/\*QUESTI APPUNTI NON INCLUDONO IL CODICE CHE FA PARTE DEL PROGRAMMA\*/

LEZIONE 1

PROCESSI:

ogni programma può essere rappresentato da uno o più processi. i processi comunicano tra di loro, sia che siano sullo stesso server, sia che siano in esecuzione su macchine diverse. i processi hanno necessità di risorse che gli derivano dal SO o da altri processi. un processo è composto da un codice, un insieme di dati e degli attributi (Process Control Block: id, state, priority, program counter, memory pointers, context data simil backup, I/O status information, accounting information). ogni processo nasce dalla fork del suo processo genitore (il primo processo è init, avviato dal SO).

fork(): processo padre crea processo figlio, copia quasi identica, tramite comando fork(). i due processi vengono eseguiti in modo concorrenziale, simultaneamente. la fork restituisce al padre l'id del processo figlio appena creato, -1 in caso di errore, 0 al figlio. il processo figlio eredita una copia identica della memoria, i registri della CPU e i file aperti dal padre.

wait(): blocca il padre finchè uno qualsiasi dei suoi figli termina. ritorna l'id del figlio o -1 se non esistono figli. waitpid() blocca il padre in attesa di un processo specifico.

exit(): ha un parametro status che serve a comunicare informazioni al processo padre. salva i risultati, esegue le funzioni specificate con i comandi atexit(fun) e on\_exit(fun), esegue fflush(), chiude i file aperti a meno di quelli condivisi con altri programmi, ed esegue \_exit(status) (che salva i risultati, dealloca la memoria, assegna ad init i processi figli del processo che la invoca, controlla che il padre abbia status alive e se è così il processo che l'ha invocata entra in uno status di zombie e mantiene il risultato salvato finchè il padre non lo richiede con la wait. se il padre non è alive, il processo termina).

THREADS:

un processo ha due caratteristiche: un aspetto fisico (memoria, spazio di indirizzi..) e un aspetto temporale (sequenza di operazioni). il SO gestisce questi due aspetti in modo indipendente. la gestione dell'aspetto temporale (flusso di istruzioni) è detta thread. la gestione dell'aspetto fisico è detta task. i SO attuali sono multiple processes, multiple threads per process. in un SO multithread ci sarà uno spazio di indirizzi di memoria virtuale, accesso protetto ai processori, ad altri processi, ai file e alle risorse I/O. ogni thread ha uno stato di esecuzione, un thread context quando non è running, variabili locali e accesso alla memoria del processo condiviso con altri thread dello stesso processo (thread control block, memoria propria, accesso a memoria condivisa). un thread ha tre stati possibili: running, ready, blocked. lo stato di un thread cambia in seguito a comandi: spawn, block, unblock, finish. i thread possono essere gestiti a livello di utente (cioè dall'applicazione) e prendono il nome di user level thread (ULT) o a livello di kernel (KLT o lightweight processes).

ULT pro: l'uso di ULT è possibile grazie al semplice import di una libreria, quindi può essere attuato su ogni OS.

ULT contro: il kernel non è a conoscenza della loro esistenza e quindi fornisce un solo core (non si sfrutta architettura multicore). una chiamata bloccante blocca tutti i thread del processo.

KLT pro: per lo stesso processo, il kernel può definire più thread su processori diversi. se un processo è bloccato, il kernel può assegnare il processore ad un altro processo. il kernel mantiene le context information del processo e dei thread.

KLT contro: l'intervento del kernel comporta dei rallentamenti.

è possibile avere approcci combinati di ULT e KLT.

PTHREAD:

PThread: libreria per ULT creata per UNIX (include pthread.h). per utilizzare PThread, un programma va strutturato in task discreti (finiti) ed indipendenti che possano essere eseguiti in modo concorrenziale. esistono due modelli di programmazione con thread: manager/worker(un thread principale assegna lavoro e input a thread sottostanti) e pipeline(un task è diviso in sotto operazioni gestite in serie ma in modo concorrente da thread diversi). un programma è detto thread-safe quando più thread possono essere in esecuzione senza interazioni inattese (dati sovrascritti, race conditions).

pthread\_create(): crea un nuovo thread eseguibile.

pthread\_exit(): termina un thread.

pthread\_cancel(): è chiamato da un altro thread.

exit(): termina il processo (quindi termina il thread che lo chiama e tutti gli altri thread del processo).

pthread\_detach(): distacca un thread (anche il main) dagli altri.

