*Università di Pisa -- Dipartimento di Informatica*
*Corso di Laurea in Informatica*

# Progetto di Laboratorio di Sistemi Operativi

a.a. 2020-21
per Corso A e Corso B
Data di pubblicazione: 4 Maggio 2021

Data di revisione: 14 Maggio 2021
- aggiunta l'API '*readNFiles*' mancante.

## 1. Introduzione

Lo studente deve realizzare un ***file storage server*** in cui la memorizzazione dei file avviene in **memoria principale**. La capacità dello storage è fissata all'avvio e non varia dinamicamente durante l'esecuzione del server. Per poter leggere, scrivere o eliminare file all'interno del file storage, il client deve connettersi al server ed utilizzare una API che dovrà essere sviluppata dallo studente (come descritto nel seguito). Il server esegue tutte le operazioni richieste dai client operando sempre e solo in memoria principale e mai sul disco.

La capacità del file storage, unitamente ad altri parametri di configurazione, è definita al momento dell'avvio del server tramite un file di configurazione testuale.

Il *file storage server* è implementato come un singolo processo multi-threaded in grado di accettare connessioni da multipli client. Il processo server dovrà essere in grado di gestire adeguatamente alcune decine di connessioni contemporanee da parte di più client.

Ogni client mantiene **una sola connessione** verso il server sulla quale invia una o più richieste relative ai file memorizzati nel server, ed ottiene le risposte in accordo al protocollo di comunicazione "richiesta-risposta". Un file è identificato univocamente dal suo **path assoluto**.

## 2. Il server

Il file storage server deve essere implementato come un **singolo processo multi-threaded** secondo lo schema "**master-worker**" in cui il numero di threads worker è fissato all'avvio del server sulla base delle informazioni di configurazione, ed ogni worker può gestire richieste di client diversi. Eventuali altri thread di "supporto" (oltre al thread main) possono essere presenti nel sistema a discrezione dello studente.

All'avvio del server, il thread main legge il file di configurazione chiamato "*config.txt"* che può risiedere in una qualsiasi directory del sistema (NOTA: tale file non è un header file quindi non va incluso ma va parsato opportunamente!). Il formato di tale file non viene specificato e deve essere deciso e descritto nella relazione dallo studente. Il file conterrà tutti i parametri che definiscono la configurazione del server, come ad esempio il numero di thread workers, la dimensione dello spazio di memorizzazione, il nome del socket file, ed altri parametri che lo studente ritiene utili.

Il processo server accetta connessioni di tipo **AF_UNIX** da parte dei client. Il nome del file associato al socket è pubblico e comunque conosciuto ai client (ad esempio "-f /tmp/LSOfiletorage.sk"). Tale processo rimane sempre attivo in attesa di nuove connessioni da parte di nuovi client oppure in attesa di richieste di operazioni da eseguire inviate attraverso le connessioni attive.

Il server termina alla ricezione dei segnali **SIGINT**, **SIGQUIT** e **SIGHUP**. Nel caso di ricezione di uno dei due segnali SIGINT o SIGQUIT, il server termina il prima possibile, ossia non accetta più nuove richieste da parte dei client connessi chiudendo tutte le connessioni attive (dovrà comunque generare il sunto delle statistiche, descritto nel seguito). Nel caso di ricezione del segnale SIGHUP, non vengono più accettate nuove richieste da parte di nuovi client, ma vengono servite tutte le richieste dei client connessi al momento della ricezione del segnale, quindi il server terminerà solo quando tutti i client connessi chiuderanno la connessione.

Lo **spazio** di memorizzazione del server ha una capacità limitata, sia nel **numero** massimo di file memorizzabili che come **spazio** effettivo di memorizzazione, ad esempio, il server potrebbe memorizzare al più 100 file e fino al raggiungimento di 100 Mbytes di spazio (tali informazioni sono passate al server attraverso il file di config).

Si chiede allo studente di gestire tale spazio di memorizzazione come una ***cache di file***. Se il server non ha capacità sufficiente ad accettare nuovi file, dovrà **espellere** file per guadagnare capacità. I file espulsi devono essere inviati al **client** la cui richiesta ne ha causato l'espulsione (il quale li gestisce come descritto più avanti), e **non sono più accessibili** in richieste successive.

Sono possibili diverse politiche di rimpiazzamento dei file nella cache del server a seguito di "capacity misses", lo studente dovrà implementare almeno la politica di rimpiazzamento FIFO, mentre politiche più sofisticate (come ad esempio LRU, LFU, etc..) possono essere implementate opzionalmente.

