Sistemi Operativi Laboratorio

UNIX = è il nome di una famiglia di Sistemi Operativi, con diverse implementazioni per le varie architetture HW, derivati dallo UNIX AT&T Unix-like sistemi operativi progettati seguendo le direttive descritte per i sistemi UNIX Linux, Mac OSX, Android, etc… sono tutti Unix-like, non UNIX.
Obiettivo: portabilità delle applicazione a livello sorgente: : 1) Programmi C 2) Script di shell 3) Programmi in altri linguaggi.
La competizione di vari costruttori per il controllo dello Unix «Standard» ha creato una situazione piuttosto complessa. Standard principali: POSIX (IEEE dal 1988, poi ISO) «Portable Operating System Interface for Unix», XPG (X/Open, dal 1989) «X/Open Portability Guide», SVID (AT&T, 1989) «System V Interface Definition», OSF (Open Software Foundation)

ZOMBIE E DEAMON = il primo è un processo che ha terminato oppure è stato ucciso, ma non riesce a segnalare l'evento al padre il daemon (demone) sono processi che girano persistentemente in background e forniscono servizi al sistema (es: webserver, file server) disponibili in qualunque momento per servire più task o utenti

GESTIONE DEI PROCESSI: I processi normalmente eseguono in foreground e hanno tre canali standard connessi al terminale (stdin, stdout, stderr) I processi attivati con & eseguono in background e sono privi di stdin (es. ./a.out &). Un processo in foreground può essere sospeso con ^Z. I processi sospesi possono essere continuati sia in foreground che in background. I processi in background possono essere riportati in foreground
SHELL: Offre due vie di comunicazione con il SO Interattivo (comandi) shell script. SCRIPT DI SHELL: è un file (di testo) costituito da una sequenza di comandi. La shell non è parte del kernel del SO, ma è un normale processo utente Ciò permette di poter modificare agevolmente l'interfaccia verso il sistema operativo.bRi-direzione dell'I/O (stdin, stdout, stderr), Pipeline dei comandi, Editing e history dei comandi, Aliasing. Gestione dei processi (foreground, background sospensione e continuazione), Linguaggio di comandi, Variabili di shell
SHELL DISPONIBILI: Bourne shell (sh): La shell originaria, preferita nella programmazione sistemistica C-shell (csh)
LA SHELL DI BERKELEY,: ottima per l'uso interativo e per gli script non di sistema
KORN SHELL (KSH): La Bourne sh riscrita dall'AT&T per assomigliare alla C-shell
TAHOE (TCSH): Dal progetto Tahoe, una C-shell migliorata

INTERFACCIA TRAMITE SYSTEM CALL: L'accesso al kernel è permesso soltanto tramite le system call, che permettono di passare all'esecuzione in modo kernel. Dal punto di vista dell'utente, l'interfaccia tramite system call funziona come una normale chiamata C. In realtà è più complicato: Esiste una system call library contenente funzioni con lo stesso nome della system call. Le funzioni di libreria cambiano il modo user in modo kernel e fanno sì che il kernel esegua il vero e proprio codice delle system call. La funzione di libreria passa un identificatore, unico, al kernel, che identifica una precisa system call. Simile a una routine di interrupt (detta operating system trap).

MAKE: Make e' un utility largamente utilizzato in ambiente UNIX (e non solo) per compilare programmi con decine/centinaia di file sorgenti, Rapidamente! Idea: ricompiliamo solo i file sorgenti del programma che sono stati modificati. Vantaggio (uno dei tanti), abbraviamo il tempo di compilazione dell'intero programma

MAKEFILE: Makefile e' uno script utilizzato per automatizzare la compilazione di un programma attraverso l'utility Make. Un makefile contine macro e rules
MAKEFILE - MACRO: Una macro in un makefile funziona come una variabile. E' defitita come: <tab><nome> = <valore> • Esempi definizione: OBJS = main.o io.o CC = clang • Esempio modificata: OBJS += support.o CC = gcc • Esempio lettura: @echo $(OBJS) (stampa -> main.o io.o support.o) @echo ${CC} (stampa -> gcc)
MAKEFILE - RULES: Una rule (regola) definisce come e quando ricompilare l'obiettivo della regola (rule's target). La regola definisce i requisiti del target, e la ricetta da usare per creare o aggiornare il target.Definizione <target> : <requisiti> <TAB> ricetta (-> come ricompilare il target)• Esempio definizione: mioprogr: main.o io.o $(LD) $(LDFLAGS) –o mioprogr main.o io.o. Se modifico main.c, make ricompila solomain.c per rigenerare main.o e "ri-linkare" mioProg
Quando ricompilare un TARGET. Un target è ricompilato se il data della sua ultima modifica è precendete a quella di almeno una delle sue dipendenze. Primo target e' il target di default, eseguito quando make e' chiamatosenza parametri omi tipici di target – all: compila e linka tutti i target – clean: cancella tutti i file generati – install: installa tutti i file generati – help: stampa una lista di target disponibili – doc: genera la documentazione
PHONY = Colleziona come dipendenze i target che non sono nomi di file generati (e.g., all, clean, help, etc..)
MAKEFILE: ESECUZIONE: Eseguire make con Makefile, usando il target di default make • Eseguire make in parallelo (opzionale: spcificare il numero massimo di thread in parallelo) make -jN• Massimo N thread in parallelo – E.g. per avere massimo 4 thread: make -j4 mentre – make -j • Nessun limite massimo di thread in parallelo. • Eseguire make con MyMake come file di configurazione: make -f MyMake • Eseguire make per costruire un target specifico (e.g., app.x) make app.x

