Sistemi Operativi e Reti di Calcolatori (SOReCa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*

Terzo Anno | Primo Semestre

A.A. 2025/2026

Esercitazione [05] Sincronizzazione: Riepilogo

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it
Paolo Ottolino paolo.ottolino@uniroma1.it
Matteo Cornacchia cornacchia@diag.uniroma1.it

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Sommario

- Soluzione esercizi precedenti (lab-04: File Pipe FIFO)
- Esercizio di Riepilogo
  - 1: fork()
  - 2: fork() + CS + parseOutput()
  - 3: fork() + pthread
  - 4: fork() – pthread + CS

Obiettivi

- Imparare ad effettuare operazioni di input e output usando i descrittori (fd) in UNIX
- Lettura/scrittura su file
- Invio/ricezioni messaggi su socket

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Soluzione: copiare un file in C

Lab04, es. 1

Descrittori in Unix

Lab04 es1: copiare un file in C

- Sorgente da completare: `copy.c`
- Argomenti
  - File sorgente S
  - File destinazione D
  - Dimensione B del batch di lettura/scrittura (opzionale, default 128 byte)
- Semantica

  Effettuare una copia di S in D tramite una sequenza di letture da S e scritture in D a blocchi di B byte per volta
- Esercizio: completare il codice dove indicato
  - Per testare la propria soluzione è disponibile lo script `test.sh`

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Descrittori in Unix

Lab04 es1: copiare un file in C

```
static inline void performCopyBetweenDescriptors(int src_fd, int dest_fd, int block_size) {
    char* buf = malloc(block_size);

    while (1) {
        int read_bytes = 0; // index for writing into the buffer
        int bytes_left = block_size; // number of bytes to (possibly) read
```

```
 while (bytes_left > 0) { //read
      int ret = read(src_fd, buf + read_bytes, bytes_left);

            // no more bytes left to read!
            if (ret == 0) break; //EOF

            if (ret == -1){
                if(errno == EINTR) // read() interrupted by a signal
                    continue;
                // handle generic errors
                handle_error("Cannot read from source file");
            }


            /* The value returned may be less than bytes_left if the
number
             * of bytes left in the file is less than bytes_left, if the
             * read() request was interrupted by a signal, or if the file
             * is a pipe or FIFO or special file and has fewer than
             * bytes_left bytes immediately available for reading */
            bytes_left -= ret;
            read_bytes += ret;
        }}
```

```
while (bytes_left > 0) { //write
     int ret = write(dest_fd, buf + written_bytes, bytes_left);


            if (ret == -1){
                if(errno == EINTR) // write() interrupted by a signal
                    continue;
                // handle generic errors
                handle_error("Cannot write to destination file");
            }

            bytes_left -= ret;
            written_bytes += ret;
        }
    }
```

```
 }

    free(buf);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Soluzione: comunicazion tramite Pipe

Lab04, es. 2

IPC tramite pipe

Lab04 es. 2: Comunicazione unidirezionale di processi via pipe con sincronizzazione

- Il processo padre crea CHILDREN_COUNT processi figlio che condividono una pipe unica:
  - nella quale WRITERS_COUNT figli scrivono (*writers*)
  - e dalla quale READERS_COUNT figli leggono (*readers*)
- I writers scrivono nella pipe in mutua esclusione tramite un semaforo il cui nome è definito nella macro WRITE_MUTEX
- I readers leggono dalla pipe in mutua esclusione grazie ad un altro semaforo, il cui nome è specificato nella macro READ_MUTEX
- All'avvio, il padre crea i semafori named assicurandosi che non esistano già, e passa come argomento ai processi figlio il puntatore all'oggetto sem_t su cui ciascun reader o writer dovrà operare
- Una volta avviati:
  - i writers devono scrivere nella pipe MSG_COUNT messaggi in totale (ognuno dovrà quindi scriverne MSG_COUNT/WRITERS_COUNT).
  - ogni reader deve leggere dalla pipe MSG_COUNT/READERS_COUNT messaggi e verificarne l'integrità
  - ogni messaggio è un array di MSG_ELEMS interi, che viene considerato integro se tutti i suoi elementi hanno lo stesso valore.
- Infine, il padre deve attendere esplicitamente la terminazione dei figli e liberare le risorse.

IPC tramite pipe

Lab04 es. 2: Comunicazione unidirezionale di processi via pipe con sincronizzazione 1/2

