SISTEMI DI CALCOLO MODULO 2

professore: Roberto Baldoni

(copia appunti precedenti)

User-Level Threads
Le applicazioni si occupano della gestione dei thread, per cui non vi è l'intervento del kernel, si cerca di organizzare uno scheduling specifico per l'applicazione in base alle esigenze di questa.
NB il kernel non è assolutamente a conoscenza dei thread.

Kernel-Level Threads
il processo è gestito dal livello kernel che inoltre gestisce i thread, lo scheduling è comunque eseguito dal kernel.

la differenza dei due approcci consiste nell'architettura tra questi: l'applicazione nell'user-level tende sepre a massimizzare il parallelismo logico tra i componenti mentre non sapend delle possibili capacita del so (multicore o multiprocessore o singolo processore) l'applicazione potrebbe non sfruttare tutta la potenza computazionale del computer.
nel kernel-leve invece, poichè questo gestisce direttamente lo scheduling, viene invertito il ragionamento.

Vantaggi:
a livello utente: scheduling indipendente dal kernel e dipende dall'applicazione che conosce bene cosa fanno i thread, parallelismo logico più efficace. le ULT tuttavia non prendono vantaggio dalle architetture multiprocessore/multicore.

a livello kernel: il kernel può direttamente schedulare i thread sfruttando al massimo l'architettura della macchina, allo stesso tempo però è più complesso sfruttare la conoscenza dell'applicazione.
se un ULT si blocca, si bloccano tutti gli altri, nel KTL se uno viene bloccato, gli altri threadm essendo fisicamente disposti su altri core/processori, continuano la loro esecuzione.
gli svantaggi del klt sono nel trasferimento di controllo tra i thread, poichè è sempre richiesto una switch mode da parte del kernel.

Approcci combinati
introdotti con il sistema solaris, con un approccio best-afford (prendo il meglio da ogni aspetto).

categorie di sistemi di calcolo
single instruction single data (sequenze di dati trasmessi a un set di processori, ognuno esegue istruzioni differenti)
single instruction multiple data (un set di processori esegue simultaneamente differenti istruzioni di differenti set di dati)
multiple instruction single data
multiple instruction multiple data

considerazioni su architetture multiprocessore
si ha simultanea o concorrente esecuzione di processi e thread
lo scheduling è fatto a livello di processore, quindi se due processi sono su due processori o core differenti possono viaggiare in parallelo perchè schedulate insieme
sincronizzazione: poichè tutto ciò che è convolto è variabile globale.
l'applicazione che gira quindi è un'applicazione globale, che deve sincronizzarsi.

Microkernel
si ricorda il kernel di un sistema operativo un programma che fa funzionare le applicazioni soprastanti (office, skype ecc), permettendo loro di lavorare a un livello di astrazione più elevato. è un «passatore» di messaggi, che fa da tramite tra applicazioni e memoria virtuale, permettendo all'applicazione di lavorare sulla memoria richiesta.

Processi e scheduler
i thread presentano il modo di parallelizzare i processi in modo leggero, possono a tal proposito essere visti come dei processi molto leggeri residenti nella sfera del processo stesso, offrendo un grande vantaggio prestazionale, nonché tuttavia svantaggi riguardo soprattutto l'accesso alle variabili.
L' elemento all'interno del os che prende e manda in esecuzione un processo piuttosto che un altro in un ambiente multiprocessing è chiamato Scheduler (in un ambiente monoprocessore manda in esecuzione UN processo, intercambiato con altri e messo in coda secondo politiche interne ad esempio lo scheduling round-robin); lo scheduler crea un interlining tra le nostre istruzioni, dal punto di vista temporale, le istruzioni che vengono eseguite dipendono dal singolo processo che in quel momento è in esecuzione.

Esempio:
processi p1, p2, p3

p1:	p2:	p3:
ist 1,1	ist 1,2	ist 1,3
ist 2,1	ist 2,2	ist 2,3
ist 3,1	ist 3,2	ist 3,3

noi sappiamo che ist 1,1 verrà eseguita necessariamente prima di ist 2,1 (esecuzione temporale locale conservate) ma NON sappiamo quando ist 1,1 verrà eseguita rispetto a ist 2,2, poiché dipende dalle politiche dello scheduler.

Tra le innumerevoli tipologie di scheduler ne troviamo un gruppo particolare che chiamiamo scheduler fair, che si distinguono per una politica di questo tipo: lo scheduler fair prima o poi da l'accesso alla cpu a tutti i processi, quindi ogni processo può evolvere, essendo garantita la sua esecuzione, secondo le politiche dello scheduler (fair) corrente.

Condivisione delle risorse tra processi
Un processo è composto da un codice sorgente (possibilmente condiviso), un certo numero di dati (variabili, parametri...) e un numero di attrbuti che descrivono lo stato del processo.
I processi possono condividere lo stesso codice, ma NON la locazione di memoria dei processi stessi, evitando di generare interferenze indesiderate.
Lo stato del processo è desctitto da diversi elementi:

- identificatore
- stato
- priorità
- program counter
- puntatori a memoria
- context data
- informazione su stato i/o
- informazioni sull'account

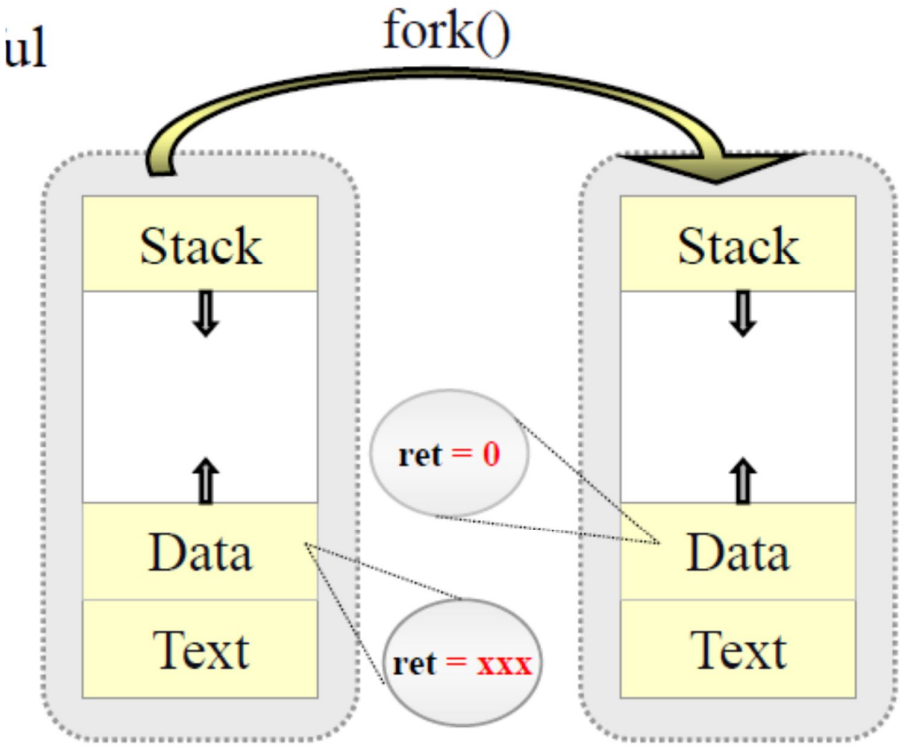
questi elementi sono tutti interamente contenuti nel PROCESSO CONTROL BLOCK, interamente

creato e gestito dal sistema operativo.

Creazione di nuovi processi
Per creare un nuovo processo è necessario fare una chiamata a sistema.
Ma cosa significa dentro il so fare una system call? Per rispondere a questa domanda prendiamo come punto di riferimento una system call all'interno di un sistema unix:
in Unix la creazione e il comportamento di un processo è determinato da 3 funzioni, fork(), wait() e exit() (rispettivamente la prima è coinvolta nella creazione del processo, la seconda ne modifica lo stato passando da running a waiting e la terza segnala il termine dell'operazione e procede con l'uccisione del processo)

NB in altri sistemi unix-like fork è sostituito da funzioni simili ma del tutto analoghe.

Meccanismo di underlining
attraverso la fork() (inserita nel main) di un processo padre viene creato un processo figlio che lavora concorrentemente con il padre: viene quindi, per quanto detto prima, effettutato un interlining tra le operazioni del processo padre e le operazioni processo figlio, che, per come lavora la fork, non è altro che un duplicato del processo da cui è stato creato. Che significa?



processo padre processo figlio

NB una volta che nascono due processi, essi sono due entità completamente separate, per cui avranno dati che possono evolvere in modo diverso (magari hanno inizialmente stesse variabili ma il figlio evolve diversamente da padre). Ogni volta che si fa una fork() si alloca memoria che è comunque limitata!!