DEBUGGING: • Verifica e debugging: Una delle fasi più complesse durante lo sviluppo SW, l'attività che richiede più tempo (70-80%) • Fondamentale usare strumenti per – Semplificare il debugging del codice – Velocizzare la correzione di bug – Evitare comportamenti strani dovuti a metodi bufferizzati nel caso di errari di segmentazione.
• Un debugger permette: – Sospensione dell'esecuzione di un programma in un punto specifico
– Eseguire un programma passo dopo passo – Ispezione i valori delle variabili a run-time

GNU DEBUGGER: • GDB (The GNU Debugger): Libero e open-source, debugger standard usato con gcc. Richiede di arricchire l'eseguibile: gcc -g compila con informazioni di debug
• gcc -ggdb come -g ma esplicitamente per gdb. Strumento da linea di comando • DDD è la più famosa interfaccia grafica. Non più mantenuta • cgdb è un'interfaccia testuale basata su curses

DB: COMANDI BASE: • Caricare un eseguibile: gdb my_exec.x gdb --args my_exec.x <program args>
• Chiudere gdb : quit / q • Assegnare eventuali argomenti all'eseguibile: set args <arguments>
• Aprire l'help di un argomento (e.g., di un comando) help <argomento>. • Breakpoint: Marcatura di un punto nel codice sorgente – L'esecuzione del programma viene sospesa quando viene raggiunto un breakpoint – Utile per ispezionare un punto del programma a run-time – Sintassi:
break / b <file:line>
break / b <methodName>
delete <numero> (rimuove il breakpoint #<numero>

GDB ESPRESSIONI • Monitoraggio: Watch point – Specifica di un espressione
• Non un punto nel codice – Utile per controllare l'evoluzione di una variabile
– Sintassi: watch <espressione>
• Valutazione: Print: Stampa il risultato di un espressione, utilizza il valore corrente delle variabili. Utile per monitorare variabili – Sintassi: print / p <espressione>

GDB: ISPEZIONE DELLO STACK: • Backtrace: Stampa la sequenza di chiamate a funzione, permette di capire l'ordine di esecuzione. Sintassi: backtrace / bt • Frame: stampa informazioni sullo stack di metodi corrente – Sintassi: frame / f descrizione corta, info frame / info f descrizione lunga – Cambiare frame: frame <numero> / f <numero>

GDB: ESECUZIONE: Eseguire un programma dall'inizio: run / r [<argomenti>]
• Continuare l'esecuzione dal punto raggiunto continue / c • Eseguire la prossima riga atomicamente next / n • Eseguire la prossima istruzione. Eventualmente entrando in metodi/funzioni step / s • Ripeti l'ultimo comando \return

DOCUMENTAZIONE: • Tipi di documentazione: Per gli utenti: Manuali, guide, tutoria, etc...
• Documentazione di API (nel caso di librerie): Per gli sviluppatori •Documentazione di API • Documentazione dell'implementazione • Altro: Use cases, UML, specifiche, etc...
Documentazione del codice sorgente: • Commenti delle API:Generazione automatica della documentazione, richiede l'utilizzo di un formato preciso. Linguaggi per la generazione della documentazione disponibili per ogni linguaggio – Obiettivo: generare la documentazione «ufficiale» del progetto
• Commenti dei dettagli implementativi: Immersi nel codice sorgente: possono utilizzare semplice linguaggio naturale. Obiettivo: rendere il codice sorgente comprensibile a terze parti o «riletture» future

DOXYGEN: Tool per la generazione automatica della documentazione di API, Multi-piattaforma, Libero ed open-source. Supporta diversi linguaggi di programmazione: C, C++, Java • Supporta diversi formati di output: HTML, Latex, RTF. Usare Doxygen • Generare un file di configurazione (Doxyfile)
Doxygen -g • Configurare il Doxyfile emacs Doxyfile • Eseguire doxygen con il Doxyfile di default Doxygen • Eseguire doxygen con un file di configurazione specifico doxygen <config file>