------------------------------------------------------------------------------------------------

LEZIONE 2

categorie di computer systems: single instruction single data, single instruction multiple data, multiple instruction single data, multiple instruction multiple data (schede grafiche attuali).

i processori in parallelo che operano con multiple instruction multiple data (MIMD) sono distinti in shared memory (a loro volta distinti in architetture master/slave e symmetric multiprocessor) e distributed memory (gestiti tramite cluster).

CONCORRENZA:

i SO lavorano con più processi/thread. quando tali processi non sono totalmente indipendenti si hanno casi di concorrenza.

operazione atomica: implementazione di una o più istruzioni indivisibili.

sezione critica: parte di codice in un processo che richiede accesso a risorse condivise che non va eseguito se un altro processo sta eseguendo la stessa sezione di codice. deve essere finita (entercritical, exitcritical).

mutua esclusione: il requisito per cui, quando un processo è in sessione critica, nessun altro processo sia in una sessione critica che accede alle stesse risorse condivise. è attuabile a livello di hardware disabilitando le interrupt (questo ovviamente crea problemi, funziona solo su sistema monoprocessore), oppure con l'istruzione atomiche compare\_and\_swap (è come un if in hardware non interrompibile) che serve all'implementazione dell'ingresso in sezione critica, o ancora con l'istruzione exchange. il problema di queste istruzioni hardware è il busy waiting (cioè tutto il tempo in cui un processo occupa il processore per controllare ad esempio la compare\_and\_swap). il deadlock rimane possibile.

race condition: situazione in cui più processi leggono e scrivono un dato condiviso e il risultato finale dipende dal timing della loro esecuzione.

deadlock: situazione in cui due o più processi non possono procedere perchè ognuno sta aspettando gli altri.

livelock: situazione in cui due o più processi cambiano continuamente il loro stato in risposta ai cambiamenti degli altri processi senza fare lavoro utile.

starvation: situazione in cui un processo runnable viene ignorato indefinitivamente dallo scheduler.

meccanismi di implementazione di concorrenza a livello software:

semaforo: costrutto basato su un valore intero usato per segnalazioni tra processi. solo tre operazioni, tutte atomiche, possono essere eseguite su un semaforo: inizializzazione, decremento (può risultare in un block), incremento (può risultare in un unblock).

semaforo binario: semaforo con unici valori 0 e 1.

mutex: simile a semaforo binario ma il processo che blocca il mutex deve essere lo stesso che lo sblocca.

condition variable: un data type usato per bloccare un processo finchè una particolare condizione diventa true.

monitor: è un oggetto semaforo, con metodi e con una coda di processi in attesa di accedere.

event flags: flag usate come meccanismo di sincronizzazione.

mailboxes/messages: due processi si scambiano indormazioni che possono essere usate per la sincronizzazione.

spinlocks: meccanismo di mutua esclusione in cui si esegue un ciclo infinito finchè una variabile di lock cambia valore ad indicare la disponibilità.

semaforo: è essenzialmente una variabile (contatore e coda) manipolabile tramite istruzioni atomiche di inizializzazione, decremento (semWait), incremento (semSignal o semPost). un processo che vuole accedere ad una risorsa disponibile decrementa la variabile. se la risorsa non è disponibile, il processo che vuole accedervi decrementa e in quel caso la variabile va da 0 a -1 e quindi il suo valore rappresenta il processo in attesa, in coda. non c'è modo di sapere se il decremento del semaforo comporterà un blocco, nè se ci sono processi in attesa, nè quale tra due processi concorrenti continuerà l'esecuzione dopo una semWait (nel caso uniprocessor). per la gestione della coda si distingono semafori strong che implementano una politica FIFO e semafori weak che non specificano l'ordine di uscita dalla coda.

