Tutoraggio di Sistemi Operativi

Comunicazione tra Processi e Thread

Matteo Federico

Lezione #8

Obiettivi della lezione

Meccanismi trattati

- Pipe anonime
- FIFO (named pipe)
- Memoria condivisa tra processi
- Memoria condivisa tra thread

Obiettivi

- Capire quando usare ogni meccanismo
- Confrontare pro e contro
- Padroneggiare le API POSIX
- Applicare con esercizi in C

TOR VERGATA

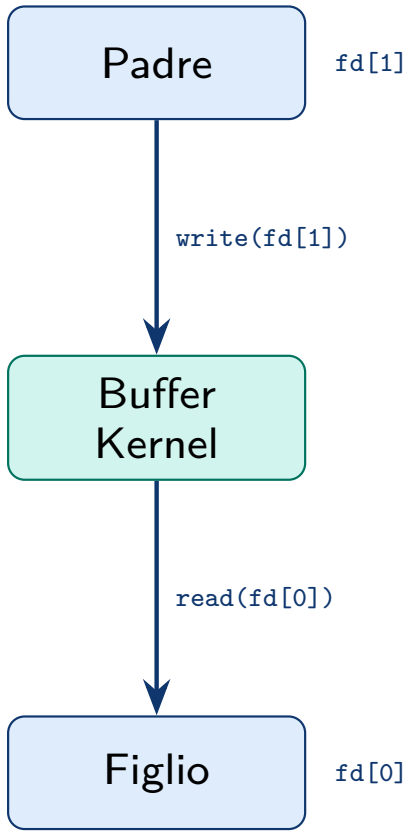
Indice

TOR VERGATA

Pipe Anonime

Pipe anonime – Concetto

- Canale **unidirezionale**, tipo byte-stream
- Usato tra processo padre e figlio dopo `fork()`
- Dati letti in ordine, solo accesso sequenziale
- Il kernel gestisce il buffer e blocca `read`/`write` quando necessario



Flusso dei dati dopo `fork()`

Pipe anonime – API principali

Creazione

```
int pipe(int fd[2]);
```

- `fd[0]`: descrittore di **lettura**
- `fd[1]`: descrittore di **scrittura**

Uso tipico dopo `fork()`

- Padre: `write(fd[1], buf, n)` oppure `dprintf(fd[1], "...")`
- Figlio: `read(fd[0], buf, n)` oppure `fdopen(fd[0],"r")` + `fgets`

Regola fondamentale

Chiudere **sempre** gli estremi inutilizzati dopo `fork()` per evitare deadlock.

TOR VERGATA

Pipe anonime – Pro e Contro

Pro

- Semplici da usare tra processi imparentati
- Sincronizzazione implicita: `read` blocca se non ci sono dati, `write` blocca se il buffer è pieno
- Nessun nome nel filesystem

Contro

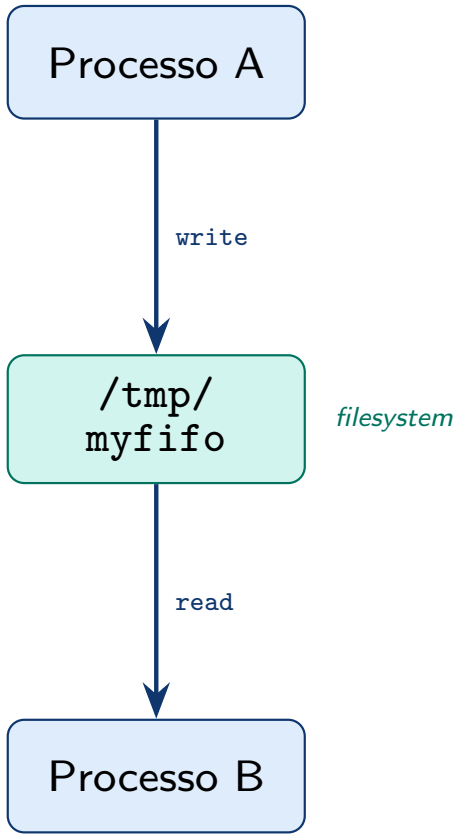
- Unidirezionali: bidirezionale $\Rightarrow$ due pipe
- Solo tra processi imparentati (tipicamente)
- Solo stream sequenziale, nessun accesso casuale

TOR VERGATA

FIFO (Named Pipe)

FIFO – Concetto

- Come una pipe, ma con un **nome nel filesystem** (es. `/tmp/myfifo`)
- Permette comunicazione tra processi **completamente indipendenti**
- File speciale creato con `mkfifo`
- Stessa semantica di lettura/scrittura delle pipe anonime



I processi si trovano tramite il path

FIFO – API principali

Creazione

```
int mkfifo(const char *pathname, mode_t mode);
```

Utilizzo (identico a un file ordinario)

- `int fd = open(pathname, O_RDONLY | O_WRONLY);`
- `read(fd, buf, count);`
- `write(fd, buf, count);`
- `close(fd);`

Cleanup

La FIFO rimane nel filesystem: rimuoverla con `unlink(pathname)`.