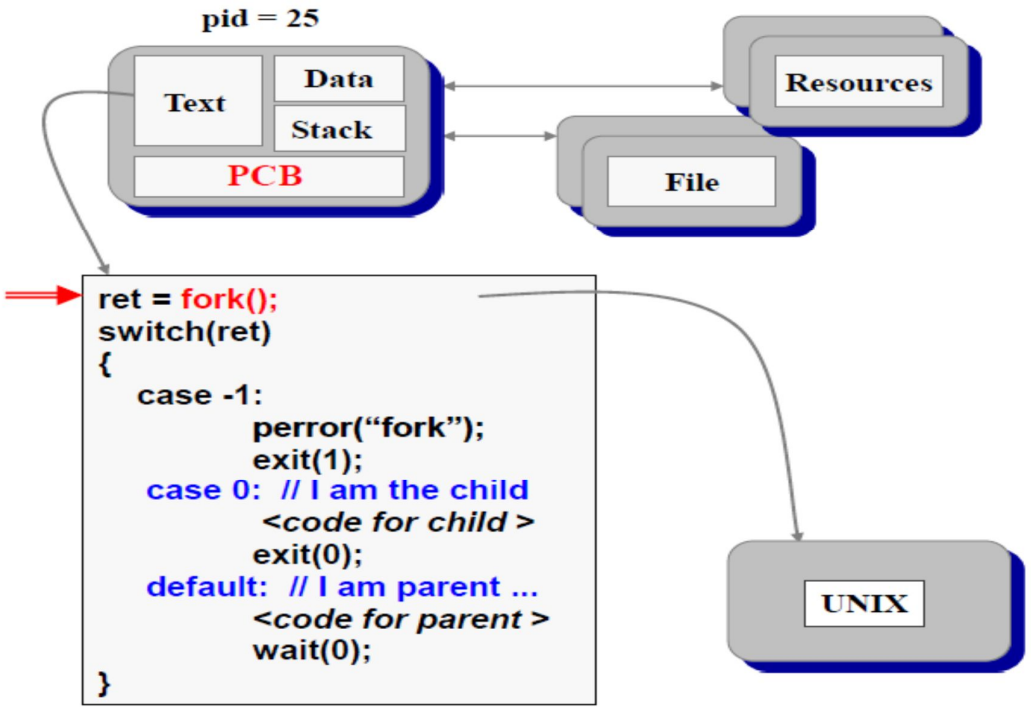
Dalla figura si nota che la fork ha due (tre, se si considera l'insuccesso) ritorni: ritorna 0 lato figlio e ritorna l'identificatore del figlio (xxx) lato padre (e ritorna -1 se l'operazione non ha avuto successo, ad esempio per insufficienza di spazio libero);

Esempio di invocazione fork
La fork non è altro che un'istruzione contenuta all'interno del mail del processo padre:
"processo padre"

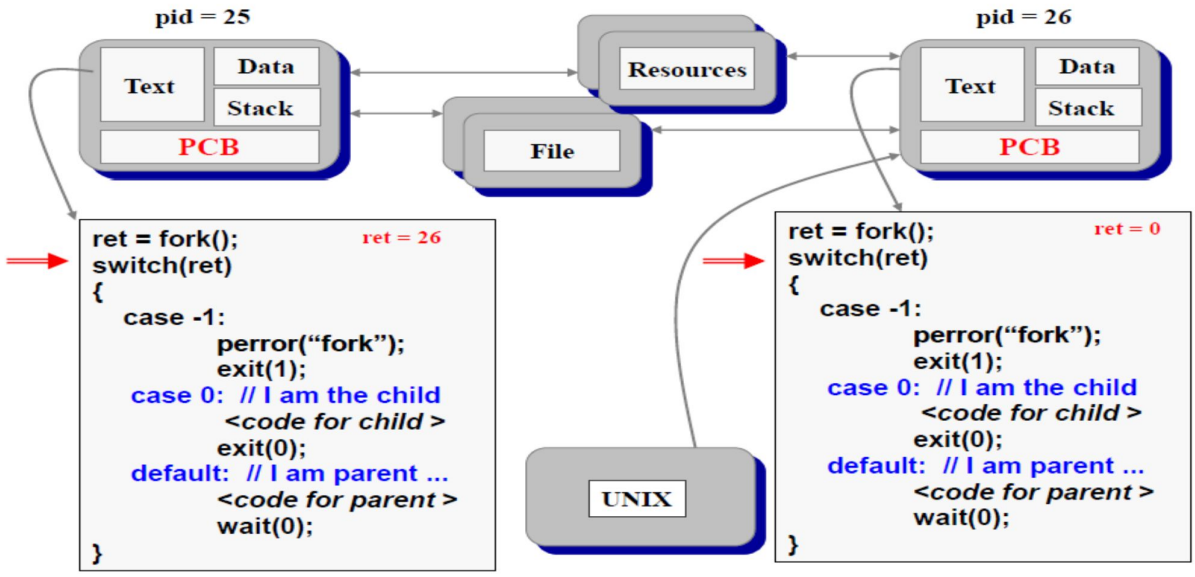
```
int main() {
        ....
        ....
        procFiglio = fork();
        ....
}
```

attraverso la fork quindi a partire da un solo processo si arriverà ad ottenere un process tree (un processo genera un processo che genera un altro processo....

Esempio di lavoro di una fork
Inizialmente si ha il processo padre:



il risultato di questo schema è:



Funzione wait()
un processo padre può aspettare che il processo figlio "muoia", perché può aspettare il risultato restituito dal processo figlio che permetterà poi al padre di riprendere l'esecuzione. L'attesa di un processo si identifica ovviamente con lo stato waiting, che viene impostato tramite la funzione wait().

Wait-exit, uccisioni e processi zombi
la wait() - exit() è una sorta di sincronizzazione tra padre e figlio (padre entra in wait prima del figlio, il che permette allo scheduler di ottenere questo risultato: il padre si mette in attesa che il figlio esegua un'operazione il cui risultato servirà al padre per riprendere la sua esecuzione);
Le cose funzionano meno bene nel caso lo scheduler metta in wait il figlio prima del padre: in questo caso il figlio diventa uno "zombi".
Quindi, se un processo senza figli muore, si perdono le sue tracce poichè il sistema operativo provvede alla cancellazione di ogni dato inutilizzato
se un processo ha figli invece, può aspettare la morte dei figli (attraverso la exit()). Se lo scheduling è fair e la exit del figlio arriva prima della wait del padre, il sistema vede padre attivo e figlio morto,di conseguenza non elimina i dati del figlio ma a quel punto fa passare il processo nello stato zombie: quando tutti i figli sono zombie, il processo padre viene sbloccato.
Alla morte di un padre che ha terminato il lavoro dopo la resurrezione dei figli, il so cancella definitivamente le tracce di padre e figli.

Funzione exit()
la memoria di un processo viene cancellata e deallocata, tornando quindi nuovamente disponibile.
NB il processo padre può anche morire prima del figlio, sebbene significhi che questo non ha fatto l'operazione di wait(). ciò equivale a dire che se il padre è vivo e il figlio è morto, il padre ha necessariamente subito una variazione di stato da running a waiting nel tempo del quale il figlio è vissuto e ha lavorato.

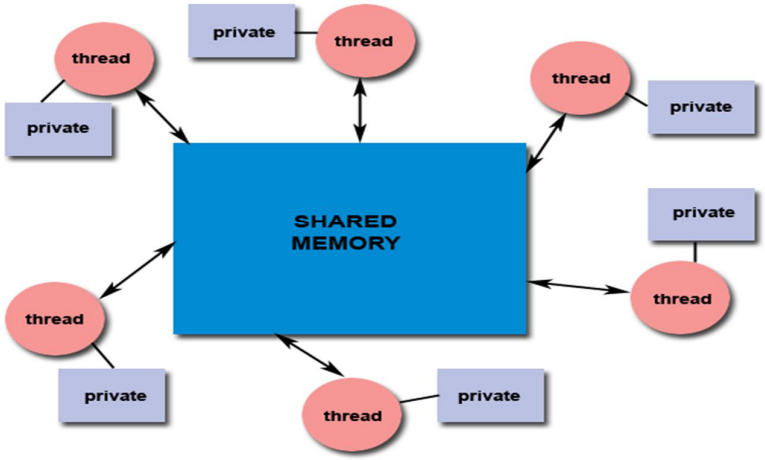
Tipi di applicazioni:
- stateless
-statefull

Pthreads
Con questo nome si identificano i POSIX THREADS, programmati nel linguaggio C e definiti nel file header <pthread.h>, incluso nell'intestazione del programma in cui devono essere utilizzati. per ottenere vantaggio da pthreads, un programma deve essere capace di essere organizzato in piccoli task che fanno delle cose possibilmente indipendenti da altre, in modo tale da poter essere messi a lavorare in parallelo.