```
int pipefd[2];
```

```
void reader(int reader_id, sem_t* read_mutex) {
    int data[MSG_ELEMS];
    printf("[READER_%d] processo reader creato.\n", reader_id);

    int ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");

    int i = 0;
    for ( ; i < MSG_COUNT/READERS_COUNT; i++) {

        ret = sem_wait(read_mutex);
        if(ret) handle_error("READER: error waiting on read mutex");

        read_from_pipe(pipefd[0], data, sizeof(data)); // sizeof(data) ==
MSG_ELEMS * sizeof(int)

        ret = sem_post(read_mutex);
        if(ret) handle_error("READER: error posting on read mutex");

        printf("[READER_%d] Letto msg #%d con valore %d\n", reader_id, i,
data[0]);
        if (!is_msg_ok(data, MSG_ELEMS))
            printf("READER: corrupted message!!!\n");
    }

    ret = sem_close(read_mutex);
    if(ret) handle_error("READER: error closing read mutex");

    ret = close(pipefd[0]);
    if(ret) handle_error("READER: error closing pipe");

}
```

```
int write_to_pipe(int fd, const void *data, size_t data_len) {

    int written_bytes = 0, ret;

    while (written_bytes < data_len) {
        ret = write(fd, data + written_bytes, data_len - written_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error writing to pipe");
        written_bytes += ret;
    }
    return written_bytes;
}
```

```
int read_from_pipe(int fd, void *data, size_t data_len) {

    int read_bytes = 0, ret;
    while (read_bytes < data_len) {
        ret = read(fd, data + read_bytes, data_len - read_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error reading from pipe");
        if (ret ==  0) handle_error("unexpected close of the pipe");
        read_bytes += ret;
    }
    return read_bytes;
}
```

IPC tramite pipe

Lab04 es. 2: Comunicazione unidirezionale di processi via pipe con sincronizzazione 2/2

```
int pipefd[2];
```

```
void writer(int writer_id, sem_t* write_mutex) {
    int data[MSG_ELEMS];
    int i,ret;
    printf("[WRITER_%d] processo writer creato.\n", writer_id);
    /**
     * COMPLETARE QUI
     *
     * Obiettivi:
     * - chiudere i descrittori non necessary
     * - gestire eventuali errori
     **/
    ret = close(pipefd[0]);
    if(ret) handle_error("error closing pipe");
    for (i = 0 ; i < MSG_COUNT/WRITERS_COUNT; i++) {
        create_msg(data, MSG_ELEMS, i);
        ret = sem_wait(write_mutex);
        if(ret) handle_error("error waiting on write mutex");
        write_to_pipe(pipefd[1], data, sizeof(data)); // sizeof(data) ==
MSG_ELEMS * sizeof(int)
        ret = sem_post(write_mutex);
        if(ret) handle_error("error posting on write mutex");
        printf("[WRITER_%d] Inviato il msg #%d\n", writer_id, i);
    }
    ret = sem_close(write_mutex);
    if(ret) handle_error("WRITER: error closing write mutex");
    /**     * COMPLETARE QUI
     *
     * Obiettivi:
     * - chiudere i descrittori rimanenti
     * - gestire eventuali errori
     **/
    ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
}
```

```
int write_to_pipe(int fd, const void *data, size_t data_len) {

    int written_bytes = 0, ret;

    while (written_bytes < data_len) {
        ret = write(fd, data + written_bytes, data_len - written_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error writing to pipe");
        written_bytes += ret;
    }
    return written_bytes;
}
```

```
int read_from_pipe(int fd, void *data, size_t data_len) {

    int read_bytes = 0, ret;
    while (read_bytes < data_len) {
        ret = read(fd, data + read_bytes, data_len - read_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error reading from pipe");
        if (ret ==  0) handle_error("unexpected close of the pipe");
        read_bytes += ret;
    }
    return read_bytes;
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