------------------------------------------------------------------------------------------------

LEZIONE 3

monitor: sono più facili da utilizzare dei semafori, vengono importati come una classe. non esistono implementazioni ufficiali di monitor in C. un monitor è costituito da più procedure la cui esecuzione è esclusiva (quando una procedura è in esecuzione su un monitor nessun altro monitor può eseguire alcuna procedura). la sincronizzazione all'interno del monitor è ottenuta attraverso variabili condizionali interne al monitor gestite con funzioni cwait e csignal. un altro metodo di implementazione di sincronizzazione e comunicazione è attraverso il message passing, che funziona non solo su sistemi operativi ma anche su sistemi distribuiti (poichè non necessita di memoria sulla stesso sistema in cui risiede invece un semaforo o un monitor). il message passing si basa su due istruzioni: send(destination, message) e receive(source, message). la receive può anche non specificare la source. send e receive possono essere blocking o nonblocking (la combinazione più comune è nonblocking send, blocking receive). il contenuto del messaggio ovviamente avrà una lunghezza fissa o variabile e potrà contenere informazioni. se si usa indirect addressing i messaggi sono inviati ad una mailbox, cioè una queue (quando in una mailbox è presente un msg, la critical section è libera).

------------------------------------------------------------------------------------------------

LEZIONE 4

DEADLOCK:

è una situazione permanente in cui un insieme di processi compete per delle risorse di sistema e si blocca. ogni processo aspetta un evento che solo un altro processo, a sua volta bloccato, può triggerare. il deadlock si può verificare in corrispondenza di concorrenza su risorse riutilizzabili o consumabili. ci sono quattro condizioni affinchè si crei un deadlock; le prime tre sono necessarie ma non sufficienti e sono: mutua esclusione, hold-and-wait e assenza di pre-emption, la quarta è condizione sufficiente ed è la circular wait.

mutual exclusion: un solo processo per volta può accedere ad una risorsa.

hold-and-wait: un processo mantiene delle risorse allocate mentre aspetta l'assegnazione di altre risorse.

no pre-emption: nessuna risorsa può essere rimossa da un processo che la sta usando.

circular wait: coda circolare di processi, tale che ogni processo sfrutta una risorsa di cui ha bisogno il processo successivo.

il SO deve gestire la possibilità di deadlock. può cercare di prevenirlo, cercare di evitarlo durante l'esecuzione, o cercare di rilevarlo monitorando le risorse.

DEADLOCK PREVENTION:

ci sono due tipi di prevenzione del deadlock, uno indiretto in cui si cerca di prevenire una delle tre condizioni necessarie, e uno diretto in cui si cerca di prevenire la condizione sufficiente. la mutua esclusione è molto difficile da prevenire. la hold-and-wait è prevenibile facendo una sola richiesta per tutte le risorse necessarie invece che n richieste consecutive (questo ovviamente può portare introdurre tempi più lunghi). no-preemption è prevenibile permettendo al SO di rimuovere risorse al processo con priorità più bassa, possibile solo quando lo stato delle risorse può essere salvato e ripristinato in seguito. la circular wait è prevenibile definendo un ordine sulle risorse necessarie, ma anche questo è molto complicato e non sempre fattibile.

DEADLOCK AVOIDANCE:

il SO evita deadlock tramite decisioni eseguite dinamicamente ogni volta che viene richiesta una risorsa che, se assegnata, può portare a deadlock. permette più concorrenza rispetto alla deadlock prevention. esistono due approcci di deadlock avoidance: process initiation denial (non si avvia un processo se il suo avvio può portare a deadlock) e resource allocation denial (non si garantisce una risorsa ad un processo se l'allocazione di tale risorsa potrebbe portare a deadlock). per la process initiation denial il SO deve conoscere la totalità di risorse di cui i vari processi avranno necessità durante la loro intera esecuzione.

DEADLOCK DETECTION: le richieste di risorse sono garantite ogni volta che è possibile, il SO non cerca di capire se avviene il deadlock o meno, ma cerca di rilevarlo quando accade. quando si rivela il deadlock si possono abortire tutti i processi in deadlock contemporaneamente, o se ne può abortire uno per volta, o si possono riportare i processi coinvolti in checkpoint precedenti in cui non erano in deadlock, o si può togliere una risorsa per volta finchè si esce dal deadlock.

------------------------------------------------------------------------------------------------

LEZIONE 5

deadlock implica starvation. non starvation implica non deadlock.

