# Sistemi Operativi

## Specifiche di Progetto di Laboratorio

Luca Geretti
luca.geretti@univr.it
Andrea Masini
andrea.masini@univr.it

Università di Verona
Dipartimento di Informatica

2024/2025

# Tema del progetto

## Un servizio di calcolo della impronta SHA-256 di file multipli

Servizio realizzabile in una a scelta fra tre possibili tecnologie di implementazione:

1. Semafori, code di messaggi e memoria condivisa in Linux / macOS
2. Pthread con monitor e FIFO in Linux / macOS
3. Code di messaggi in MentOS orientato ad aspetti di schedulazione

# Cos'è SHA-256?

SHA sta per Secure Hash Algorithm ed appartiene ad una famiglia di algoritmi che convertono una stringa di byte di lunghezza arbitraria in una impronta (digest in inglese) di lunghezza fissata in bit. Nel nostro caso si tratta di una impronta di 256 bit, ovvero 32 byte rappresentabili visivamente come una stringa di 64 simboli esadecimali.

Viene utilizzata per sintetizzare contenuti informativi complessi in un formato compatto: è estremamente improbabile che due input diversi abbiano la stessa impronta. Usi classici sono per indicizzazione di dati tramite hashing oppure per verifica di integrità, per esempio controllando che l'impronta di un file scaricabile pubblicata ufficialmente combaci con quella computata sulla copia scaricata del file.

# Struttura generale del progetto

L'obiettivo è anzitutto realizzare un server che permetta multiple computazioni di impronte SHA-256. Il tempo di calcolo per impronta è proporzionale al numero di byte dell'ingresso (ovvero il file), dipendente dalla piattaforma e dalla implementazione dell'algoritmo ($\sim$1 secondo per 70 MB su laptop moderno nella implementazione OpenSSL, $\sim$1 secondo per 30 KB su MentOS in QEMU). Successivamente va realizzato un client che invii l'informazione di file di input al server e riceva l'impronta risultante appena computata.

# Struttura generale del progetto

L'obiettivo è anzitutto realizzare un server che permetta multiple computazioni di impronte SHA-256. Il tempo di calcolo per impronta è proporzionale al numero di byte dell'ingresso (ovvero il file), dipendente dalla piattaforma e dalla implementazione dell'algoritmo ($\sim$1 secondo per 70 MB su laptop moderno nella implementazione OpenSSL, $\sim$1 secondo per 30 KB su MentOS in QEMU). Successivamente va realizzato un client che invii l'informazione di file di input al server e riceva l'impronta risultante appena computata.

Ognuna delle tre opzioni tecnologiche ha requisiti specifici che la contraddistinguono, in particolare:

1. Requisiti minimi per la sufficienza
2. Multipli requisiti addizionali per la massima valutazione
3. Suggerimenti su eventuali funzionalità opzionali per ricevere la lode nel progetto

# Modalità di esecuzione

1. Realizzato come progetto individuale dello studente
2. Consegna: relazione testuale con allegato codice sorgente
3. Termine massimo di consegna e validità: sessione invernale 2025/2026 inclusa
4. Peso valutazione: 25% del voto complessivo, tuttavia permette in maniera esclusiva di raggiungere la lode complessiva a fronte del massimo voto nello scritto

# Materiale di supporto fornito

Esempio di codice per la creazione di una impronta SHA-256 a partire da una stringa ed a partire da un file.

- Versione per libreria OpenSSL (Linux e MacOS) per tecnologie 1 e 2;
- Versione dedicata MentOS per tecnologia 3;

N.B. il codice che processa il file non è pensato per essere massimamente efficiente ed anzi risulta più pratico per il progetto avere tempi di calcolo significativi anche per file piccoli.

# Esempio di creazione impronta da stringa (OpenSSL)

```c
int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage: %s <string with no spaces>\n", argv[0]);
        exit(1);
    }

    SHA256_CTX ctx;
    SHA256_Init(&ctx);

    SHA256_Update(&ctx, argv[1], strlen(argv[1]));

    unsigned char hash[32];
    SHA256_Final(hash, &ctx);

    char char_hash[65];
    for(int i = 0; i < 32; i++)
        sprintf(char_hash + (i * 2), "%02x", hash[i]);

    printf("%s\n", char_hash);

    return 0;
}
```