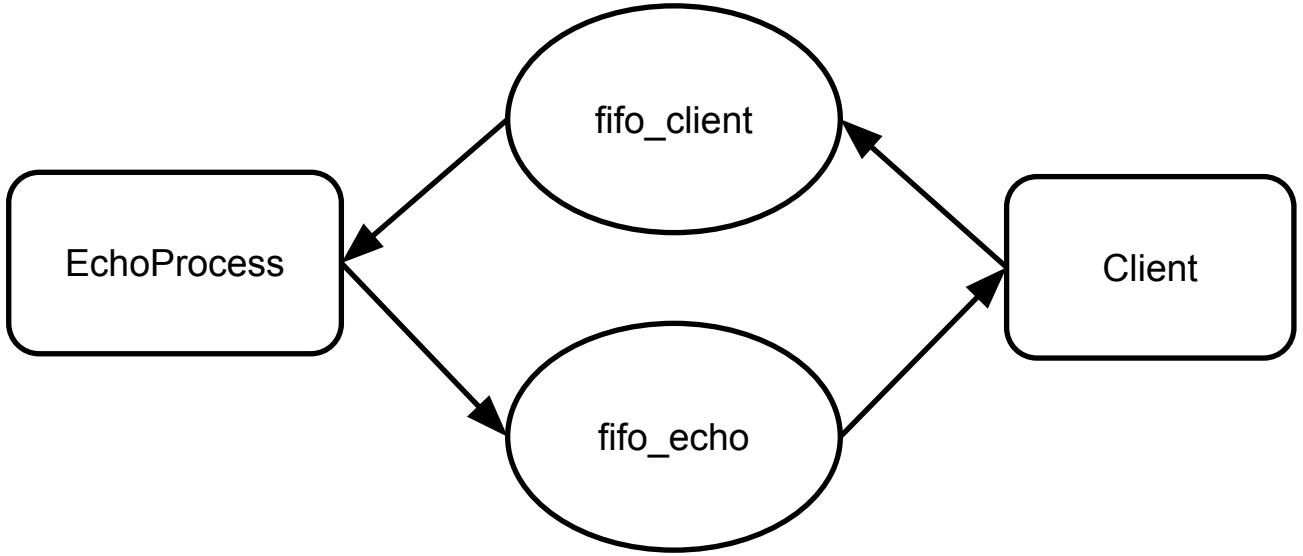
SAPIENZA
UNIVERSITÀ DI ROMA

Soluzione: comunicazion tramite Pipe

Lab04, es. 3

Named Pipe (FIFO)
Lab04 es3: EchoProcess su FIFO

- Il server prepara (crea) due FIFO
  - `echo_fifo` per inviare messaggi al client
  - `client_fifo` per ricevere messaggi dal client
- La comunicazione client-server avviene tramite queste due FIFO
- Esercizio: completare codici di client (`client.c`) e server (`echo.c`) e lettura/scrittura (rw.c)



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti
Sapienza Università di Roma

Named Pipe (FIFO)
Lab04 es3: EchoProcess su FIFO 1/2

client.c

```
/** Client component **/
int main(int argc, char* argv[]) {
    int ret;
    int echo_fifo, client_fifo;
    char buf[1024];

    char* quit_command = QUIT_COMMAND;
    size_t quit_command_len = strlen(quit_command);

    echo_fifo = open(ECHO_FIFO_NAME, O_RDONLY);
    if(echo_fifo == -1) handle_error("Cannot open Echo FIFO for reading");
    client_fifo = open(CLNT_FIFO_NAME, O_WRONLY);
    if(client_fifo == -1) handle_error("Cannot open Client FIFO for writing");

    memset(buf,0,1024);
    int bytes_read = readOneByOne(echo_fifo, buf, '\n');

    buf[bytes_read] = '\0';
    printf("%s", buf);

    // main loop
    while (1) {
        printf("Insert your message: ");

        // read a line from stdin (including newline symbol '\n')
        if (fgets(buf, sizeof(buf), stdin) != (char*)buf){
            handle_error("Error while reading from stdin, exiting...\n");
        }

        // send message to Echo process
        int msg_len = strlen(buf);
        writeMsg(client_fifo, buf, msg_len);

        /* After a quit command we won't receive data from the server
         * anymore, thus we must exit the main loop. */
        if (msg_len - 1 == quit_command_len && // -1 to ignore trailing '\n'
            !memcmp(buf, quit_command, quit_command_len)) break;

        // read message from Echo process
        bytes_read = readOneByOne(echo_fifo, buf, '\n');
        buf[bytes_read] = '\0';
        printf("Server response: %s", buf);
    }

    // close the descriptors
    ret = close(echo_fifo);
    if(ret) handle_error("Cannot close Echo FIFO");
    ret = close(client_fifo);
    if(ret) handle_error("Cannot close Client FIFO");

    exit(EXIT_SUCCESS);
}
```