ALGORITMO DI DIJKSTRA:

ha delle variabili globali: un array di dimensione pari al numero di processi che indica quali processi sono interessati ad entrare in sezione critica. un array di dimensione pari al numero di processi che indica quali processi sono passati fino ad un certo punto e potrebbero potenzialmente arrivare in sezione critica. una variabile k che rappresenta il turno. è presente anche una variabile locale, un intero che identifica il processo. non garantisce no starvation.

il processo i è interessato e quindi imposta intrested[i]=true. se il turno è di quel processo (e questo si controlla con un while k!=i), questo imposterà direttamente passed[i]=true, altrimenti se il turno è di qualcun altro il processo corrente imposta passed[i]=false e controlla di chi è il turno. se il processo di cui è il turno non è interessato, allora il processo corrente imposterà turno k=i. dopo di che si scorrono tutti i processi tranne quello corrente di indice i. se qualche altro processo è impostato su passed[j]=true, si ritorna al while. se a questo punto si vede che il turno non è del processo corrente, si reimposta passed[i]=false. solo quando tutti gli altri processi sono in passed[i]=false e il turno coincide col processo corrente, tale processo può entrare in sezione critica. il processo reimposta poi sia passed[i] che intrested[i] a false.

INTER PROCESS COMUNICATION:

un processo può accedere solo al suo spazio di indirizzi (area di memoria). il kernel può accedere a tutte le aree di memoria. quando i processi sono cooperativi devono poter comunicare e condividere informazione. la comunicazione è attuabile tramite memoria condivisa o con data transfer (byte stream con pipe e FIFO). avere memoria condivisa non implica la sincronizzazione (saranno quindi necessari semafori), mentre il message passing consente sia la comunicazione che la sincronizzazione.

SHARED MEMORY:

il meccanismo di ICP con shared memory è descrivibile con la logica producer/consumer.

posix shared memory API: shm\_open() (crea/apre shared memory page senza definirne la dimensione. ritorna un file descriptor), ltrunc() e ftruncate() (limitano la size della shared memory page), mmap() (restituisce il puntatore all'area di memoria shared memory), close() (chiude il file descriptor), shm\_unlink() (libera la shared memory).

MESSAGE PASSING:

la comunicazione è attuata tramite due primitive send(destination, message) e receive(source, message) (destination e source possono essere omesse) e una struttura dati del messaggio header+body (header: message type, destination id, source id, message length, control information). ciò che permette la comunicazione sono le pipes (per processi tra loro relazionati con fork()) o named-pipes (per processi non relazionati tra di loro, sono dette FIFO), che sono essenzialmente buffer (di solito 4096 byte) che permettono di avere un canale di comunicazione unidirezionale padre-figlio o tra processi. una pipe è chiamata con una call pipe() e viene eliminata alla terminazione del processo. una pipe aperta consiste di due descrittori, uno per ogni processo. il processo con descrittore 0 è quello che scrive e quello con descrittore 1 è quello che legge. una named-pipe non è chiusa alla terminazione del processo. un processo che prova a leggere una pipe vuota o uno che prova a scrivere su una pipe piena rimane bloccato finchè qualcuno non scrive sulla pipe vuota o non rimuove dati scritti da quella piena. una scrittura di n<=PIPE\_BUFF byte è atomica, una più lunga potrebbe essere inframezzata da altre write. la pipe è un dispositivo logico, non fisico, quindi definiamo una pipe come chiusa quando nessuno scrive più su di essa, quindi quando tutti i processi con accesso in scrittura che condividevano il descrittore fd[1] lo hano chiuso; in questo caso la chiamata read() effettuata da un lettore restituisce 0 e a sua volta uno scrittore che prova a scrivere su una pipe che non ha lettori riceve un segnale di errore. per evitare deadlock è necessario che tutti i processi chiudano i descrittori di pipe che non gli servono più usando una close().