FIFO – Pro e Contro

Pro

- Processi completamente indipendenti
- Discovery tramite path nel filesystem
- Comportamento già noto (come pipe)
- Adatta a piccoli sistemi di messaggistica

Contro

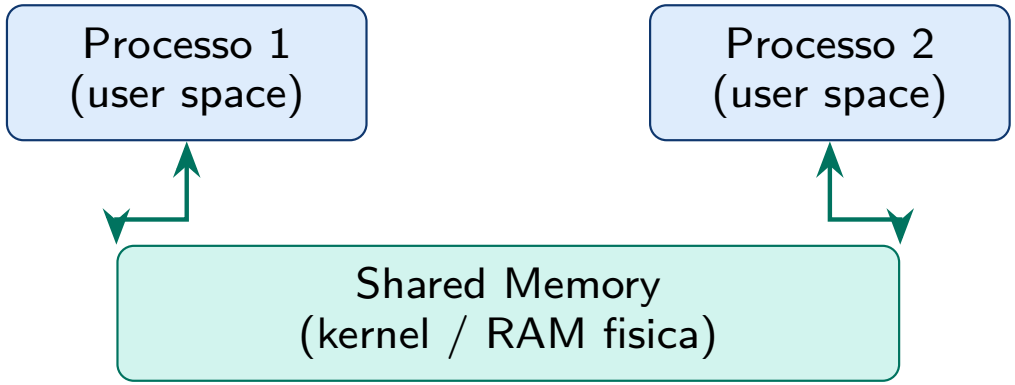
- Solo stream sequenziale, nessun accesso casuale
- Gestione permessi e cleanup nel filesystem
- Attenzione con più reader/writer sulla stessa FIFO

TOR VERGATA

Memoria Condivisa tra Processi

Memoria condivisa tra processi – Concetto

- Segmento di memoria mappato in **più processi**
- POSIX: `shm_open` + `ftruncate` + `mmap`
- Ideale per grandi quantità di dati e accesso casuale (array, ring buffer, strutture dati)
- I processi vedono la **stessa pagina fisica**: zero copie



Stessa area fisica, due mapping

Memoria condivisa tra processi – API principali

Ciclo di vita POSIX

1. `shm_open("/nome", O_CREAT|O_RDWR, 0660)` — crea/apre
2. `ftruncate(fd, size)` — dimensiona
3. `mmap(NULL, size, PROT_R|PROT_W, MAP_SHARED, fd, 0)` — mappa
4. *...uso...*
5. `munmap(ptr, size)` — de-mappa
6. `shm_unlink("/nome")` — rimuove dal fs

Compilazione

Aggiungere `-lrt` (su alcuni sistemi) per `shm_open` e `shm_unlink`.

TOR VERGATA

Memoria condivisa tra processi – Pro e Contro

Pro

- Ottime prestazioni: nessuna copia esplicita
- Accesso casuale: array, ring buffer, strutture dati condivise
- Tra processi completamente indipendenti