rw.c

```
int readOneByOne(int fd, char* buf, char separator) {

    int ret;
    int bytes_read = 0;
    do {
        ret = read(fd, buf + bytes_read, 1);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot read from FIFO");
        if (ret ==  0){
            printf("%s\n",buf);
            fflush(stdout);
             handle_error_en(bytes_read,"Process has closed the FIFO unexpectedly!
Exiting...");
        }
        // we use post-increment on bytes_read so that we first read the
        // byte that has just been written, then we do the increment
    } while(buf[bytes_read++] != separator);
    printf("Read %d bytes\n",bytes_read);
    fflush(stdout);
    return bytes_read;
}
```

```
void writeMsg(int fd, char* buf, int size) {
    int ret;
int bytes_sent = 0;
    while (bytes_sent < size) {
        ret = write(fd, buf + bytes_sent, size - bytes_sent);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot write to FIFO");
        bytes_sent += ret;
    }
    printf("Sent %d bytes\n",bytes_sent);
    fflush(stdout);
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Named Pipe (FIFO)
Lab04 es3: EchoProcess su FIFO 2/2

echo.c

```
int main(int argc, char* argv[]) {
    int ret;
    int echo_fifo, client_fifo;
    char buf[1024];

    char* quit_command = QUIT_COMMAND;
    size_t quit_command_len = strlen(quit_command);

    // Create the two FIFOs
    unlink(ECHO_FIFO_NAME);
    unlink(CLNT_FIFO_NAME);
    ret = mkfifo(ECHO_FIFO_NAME, 0666);
    if(ret) handle_error("Cannot create Echo FIFO");
    ret = mkfifo(CLNT_FIFO_NAME, 0666);
    if(ret) handle_error("Cannot create Client FIFO");

    /** [SOLUTION] OPEN THE TWO FIFOS **/
    echo_fifo = open(ECHO_FIFO_NAME, O_WRONLY);
    if(echo_fifo == -1) handle_error("Cannot open Echo FIFO for writing");
    client_fifo = open(CLNT_FIFO_NAME, O_RDONLY);
    if(client_fifo==-1) handle_error("Cannot open Client FIFO for reading");

    // send welcome message
    sprintf(buf, "Hi! I'm an Echo process based on FIFOs. I will send you back through a FIFO whatever"
            " you send me through the other FIFO, and I will stop and exit when you send me %s.\n",
quit_command);

    writeMsg(echo_fifo, buf, strlen(buf));

    while (1) {
        memset(buf,0,1024);
        int bytes_read = readOneByOne(client_fifo, buf, '\n');

        if (DEBUG) {
            buf[bytes_read] = '\0';
            printf("Message received: %s", buf);
        }

        // check whether I have just been told to quit...
        if (bytes_read == quit_command_len && !memcmp(buf, quit_command, quit_command_len)) break;

        // ... or if I have to send the message back through the Echo FIFO
        writeMsg(echo_fifo, buf, bytes_read);
    }

    // close the descriptors and destroy the two FIFOs
    cleanFIFOs(echo_fifo, client_fifo);
    exit(EXIT_SUCCESS);
}
```

rw.c

```
int readOneByOne(int fd, char* buf, char separator) {

    int ret;
    int bytes_read = 0;
    do {
        ret = read(fd, buf + bytes_read, 1);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot read from FIFO");
        if (ret ==  0){
            printf("%s\n",buf);
            fflush(stdout);
             handle_error_en(bytes_read,"Process has closed the FIFO unexpectedly!
Exiting...");
        }
        // we use post-increment on bytes_read so that we first read the
        // byte that has just been written, then we do the increment
    } while(buf[bytes_read++] != separator);
    printf("Read %d bytes\n",bytes_read);
    fflush(stdout);
    return bytes_read;
}
```