------------------------------------------------------------------------------------------------

LEZIONE 6

LAMPORT'S BAKERY ALGORITHM:

è un algoritmo in cui i processi comunicano attraverso variabili condivise in cui si scrive/legge (read/write non sono operazioni atomiche!), come nell'algoritmo di Dijkstra. basato su due fasi, una di doorway in cui si "prende il numero" e una di waiting in cui il processo attende di entrare in critical section. è inoltre necessario l'identificatore "choosing" per ogni processo che è impostato a true quando un processo sta leggendo i valori degli altri processi e sta impostando il suo. quando un processo sa che nessun altro sta impostando il proprio numero del posto in fila inizia un controllo su tutti gli altri processi e controlla sia la priorità del numero del turno che la priorità dell'id del processo. ha priorità chi ha il turno con numero minore o, a parità di turno, chi ha l'id minore. va notato che il proprio turno va impostato come il massimo+1 tra tutti i turni degli altri processi e la ricerca del massimo va fatta sfruttando variabili locali e non modificando ogni volta il valore vero e proprio onde evitare confusione (se un processo aggiorna il valore del suo turno ogni volta che controlla un altro processo, un terzo processo che si basa su di lui potrebbe scegliere il suo turno sulla base di valori sbagliati).

come dimostro che c'è mutua esclusione? si dimostra per assurdo. poniamo di avere due processi Pi e Pj, con i<j, entrambi in critical section (Pj entra in CS e poi anche Pi poichè a parità di turno ha indice minore). Pi e Pj hanno num=5 e choosing=false. allora prima di entrare in CS, Pi deve aver controllato che num[k]=0 o che (num i, i)<(num k, k). allo stesso modo, prima di entrare in CS Pj deve aver controllato che num[k]=0 o che (num j, j)<(num i, i) e quindi è entrato in CS se num k=0 o se numk>5. ma numk non poteva essere >5, altrimenti avrebbe voluto dire che Pi e Pj che hanno num=5 sarebbero già entrati in CS poichè hanno precedenza su qualsiasi processo con numk>5, o avrebbero scelto numeri di turno più alti di 5 per mettersi in coda. numk può essere =0 se Pk è uscito dalla CS ma quindi non sarebbe più interessato, oppure se sta impostando choosing a true, oppure se è nel mezzo delle azioni per scegliere il suo turno. ma se Pk fosse nel mezzo delle azioni per scegliere il suo turno, i controlli di Pj riscontrerebbero un'incongruenza al controllo di number[k]!=0.

RETE:

la rete si classifica in base alla scala, a partire dalla personal area network, per poi estendersi alla local area network (più computer collegati a access point o ethernet switch che a loro volta connettono alla rete), metropolitan area network (ad esempio per la tv cablata negli USA, delle intere zone abitative sono collegate alla stessa rete), wide area network (collegano zone intere geograficamente immense, usando cavi propri o di un internet service provider, oppure tramite connessione virtualizzata attraverso internet), internet nel senso più esteso.

STRUTTURA A TRE LIVELLI DI UNA RETE DI CALCOLATORI:

area applicativa: livelli di applicazione, presentazione e sessione del modello OSI. vale l'indirizzamento DSN.

area interoperabilità trasporto dell'informazione: livelli di trasporto e rete. vale l'indirizzamento IP.

area infrastruttura di trasporto dell'informazione: livelli di collegamento e fisico. vale l'indirizzamento MAC.

PROTOCOLLO IP:

indririzzamento di rete e instradamento dei pacchetti. frammentazione e riassemblaggio. corrispondenza con indirizzi dei livelli sottostanti (ARP).

INDIRIZZAMENTO:

indirizzo IP: 32 bit; NetId + HostId; classe A(7 bit netid, 24 bit hostid), classe B(14 bit netid, 16 bit hostid), classe C(21 bit netid, 8 bit hostid), classe D, classe E;

------------------------------------------------------------------------------------------------

LEZIONE 7

protocollo IP. TCP. UDP.

DOMAIN NAME SYSTEM:

serve ad una macchina che fa una richiesta per un sito.com di cui non sa l'indirizzo IP. allora chiede ad un DNS che trova l'IP di tale sito nei suoi record o in quelli dei DNS circostanti.

TCP/IP:

fornisce connessione logica tra due host, o meglio tra due porte corrispondenti a processi attivi a livello applicativo in due host. le più famose applicazioni di TCP/IP sono quelle di: simple mail transfer protocol SMTP, file transfer protocol FTP, secure shell SSH.

