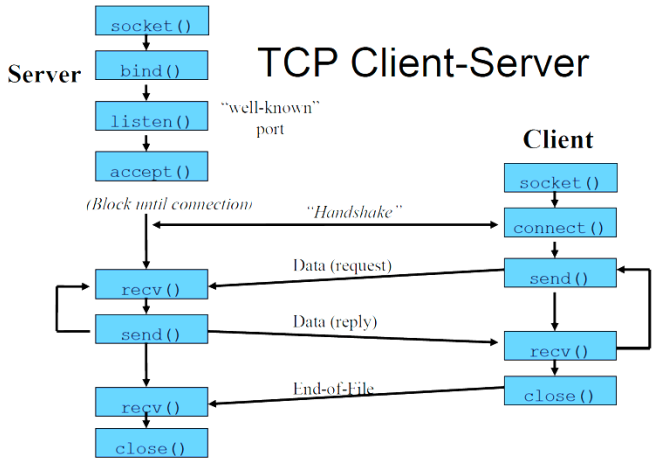
In sistemi Unix una **Pipe** attiva un collegamento monodirezionale fra due processi padre e figlio, quando il processo termina la pipe è eliminata, si basa sulla system call pipe(). Le pipe permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali, le informazioni lette 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 equivalente a 4096 byte in linux (16 pagine di memoria). È possibile rendere la lettura e la scrittura non bloccanti, ma non lo trattiamo. Una scrittura di dimensione minore del buffer è **atomica**, se il numero di byte da scrivere è maggiore della dimensione del buffer il processo si bloccherà ed è possibile che altri processi riprendano a scrivere prima di me nel momento in cui la pipe si inizia a svuotare. 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.

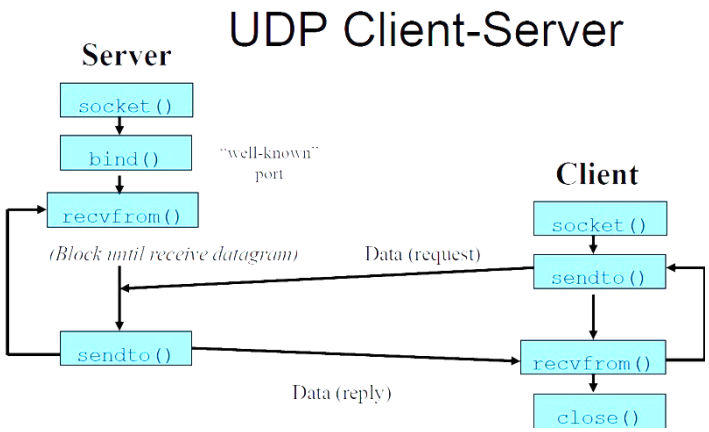
- **Socket**

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.

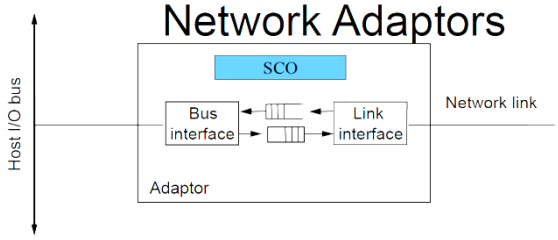
Una **Socket** permette comunicazione fra un processo client e un server. Può essere una **stream socket** (TCP - con connessione, affidabile), una **datagram socket** (UDP - senza connessione, non affidabile) o una raw socket (accesso diretto ai livelli bassi del protocollo). Per identificare una socket si utilizzano **l'indirizzo IP** dell'host e un **numero di porta**. Nel funzionamento il client manda una request con un numero di porta sconosciuto (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. La chiamata alla funzione socket() restituisce un descrittore. Le altre **primitive necessarie** per la comunicazione sono listen() per bloccare il processo in attesa di ricevere dei dati, connect() che tenta di stabilire una connessione, send() per inviare i dati, receive() per bloccare il processo in attesa di ricevere dati, disconnect() per avvisare che si vuole chiudere connessione. Le primitive per le socket TCP di Berkeley (Unix) sono socket() per creare comunicazione, bind() per assegnare il mio indirizzo alla socket, listen() per prepararsi ad accettare connessioni, accept() per accettare una richiesta di connessione(), connect() con l'indirizzo del destinatario per stabilire una connessione, send() per mandare dati, recv() per ricevere dati, close() per chiudere 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.

```
struct in_addr {
        in_addr_t s_addr;                   /32 bit IPv4 address
}
struct sockaddr_in {
        uint8_t         sin_len;            /lunghezza struttura
        sa_family_t     sin_family;         /AF_INET (IPv4)
        in_port_t       sin_port;           /port num
        struct in_addr  sin_addr;           /IPv4 address
        char            sin_zero[8];        /unused
}
/*inizializzare a zero sockaddr prima di usarla*/
```