```
void writeMsg(int fd, char* buf, int size) {
    int ret;
int bytes_sent = 0;
    while (bytes_sent < size) {
        ret = write(fd, buf + bytes_sent, size - bytes_sent);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot write to FIFO");
        bytes_sent += ret;
    }
    printf("Sent %d bytes\n",bytes_sent);
    fflush(stdout);
}
```

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Sincronizzazione: Riepilogo

- Lab05, es. 1
- Lab05, es. 2
- Lab05, es. 3
- Lab05, es. 4

Sezione Critica

Lab05

[Esercizio di riepilogo su quanto visto finora in laboratorio]

- Sviluppare un'applicazione in C con questa semantica
  - Il processo «main» crea N processi figlio tramite fork
  - Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo «main»
  - L'attività dei processi figlio consiste nel lanciare M thread per volta
    - La sezione critica di ciascun thread consiste nello scrivere in append su un file l'identità del processo scrivente
  - Passati T secondi, il processo «main» deve notificare i processi figlio di cessare la loro attività e terminare
    - Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione
    - Suggerimento: fare uso di una shared memory
  - Infine, il processo «main» deve identificare il processo che ha effettuato più accessi in sezione critica

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Creazione e Notifica ai processi figlio

Lab05 es1: `main`

- Processo «main»
  - Crea N processi figlio
  - Notifica gli N processi figlio di avviare la loro attività
  - Attende T secondi
  - Notifica gli N processi figlio di cessare la loro attività e terminare
  - Attende la terminazione degli N processi figlio
  - Identifica il processo che ha acceduto in sezione critica più volte (usare la funzione `parseOutput()`)
  - Termina

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Attività dei processi figlio

Lab05 es1: `child`

- Processo figlio
  - Attende la notifica di avvio dal processo «main»
  - Ciclo
    - Lancia M thread
    - Attende il termine degli M thread
    - Verifica se il processo «main» ha notificato di cessare l'attività
      - In caso positivo, esce dal ciclo
  - Termina

Thread dei processi figlio

Lab05 es1: `child thread`

- Thread di un processo figlio
  - Richiede l'accesso in sezione critica
  - Una volta in sezione critica
    - Apre il file in append
    - Scrive l'identità del processo figlio
    - Chiude il file
    - Esce dalla sezione critica
  - Termina

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Homework

Homework

esercizi per casa

1. Modificare il sorgente per non usare i file
   - Un contatore per ogni processo figlio viene posto nella memoria condivisa e viene incrementato dai thread
2. Implementare l'esercizio usando solo semafori
   - Suggerimento:
     - Il main aspetta T secondi (`sleep`) e notifica i figli di terminare le loro attività tramite un semaforo
     - Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione, ma non può bloccarsi sul semaforo, quindi deve leggere il suo contenuto (`sem_getvalue`)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Valutazione

https://forms.gle/4bBUdq6UfJH3HoUCA

Sincronizzazione: Soluzione

Sincronizzazione: Soluzione

Sol.: 1. processo main crea N processi figlio tramite fork

```
for (i = 0; i < n; i++) {
   pid_t pid = fork();
   if (pid == -1) {
      exit(EXIT_FAILURE);
   } else if (pid == 0) {
            // child process, its id is i
      break;
   } else {
            // main process, go on creating processes
      continue;
   }
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Sincronizzazione: Soluzione

Sol. 2. i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main

- Richiede due diverse sincronizzazioni da effettuare in sequenza
  1. Il main deve aspettare che tutti i figli siano partiti (prima istruzione eseguita)
  2. I figli devono aspettare il «via» dal main, in modo che tutti possano avviare le proprie attività approssimativamente nello stesso istante
     - L'approssimazione è dovuta al fatto che il main «sveglia» un processo per volta e al non-determinismo nell'allocazione dei core ai thread
     - Si può considerare un best-effort, comunque migliore rispetto al non imporre alcuna sincronizzazione all'avvio