Modelli per programmi con thread

- Manager/worker: un manager assegna lavoro agli altri thread worker; il manager prende l'input e cattura il risultato degli altri task (Client/Server)
- Pipeline: il task si divide in una serie di suboperazioni concorrenti (P2P)

- Modello a memoria condivisa
  i thread nascono da un processo, hanno di fatto due tipi di memoria, una privata e una condivisa, molto piccola affinché il discorso del thread abbia un senso contenuto, che contiene le variabili del processo padre; questa memoria corrisponde di fatto alla memoria dove risiedono tutti i dati del processo e i vari thread lavorano su tali variabili create dal processo stesso.



[modello a memoria condivisa]

Sicurezza thread
Per garantire un comportamento sicuro dei thread di un processo, si fa riferimento a una proprietà dell'applicazione denominata thread-safeness, indicante la capacità di questa di eseguire più thread simultaneamente senza generare errori nella memoria condivisa o creare condizioni di "gara" tra

thread.
Per meglio specificare quanto detto, cosnideriamo un'applicazione che crea numerosi thread, ognuno dei quali effettua una chiamata alla stessa libreria; Questa libreria accede a o modifica una struttura globale o una locazione in memoria; Quando ogni thread chiama questa routine è possibile che questi provino a modificare l'elemento nello stesso tempo: l'unico modo possibile per prevenire la corruzione dei dati, e quindi per ottenere thread-safeness, è impiegare un meccanismo di sincronizzazione dei thread.

Utilizzo dei Pthread
NB è importante ricordare che il metodo main() comprende un singolo thread preimpostato.
Per creare un nuovo thread e renderlo eseguibile è necessario invocare il metodo pthread_create(), ovviamente il numero massimo di thread creabili dipende dall'impliementazione corrente del processo; Una volta creati, i thread sono tra loro uguali e distinti, e possono creare a loro volta altri thread.
Per terminare un thread abbiamo invece diversi modi:

- Il thread ha completato il suo lavoro, esegue il return e "muore" da solo
- Chiamo il metodo pthread_exit()
- Invoco il metodo pthread_cancel()
- Chiamo il metodo exit()

Concentriamoci su **pthread_exit()**:
pthread_exit viene chiamato dopo che un thread ha completato il suo lavoro e non ha più ragione di esistere. Se il programma padre finisce prima che lo facciano i figli, se esiste un metodo pthread_exit() essi continuerano a svolgere il loro lavoro fino all'invocazione di questo.
E' inoltre importante sapere che questo metodo non chiude i file, di conseguenza ogni file aperto all'interno del thread resta per default aperto, ne occorre pertanto una chiusura "manuale".

Concorrenza: mutua esclusione e sincronizzazione

Premessa

é qui riportata una tabella con definizioni utili alla comprensione del capitolo.

La concorrenza ha a che fare con in modo in cui i processi e i thread lavorano su piattaforme che permettono l'esecuzione di più processi/thread contemporaneamente.

La concorrenza si presenta in tre differenti contesti:

- Applicazioni multiple: permette la condivisione del tempo di elaborazione tra applicazioni attive (prima gira un'applicazione, poi un'altra, secondo le regole di scheduling)
- Applicazioni strutturate: estensione della struttura modulare del programma
- Struttura del sistema operativo: i sistemi operativi stessi implementano un numero proprio di processi e thread per il corretto funzionamento di questo

IMPORTANTE: l'output del processo deve essere indipendente dalla velocità di esecuzione di altri processi concorrenti.

Principi e proprietà della concorrenza
La concorrenza, come già detto, ha a che fare con l'esecuzione contemporanea di più processi/thread che lavorano allo stesso tempo.
Quando si ha a che fare con questa, la velocità relativa di esecuzione di un processo non può essere predetta a priori, in quanto dipende da fattori esterni alla natura del processo: fattori influenzanti sono quindi le attività di altri processi (esempio: un processo che vuole accedere a una locazione di memoria non può farlo perchè in quel momento un altro processo sta lavorando su tale spazio); il modo in cui il sistema operativo invoca degli interrupt(); le stesse politiche di scheduling del sistema.

Problematiche della concorrenza
La concorrenza, per sua definizione, porta con sé delle problematiche su cui è opportuno soffermarsi a fini preventivi: Attraverso la concorrenza possono manifestarsi problemi sulla gestione di risorse condivise; il sistema operativo inoltre può trovare difficoltà nel gestire ottimamente le allocazioni di risorse, soprattutto nel caso in cui ha a che fare in thread del tipo user-

level; Altri problemi possono scaturire da vari controlli sugli errori dei programmi.

Race condition: i thread gareggiano
Quando due o più thread/processi accedono a una locazione di memoria per leggere o scrivere dati, questi iniziano a concorrere: tutti i processi/thread interverranno su tali dati eseguendo le loro azioni, finchè l'ultimo, il "perdente" non aggiornerà con il suo intervento il dato considerato, determinandone il valore finale.

Concorrenza e sistema operativo
Con l'avvento della concorrenza sono cambiate anche le caratteristiche di gestione dei programmi di un sistema operativo, che adesso ha a che fare con l'esecuzione contemporanea di più processi e servizi. Su questa base il sistema operativo deve attenersi a quattro compiti:

- Il sistema operativo deve essere capace di tenera traccia di più processi (altrimenti ovviamente non sarebbe possibile la concorrenza)
- Deve allocare e deallocare risorse per ogni processo attivo
- Deve proteggere i dati e le risorse fisiche di ogni processo dall'interferenza degli altri
- Deve garantire che i processi e gli output siano indipendenti dalla velocità di elaborazione

Qui riportiamo ora una tabella di riferimento per le interazioni tra i processi:

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Analizziamo ora più nel dettaglio queste tre relazioni tra processi.

Competizione
La competizione tra processi è una particolare situazione di conflitto che si genera quando tali processi vogliono utilizzare le stesse risorse (memoria, dispostivi di I/O....): inizia tra questi processi quindi una sorta di "battaglia" per ottenere il possesso delle risorse richieste.
In caso di competizione possono essere riscontrati problemi di deadlock o starvation: per evitare tali imprevisti si fa uso della mutua esclusione.

La mutua esclusione fa si che un processo che si arresta lo possa fare senza interferire con altri processi, evitando quindi le due sopra citate situazioni pericolose; A tale processo non viene inoltre negato l'accesso alla sezione critica quando non vi sono altri processi che la stanno usando, e vi rimane dentro solo per un tempo finito.

Da un punto di vista hardware, la mutua esclusione può essere garantita nei sistemi uniprocessore tramite disabilitazione degli interrupt(), portanto però con sé svantaggi di carattere prestazionale (l'efficienza di esecuzione può notevolmente peggiorare) e di carattere portabile (questo approccio non funzionerà in un'architettura multiprocessore).

Istruzioni Macchina Speciali
Per gestire fenomeni come la mutua esclusione si fa uno di istruzioni macchina speciali: queste sono caratterizzate dal fatto che oltre alla loro semplicità di verifica e la loro portabilità (sono applicabili a un qualunque numero di processi sia che si trovino su un singolo processore o un sistema multiprocessore con condivisione della memoria principale) sono utilizzabili per supportare sezioni critiche multiple; ogni sezione critica può essere definita da una propria variabile.
Queste istruzioni non sono comunque esenti da svantaggi: è infatti impiegato il busy-waiting ai processi, quindi quando un processo è nello stato wait in attesa di accedere a una sezione critica questo continuerà a consumare il processor time; é inoltre possibile la starvation, più precisamente è possibile quando un processo lascia la sezione critica e più di un processo si trova nello stato wait. Allo stesso modo è possibile il deadlock.

Semafori
Un semaforo è un meccanismo di concorrenza. Possiamo definire un semaforo come una variabile intera utilizzata per segnalazioni tra processi. Un semaforo può eseguire soltanto tre operazioni atomiche, ovvero l'inizializzazione (a un intero non negativo) e l'incremento (semSignal) e decremento (semWait) di se stesso. Il decremento di un semaforo corrisponde al blocco di un processo, intuitivamente quindi l'incremento del semaforo corrisponde allo sblocco. Non c'è modo di ispezionare o maneggiare un semaforo oltre a queste operazioni, portanto a queste conseguenze:

- Non è possibile sapere prima che un processo decrementi un semaforo se questo lo bloccherà o no
- Non c'è modo di sapere quale processo continuerà immediatamente su un sistema uniprocessore quando due processi stanno lavorando concorrentemente
- Non è possibile sapere se un altro processo è in waiting, quindi il numero di processi sbloccati può essere zero o uno.