# Esempio di creazione impronta da stringa (MentOS)

```c
int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage: %s <string with no spaces>\n", argv[0]);
        exit(1);
    }

    SHA256_ctx_t ctx;
    sha256_init(&ctx);

    sha256_update(&ctx, (uint8_t *)argv[1], (uint32_t)strlen(argv[1]));

    uint8_t hash[32];
    sha256_final(&ctx, hash);

    char char_hash[65];
    sha256_bytes_to_hex(hash, 32, char_hash, 65);

    printf("%s\n", char_hash);

    return 0;
}
```

# Esempio funzione di creazione impronta da file (OpenSSL)

```
void digest_file (const char *filename, uint8_t * hash) {

    SHA256_CTX ctx;
    SHA256_Init(&ctx);

    char buffer [32];

    int file = open(filename, O_RDONLY, 0);
    if ( file == -1) {
        printf (" File %s does not exist \n", filename);
        exit (1);
    }

    ssize_t bR;
    do {
        // read the file in chunks of 32 characters
        bR = read( file , buffer , 32);
        if (bR > 0) {
            SHA256_Update(&ctx,(uint8_t *)buffer,bR);
        } else if (bR < 0) {
            printf ("Read failed \n");
            exit (1);
        }
    } while (bR > 0);

    SHA256_Final(hash, &ctx);

    close( file );
}
```

# Esempio funzione di creazione impronta da file (MentOS)

```
void digest_file (const char *filename, uint8_t * hash) {

    SHA256_ctx_t ctx;
    sha256_init (&ctx);

    char buffer [32];

    int file = open(filename, O_RDONLY, 0);
    if ( file == -1) {
        printf (" File %s does not exist \n", filename);
        exit (1);
    }

    ssize_t bR;
    do {
        bR = read( file , buffer , 32);
        if (bR > 0) {
            sha256_update(&ctx,(uint8_t *) buffer ,bR);
        } else if (bR < 0) {
            printf ("Read failed \n");
            exit (1);
        }
    } while (bR > 0);

    sha256_final (&ctx, hash);

    close ( file );
}
```

# Specifiche opzione 1 (IPC su Linux / macOS)

Modello di trasferimento: il client invia al server il file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare client che invia richiesta e riceve risposta, usando code di messaggi
- Trasferire il file da client a server attraverso memoria condivisa

# Specifiche opzione 1 (IPC su Linux / macOS)

Modello di trasferimento: il client invia al server il file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare client che invia richiesta e riceve risposta, usando code di messaggi
- Trasferire il file da client a server attraverso memoria condivisa

## Per arrivare al massimo voto:

- Istanziare processi distinti per elaborare richieste multiple concorrenti
- Introdurre un limite al numero di processi in esecuzione, modificabile dinamicamente da un secondo client
- Schedulare le richieste pendenti in ordine di dimensione del file
- Utilizzare almeno un semaforo per sincronizzare il flusso di comunicazione e/o processamento

# Specifiche opzione 1 (IPC su Linux / macOS)

Modello di trasferimento: il client invia al server il file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare client che invia richiesta e riceve risposta, usando code di messaggi
- Trasferire il file da client a server attraverso memoria condivisa

## Per arrivare al massimo voto:

- Istanziare processi distinti per elaborare richieste multiple concorrenti
- Introdurre un limite al numero di processi in esecuzione, modificabile dinamicamente da un secondo client
- Schedulare le richieste pendenti in ordine di dimensione del file
- Utilizzare almeno un semaforo per sincronizzare il flusso di comunicazione e/o processamento

## Alcuni suggerimenti per la lode:

- Offrire multipli algoritmi di schedulazione delle richieste pendenti (p.e. FCFS), configurabile alla partenza del server
- Implementare un meccanismo di interrogazione del server sul suo stato corrente di richieste pendenti ed in processamento

# Specifiche opzione 2 (Pthread e FIFO su Linux / macOS)

Modello di trasferimento: il client invia al server il percorso del file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando FIFO
- Implementare client che invia richiesta e riceve risposta, usando FIFO
- Istanziare thread distinti per elaborare richieste multiple in modo concorrente

# Specifiche opzione 2 (Pthread e FIFO su Linux / macOS)

Modello di trasferimento: il client invia al server il percorso del file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando FIFO
- Implementare client che invia richiesta e riceve risposta, usando FIFO
- Istanziare thread distinti per elaborare richieste multiple in modo concorrente

## Per arrivare al massimo voto:

- Schedulare le richieste pendenti in ordine di dimensione del file
- Introdurre un limite al numero di thread in esecuzione fissato
- Implementare il caching in memoria delle coppie percorso-hash già servite, così da restituire i valori computati nel caso di percorsi ripetuti
- Gestire richieste multiple simultanee per un dato percorso processando una sola richiesta ed attendendo il risultato nelle restanti richieste

