

# Esercizio di riepilogo [1/10]

- Esercizio di riepilogo su processi, thread e semafori
- Sviluppare un'applicazione in C con questa semantica
  1. Il processo main crea N processi figlio tramite fork
  2. Tutti i processi figlio si sincronizzano per iniziare la loro attività, avviata dal processo main
  3. L'attività dei processi figlio consiste nel lanciare M thread per volta
    - a. Competono per l'accesso in sezione critica, gestito dal processo main
    - b. Una volta in sezione critica, devono scrivere in append su un file l'identità del processo scrivente
  4. Passati T secondi, il processo main deve notificare i processi figlio di cessare la loro attività e terminare (usare la `sem_getvalue`)
    - a. Prima di terminare, un processo deve attendere la fine dei thread attualmente in esecuzione
  5. Infine, il processo main deve identificare il processo che ha effettuato più accessi in sezione critica

# Esercizio di riepilogo [2/10]

## 1. Il 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;  
    }  
}
```

# Esercizio di riepilogo [3/10]

2. Tutti 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

# Esercizio di riepilogo [4/10]

2. Tutti 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);
```

# Esercizio di riepilogo [5/10]

2. Tutti 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);`

# Esercizio di riepilogo [6/10]

## 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);
```

# Esercizio di riepilogo [7/10]

## 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**



## Esercizio di riepilogo [8/10]

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

- Semaforo `end_children_activities`
  - 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 (`sem_post`)

```
sleep(t);  
sem_post(end_children_activities);
```
- 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 con una `sem_getvalue`

```
sem_getvalue(end_children_act, &main_notification);  
if (main_notification) break;
```



## Esercizio di riepilogo [9/10]

5. il processo 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);
```

## Esercizio di riepilogo [10/10]

5. il processo 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