Un semaforo è implementato come una struct contenente due campi: il primo è una variabile intera non negativa (o un enum tra zero e uno nel caso di semaforo binario), il valore vero e proprio del semaforo; il secondo contiene una coda di tipo arbitrario, che viene riempita da un processo con la semwait e svuotata con la semsignal. Una coda è usata per bloccare un processo nello stato wait con il semaforo. Possiamo identificare due tipi di semafori: gli strong semaphores fanno in modo che il processo che è stato bloccato da più tempo sia rilasciato dalla coda per primo; i weak semaphores non hanno un ordine specificato che indichi come i processi vengano rimossi dalla coda.

Producer/Consumer
La producer/consumer è una configurazione nella quale uno o più processi producer generano dati e li mettono in un buffer; quindi un singolo processo consumer prende e porta fuori, "consumandoli", un elemento alla volta dal buffer. Questa configurazione prevede che ai dati possa accedere un singolo processo alla volta, indipendentemente dal fatto che sia un producer o un consumer. Il problema che ci poniamo è il seguente, quello di assicurare che il producer non possa aggiungere dati in un buffer pieno e che il consumer non possa rimuovere dati da un buffer vuoto. Per la soluzione si vedano le slide.

Buffer nel caso producer-consumer
(riferimento a figura 5.12)
in questo caso avremo due puntatori: un puntatore in del producer che inserisce dati in una zona disponibile del buffer e un puntatore out che consuma i dati in una zona piena. Quando in raggiunge out (ovvero quando in o out puntano alla stessa locazione di memoria del buffer) lo spazio è pieno e non è possibile aggiungere elementi, ovviamente non esiste un buffer reale infinito. Per non permettere ai due puntatori di arrivare a puntare una zona di memoria non valida o comunque esterna al buffer , quando si arriva all'ultimo byte della zona di buffer si torna all' inizio; Tramite la circolarità è possibile continuare il processo.

Monitor
La sincronizzazione e la concorrenza sono problemi che hanno side effect, come visto fino ad ora. I semafori sono uno strumento di gestione della concorrenza e permettono di effettuare sincronizzazione tra i processi.
Esistono tuttavia altri strumenti atti a questo scopo, ad esempio in java, come il costrutto di monitor. Mentre il semaforo ha una interazione con lo scheduler del sistema operativo, con i monitor questo non avviene, poiché siamo a un livello un po' più elevato.

Un monitor è una sorta di struttura con all'interno diverse procedure (da 1 a k), utilizzabili dai processi che per poter entrare e usarne una si mettono in coda.
I monitor sono caratterizzati da variabili condizione, accessibili solo dalle procedure del monitor e non da procedure esterne, che possono sembrare simili ai semafori, chiamate cWait e cSignal: queste due system call non hanno nessuna interazione (diretta) con il sistema operativo (come invece hanno i semafori). In questo caso nel momento in cui ad esempio un buffer è vuoto un consumer si ferma
su una delle code.
Fondamentalmente anche qui abbiamo a che fare con semafori, che sono però più a livello sw, che hanno solo un'interazione indiretta con il sistema operativo:

- Mentre semWait decrementa il valore del semaforo, cwait(c) sospende l'esecuzione del processo chiamato dal parametro c
- Mentre semSignal incrementa il valore del semaforo, csignal(c) riprende l'esecuzione di alcuni processi bloccati precedentemente da una cwait

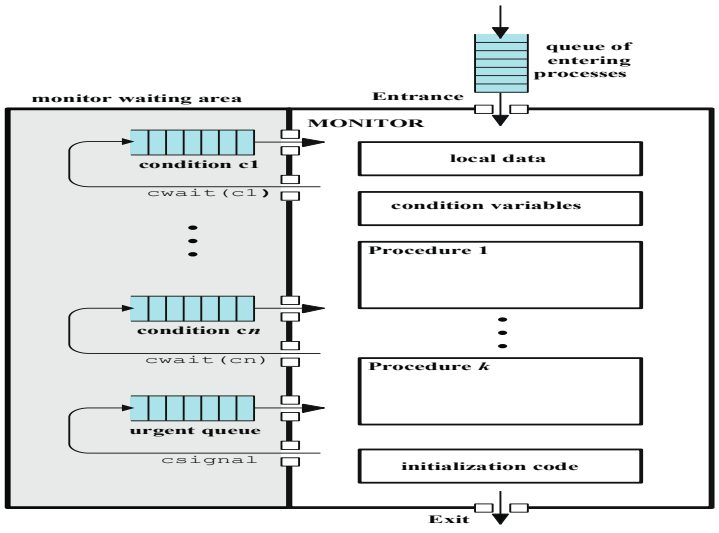


Figure 5.15 Structure of a Monitor

il monitor ha una gestione della sua coda indipendente dall'elemento stesso, quando invece vi era un legame indispensabile a fini di sincronizzazione per i semafori: non vi è dunque un legame con l'elemento, assenza di legame che mostra un distaccamento dagli ingranaggi del sistema operativo. Lavorando su un livello di coda di più alto profilo rispetto a quella del semaforo, viene meno il problema nella figura 5.9
Quindi quando il buffer è vuoto il processo si addormenta sul cwait(notempty), non appena arriva qualcuno che lo sveglia, prendo il primo processo addoermentato e lo porto in esecuzione. (riferimento a figura 5.16)

NB Qui tuttavia, dato questo comportamento e il livello di astrazione diventa più alto, perdiamo in termini di concorrenza reale (conseguenza normale quando si programma a livello più alto).

Message Passing
Quando un processo interagisce con un altro processo devono essere soddisfatte due condizioni: sincronizzazione (che forza la mutua esclusione) e comunicazione (per scambiare informazioni).
Il message passing è un approccio che provvede a entrambe queste due funzioni, inoltre lavora anche su sistemi distribuiti.
Il message passing fa uso di due funzioni primitive, la send(destination, message) e la receive(source, message), che soddisfano queste regole:

- Un processo invia informazioni nella forma di un messaggio a un altro processo designato da una destinazione
- Un processo riceve informazioni dall'esecuzione della primitiva receive, che indica la sorgente e il messaggio

Nonblocking e blocking
Le funzioni primitive ora viste possono essere di tipo blocking o unblocking: nel caso ad esempio di una nonblocking send, una volta inviato il messaggio l'esecuzione del processo continua normalmente.

Addressing
Gli schemi per i processi send e receiver ricadono in due categorie: indirizzamento diretto (implicito e/o esplicito) e indirizzamento indiretto.
Quest'ultimo viene normalmente sviluppato attraverso mailbox: i messaggi vengono inviati a mailbox che successivamente vengono lette da più entità che possono anche scrivere messaggi (poi inseriti nella mailbox stessa).

Problema del buffer limitato produttore-consumatore
all'inizio vengono create due mailbox: una mayproduce e una mayconsume. Viene poi creato un ciclo for che va da 1 alla capacità, tramite i quali alla mayproduce sono inviati messaggi null, che non sono altro che tipi di token che i producer hanno a disposizione per scrivere, finché il buffer non sarà pieno. Il producer riceve su mayproduce, cercando l'esistenza di uno di questi token, dove all'interno ci sono tanti messaggi quanto è la grandezza del buffer. Successivamente produce il dato, che viene mandato sulla mailbox mayconsume. Quindi dall'avere n elementi dentro mayproduce abbiamo un elemento che è stato prodotto dentro mayconsume. Chiaramente il producer prende un elemento al la volta, e via via mayproduce si depaupera di elementi che adesso si troveranno in mayconsume.
Ogni volta che il producer invia un messaggio consuma un token, fino allo svuotameto della casella di posta, dove si blocca perché non trova più token disponibili e la receive diventa blocking.

Per quanto riguarda la consumer invece, anche questa entra in un ciclo, while(true), andando a guardare la mailbox mayconsume (che poi va a ricaricare la mailbox mayproduce), quindi se il buffer è vuoto il producer si blocca, dopodiché consuma il primo dato e viene trasmesso verso il sender.
Molte delle architetture contemporanee sono basate su scambio di messaggi; sono sistemi per risolvere problemi che stanno sempre più prendendo piede rispetto ad altri meccanismi come semafori e monitor.

Problemi dei readers/writers

un'area dati condivisa tra molti processi, alcuni che leggono e alcuni che scrivono. Devono essere soddisfatte alcune condizioni: ogni numero di reader può simultaneamente leggere il file; solo un writer alla volta possono scrivere il file; se però un writer sta scrivendo un file, nessun reader può leggerlo. All'interno di computer o applicazioni di tipo commerciale si fanno tantissime read e pochissime write.