CONFIGURAZIONE DI DOXYGEN: La configurazione avviene settando parametri nel Doxyfile Parametri principali. • PROJECT_NAME (e.g., MyLib) = Nome del progetto • OUTPUT_DIRECTORY (e.g. doc)= Directory dove salvare la documentazione. • INPUT / FILE_PATTERNS (e.g. src) = Lista dei file (o directory) di ingresso. • RECURSIVE (e.g. YES) se YES, allora visita le cartelle ricorsivamente. • EXCLUDE / EXCLUDE_PATTERNS (*.java) = File da escludere. • GENERATE_* (e.g. GENERATE_HTML) = Specifica del formato di uscita
FORMATO DEI COMMENTI: commenti in doxygen devono rispettare una forma particolare: 1) /** Commento parserizzato da Doxygen */ 2)/// Commento parserizzato da Doxygen – Altri formati di commento vengono ignorati
INFORMAZIONI SPECIALI PER LA DOCUMENTAZIONE: vengono date tramite tag speciali nella forma: @tag \tag
TAG PRINCIPALI DI DOXYGEN: 1) @brief <commento> Un breve commento sulla parte di codice seguente. Utilizzato prima di blocchi di codice 2) @param <nome parametro> <commento> – <commento> spiega il significato/utilizzo del parametro <nome parametro>: Utilizzato per commentare dichiarazioni di metodi 3) @return <comment> Commenta il comportamento del valore di ritorno di un metodo: Utilizzato per commentare dichiarazioni di metodi 4)@throw <nome eccezione> <commento> Commenta eccezioni lanciate da metodi (non per C) : Utilizzato per commentare dichiarazioni di metodi

TAG DI DOCUMENTAZIONE GLOBALE: 1)Informazioni sul contenuto di un file /** @file * <comment > */ 2) Raggruppare metodi con una connessione logica /** @name <groupName>*/ /*@{ */<methodsWithTheirDocumentation> /*@} */
Esempio
/** @name List accessors. */
/*@{ */
/** @brief Gets the element at given position.
* Linear complexity.
* @param l The list.
* @param pos The position.
* @return The stored element.
*/
void * getListElement(List * l, int pos);
/*@} */

UNIX E IPC: • ipcs: riporta lo stato di tutte le risorse, o selettivamente, con le seguenti opzioni: -s informazioni sui semafori; -m informazioni sulla memoria condivisa; -q informazioni sulle code di messaggi. • ipcrm: elimina le risorse (se permesso) dal sistema. Nel caso di terminazioni anomale, le risorse possono rimanere allocate Le opzioni sono quelle ipcs Va specificato un ID di risorsa, come ritornato da ipcs

THREAD: Thread multiple eseguono all'interno dello stesso spazio di indirizzamento. Conseguenze Modifiche effettuate da una thread ad una risorsa condivisa sono visibili da tutte le altre thread. Possibile lettura e scrittura sulla stessa locazione di memoria. È necessaria esplicita sincronizzazione da parte del programmatore. Thread sono paritetiche (no gerarchia). Perché thread? Motivo essenziale: prestazioni. Costo rispetto alla creazione di un processo molto inferiore. Costo di IPC molto inferiore. Aumento del parallelismo nelle applicazioni Quali thread? • Numerose implementazioni: Implementazione standard − POSIX thread (pthread)◦ Definite come API in C Implementate nella libreria libpthread.a e utilizzabili includendo lo header file pthread.h − In Linux ◦ gcc <file.c> -lpthread ◦ <file.c> deve includere pthread.h
PROGRAMMI MULTI-THREADED: Per sfruttare i vantaggi delle thread, un programma deve poter essere organizzato in una serie di task indipendenti eseguibili in modo concorrente. Esempi: 1) Procedure che possono essere sovrapposte nel tempo o eseguite in un qualunque ordine 2)Procedure che si bloccano per attese potenzialmente lunghe 3)Procedure che devono rispondere ad eventi asincroni 4)Procedure che sono più/meno importanti di altre Modelli tipici di programmi multithreaded Manager/worker 5)Thread manager gestisce gli input e assegna operazioni a altre threadworker
6) Pipeline; Un task viene spezzato in una serie di sottooperazioni (implementate da thread distinte) da eseguire in serie e in modo concorrente 7)Peer: Simile al modello manager/worker Dopo che la thread principale crea le altre, partecipa al lavoro

CONDITION VARIABLE: Meccanismo addizionale di sincronizzazione. Permettono di sincronizzare thread in base ai valori dei dati senza busy waiting. • Senza condition variable, le thread devono testare in busy waiting (eventualmente in una sezione critica) il valore della condizione desiderata • Le variabili condition sono sempre associate all'uso di un mutex