# Specifiche opzione 2 (Pthread e FIFO su Linux / macOS)

Modello di trasferimento: il client invia al server il percorso del file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando FIFO
- Implementare client che invia richiesta e riceve risposta, usando FIFO
- Istanziare thread distinti per elaborare richieste multiple in modo concorrente

## Per arrivare al massimo voto:

- Schedulare le richieste pendenti in ordine di dimensione del file
- Introdurre un limite al numero di thread in esecuzione fissato
- Implementare il caching in memoria delle coppie percorso-hash già servite, così da restituire i valori computati nel caso di percorsi ripetuti
- Gestire richieste multiple simultanee per un dato percorso processando una sola richiesta ed attendendo il risultato nelle restanti richieste

## Alcuni suggerimenti per la lode:

- Implementare un "thread pool", ovvero i thread in numero fissato sono sempre in "esecuzione" ed appena completano una richiesta prelevano una nuova richiesta pendente
- Implementare un meccanismo di interrogazione della cache delle richieste già processate

# Specifiche opzione 3 (IPC e schedulazione su MentOS)

Modello di trasferimento: il client invia al server il percorso del file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare client che invia richiesta e riceve risposta, usando code di messaggi
- Modificare la schedulazione basata su vruntime per suddividere equamente il tempo della CPU fra i processi che hanno lo stesso pgid

# Specifiche opzione 3 (IPC e schedulazione su MentOS)

Modello di trasferimento: il client invia al server il percorso del file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare client che invia richiesta e riceve risposta, usando code di messaggi
- Modificare la schedulazione basata su vruntime per suddividere equamente il tempo della CPU fra i processi che hanno lo stesso pgid

## Per arrivare al massimo voto:

- Istanziare processi distinti, con lo stesso pgid, per elaborare richieste multiple concorrenti
- Definire un numero massimo di processi in esecuzione, fissato come argomento della chiamata del server
- Schedulare le richieste pendenti in ordine di dimensione del file
- Implementare il salvataggio su file delle coppie percorso-hash già servite, così da restituire i valori computati nel caso di percorsi ripetuti, anche dopo la ripartenza del server

# Specifiche opzione 3 (IPC e schedulazione su MentOS)

Modello di trasferimento: il client invia al server il percorso del file e riceve l'impronta.

## Per la sufficienza:

- Implementare server che riceve richieste ed invia risposte, usando code di messaggi
- Implementare client che invia richiesta e riceve risposta, usando code di messaggi
- Modificare la schedulazione basata su vruntime per suddividere equamente il tempo della CPU fra i processi che hanno lo stesso pgid

## Per arrivare al massimo voto:

- Istanziare processi distinti, con lo stesso pgid, per elaborare richieste multiple concorrenti
- Definire un numero massimo di processi in esecuzione, fissato come argomento della chiamata del server
- Schedulare le richieste pendenti in ordine di dimensione del file
- Implementare il salvataggio su file delle coppie percorso-hash già servite, così da restituire i valori computati nel caso di percorsi ripetuti, anche dopo la ripartenza del server

## Alcuni suggerimenti per la lode:

- Implementare un buffer in memoria di un numero fissato di coppie del file di salvataggio, gestendo i casi di buffer hit, buffer miss e file miss
- Abilitare la modifica dinamica della priorità dei processi che elaborano le impronte

# Requisiti sulla consegna

Da inviare a luca.geretti@univr.it.

## Relazione (formato pdf)

- Organizzare chiaramente per specifiche implementate
- Descrivere le scelte implementative e difficoltà riscontrate, eventualmente con ausilio grafico
- Se strettamente necessario, menzionare frammenti di codice

# Requisiti sulla consegna

Da inviare a luca.geretti@univr.it.

## Relazione (formato pdf)

- Organizzare chiaramente per specifiche implementate
- Descrivere le scelte implementative e difficoltà riscontrate, eventualmente con ausilio grafico
- Se strettamente necessario, menzionare frammenti di codice

## Codice sorgente (formato zip)

- Opzioni 1 e 2: strutturare come progetto CMake (esempio disponibile nel codice SHA-256 fornito)
- Opzione 3: strutturare come albero di cartelle e files modificati rispetto al ramo master di MentOS (esempio disponibile nello zip di soluzioni della lezione sulle chiamate di sistema)
- Deve poter compilare ed eseguire correttamente