Sistemi operativi e rete

quando si è progettata l'architettura di internet abbiamo un backbone, e dei punti di rete locale a cui è possibile accedere
fino agli anni 70 non esisteva solo l'architettura internet (spesso alternative proprietarie), sebbene internet sia prevalsa.
Le architetture che coesistevano errano difficilmente compatibili: era necessario conoscere tutta la serie di indirizzi atti alla comunicazione

successo di internet
Internet non aveva dietro a se un vendor, per tale motivo ricevette una spinta iniziale molto forte, accompagnate dallo sviluppo di architetture delle API di internet open-ource. La grande capacità di internet è quella di potersi “autorigenerare”, per mezzo di algoritmi come quelli di routing, che conferiscono flessibilità al comportamento della rete anche in caso di imprevisti di portata considerevole, inoltre la sua dimensione è tale che qualunque operatore per quanto grande sia, se decidesse di staccarsi dalla rete, non ci saranno ripercussioni.
Con il passare degli anni i governi di molti paesi a fini tutelari e anti-spionaggio hanno iniziato ad apportare una serie di regole per stabilire se l'entità può essere connesso alla rete.

Per scambiare informazioni con un server presente in un'altra rete, i pacchetti passeranno prima nella rete dove ci si trova, i dati verranno scambiati con un altro operatore, in una staffetta con destinazione l'operatore del sito da visiare.

Internet excancge point: è una stanza in cui i vari operatori interessati portano i propri router (a pagamento).

Middle-access point
in italia si trovano a roma e a milano, successivamente se ne sono aggiunti altri: più middle-access point significa maggior velocità di comunicazione: ovviamente si tratta di un sistema nel quale bisogna tener conto del giusto posizionamento dei punti di accesso per evitare di generare considerevoli bottleneck.

Google
google ha iniziato a mettere servizi su internet, gestiti attraverso datacenter interconnessi (si predilige il nord per il raffreddamento, sfruttando anche corsi d'acqua fredda e impianti per aumentare le performance dei propri server). Google è in grado di spostare tutto il traffico di gmail in soli 30 minuti.

Google ha poi un'altra rete, chiamata user network, composta da NAP in modo capillare, che permettono collegamenti con i vari operatori che poi forniranno la rete all'utente finale.

Il protocollo di internet è agnostico alla tecnologia. Ciò che ha è una network interface, che ancorano internet alla tecnologia attraverso poche funzioni chhe permettono ad esempio lo scambio tra indirizzo di rete e indirizzo IP.

Protocol stack
è una rappresentaione di come si impilano i protocolli della macchina data. I primi due livelli del livello OSI sono gli xDSL; il terzo è dato dai protocolli ip e tcp,il quarto invece è il protocollo del collegamento fisico, ad esempio ethernet.

Ognuno di questi livelli compie tre azioni: inddirizzamento, routing e frammentazione (e/o riassemblaggio).
Un esempio di sistema relativo a quanto detto può essere un sistema p2p come torrent.

Sockets
I sockets sono interfacce che si piazzano tra applicazione e il kernel del sistema operativo, presentandosi quindi come strumento di dialogo tra queste due entità.
In un sistema distribuito il socket ha il compito di dialogare tra processi che non si trovano nella stessa macchina (ovviamente si può anche dialogare tra processi di una sola); E' inoltre utile per utilizzare il protocollo tcp/ip.

Un socket può essere immaginato da un punto di vista figurativo come una “scatola nera”, dal punto di vista del so può essre visto come un file: di fatto le modalità di utilizzo sono simili a quelle di gestione file comuni (lettura, scrittura, anche se la scrittura significa inviare un messaggio a un altro socket).
I socket diventano operativi quando è possibile scambiare informaazioni tra due socket, ad esempio quando un trasmettitore e un ricevitore vengono messi in modo da poter comunicare sulla base di protocolli tcp e udp (con connessione e senza connessione).

End point
un end point innanzitutto deve essere riconoscibile, e poiché due end point giacciono su due macchine differenti, è di rilevante importanza l'indirizzo host di questo.
Il motore tra i due end point comunicativi è determinato dai protocolli udp e tcp, protocolli implementati a livelli di rete: quando parliamo di questi protocolli dobbiamo fare un sallto in avanti a livello di astrazione.

NB nell'end to end la macchina ricevente può avere moltissime connessioni attive!!

Fondamentalmente quando si accende la macchina già di fatto partono tutta una serie di applicazioni che utilizzano la rete, dunque vengono utilizzati i socket.
L'host quando riceve l'informazione, deve indirizzare i pacchetti nella giusta connessione (dropbox, gmail, google...): c'è una fase in cui il traffico di pacchetti che arrivano in una macchina devono essere smistati nei socket corretti.
Per indirizzare i socket si utilizzano i port number: la coppia ip e port number identifica univocamente la connessione tra i socket. Il port number può essere visto come identificatore del socket, da un punto di vista della connessione.
(Il multiplexing/demultiplexing dei paccetti viene sempre fatto dai protocolli tcp/udp)

Porta
il concetto di porta è stato abbastanza elaborato nel tempo.
Quando si progettano applicazioni su rete, si usa un concetto di tipo client/server: il server compie delle operazioni e inviano la risposta al client (esempio: connessione http)
Come si identifica un server?
Ci sono dei servizi che hanno delle well-known port (porte che tutti conoscono), tutta una serie di servizi con indirizzi noti. (ad esempio ftp è la porta 21 e usa protocollo tcp)
Esiste un certo range di numeri dove è possibile assegnare well-known port (che diventano tali quando il servizio offerto è talmente riconosciuto e importante da poterselo permettere)
NB le well-known port hanno tutti numeri piuttosto bassi
0-1023: indirizzi di porta riservati, applicazioni che per essere usate necessitano i permessi di root (un utente comune non può usarle).
1024-5000: indirizzi che è possibile utilizzare anche dall'utenza comune: a differenza di well-known port che sono create e restano fisse fino allo spegnimento, queste porte sono effimere: queste porte possono essere date da programmi creati dall'utente.
Ci sono diversi programmi che possono essere lanciati da client e generano una porta nell'indirizzo 0-1023: questo perché il processo di autenticazione non funziona client/server (lo è solo all'inizio), perché il server diventa poi a sua volta il client dell'utente.

[inserisci schema porte]

socket come box
l'applicazione deve poter comunicare con un processo in un altro host: il socket ci permette di interagire col sistema operativo, che inizia a impacchettare i dati sottoforma di http in caso di connessione web, tcp/ip ecc. e, una volta finito, il so effettua una chiamata alla scheda di rete per avvisare che il pacchetto è pronto: il processore della scheda di rete prende quindi l'informazione e la trasferisce.

[inserisci immagine]

quando viene creato il socket, all'utente ritorna un intero che rappresenta l'identificatore del socket: da un punto vista unix tutti i file hanno un identificatore. Da un punto di vista programmativo la funzione di creazione di un socket restituisce un intero.

Quando si ha a che fare su un socket, da una parte ci sono le system call che permettono la loro gestione, queste vengono attivate attraverso un passaggio di parametri dove ad esempio deve essere definita una connessione con indirizzo port number ecc.
sockaddr_in è una delle strutture fondamentali, i cui campi possono essere riempiti sia da utente che da so (vedi struttura su slide).
sin_family permette di discriminare la rete su cui ci si trova, prima era dedicato a AF_INET.
NB prima di usare questa struttura VA AZZERATA usando bzero().

System call utilizzate
si dividono in funzioni che passano indirizzo da utente a so e viceversa.
Le prime sono bind, connect, sendto
Le seconde sono accept, recvfrom

[inserisci slide con schema verde]

socket() crea un socket. Quando si chiama un socket, si mette in input la famiglia, il tipo e il protocollo.

Bind() assegna un protocol address name a un socket.

Il port number può essere o well-known (quindi è un server di root che parte con il boot della macchina), se nel port number viene messo 0, il sistema operativo provvede a dare un port number effimero al programma. Bind() di solito restituisce 0, ma se vogliamo reinstallare un indirizzo già in uso si andrà in errore.

Listen() prende il sockfd, che sarebbe l'identificatore, e fa una coda di un numero backlog per le connessioni incomplete (storicamente si mette 5, raramente 15).

accept() è la funziona lato server dove il server si blocca in attesa di connessione, aspetta qualcuno che gli invii una connessione per effettuare una certa connessione: una volta arrivata si sblocca l'accept e normalmente crea o un figlio o un thread che gestisce la connessione richiesta.

Una volta che il server ha inviato si blocca sulla recv()

la close serve anche per strutturare in modo appropriato il lato server, perché non vogliamo che la connessione sia bloccata su un singolo client per ore
se usiamo queste strutture è importante gestire in modo adeguato gli identificatori.