Durante l'esecuzione, il processo server storage effettua il logging, su un **file di log** specificato nel file di configurazione, di *tutte le operazioni* che vengono richieste dai client o di gestione interna del server (ad esempio, l'arrivo di una nuova connessione, il nome del file letto e l'ammontare dei byte restituiti al client, la quantità di dati scritti, se è stata richiesta una operazione di lock su un file, se è partito l'algoritmo di rimpiazzamento dei file della cache e quale vittima è stata selezionata, etc.). La scelta del formato del file di log è lasciata allo studente.

Al termine dell'esecuzione, il server stampa sullo *standard output* in modo formattato un sunto delle operazioni effettuate durante l'esecuzione, e tra queste almeno le seguenti informazioni:

1. numero di file massimo memorizzato nel server;
2. dimensione massima in Mbytes raggiunta dal file storage;
3. numero di volte in cui l'algoritmo di rimpiazzamento della cache è stato eseguito per selezionare uno o più file "vittima";
4. lista dei file contenuti nello storage al momento della chiusura del server.

## 3. Il client

Il programma client è un programma separato dal server con una propria funzione main. E' costruito per inviare richieste per creare/rimuovere/aprire/scrivere/… file nel/dal file storage server esclusivamente attraverso l'API descritta nella sezione successiva. Il programma client invia richieste al server sulla base degli argomenti specificati sulla riga di comando. In generale, il protocollo tra client e server è di tipo "richiesta-risposta". Particolarmente degno di nota è il caso di richiesta di scrittura di un file che provoca un "capacity miss" e l'espulsione dallo storage di un file che era stato precedentemente modificato (ad esempio dall'operazione 'appendToFile' -- vedere la sezione successiva). In questo caso, il server risponde alla richiesta di operazione di scrittura inviando il file espulso dalla cache. Sulla base degli argomenti a linea di comando specificati per il client, tale file verrà buttato via oppure salvato in una opportuna directory (vedere l'opzione '-D').

Il client accetta un certo numero di argomenti da linea di comando. Tra questi, almeno i seguenti:

***-h*** : stampa la lista di tutte le opzioni accettate dal client e termina immediatamente;

***-f filename*** : specifica il nome del socket AF_UNIX a cui connettersi;

***-w dirname[,n=0]*** : invia al server i file nella cartella 'dirname', ovvero effettua una richiesta di scrittura al server per i file. Se la directory 'dirname' contiene altre directory, queste vengono visitate ricorsivamente fino a quando non si leggono 'n' file; se n=0 (o non è specificato) non c'è un limite superiore al numero di file da inviare al server (tuttavia non è detto che il server possa scriverli tutti).

***-W file1[,file2]***: lista di nomi di file da scrivere nel server separati da ',';

***-D dirname*** : cartella in memoria secondaria dove vengono scritti (lato client) i file che il server rimuove a seguito di capacity misses (e che erano stati modificati da operazioni di scrittura) per servire le scritture richieste attraverso l'opzione '-w' e '-W'. L'opzione '-D' deve essere usata quindi congiuntamente all'opzione '-w' o '-W', altrimenti viene generato un errore. Se l'opzione '-D' non viene specificata, tutti i file che il server invia verso il client a seguito di espulsioni dalla cache, vengono buttati via. Ad esempio, supponiamo i seguenti argomenti a linea di comando "-w send -D store", e supponiamo che dentro la cartella 'send' ci sia solo il file 'pippo'. Infine supponiamo che il server, per poter scrivere nello storage il file 'pippo' deve espellere il file 'pluto' e 'minni'. Allora, al termine dell'operazione di scrittura, la cartella

'store' conterrà sia il file 'pluto' che il file 'minni'. Se l'opzione '-D' non viene specificata, allora il server invia sempre i file 'pluto' e 'minni' al client, ma questi vengono buttati via;

***-r file1[,file2]*** : lista di nomi di file da leggere dal server separati da ',' (esempio: **-r pippo,pluto,minni**);

***-R [n=0]*** : tale opzione permette di leggere 'n' file qualsiasi attualmente memorizzati nel server; se n=0 (o non è specificato) allora vengono letti tutti i file presenti nel server;

***-d dirname*** : cartella in memoria secondaria dove scrivere i file letti dal server con l'opzione '-r' o '-R'. L'opzione -d va usata congiuntamente a '-r' o '-R', altrimenti viene generato un errore; Se si utilizzano le opzioni '-r' o '-R' senza specificare l'opzione '-d' i file letti non vengono memorizzati sul disco;

***-t time*** : tempo in millisecondi che intercorre tra l'invio di due richieste successive al server (se non specificata si suppone -t 0, cioè non c'è alcun ritardo tra l'invio di due richieste consecutive);

***-l file1[,file2]*** : lista di nomi di file su cui acquisire la mutua esclusione;

***-u file1[,file2]*** : lista di nomi di file su cui rilasciare la mutua esclusione;

***-c file1[,file2]*** : lista di file da rimuovere dal server se presenti;

***-p*** : abilita le stampe sullo standard output per ogni operazione. Le stampe associate alle varie operazioni riportano almeno le seguenti informazioni: tipo di operazione, file di riferimento, esito e dove è rilevante i bytes letti o scritti.