La funzione int gethostname(char* hostname, int bufferLenght) richiede il nome dell'host su cui il programma sta girando. La funzione unsigned long inet_addr(char* address) converte un indirizzo ip dalla notazione puntata in un valore numerico a 32 bit. La funzione char* inet_ntoa(struct in_addr address) converte un indirizzo ip da struttura in_addr in notazione puntata. La funzione struct hostent* gethostbyname(const char* hostname) restituisce le informazioni riguardanti un host name (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.



Un **network adaptor** (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, un registro in cui entrambi possono 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 dispostivi che non fanno altro come i router) o **Interrupt** (l'adaptor invia un interrupt all'host che va quindi a leggere il CSR per capire cosa fare). Il **trasferimento dei dati** dall'adaptor alla memoria e viceversa può essere **Direct Memory Access** (non coinvolge la CPU, l'host ha un'area di memoria assegnata in cui vengono immediatamente inviati i frame, bastano pochi bytes sulla scheda) o **Programmed I/O** (lo scambio dati tra memoria ed adapter passa per la CPU, bisogna bufferizzare almeno un frame sulla CPU, la memoria deve essere dual port per cui sia processore che adaptor devono poter leggere e scrivere). Nel caso di Direct Memory Access la memoria dove allocare i frames 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 buffers da 1500bytes. Per i frame che arrivano dall'adapter 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. Tale messaggio viene processato da tutti i livelli che aggiungono le header. Quando il messaggio è pronto viene avvertita la SCO dell'adapter 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 dell'adaptor.

- **IoT**

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.

- **Sistemi distribuiti**

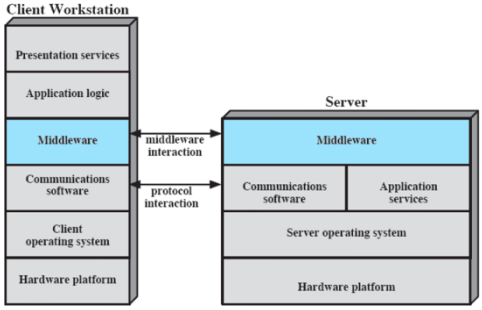
Un sistema distribuito è un insieme di entità separate spazialmente, non più nello stesso computer. Con **entità** indichiamo processi e thread ad esempio. Devono poter **comunicare e coordinarsi.** I sistemi distribuiti sono già molto presenti in tutti gli ambiti (network di workstation, sistemi manufatturieri 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

La **difficoltà** maggiore sta nel far comunicare i sistemi che lavorano in ambiente distribuito. Inoltre viene generato molto traffico dati nella rete e aumento i rischi legati alla sicurezza. 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 un dato sia bloccato in una macchina affinché rimanga disponibile
- Scalabilità: bisogna controllare le risorse fisiche, evitare perdita di prestazioni dovuta all'overhead, evitare colli di bottiglia nel sistema
- Affidabilità: non ho problemi se fallisce una sola macchina ma devo trovare, tollerare e risolvere eventuali failure. Potrei quindi necessitare di ridondanza
- 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 tempi 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 e ad ogni applicazione grazie a specifiche API.