UDP Client-server
quando faccio la connect dichiariamo il tipo di protocollo che ci interessa, in questo caso udp
in udp lato server le prime due operazioni sono identiche con l'eccezione che viene dichiarato il socket di tipo udp, a quel punto ricordando che è un protocollo senza connessione non abbiamo bisogno di accept e connect. Visto che non abbiamo connessione si trasmette direttamente, pur secondo modalità diverse: prima l'accept e la connect creavano il canale sfruttato da send e receive. Il questo caso usiamo la sendto(ip address, port number; nel lato client e nel lato server per ricomunicare col client) e receivefrom(nel lato server). Mentre nel lato client una close sul protocollo tcp “uccide” il server, poiché non abbiamo connessione perché siamo in udp questo problema è aggirato, infatti il server non viene distrutto ma torna bloccato.

Recvform riempie l'indirizzo da dove i pacchett provengono
[scrivi utilità delle altre funzioni]

il primo tipo di socket sul mercato è il BSDSocket, socket unix sviluppati a Berkley.
Su windows abbiamo il WinSock
[riporta tabella confronti]

nel caso di winsock abbiamo anche una nonblocking mode dove l'accept non blocca, ma sempliemente ritorna uno stato.

In bsd la funzione select(). Ci sono 50 funzioni in cui diversi client possono inviare informazioni. La select prende tutti gli identificatori dei socket e quando riceve informazioni la select ritorna l'identificatore del socket che s è sbloccato, aggirando il problema dello sblocco/blocco determinato dagli accept.

Implementazione dei socket in un sistema operativo
ci sono due modi per implementare i socket: process model e message buffer.
La cosa peggiore in un sistema operativo è prendere blocchi di memoria e copiarli da un'altra parte: questa operazione richiede un sacco di tempo. Altra cosa a cui prestare attenzione è il context

switching, presente ogni volta che c'è un cambio di processo, che si traduce anche in tempo perso dalla CPU. Occorre minimizzare questi due fattori.
[inserisci immagine implementazione socket]

[…]

network adaptors
tutto il sistema di una network adaptor è governato da un sottosistema di controllo della scheda chiamato SCO, questo controlla sia il microcodice che organizza il frame ethernet e nello stesso tempo ha tutta l'iterazione con la memoria e con la cpu.
Si necessita di una memoria interna che permette di accomodare una differenza di velocità tra l'andamento del computer e l'andamento della rete.
All'interno di un computer si ha quindi la cpu che tramite un clock di sistema organiza le attività, controllando i bus interni (memoria e rete). Quello che il n.a. deve fare è definire un protocollo di comunicazione con la cpu, percheè entrambe non possono contemporaneamente eseguire operazioni sulla memoria, questo perché porterebbe alla distruzione dell'informazione sul bus. La soluzione è analoga a quella dei semafori per i processi, rappresentato dal protocollo di comunicazione con la cpu. Dentro la cpu si ricorda la presenza di un control status register CSR, presente anche nella sco. Questo permette a cpu e sco della rete di dialogare tra loro perché lo status register della sco è foormato da bit che vengono settati dalla sco e letti da cpu (o viceversa), questo scrivi e leggi rappresenta la metodologia di sincronizzazione. Per convenzione la trasmissione è rappresentato dal settaggio dei bit, l'attivazione del microcodice prepara e trasmette l'informazione.
NB ogni bit ha uno specifico significato, questi bit vengono inoltre nel codice definiti come macro.

Controllo da parte dell'host
busy waiting: host si posiziona li, non si muove e continua a leggere il CSR (ovviamente quando si fanno cose del genere il computer deve essere specializzato per una certa attività, come ad esempio un computer che funge da router, ma non va bene per un computer consumer).
Interrupt: metodo privilegiato sui computer consumer, l'adaptor invia un interrupt all'host che invia un interrupt handler che va a leggere CSR per capire l'operazione da eseguire.

Traferimento dati da adaptor a memoria
data memory access
non c'è coinvolgimento cpu nello scambio dati e i frame vengono drettamente inviati alla memoria di lavoro: c'è un dispositivo implementato nel computer (presente nelle schede di rete) per cui il trasferimento da memoria da adattatore di rete a memoria di lavoro (destinazione) viene fatta senza coinvolgere la cpu del computer, che si sgancia dai bus mettendosi nello stato di halt (in termini di elettricità la cpu si stacca mettendosi in alta impedenza)
programmed i/o
qui la cpu non si mette in halt ma quando l'adapter avvisa di un frame pronto la cpu fa partire una routine che prende dalla memoria del network adaptor e la porta sulla memoria gestita dai socket: la cpu in questo caso è responsabile di tutto il traferimento dei dati.
La tecnica di trasferimento dati dipende dall'architettura della macchina
organizzazione zona di memoria dello scambio dati tra na e cpu
questa si dice BD, buffer descriptor list, dove vengono tipicamente preallocati 64 buffer (si tratta quindi di un vettore di puntatori a buffer) quindi il numero massimo di processi che possono lavorare su quella zona di memoria è 64 (perché può esserci un solo processo per buffer).
Poiché questa è una zona di memoria condivisa, viene regolata da un semaforo.
scatter read / gather write
“frame distinti sono allocati in buffer distinti, un frame puo’
essere allocato su piu’ buffer”

viaggio di un messaggio attraverso il so.
Il messaggio che si vuole inviare deve essere preso dal so e portato in una zona di BD, se non è già completamente occupata. Una volta portato il contenuto del messaggio dentro il bd quello che accate è che si iniziano a inserire gli header dei vari protocolli di rete TCP e IP.
Quando il messaggio è pronto all'interno della bd, la scheda di rete viene avvisata e a questa viene ordinato di mandare il messaggio sulla rete: questo avviene settando i bit del CSP della sco, cche quando vede il cambiamento dei bit sa cosa deve fare.
La sco dell'adaptor quindi prende il messaggio e lo manda sulla rete. Alla cpu una volta terminata la trasmissione viene notificato l'evento tramite il settaggio del bit di riferimento del CSR che scatena una interruzione.
L'interruzione fa partire una procedura dentro la cpu detta interrupt handler prende atto che la trasmissione è avvenuta, e procede con il reset dei bit e la liberazione delle opportune risorse (entra quindi nel BD e distrugge il messaggio, poiché questo è stato inviato).

Device driver
il decive driver è una collezione di routines di sistema che serve per ancorare il so all'hardware sottostante specifico dell'adaptor.

Middleware
l'eterogeneità che è data da due sistemi operativi viene data da uno strato intermedio tra applicazione e sistemi operativi inferiori: si maschera così tale eterogeneità tramite uno strato sw appropriato chiamato middleware.
Il middleware quindi serve a masceerare l'eterogeneità inferiore delle piattaforme.
Da un punto di vista astratto la divisione middleware-sw in realtà non è fissa, dal momento che il sw si evolve con le applicazioni.

Computazioni client/server
il server di solito è una macchina da prestazioni molto elevate rispetto ai computer client. Se si usa un server normalmente si usa per condivisione di elementi.
L'idea di base è di alzare al massimo la capacità computazionale di un server a cui si collegano client di potenza inferiore.

Struttura a più livelli: su un livello lavora l'applicazione, un livello sottostante in cui si fa comunicazione.
L'applicazione è strutturata in sei livelli, tuttavia se ne individuano tre, che possono risiedere sul server oppure via via sul client, in questo caso il client si appesantirà.

Remote Procedure Calls
l'applicazione giace su due host, una parte client e una parte server: l'applicazione quindi non si trova più sul singolo computer (si parla di un middleware).

Le remote procedure calls permettono ai programmi di comunicare con macchine anche di diversi vendor tramite semplici procedure con semantica call/return.
Il cliente manda la richiesta sul server, che diventa un messaggio, poi elaborato dal server che manda la risposta al client.
Nel periodo in cui la richiesta è mandata il client è bloccato. Rispetto a un ambiente centralizzato vi è questo sistema di messaggi, che si muovono su un sistema che non è affidabile, il che può portare a perdite di informazioni. Si può affermare che le certezze in un sistema centralizzato vengono meno quando si ha a che fare con un sistema distribuito.

In caso di perdita di risposta il client starà in attesa: il client sarà quindi bloccato, e non è per questo semplice capire se la causa è il guasto del server o la perdita di informazioni. Questi due incidenti portano a conseguenze sintattiche differenti.