Contro

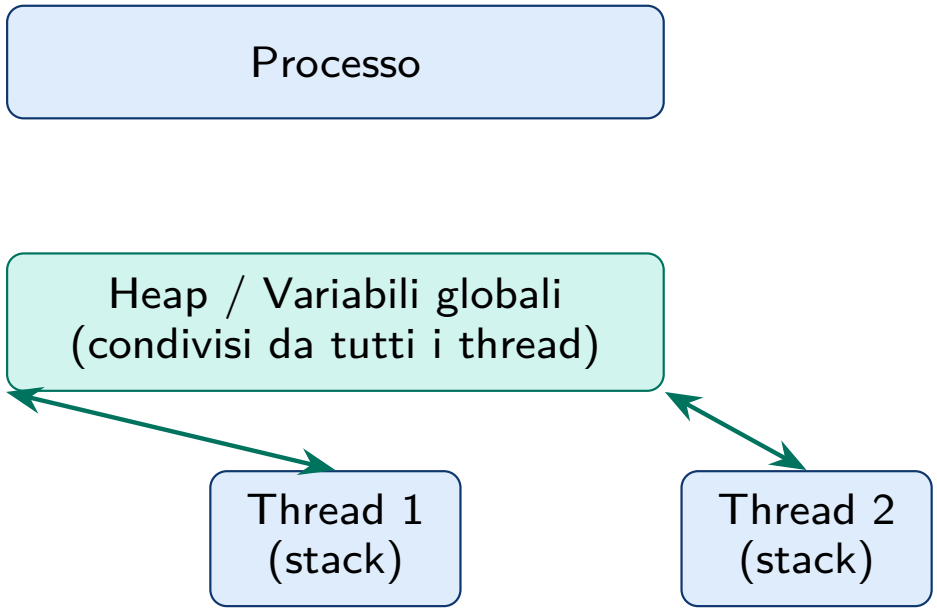
- **Nessuna sincronia automatica**: servono mutex o semafori con `PTHREAD_PROCESS_SHARED`
- Gestione più complessa (protocolli, cleanup)
- Layout di memoria da definire e mantenere compatibile

TOR VERGATA

Memoria Condivisa tra Thread

Memoria condivisa tra thread – Concetto

- Thread dello **stesso processo** condividono automaticamente lo spazio di indirizzamento:
  - codice, variabili globali, heap
  - stack **privato** per ogni thread
- Nessun IPC esplicito: basta una `struct` globale
- Il problema principale diventa la **race condition**, non la comunicazione



Memoria condivisa tra thread – API principali

Creazione e attesa thread

```
pthread_create(&tid, NULL, fn, arg);
pthread_join(tid, NULL);
```

Sincronizzazione con mutex

- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- `pthread_mutex_lock(&m);` / `pthread_mutex_unlock(&m);`
- `pthread_mutex_destroy(&m);`

Attributi process-shared (shm tra processi)

```
pthread_mutexattr_setpshared(&attr,
  PTHREAD_PROCESS_SHARED);
```

TOR VERGATA

Memoria condivisa tra thread – Pro e Contro

Pro

- Nessun IPC aggiuntivo
- Context switch più leggero rispetto ai processi
- Comodo per parallelizzare parti dello stesso programma

Contro

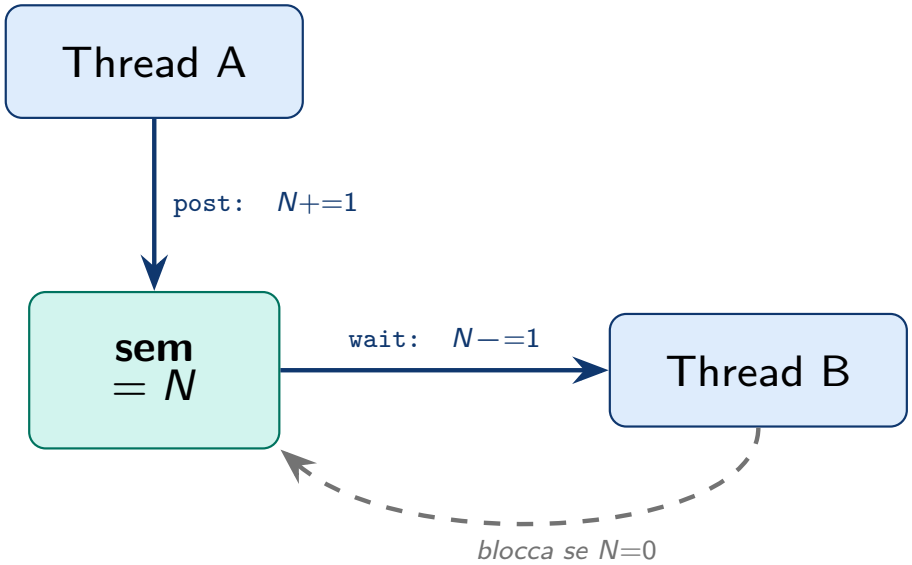
- Race condition e data race se i lock non sono usati correttamente
- Debug complesso: deadlock, starvation, Heisenbugs
- Errori di concorrenza difficili da riprodurre

TOR VERGATA

Semafori POSIX

Semafori POSIX – Concetto

- Variabile intera **non-negativa** con operazioni atomiche
- Due operazioni fondamentali:
  - **wait** (P): se valore > 0 decrementa;
    altrimenti **blocca** il chiamante
  - **post** (V): incrementa; se ci sono thread bloccati, ne sveglia uno
- **Binario** $(0/1) \rightarrow$ mutual exclusion
- **Contatore** $(\geq 0) \rightarrow$ signaling, bounded buffer