SOCKETS:

una socket permette la comunicazione tra client e server grazie all'identificazione univoca con IP e id della socket. può essere orientata alla conessione (socket TCP/IP, o Stream socket) o no (socket UDP/IP, o Datagram socket). la socket diventa quindi un collegamento logico tra due livelli applicativi di due macchine distinte, anche se in realtà è un'interfaccia tra il livello applicativo e quello di trasporto. una connessione è quindi creata tra due coppie (IP, id socket). quando il client conosce solo l'IP invia una request verso una "well-known port". il server risponderà verso la stessa porta da cui ha ricevuto la request, ma a partire da una porta diversa dalla sua "well-knwon" così da indicare al client su quale socket continuare la comunicazione.

il SO utilizza file descriptor per gestire le socket. il comando socket() restituisce l'id della socket creata. le primitive a livello logico sono listen, connect, send, receive, disconnect; quelle a livello pratico sono: socket (crea nuovo end point di comunicazione), bind (associa indirizzo locale alla socket), listen (annuncia la volontà di accettare connessioni. permette la ricezione ma non accetta la connessione), accept (è bloccante nel server, rappresenta l'accettazione della connect lato client e blocca fino allo stabilirsi della connessione con client), connect (è bloccante lato client), send, receive, close (bloccante finchè tutti i dati non sono stati inviati). i comandi in cui specifichiamo l'indirizzo dall'user al SO sono bind, connect (solo TCP), sendto (solo UDP). quelli con cui passiamo l'indirizzo da OS ad user sono accept (solo TCP), recvfrom (solo UDP).

NETWORK ADAPTORS:

scheda di rete. è costituita da due parti separate che interagiscono attraverso una FIFO che maschera l'asincronia tra la rete e il bus interno. la prima parte interagisce con la CPU della scheda, la seconda con la rete. tutto il sistema è controllato da una SCO (sottosistema di controllo della scheda). l'adaptor esporta verso la CPU uno o più registri macchina (control status register). CSR è il mezzo di comunicazione tra SCO della scheda e CPU. l'host può controllare cosa accade in CSR con busy waiting o con interrupt. il trasferimento dati da adaptor a memoria e vv può avvenire con direct memory access o con programmed i/o. la direct memory access DMA si basa su una buffer descriptor list (array di buffer), così che la CPU possa mettere dati su questi buffer per poi segnalare la posizione su tale array in cui ha messo dati.

DEVICE DRIVER:

è una collezione di routine di OS che serve per ancorare il SO all'hardware sottostante specifico dell'adaptor.

------------------------------------------------------------------------------------------------

LEZIONE 8

SISTEMI DISTRIBUITI:

sistema che coinvolge più macchine. è un insieme di entità indipendenti e spazialmente separate, ognuna con un potere computazionale, che sono in grado di comunicare e coordinarsi per un obiettivo comune tale che appaia al suo user come un singolo sistema (es Local Area Network, Intranet, Internet, WWW). l'obiettivo primario è la condivisione di dati e risorse, che necessita quindi di sincronizzazione e coordinazione. il coordinamento presenta difficoltà a causa della concorrenza sia temporale che spaziale, dell'assenza di clock globale, della possibilità di failure e di ritardi non predicibili. altre difficoltà nel design di sistemi distribuiti sono: eterogeneità (diverse reti, hardware, SO, linguaggi di programmazione, implementazioni..), sicurezza (confidenzialità, integrità cioè protezione rispetto ad alterazioni o corruzioni, disponibilità cioè protezione contro interferenze che potrebbero impedire l'accesso alla risorsa), scalabilità, affidabilità, flessibilità, trasparenza. nei sistemi distribuiti si comunica in base a modelli client/server (un server centralizzato che gestisce richieste e risposte) o peer-to-peer (non c'è server centralizzato ma entità che comunicano allo stesso livello). tra l'applicazione e il SO c'è il middleware, che è ciò che permetterà alle macchine di comunicare fornendo modi uniformi di accesso alle risorse di sistema. le applicazioni distribuite e i servizi del middleware sono distribuite su tutte le macchine, ovvero sono proprie delle macchine su cui operano ma appaiono come se fossero funzionanti su una sola macchina (cioè l'applicazione funziona "in cima" ad ogni macchina e comunica solo con gli strati adiacenti di tale macchina, ma deve sembrare come se tutti i livelli applicativi di tutte le entità su cui il sistema distribuito agisce fossero appartenenti alla stessa macchina e funzionassero in modo organico).