Nel caso di **Client-Server computing**, ho un client con basse prestazioni e un server che supplisce alle prestazioni del client. Sono connessi in Lan o tramite internet. L'applicazione si svolge nel server mentre il client viene usato solo per mostrare il risultato finale. È importante distinguere i compiti fra il client e il server. Supponiamo 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
- cooperative processing: presentation e application sul client. Application, database e DBMS sul server
- client based processing: presentation, application, database sul client. Database e DBMS sul server

Nel caso di **Client/Server a 3 livelli** ho un server di mezzo che distribuisce le richieste fra i vari server. Una **SOA** Service Oriented Architecture è spesso usata nelle aziende ed organizza le funzioni in una struttura modulare interconnessa. Ho una serie di servizi e di clienti che sono collegati tramite interfacce che permettono la comunicazione. I Service provider rendono noti i loro servizi al Service broker, quando un Service requester ha bisogno di qualcosa chiede al broker che lo rimanda al provider corretto.

Per comunicare nei sistemi distribuiti è necessario il **message passing** che avviene fra i middleware. Una **Remote Procedure Call** RPC è una funzione eseguibile da remoto, si necessita di standardizzazione. Client e server comunicano attraverso degli RPC stub programs. La difficoltà maggiore che si incontra in questo caso è il passaggio di puntatori per cui si deve trovare delle soluzioni alternative. Posso avere connessioni persistenti o non persistenti. Le RPC possono essere sincrone (chi le chiama attende il risultato) o asincrone (chi le chiama prosegue nell'elaborazione senza attendere il risultato che arriverà più avanti). Nel caso in cui ci siano dei **problemi** posso scegliere quale politica applicare. 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. At most once prevede che non si faccia nulla nel caso in cui non si riceva il risultato. Exactly one è la più difficile da implementare.

- **Sicurezza**

La sicurezza è un processo che va gestito correttamente, nulla è completamente sicuro ma bisogna lavorare per avere un livello più alto possibile. 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 sicurezza di un sistema è pari alla sicurezza del componente meno sicuro. La crittografia è un ottimo strumento ma non basta. Il **cybercrime** è un problema reale che va affrontato. La Computer Security è la protezione di un sistema di calcolo per preservare l'integrità, la disponibilità e la confidenzialità di informazioni e risorse. La **triade CIA** comprende i tre obiettivi della sicurezza: **Confidentiality**, **Integrity** e **Availability**. Ho in aggiunta due concetti fondamentali: **Authenticity** (verificare se una informazione è genuina e se proviene dal mittente indicato) e **Accountability** (poter tracciare le responsabilità di una falla nella sicurezza). Le possibili **minacce** riguardano l'hardware, il software, i dati o le comunicazioni.

Gli **attacchi** si dividono in **attivi** (leggono, modificano, generano e distruggono informazioni) o **passivi** (leggono informazioni). Gli attivi possono essere di quattro categorie: replay (catturo informazioni e le rimando), masquerade (mi spaccio per qualcun altro), modification of messages, denial of service (mando informazioni non richieste per intasare una rete). 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. Un **malware** è un software malevolo. Una **backdoor** è una porta di servizio, spesso installata volontariamente per avere accesso secondario. Un **trojan horse** è un programma apparentemente innocuo che nasconde software malevolo. Un **virus** infetta processi e si riproduce. Un **multiple-threat malware** infetta vari files, un **blended attack** usa vari metodi di infezione e trasmissione per potersi diffondere più velocemente e gravemente. Un **rootkit** è un programma malevolo che fa ottenere permessi di amministratore nel sistema, può essere persistente se sopravvive a un reboot o memory based altrimenti, può essere in user mode e quindi sfruttare le attività dell'utente modificandole o in kernel mode.