Semafori POSIX – API

Inizializzazione e distruzione

```
sem_init(&sem, pshared, valore_iniziale);
```

`pshared=0`: condiviso tra thread dello stesso processo
`pshared=1`: condiviso tra processi distinti (shm)

```
sem_destroy(&sem);
```

Operazioni

- `sem_wait(&sem);` — P — decrementa; blocca se = 0
- `sem_post(&sem);` — V — incrementa; sveglia un waiter
- `sem_trywait(&sem);` — non bloccante; ritorna `EAGAIN` se = 0

Differenza chiave dal mutex

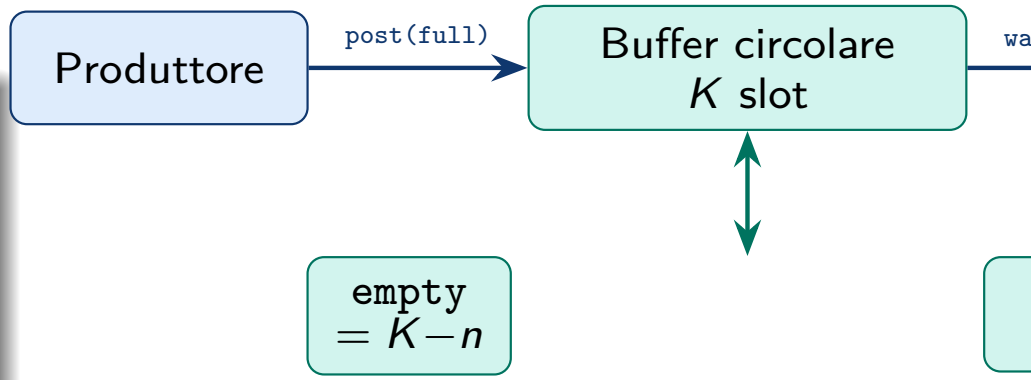
Il semaforo **non ha ownership**: chiunque può chiamare `post`, anche chi non ha eseguito `wait`. Questo lo rende ideale per il signaling tra thread diversi.

TOR VERGATA

Semafori POSIX – Pattern: Bounded Buffer

Due semafori contatori

- `sem_empty` (init K): slot liberi
- `sem_full` (init 0): item pronti
- Mutex aggiuntivo per accedere in sicurezza a `head`/`tail`



n = item attualmente nel buffer

Produttore

```
sem_wait(&empty);
scrivi nel buffer
sem_post(&full);
```

Consumatore

```
sem_wait(&full);
leggi dal buffer
sem_post(&empty);
```

Mutex e Variabili di Condizione

Mutex – Concetto e API

Concetto

- Garantisce **accesso esclusivo** a una sezione critica
- **Owner-based**: solo il thread che ha acquisito il lock può rilasciarlo
- Tentare il doppio lock dallo stesso thread → deadlock
- Più efficiente del semaforo binario per la mutual exclusion

API POSIX

```
pthread_mutex_t m =
  PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_lock(&m);
```
blocca se il mutex è occupato

```
pthread_mutex_unlock(&m);
```

```
pthread_mutex_trylock(&m);
```
non bloccante; ritorna `EBUSY`

```
pthread_mutex_destroy(&m);
```

Pattern sezione critica

```
pthread_mutex_lock(&m);   ...dato condiviso...   pthread_mutex_unlock(&m);
```

TOR VERGATA

Variabili di Condizione – Concetto e API

Concetto

- Permettono a un thread di **aspettare** che una condizione diventi vera
- Usate **sempre** insieme a un mutex
- `cond_wait`: **atomicamente** rilascia il mutex e dorme
- Al risveglio **riprende** il mutex automaticamente

Regola d'oro: WHILE, non IF

I riattivi spurii (spurious wakeup) sono possibili.
Controllare sempre la condizione in un `while`.