Una remote procedure call è quindi una chiamata a procedura remota che trasforma l'iterazione client/server in una chiamata a procedura, simile a quella locale, nascondendo al programmatore la maggiore parte dei meccanismi implementativi che la compongono.
E' tuttavia una tecnologia che impone determinati vincoli che devono essere osservati dal programmtore, nonché elementi che a questo verranno nascosti.
A livello de codice il programmatore deve essere consapevole che alcune delle procedure che chiamerà sono procedure remote e non locali. Occorre quindi portare le definizioni delle interfacce dei socket ad esempio.
La differenza è che nel rpc è una struttra che si va a creare con un certo linguaggio, che può essere chiamata solo in un determinato modo importato nel codice.
A tempo di esecuzione per far funzionare questo meccanismo di interazione client/server si necessita di un supporto runtime, che si trasforma in un processo server che è in grado di interpretare le rpc. Il runtime support deve inoltre trovare il server appropriato, questa operazione è effettuata lato client.
Durante la compilazione poi per ogni chiamata a procedura remota vengono agganciate linee di codice non visibili che servono a strumentare da una parte i parametri e dall'altra le chiamate al supporto runtime, che servono ad esempio a conoscere il nome del server che effettua la procedura. Ogni passaggio quindi effettua la sua parte di mascheramento.

I meccanismi per l'rpc consistono di un protocollo che nasconde le insidie di rete come la perdita dei pacchetti, e di un meccanismo di impacchettamento di argomenti lato chiamante e di spacchettamento nel lato chiamato.

Localizzazione del server
la localizzazione può avvenire tramite un metodo statico, consistente nella cablazione all'interno del client dell'indirizzo IP del sever; un metodo dinamico, che coinvolge lo stub del client.

Semantica delle rpc
[…]
exactly once: per poterla usare occorre distinguere la prima esecuzione dalla seconda esecuzione, in modo tale da inviare direttamente i risultati, immagazzinati nel server. Qando al server arriva a una richiesta gia effettuata il risultato dovrà essere preso dal file di log: questo accade per evitare una seconda esecuzione, il cui risultato è preso dal disco dove si è immagazzinato il risultato della prima esecuzione.
Questo viene codificato via software.
Per notificare che c'è stato un problema del client (che si suppone sia morto), nelle richieste inviate al server s aggiunge un numero di reincarnazione, poiché questo viene tirato nuovamente su per completare la richiesta: questo rappresenta un identificatore univoco della richiesta, sul quale il server può speculare per verificare che la richiesta sia già stata eseguita.

simple rpc stack
è una struttura basata su cinque livelli: (vedi figura) i cui ultimi 3 sono quelli che implementano il protocollo di comunicazione.

Protocolli stack
blast

protocollo chiamato bulk trasfer
prende i pacchetti (i parametri del client) e inizia a dividere questi in frammenti, ognuno dei quali viene inviato attraverso udp, uno dietro l'alto. La strategia si compone di una ritrasmissione selettiva accompagnata da partial aknowlegment; vengono utilizzati inoltre tre timer: done, last_frag, retry.

Lato trasmittente, viene settato un timer done quando i frammenti nella memoria locale vengono ivnati tutti
in presenza di una srr, rispedisco i pacchetti mancanti e resetto il done
se la srr notifica che tutti i frammenti sono stati ricevuti, il sender libera i frammenti (cosa che accade anche nel caso in cui si esaurisce il done, questo può portare quindi a una perdita di informazioni perché il sender non trasmette più nulla).

Lato ricevitore
quando arriva il primo frammento si mette un timer last_frag, entro il quale ci si aspetta l'ultimo frammento: se avviene si prende il blocco di info e viene passato a livello superiore.
Se l'ultimo frammento non è arrivato il ricevitore trasmette un srr e posiziona un timer retry.
(succede anche nel caso in cui arriva l'ultimo frammento ma il messaggio è incompleto)
se il retry scade una o due volte, si manda una srr e si reimposta retry, alla terza volta però ci si arrende e si libera il messaggio incompleto.

Componenti RCP
si è creata una struttura protocollare che non abbia le problematiche del tcp, poiché in questo in caso di imprevisti la connessione deve ricominciare da capo, perdendo tempo.
Ci si struttura su 3 livelli protocollari: blast, chan, select.
Si nota che ci possono essere più chiamate rcp, che la macchina gestisce in modalità server: il select cercherà la rpc giusta della macchina a cui inviare i paramentri, facendo multiplexing. Il chan gestisce la request e la retry dei messaggi che partono e arrivano. All'interno del chan si gestisce la semantica delle rpc, definita come at least once, che permette di ragionare sul singolo invio dei paramentri. Al livello blast la req con tutti i parametri può essere un messaggio molto pesante, quello che accade è che il req si trasforma in ondate da 32 pacchetti ripetute il numero di volte necessario alla trasmissione di tutto il messaggio. Nel blast ci sono meccaniscmi che permettono di ridurre gli aknowlegment, inviando le ack tramite settaggio dei bit della maschera di rete mancanti: questo è l'srr.

Rispetto a tcp questo protocollo, in caso di connessione a banda molto larga, grazie a questo meccanismo mentre TCP ci metterebbe un certo tempo per usare molta banda, questo inizia a sparare immediatamente i blocchi da 32 senza mai fermarsi, aspettando solo gli acknowlegment, in questo modo sfrutta tutta la banda a disposizione.

Select
il select seleziona la rpc a cui devono arrivare i risultati, seguendo un ragionamento a livello di middleware (a differenza di TPC/IP che ragiona a livello di rete, più basso).
In questo caso abbiamo l'indirizzamento flat, indirizzo ip unico per ogni rpc, oppure di tipo gerarchico, legato al main che contiene la rpc. Una main è composta da 4 rpc che possono essere chiamate indipendentemente.

Formattazione dei dati
Chiamante e chiamato si suppongono su macchine distinte, il che implica processori e componentistica di vendor diversi: per questo lo stream deve essere strutturato, perché client e server devono sapere quali sono i parametri di interazione. Il concetto è prendere e codificare dati, per poi una volta ottenuto i parametri codificati, si inviano attraverso la connessione mediante

messaggi.
La prima difficoltà è la raprresentazione dei numeri nelle memorie dei calcolatori, perché esistono due problematiche dentro il singolo computer, e sono la rappresentazione degli interi e la rappresentazione dei floating point. Ad esempio all'interno della parte interi, la rappresentazione può essere big o little endian. Nella floating point esiste una rappresetnazione standard, ma anche molte altre non standard. Nelle rpc bisogna risolvere un problema di interoperabilita, attraverso una sorta di modello di informazione intermedio più astratto che permette a tutti i calcolatori di riferirsi a quel modello: questi modelli sono detti forme canoniche.
Un altro approccio di risoluzione è il Receiver makes right: in questo caso il receiver fa tutto il lavoro, capisce quale forma canonica deve applicare.

Tagged dei dati
un dato può essere tagged o untagged: i primi sono dati con uno o più campi informativi, che indicano tipo, dimennsione e architettura.
Un untagged non contiene campi dati sulla dimensione, e si presuppone che ci sia un accordo a priori tra sender e receiver su quale sarà la loro dimensione.
Di fatto il tag è una forma canonica, codificabile dal sender e receiver.

XDR
La forma canonica più importante è la XDR (external data representation).
E' una forma canonica intermedia, usata in modalità untagged eccetto se usata sulle strutture dinamice, per le quali è necessario il tagged.

Abstract Syntax Notation One (ASN-1)
E' uno standard usato in tutta una serie di comunicazioni standardizzate, come i protocolli SMP, tutti i protocolli che permettono di controllare le macchine da remoto, nonché gli elementi della rete. E' una forma canonica che usa il tagged, peraltro in una forma molto pesante: tipo dato, lunghezza,
dato; questo viene opportunamente annidato.

Si nota che ad oggi non esiste uno standard “vincente”, molti sono tra loro affiancati e scelti in base alle esigenze delle applicazioni.

Un esempio di receiver-makes-right è il NDR, dove nei pacchetti è scritto come devono essere rappresentati i dati all'interno del receiver.

Sistemi distribuiti
col client/server abbiamo fondamentalmente coordinato le due macchine per poter eseguire un certo lavoro: parlando invece in un sistema distribuito, composto da più macchine, il concetto è più articolato.

I problemi della condivisione dei dati sono dati dalla coordinazione e dalla sincronizzazione delle macchine all'interno del sistema. Se guardiamo i sistemi concorrenti e i sistmei distribuiti, nei sistemi concorrenti esiste una nozione importante che è quella del clock: le attività della macchina sono cadenzate da questo clock.
Imprevisti su sistemi distribuiti possono essere ad esempio la rottura di un server, ma un server può essere caricato troppo, provocando un sovraccarico di informazioni che mandano il server in crash. Questo descritto è un problema tipico dell'approccio client/server.

Il un sistema distribuito si ha concorrenza sia temporale che spaziale: ovvero diversi processi

possono girare su macchine diverse: grazie al clock i processi hanno intrinsecamente un ordine totale.