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, 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). Una tecnica consiste nell'utilizzo di **canarini**: si inserisce una porzione di memoria nello stack fra il return address e le variabili locali e prima di proseguire dove indicato dal return address si controlla che il contenuto del canarino non sia stato modificato. Altrimenti potrei dividere le sezioni di memoria in cui è permesso scrivere da quelle in cui è permesso leggere (WOX) o dividere quelle di codice eseguibile da quelle modificabili (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 vari 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, modificare chat e manipolare giochi.

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. Può riguardare banda di rete, risorse di sistema o risorse di applicazioni. Un classico esempio è il flooding di ping, ovvero l'invio di dati da una rete che ha più capacità ad una con meno capacità per intasarla e causare perdita di traffico. Si rischia che il destinatario dell'attacco possa risalire alla sorgente da cui è partito. Altrimenti si può fare ICMP flood, UDP flood o TCP SYN flood (mando dei SYN, 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 providers 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.

Tecniche di protezione:

- **Crittografia**: è un elemento molto importante e necessario ma non sufficiente. Alla base della crittografia abbiamo numeri casuali, che nei computer sono in realtà semi-casuali. Crittografia **simmetrica**: si genera una chiave specifica per una comunicazione con cui si cifra e decifra il messaggio tramite un algoritmo. Un attacco basato sulla brute force non è realizzabile perché le chiavi sono di una lunghezza tale che si necessiterebbe di anni per trovare quella giusta. 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. Crittografia **asimmetrica**: chi vuole ricevere dei dati genera due chiavi, una pubblica e una privata. Con quella pubblica i dati vengono cifrati e solo io che ho quella privata posso decifrarli. 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. **MAC** – Message Authentication Code: si utilizza per autenticare il messaggio. Data una chiave calcolo il MAC del messaggio e lo aggiungo in coda. Dopo la

trasmissione il ricevente può ricalcolare il MAC del messaggio ricevuto e controllare che sia uguale al MAC calcolato in partenza. Si utilizzano funzioni di hash che permettono di avere come output un numero di lunghezza fissa indipendentemente dal messaggio in input.

- **Autenticazione**: ho vari metodi. **Password utente**: quando inserisco una password viene calcolato e salvato il suo **hash** in modo tale che sia impossibile risalire alla password. Per evitare che utilizzando la stessa password si abbia lo stesso hash viene anche inserito un **sale** che fa una prima codifica della password, questo sale viene poi salvato insieme all'hash finale della password ed è utilizzato per i successivi tentativi di autenticazione. **Token**: un token non ha potenza di calcolo ma contiene informazioni per identificarci, una smart card ha anche della potenza di calcolo. **Biometrie**: 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.
- **Controllo di accesso**: assegna il tipo di accesso in base alle circostanze o all'utente che lo richiede. Le politiche di access control sono di tre tipi: **DAC** – discretionary access control (ho il controllo in base alla mia identità e posso garantire il controllo anche ad altre identità), **MAC** – mandatory access control (ho il controllo in base alla mia identità) o **RBAC** – role-based access control (ho il controllo in base al mio ruolo nel sistema, posso avere più ruoli).
- **Antivirus**: 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. 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.
- **Firewall**: si occupa di bloccare pacchetti non legittimi provenienti dalla rete. Può lavorare a livello di trasporto bloccando alcune porte, a livello di rete bloccando alcuni indirizzi IP o a livello di applicazione. La gestione può prevedere il blocco di tutti i pacchetti eccetto quelli esplicitamente consentiti o alternativamente il blocco dei soli pacchetti specificati. I firewall sono generalmente numerosi in una rete, se ne inserisce uno all'ingresso dalla rete esterna, uno su ogni macchina e altri eventuali fra gli switch.
- **Intrusion detection**. Potrei monitorare il traffico di ogni host per creare delle statistiche e riconoscere immediatamente un livello anormale di traffico dovuto ad un'intrusione. 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. Posso inserire honeypots anche all'interno della mia rete per controllare quali attacchi sono riusciti a superare il firewall.

La sicurezza può essere garantita solo da codice di buona qualità -> security by design.