Gli argomenti a linea di comando del client possono essere ripetuti più volte (ad eccezione di '-f', '-h', '-p'). Ogni argomento va interpretato come una o più richieste che il client invia al server. La scelta di come gestire le richieste da parte dei client è lasciata allo studente, tenendo conto, però, che argomenti da linea di comando distinti corrispondono a richieste distinte al server. Ad esempio, la richiesta "-r file1 -d /dev/null -w ./mydir", si traduce in una richiesta di lettura di file1 ed una o più richieste di scrittura per i file contenuti nella directory ./mydir.

Il processo client, qualora debba inviare più richieste al processo server, aspetta che una richiesta sia stata completata dal server (protocollo di comunicazione "richiesta-risposta"). Questo vuol dire che il server risponde sempre ad una richiesta con un messaggio che può contenere o i dati richiesti dal client (ad esempio, per richieste di lettura), oppure una notifica di esito dell'operazione (ad esempio, successo dell'operazione, oppure fallimento con eventuale messaggio o codice di errore associato).

## 4. Interfaccia per interagire con il file server (API)

***- int openConnection(const char* sockname, int msec, const struct timespec abstime)***
Viene aperta una connessione AF_UNIX al socket file *sockname*. Se il server non accetta immediatamente la richiesta di connessione, la connessione da parte del client viene ripetuta dopo '*msec*' millisecondi e fino allo scadere del tempo assoluto '*abstime'* specificato come terzo argomento. Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int closeConnection(const char* sockname)***
Chiude la connessione AF_UNIX associata al socket file *sockname*. Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int openFile(const char* pathname, int flags)***
Richiesta di apertura o di creazione di un file. La semantica della *openFile* dipende dai flags passati come secondo argomento che possono essere O_CREATE ed O_LOCK. Se viene passato il flag O_CREATE ed il file esiste già memorizzato nel server, oppure il file non esiste ed il flag O_CREATE non è stato specificato, viene ritornato un errore. In caso di successo, il file viene sempre aperto in lettura e scrittura, ed in particolare le scritture possono avvenire solo in append. Se viene passato il flag O_LOCK (eventualmente in OR con O_CREATE) il file viene aperto e/o creato in modalità locked, che vuol dire che l'unico che può leggere o scrivere il file '*pathname*' è il processo che lo ha aperto. Il flag O_LOCK può essere esplicitamente resettato utilizzando la chiamata *unlockFile*, descritta di seguito.
Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int readFile(const char* pathname, void** buf, size_t* size)***

Legge tutto il contenuto del file dal server (se esiste) ritornando un puntatore ad un'area allocata sullo heap nel parametro '*buf*', mentre '*size*' conterrà la dimensione del buffer dati (ossia la dimensione in bytes del file letto). In caso di errore, '*buf*'e '*size*' non sono validi. Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int readNFiles(int N, const char* dirname)***
Richiede al server la lettura di 'N' files *qualsiasi* da memorizzare nella directory '*dirname*' lato client. Se il server ha meno di 'N' file disponibili, li invia tutti. Se N<=0 la richiesta al server è quella di leggere tutti i file memorizzati al suo interno. Ritorna un valore maggiore o uguale a 0 in caso di successo (cioè ritorna il n. di file effettivamente letti), -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int writeFile(const char* pathname, const char* dirname)***
Scrive tutto il file puntato da *pathname* nel file server. Ritorna successo solo se la precedente operazione, terminata con successo, è stata *openFile(pathname, O_CREATE| O_LOCK)*. Se *'dirname*' è diverso da NULL, il file eventualmente spedito dal server perchè espulso dalla cache per far posto al file '*pathname*' dovrà essere scritto in '*dirname*'; Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int appendToFile(const char* pathname, void* buf, size_t size, const char* dirname)***
Richiesta di scrivere in append al file '*pathname*' i '*size*' bytes contenuti nel buffer '*buf*'. L'operazione di append nel file è garantita essere atomica dal file server. Se *'dirname*' è diverso da NULL, il file eventualmente spedito dal server perchè espulso dalla cache per far posto ai nuovi dati di '*pathname*' dovrà essere scritto in '*dirname*'; Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int lockFile(const char* pathname)***
In caso di successo setta il flag O_LOCK al file. Se il file era stato aperto/creato con il flag O_LOCK e la richiesta proviene dallo stesso processo, oppure se il file non ha il flag O_LOCK settato, l'operazione termina immediatamente con successo, altrimenti l'operazione non viene completata fino a quando il flag O_LOCK non viene resettato dal detentore della lock. L'ordine di acquisizione della lock sul file non è specificato. Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int unlockFile(const char* pathname)***
Resetta il flag O_LOCK sul file '*pathname*'. L'operazione ha successo solo se l'owner della lock è il processo che ha richiesto l'operazione, altrimenti l'operazione termina con errore. Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int closeFile(const char* pathname)***
Richiesta di chiusura del file puntato da '*pathname*'. Eventuali operazioni sul file dopo la *closeFile* falliscono. Ritorna 0 in caso di successo, -1 in caso di fallimento, *errno* viene settato opportunamente.