Mutex
Il mutex deve avere delle proprietà di correttezza ben precise:
1- mutex: al più un processo può entrare nella sezione critica alla volta
2- no_deadlock: un processo tra n rimane bloccato e non accede alla sezione critica mentre esiste almeno un processo che accede alla sezione critica, dove vengono fatte operazioni con la risorsa condivisa. Il deadlock ci sarebbe nel momento in cui tutti i processi sono bloccati, situazione per quanto detto evitata.
Rispetto a questo strato il codice utente sta sopra al protocollo, diviso da questo tramite il mutex.
Il protocollo sottostante si basa su delle assunzioni che definiscono il sistema di calcolo.
Il primo modello di sistema considerato si compone di una cpu che può avere più processi (e di conseguenza si ha uno scheduler fair) e una memoria.

NB un'operazione atomica non può essere interrotta.

Si ricorda che scheduler fair significa che da un punto di vista matematico in un'esecuzione infinita non esiste un processo che resta in esecuzione per un tempo infinito.

Come progettare un protocollo che permetta l'accesso in mutua esclusione a una risorsa che soddisfi quelle proprietà che ha il mutex?

Algoritmo di Dykstra

```
REPEAT
1       NCS
2       Y[i] := TRUE
3       X[i] := FALSE
4       WHILE k != i DO
5               IF NOT Y[k] THEN k := i
6       X[i] := TRUE
7       FOR j := 1 TO n DO
8               IF i != j AND X[j] THEN GOTO 3
9       <<CRITICAL SECTION>>
10      Y[i] = X[i] = FALSE
FOREVER
```

Mutex
mutua esclusione
nella concorrenza normalmente si lavora per “contraddizione”: assumiamo che due processi i e j siano in sezione critica contemporaneamente. Se i è in sezione critica allora X[j] = false.
Tutto questo implica che i.8 precede j.6, ovvero che il processo P[j] non è ancora riuscito a fare j.6.
Dall'altra parte si può dire che j.6 precede j.8 per la sequenza locale che rispetta un ordine totale.
Se j è in sezione critica allora ha trovato X[i] = false, implicando che j.8 precede i.6, inoltre i.6 precede i.8
Quindi si è trovato che i.8 precede j.8 e j.8 precede i.8, ottenendo che i.8 precede i.8, ovviamente questo è impossibile. Abbiamo dimostrato per assurdo che un solo processo per volta può stare in sezione critica.

No deadlock
il deadlock è la situazione in cui tutti i processi sono bloccati; da un punto di vista matematico

nessun peocesso riesce in un tempo infinito ad accede infinite volte alla sezione critica: non significa comunque che non ci siano processi che riescano ad entrarci.
L'algoritmo di dykstra è stato pensato appositamente per non avere deatlock

dimostrazione per assurdo:
si assumono D processi che sono bloccati nella trying section. Per ogni i' appartenente a D si ha che Y[i'] = true.
Esiste un i appartenente a D tale che k = i, ovvero se si ha tutti i processi bloccati nel trying section esiste sicuramente un k che riesce a scrivere per ultimo la variabile k. Se questo non fosse significa che questo i non appartiene a D, quindi non appartiene al set di processi bloccati nella trying, diciamo quindi che i è uscito, e qualora mettesse Y[i] a false, esisteranno un sottoinsieme di D processi che vedranno il cambio di variabile, che riusciranno a entrare a settare k. Qualcuno però, a seconda dello scheduler, entrerà per ultimo e setterà k al valore i.

Per ogni processo i' appartenente a D/i (D escluso i), prima o poi X[i'] = false
Questo significa che prima o poi tutti i processi si bloccano nella trying section eccetto i

Si considera ora un sistema composto da più cpu e più memorie unite da bus
Qual è il concetto di scrivere una variabile memorizzata in un multisistema?
Quindi write(a) è un'operazione non istantanea (da un punto di vista di sequenza di operazioni atomiche del modello di dykstra)

Algoritmo del panettiere

```
REPEAT
1       NCS
2       CHOOSING i := TRUE
3       NUM[i] := 1 + MAX{NUM[j] : i<= j <= N}
4       CHOOSING [i] := FALSE
5       FOR j := 1 TO N DO BEGIN
6               WHILE CHOOSING [j] DO SKIP
7               WHILE NUM[j] != 0 AND {NUM[j], j} < {NUM[I}, i}
8       END
9       <<CRITICAL SECTION>>
10      NUM[i] := 0
FOREVER
```

questo algoritmo si basa su un concetto “elimina code”
si cerca di capire chi è l'ultimo in coda (si ricorda che in un sistema un processo può leggere le variabili degli altri processi). Il problema da risolvere consiste nel costruire in un modo distribuito un meccanismo di cooperazione tra i processi affinché si possa attribuire un numero a i (più grande del massimo che era prima nella coda).

[trying: 2-8]
[exit: 10]

righe 2-4: si scrive il numero del “biglietto” NUM[i]. questa parte viene chiamata doorway.
Righe 5-8: “bancone del bakery”, una volta scritto il “numeretto”, si aspetta in fila.

Quando si usa questo algoritmo si assume che un processo può leggere le variabili di un altro processo; questa assunzione significa fondamentalmente la necessita di strumenti che consentono di poter leggere nella memoria dei processi, cosa non vera all'interno di un sistema distribuito su più nodi, fisicamente separati; questo ci fa passare da un concetto di lettura senza che gli altri se ne accorgono, a una tecnica di lettura basata su permessi conferiti dai processi le cui variabili devono essere lette. Tutto questo ha un forte impatto sul codice. Per quanto riguarda l'algoritmo del panettiere in un sistema distribuito:

```
1       NCS
2       CHOOSING := TRUE
3       FOR J != I
4               SEND NUM TO J
5               RECEIVE REPLY(V) FROM J
6               NUM := MAX(V, NUM)
7       NUM := NUM + 1
8       CHOOSING := FALSE
```

(corrispondono alle righe 1...4 del codice sopra)
quando si ragiona a livello di sistemi distribuiti non importa solo il codice, ma anche i messaggi scambiati.

```
9       FOR J = 1 TO N
10              REPEAT
11                      SEND CHOOSING TO J
12                      RECEIVE REPLY(V) FROM J
13              UNTIL V
```

da un punto di vista client/server il codice è tradotto in un ciclo for di richieste send-receive, nel quale ripetutamente si chiede la choosing, se è true occorre ciclare, se è false vado alle righe successive del codice
dopo la doorway quindi si attacca un altro pezzo di codice, dove si inizia a chiedere choosing, e ci sono due possibilità: se non sta scegliendo si va avanti, altrimenti continuo a mandare la richiesta.

```
14      SEND NUM TO J
15      RECIVE REPLY(V) FROM J
16      UNTIL V != 0 OR (NUM, I) > {V, J}
18      END
19      CRITICAL SECTION
20      NUM = 0
```

NB da questo si evince che non c'è compatibilità tra algoritmi di sistemi paralleli e algoritmi di sistemi distribuiti, pertanto occorre riaddattarli.

Occorre avere un pattern di messaggi adeguato per il sistema distribuito, inoltre si necessita che l'algoritmo sia prevedibile, ovvero è necessario conoscere un numero di messaggi necessario per completare l'azione: il problema si trova nel repeat/until.

In un ambiente distribuito il codice del processo si compone anche di quella parte che gli permette di rispondere alle chiamate: questi si chiamano thread di risposta.

```
UPON THE ARRIVAL OF A NUM MESSAGE
        SEND REPLY(NUM) TO J

UPON THE ARRIVAL OF CHOOSING MESSAGE FROM J
        SEND REPLY(CHOOSING) TO J
```

tutto questo sta nello stesso processore, quindi potrebbe trattarsi anche di due thread di uno stesso processo.

L'arrivo dei messaggi è asincrono, poiché non si conosce il tempo in cui un messaggio arriva.

```
BEGIN
1       STATE := REQUESTING
2       NUM := NUM + 1; LASTREQ :=  NUM
3       SEND REQUEST(LASTREQ) TO P1...PN
4       WAIT UNTIL RECEIVER = N - 1
5       STATE := CS
6       CS
7       SEND REPLY TO ANY REQUEST IN Q
7'      Q := 0 ; STATE := NCS; REPLIES := 0      UPON RECEIPT FROM PJ 12.#REPLIES++;
END
```

MOD.SISTEMA
1- i messaggi affidabili ma i tempi di trasmissione imprescindibili ma finiti
2- le variabili sono proprietà dei processi; non si possono r/w variabili di altri processi
3- scheduler fair

si assume che PI E PJ siano nella sezione critica contemporaneamente.
Si hanno 3 casi possibili: