

# Sistemi Operativi e Reti di Calcolatori (SORECa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*  
Terzo Anno | Primo Semestre  
A.A. 2024/2025

## Esercitazione [05] Sincronizzazione: Riepilogo

Riccardo Lazzeretti [lazzeretti@diag.uniroma1.it](mailto:lazzeretti@diag.uniroma1.it)

Paolo Ottolino [paolo.ottolino@uniroma1.it](mailto:paolo.ottolino@uniroma1.it)

Edoardo Liberati [e.liberati@diag.uniroma1.it](mailto:e.liberati@diag.uniroma1.it)

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 es1: `fork()`

*[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

# Creazione e Notifica ai processi figlio

Lab05 es2: `parseOutput()`

- 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

# Attività dei processi figlio

## Lab05 es3

- 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 es4: thread di un processo figlio

- 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

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;  
    }  
}
```



## 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);
```

# 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);
```

**critical section**

# 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



# Homework

# Homework

## esercizio 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`)