Sincronizzazione: Soluzione

Sol.: 2. i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main

- Per la prima sincronizzazione
  - Il main deve bloccarsi → `sem_wait`
  - I processi figlio devono sbloccare il main → `sem_post`
  - Come usare il semaforo `main_waits_for_children`
    - Inizialmente il main deve bloccarsi anche se nessun figlio ha ancora notificato il proprio avvio → semaforo inizializzato a 0
    - Il main deve aspettare che tutti i figli abbiano notificato il proprio avvio

```
for (i = 0; i < n; i++)
 sem_wait(main_waits_for_children);
```

    - Ogni figlio deve notificare il proprio avvio

```
sem_post(main_waits_for_children);
```

Sincronizzazione: Soluzione

Sol.: 2. i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main

- Per la seconda sincronizzazione
  - I processi figlio devono bloccarsi → `sem_wait`
  - Il main deve sbloccare i figli → `sem_post`
  - Come usare il semaforo `children_wait_for_main`
    - Ogni processo figlio deve potersi bloccare → semaforo inizializzato a 0

```
sem_wait(children_wait_for_main);
```

    - Il main deve consentire a tutti i processi figlio di sbloccarsi

```
for (i = 0; i < n; i++)
 sem_post(children_wait_for_main);
```

Sincronizzazione: Soluzione

Sol.: 3. L'attività dei processi figlio consiste nel lanciare M thread per volta

```
pthread_t* thread_handlers =
 malloc(m * sizeof(pthread_t));
……
for (j = 0; j < m; j++) {
 thread_args_t *t_args = ...;
    t_args->process_id = process_id;
 t_args->thread_id = thread_id++;
 pthread_create(&thread_handlers[j], NULL,
    thread_function, t_args);
}
```

- E poi deve attenderne il termine

```
for (j = 0; j < m; j++)
 pthread_join(thread_handlers[j], NULL);
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Sincronizzazione: Soluzione

Sol.: Operazioni del singolo thread

- Accesso in sezione critica
- Scrittura in append su un file dell'identità del processo

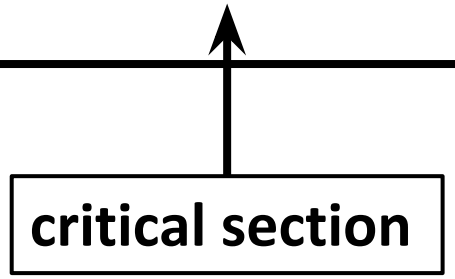
```
thread_args_t *args = (thread_args_t*)arg_ptr;

sem_wait(critical_section);

int fd = open(FILENAME, O_WRONLY | O_APPEND);
write(fd, &(args->process_id), sizeof(int));
close(fd);

sem_post(critical_section);

free(args);
```



Sincronizzazione: Soluzione

Sol.: 4. trascorsi T secondi, il main deve notificare i processi figlio di cessare la loro attività e terminare

- Shared memory `/shmem-notification`
  - Valore 0: continuare le attività (valore iniziale)
  - Valore 1: terminare le attività
- Il main aspetta T secondi (`sleep`) e notifica i figli di terminare le loro attività

```
sleep(t);
*data = 1;
```

- Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione
  - Dopo il ciclo di pthread_join, il processo figlio può verificare la presenza di tale notifica controllando il contenuto della memoria condivisa

```
if (*data) break;
```

Sincronizzazione: Soluzione

Sol.: 5. main deve identificare il processo che ha effettuato più accessi in sezione critica

- Attesa del termine effettivo di tutti i figli

```
int child_status;
for (i = 0; i < n; i++)
 wait(&child_status);
```

- Lettura statistiche di accesso da file

```
int *access_stats = (int*)calloc(n, sizeof(int));
int fd = open(FILENAME, O_RDONLY);
size_t read_bytes;  int read_byte;
do { read_bytes = read(fd, &read_byte, sizeof(int));
 if (read_bytes > 0) access_stats[read_byte]++;
} while(read_bytes > 0);
close(fd);
```

Sincronizzazione: Soluzione

Sol.: 5. main deve identificare il processo che ha effettuato più accessi in sezione critica

- Identificazione del processo che ha effettuato più accessi

```
int max_process_id = -1, max_accesses = -1;
for (i = 0; i < n; i++) {
 if (access_stats[i] > max_accesses) {
    max_accesses = access_stats[i];
    max_process_id = i;
 }
}
```

- Cleanup
  - Close e unlink di tutti i semafori
  - Free della memoria allocata