***- int removeFile(const char* pathname)***
Rimuove il file cancellandolo dal file storage server. L'operazione fallisce se il file non è in stato locked, o è in stato locked da parte di un processo client diverso da chi effettua la *removeFile.*

## 5. Makefile

Il progetto dovrà includere un Makefile avente, tra gli altri, i target *all* per generare gli eseguibili del programma server e del programma client; *clean/cleanall* per ripulire la directory di lavoro dai file generati, moduli oggetto, socket file, logs, librerie, etc.; tre target di test: *test1, test2* e *test3.*

Il target *test1*: deve far partire il processo server con una dimensione dello storage pari a 10000 file ed una capacità in bytes di 128MBytes ed un pool di threads worker pari ad 1. Il processo server deve essere eseguito con *valgrind* facendo precedere il comando di avvio del server dalla stringa "*valgrind –leak-check=full*". Al termine dell'esecuzione dovrà risultare che il numero di malloc ed il numero di free devono coincidere e non ci devono essere errori. Il target del Makefile dopo aver avviato il server, esegue uno script Bash che lancia uno o più processi client configurati per testare singolarmente tutte le operazioni implementate dal server, con un ritardo tra diverse operazioni di 200 millisecondi. Il server deve essere terminato con il segnale SIGHUP al termine dell'esecuzione dello script. Lo script Bash lancia tutti client con l'opzione '-p'.

Il target *test2*: deve eseguire il server (senza usare valgrind) con una dimensione dello storage pari a 10 file ed una capacità in bytes di 1MBytes ed un pool di worker threads pari a 4. Anche in questo caso viene lanciato uno script Bash che esegue più processi client che inviano richieste al server ed al termine viene inviato un segnale di tipo SIGHUP. Lo scopo del test è quello di dimostrare (tramite l'output prodotto dal server al termine dell'esecuzione e l'output prodotto dai client con l'opzione '-p') la funzionalità dell'algoritmo di rimpiazzamento dei file nella cache del server.

Il target test3: deve eseguire il server (senza usare valgrind) con un pool di thread pari a 8 con una dimensione dello storage di 100 file ed una capacità in bytes di 32MBytes. Viene lanciato uno script Bash che esegue ininterrottamente un numero di processi client (senza usare l'opzione '-p') in modo tale che ce ne siano sempre almeno 10 in esecuzione contemporaneamente. Ogni processo client invia più richieste (almeno 5 richieste) al server con '-t 0'. Il test dopo 30 secondi si interrompe ed invia il segnale SIGINT al server e termina (senza aspettare l'eventuale terminazione dei processi client ancora attivi). Lo scopo di questo test è quello di eseguire uno stress test del server. Il test si intende superato se non si producono errori a run-time lato server e se il sunto delle statistiche prodotto dal server riporta "valori ragionevoli" (cioè, non ci sono valori negativi, valori troppo alti, campi vuoti, etc...).

## 6. Script Bash per le statistiche

Lo studente dovrà realizzare uno script Bash con nome *statistiche.sh* che effettua il parsing del file di log prodotto dal processo server durante la sua esecuzione, e produce un output formattato contenente le operazioni effettuate. Nello specifico, lo script produce sullo standard output, almeno le seguenti informazioni.

- n. di read e size media delle letture in bytes
- n. di write e size media delle scritture in bytes
- n. di operazioni di lock
- n. di operazioni di open-lock
- n. di operazioni di unlock
- n. di operazioni di close
- dimensione massima in Mbytes raggiunta dallo storage
- dimensione massima in numero di file raggiunta dallo storage
- numero di volte in cui l'algoritmo di rimpiazzamento della cache è stato eseguito per selezionare un file vittima
- n. di richieste servite da ogni worker thread
- massimo n. di connessioni contemporanee.