CLIENT/SERVER COMPUTING:

è un tipo di distributed process particolarmente user friendly, con enfasi sulla centralizzazione di alcune operazioni come quelle di gestione del database, modulare e indirizzato alla connessione (binding). l'applicazione è svolta nel server, mentre nel client "appare" il risultato di tale applicazione e quindi il lato client avrà anche dei servizi di presentazione. i protocolli di comunicazione devono essere uguali all'interno dello stesso sistema. le applicazioni client/server sono di 4 tipi: host-based processing (tutto è gestito lato server, a parte l'interfaccia utente dell'applicazione), server-based processing (come l'host-based ma il lato client ha la presentation logic, e quindi un minimo di capacità di calcolo), cooperative processing (i dati sono lato server, l'applicazione è distribuita), client-based processing (il processing dell'applicazione è fatto tutto lato client). è possibile anche una struttura di applicazione client/server a tre livelli (Service-Oriented Architecture SOA), in cui sono presenti due livelli di server, con un server che comunica col client e ridistribuisce le richieste che gli provengono da altri server. i middleware comunicano tramite message passing.

REMOTE PROCEDURE CALL (RPC): funzioni che possono essere eseguite da remoto. necessitano di una standardizzazione tramite stub. possono essere sincrone (bloccanti, il client si ferma fino a che non ottiene il return delle procedure) o asincrone (non bloccanti). seguono politiche di "at least once" (si aspetta almeno un risultato come risposta), "at most once" (si richiede al più un risultato quindi in caso di errori si preferisce non inviare nessuna return) ed "exactly one" (è il caso più difficile da gestire poichè la risposta deve essere tracciata esattamente dal suo richiedente).

PARAMETER PASSING/REPRESENTATION: è la difficoltà maggiore è la rappresentazione dei dati, poichè c'è difficoltà nel passare dati tramite puntatori.

IOT: ogni oggetto connesso ad internet e con capacità di calcolo. sono composti da sensori, attuatori anche non fisici, microcontrollori, transricevitori, radio-frequency identification RFID. hanno poca RAM e ROM, non hanno unità di gestione della memoria.

------------------------------------------------------------------------------------------------

LEZIONE 9

SYSTEM AND NETWORK SECURITY:

la computer security è la protezione fornita da un sistema di calcolo con lo scopo di garantire e preservare confidenzialità, integrità (la data integrity si assicura che informazioni e programmi vengano modificati soltanto in modi specifici e autorizzati. la system integrity si assicura che un sistema performi nel modo in cui è stato inteso che funzioni), disponibilità di tutte le risorse del sistema informativo (cioè hardware, software, firmware, informazioni e telecomunicazioni). oltre a questi tre criteri fondamentali si aggiungono i concetti di autenticità, affidabilità (accountability).

CYBER ATTACKS:

attivi (per leggere, modificare, generare, distruggere informazioni), o passivi (per leggere informazioni). gli attacchi attivi ricadono in 4 categorie: replay (cattura passiva di informazioni che poi vengono ritrasmesse per ottenere effetti non autorizzati come identificazioni fittizie o danni a sistema), masquerade (un'entità finge di essere un'entità diversa), modification of messages (viene alterata parte di un messaggio autorizzato), denial of service (impedisce il normale uso di risorse comunicative).

CYBER ATTACKERS:

hackers, insiders, criminal enterprise, advanced persistent threat (state sponsored)

MALWARE:

software o componente di esso creata per causare danno o sfruttare risorse del computer target. alcuni tipi sono: virus (ha tre parti: infection mechanism, trigger, payload), worm, logic bomb, trojan horse, backdoor/trapdoor, platform independent code, exploit, auto-rooter, kit, spammer program, flooders, key logger, rootkit (programma che garantisce diritti di amministratore. può essere persistente, memory based, user mode o kernel mode), zombie, bot, spyware, adware.

------------------------------------------------------------------------------------------------

LEZIONE 10

i malware sfruttano vulnerabilità dei SO o dei programmi, come il buffer overflow che è un errore che si verifica spesso soprattutto con linguaggi di basso livello, e che rende facile spostarsi su zone di memoria diverse da quelle intended to access, che potrebbero contenere info sensibili o potrebbero dar modo ad un attacker di eseguire altro codice dannoso. i meccanismi di protezione da stack buffer overflow ricadono in due categorie: compile-time defenses (es. canaries) e stack protection mechanisms.

BOT:

programma che prende il controllo di un computer collegato ad internet per usare tale computer per lanciare altri attacchi che sono quindi difficili da ricondurre al creatore del bot. una volta entrato nel computer che intende controllare, può eseguire il codice dannoso in un momento successivo, all'attivazione da parte dell'attacker. un insieme di bot coordinati è detto botnet. tra gli usi dei bot ci sono: distributed denial of service, spamming, sniffing traffic, keylogging spreading new maleare, installing advertismet add-ons and browser helper objects, attacking internet relay chat, manipulating online polls/games.

PROTECTION TECHNIQUES:

CRYPTOGRAPHIC TOOLS: basata sui numeri casuali (pseudocasuali), la crittografia può essere simmetrica (due enti condividono una chiave che permette ad entrambi la decifrazione dei messaggi), verso la quale un eventuale attacco dovrà essere basato su crittoanalisi (complessa e pesante) o su attacchi brute-force, o asimmetrica (o a chiave publicca, public key). la cifratura nel caso di crittografia simmetrica può essere di tipo block o di tipo stream. la cifratura nel caso di chiave pubblica si basa su due chiavi: chi vuole ricevere genera una chiave pubblica e una privata in modo che quella pubblica sia utilizzabile da chiunque voglia cifrare un messaggio da inviargli; la chiave privata invece non è ricostruibile sulla base di quella pubblica, è ovviamente molto complessa; chi riceve è l'unico che conosce la chiave privata; la complessità computazionale e le dimensioni del messaggio cifrato sono entrambe molto elevate. alcuni tipi di public key algorithms sono quelli RSA, Diffie-Hellman key exchange (che sfrutta la crittografia asimmetrica esclusivamente per lo scambio della chiave per la crittografia simmetrica, evitando quindi tutti i costi di computazione e di dimensione del messaggio che deriverebbero da un uso continuato di crittografia asimmetrica), Elliptic curve cryptography (ECC, sicuro come RSA ma con chiavi più piccole), Homomorphic encription (HE, in cui si permette a chi non è in possesso della chiave privata di svolgere comunque calcoli su un messaggio che non può decriptare).

PASSWORD-BASED AUTHENTICATION: oltre alla cifratura è importante fare in modo che ad un sistema si acceda solo in seguito ad autenticazione tramite password. alle password inserite va applicato un hash (slow hash function, importante che sia estremamente lenta per rendere computazionalmente pesanti attacchi di tipo brute-force) per proteggerle quando vengono "memorizzate" nei database. oltre all'hash viene usato un "salt". il valore di questo sale viene memorizzato in chiaro.

TOKEN-BASED AUTHENTICATION: l'identificazione avviene in base a qualcosa che "si ha", tipo tessera, badge, microchip, smart card (es. NFC).

STATIC BIOMETRIC AUTHENTICATION: impronte digitali, geometria della mano, caratteristiche facciali, pattern di retina e iride, voiceprint, firma, biometrie comportamentali.

ACCESS CONTROL: è un tipo di politica di accesso alle risorse. le politiche in questa categoria sono di tipo: Discretionary AC, Mandatory AC, Role-based AC.

FIREWALLS: servono a bloccare i pacchetti illegittimi prima che entrino in una rete/LAN. sono di tipo: packet filtering, stateful inspection, application proxy, circuit-level proxy.

INTRUSION DETECTION: si basa sullo studio degli utenti autorizzati per rilevare attività insolite/anomalie in caso di intrusioni.

HONEYPOTS: sono sistemi esca che simulano la rete interna usati per captare attacchi malware in modo da studiare come bloccarli e come questi agiscono sul sistema per reagire in caso di intrusione sulla rete reale.