## 7. Sviluppo opzionale

Oltre all'implementazione di politiche di rimpiazzamento della cache più sofisticate rispetto alla politica FIFO richiesta, lo studente può, opzionalmente, implementare la memorizzazione dei file nella cache in formato compresso. In altri termini, il file viene ricevuto e spedito dai/ai client in formato non compresso, mentre la memorizzazione internamente al server avviene in formato compresso. Il formato della compressione è lasciato come libera scelta dello studente.

## 8. Note finali

Ci si attende che tutto il codice sviluppato sia, per quanto possibile, conforme POSIX. Eventuali eccezioni (come ad esempio l'uso di estensioni GNU) devono essere documentate nella relazione di accompagnamento e/o nel file in cui ne viene fatto uso.

La consegna dovrà avvenire, entro i termini previsti per la consegna in ogni appello, attraverso l'upload sul portale Moodle del corso di laboratorio di un archivio con nome ***nome_cognome-CorsoX*.tar.gz** contenente tutto il codice necessario per compilare ed eseguire il progetto (CorsoX sarà CorsoA o CorsoB a seconda del corso di appartenenza

dello studente). Scompattando l'archivio in una directory vuota, dovrà essere possibile eseguire il comandi *make* (con target di default) per costruire tutti gli eseguibili. *make test1, make test2* e *make test3* per compilare ed eseguire i test corrispondenti e vederne i risultati come da specifica.

L'archivio dovrà anche contenere al suo interno una relazione in formato PDF (di massimo 5 pagine – Times New Roman, 11pt, margini di 2cm formato A4, interlinea singola) in cui sono descritti gli aspetti più significativi dell'implementazione del progetto e le scelte progettuali fatte. Il progetto può essere realizzato su un qualsiasi sistema Linux a 64bit multi-cores. In ogni caso, i test devono girare senza errori almeno sulla macchina virtuale Xubuntu fornita per il corso e configurata con almeno 2 cores (la VM non deve essere aggiornata rispetto alla versione presente sul sito didawiki del corso).

Il codice scritto deve essere adeguatamente documentato, in particolare per le parti più significative al fine di facilitarne la comprensione. L'utilizzo di librerie di "terze parti" è consentito solo se: 1. tutto il materiale viene fornito nell'archivio consegnato; 2. tutto quanto necessario viene compilato automaticamente dai targets del Makefile; 3. è riportato nella relazione associata al progetto quale parte del codice di terze parti è stato utilizzato con gli opportuni riferimenti web e copyright notes; 4. sia indicato all'inizio degli eventuali file di include utilizzati i credits previsti dall'autore del software.

Allo studente che svilupperà il codice in un repository Git pubblico (ad es. github) con una commit history ragionevole (cioè non un unico commit grosso o pochissimi commit grossi) verrà assegnato un bonus di 2 punti extra sulla valutazione del progetto. Il link al repository Git dovrà essere riportato nella relazione finale.

Infine, si ricorda che per sostenere l'esame **è necessario iscriversi** sul portale esami, e che la data pubblicata sul portale esami relativamente all'appello di Sistemi Operativi e Laboratorio si riferisce alla data ultima di consegna del progetto per un dato appello.

## PER CHI CONSEGNA IL PROGETTO ENTRO L'APPELLO DI LUGLIO 2021

Per chi consegna il progetto entro l'appello di Luglio 2021 (compreso), **è possibile** consegnare una versione **ridotta e semplificata del progetto** nel modo seguente rispetto a quanto specificato nelle sezioni precedenti:

- Il server non deve produrre un file di log delle operazioni effettuate e lo script *statistiche.sh* non si deve realizzare;
- Il test che ha come target del Makefile *test3* non deve essere realizzato. Il Makefile avrà solamente i target test1 e test2.
- Le operazioni *lockFile* ed *unlockFile* non si devono realizzare.
- L'opzione del client '-D' non è supportata; cioè gli eventuali file vittima dell'algoritmo di rimpiazzamento della cache vengono tolti dal file di storage senza essere inviati al client per essere memorizzati nella cartella specificata con l'opzione '-D'.

Per le consegne entro Luglio 2021 queste funzionalità sono da considerarsi **opzionali:** è possibile consegnare la versione semplificata, o realizzare alcune o tutte queste funzionalità (specificandolo nel report). Chiaramente eventuali funzionalità opzionali vengono valutate positivamente e contribuiscono al raggiungimento di un voto eccellente.