API POSIX

```
pthread_cond_t c =
  PTHREAD_COND_INITIALIZER;
```

```
pthread_cond_wait(&c, &m);
```
rilascia `m`, dorme, riprende `m`

```
pthread_cond_signal(&c);
```
sveglia **un** thread

```
pthread_cond_broadcast(&c);
```
sveglia **tutti** i thread

```
pthread_cond_destroy(&c);
```

TOR VERGATA

Variabili di Condizione – Pattern d'uso

Thread in attesa

```
pthread_mutex_lock(&m);

/* WHILE, non if */
while (!condizione)
    pthread_cond_wait(&c, &m);

/* condizione vera: lavora */
...

pthread_mutex_unlock(&m);
```

Thread che modifica la condizione

```
pthread_mutex_lock(&m);

/* modifica il dato */
condizione = true;

/* sveglia chi aspetta */
pthread_cond_signal(&c);
/* oppure broadcast */

pthread_mutex_unlock(&m);
```

Perché `broadcast` invece di `signal`?

Usare `broadcast` quando più thread possono avanzare (es. Readers-Writers) o quando la condizione si azzera (es. tutti i lettori terminano → sveglia lo scrittore).

TOR VERGATA

Confronto dei Meccanismi IPC

Tabella comparativa

Meccanismo	Processi indipendenti?	Accesso casuale?	Sincronia automatica?	Prestazioni
Pipe anonima	**No**	**No**	**Sì**	Buone
FIFO	**Sì**	**No**	**Sì**	Buone
Shm tra processi	**Sì**	**Sì**	**No**	**Ottime**
Shm tra thread	*N/A*	**Sì**	**No**	**Ottime**

Regola generale

Più alte sono le prestazioni, più la sincronizzazione è a carico del programmatore.

TOR VERGATA

Esercizi

Esercizio 1 – Eco Filtrato (Pipe anonima)

Eco Filtrato

Sviluppare un'applicazione C (POSIX) con una **pipe anonima** tra processo padre e figlio.

- Il padre legge parole da `stdin`:
  - Parola **non** inizia con # $\rightarrow$ invia al figlio tramite pipe.
  - Parola inizia con # $\rightarrow$ stampa rimuovendo il carattere #.
- Il figlio riceve la parola, sostituisce le **vocali con** * e la stampa.

Soluzione 1 – es1_pipe.c

```c
void replace_vowels(char *w) {
    for (int i = 0; w[i]; i++) {
        switch (tolower((unsigned char)w[i])) {
            case 'a': case 'e': case 'i':
            case 'o': case 'u': w[i] = '*';
        }
    }
}
int main(void) {
    int fd[2];  pipe(fd);
    if (fork() == 0) {                  /* FIGLIO */
        close(fd[1]);
        FILE *fin = fdopen(fd[0], "r");
        char word[256];
        while (fscanf(fin, "%255s", word) == 1) {
            replace_vowels(word);
            printf("[figlio] %s\n", word);
        }
        fclose(fin); exit(0);
    }
    close(fd[0]);                       /* PADRE  */
    char word[256];
    while (scanf("%255s", word) == 1) {
        if (word[0] == '#') printf("[padre] %s\n", word+1);
        else                dprintf(fd[1], "%s\n", word);
    }
    close(fd[1]);  wait(NULL);
}
```

Esercizio 2 – Command & Executor (FIFO)

Command & Executor

Progettare una comunicazione tramite FIFO tra due programmi separati:

- **command**: invia comandi su una FIFO esistente. Se la FIFO non ha lettori, attende 5 secondi e riprova (tempo indefinito). Legge `time` o `whoami` dall'utente.
- **executor**: crea le FIFO, si mette in ascolto, crea un processo figlio che esegue il comando tramite `exec` redirigendo l'output sulla FIFO di risposta.

TOR VERGATA

Soluzione 2 – es2_executor.c (frammento)

```c
mkfifo(FIFO_CMD, 0660);  mkfifo(FIFO_OUT, 0660);
for (;;) {
    /* Aperture bloccanti: si sincronizzano con command */
    int fd_in  = open(FIFO_CMD, O_RDONLY);
    int fd_out = open(FIFO_OUT, O_WRONLY);
    char cmd[256] = {0};
    read(fd_in, cmd, sizeof(cmd)-1);
    close(fd_in);
    cmd[strcspn(cmd, "\n")] = '\0';

    char *exe = strcmp(cmd,"time")   == 0 ? "date"   :
                strcmp(cmd,"whoami") == 0 ? "whoami" : NULL;
    if (!exe) { write(fd_out,"ERRORE\n",7); close(fd_out); continue; }

    pid_t pid = fork();
    if (pid == 0) {
        dup2(fd_out, STDOUT_FILENO);  /* stdout -> FIFO_OUT */
        close(fd_out);
        execlp(exe, exe, NULL);
    }
    waitpid(pid, NULL, 0);
    close(fd_out);  /* EOF per command */
}
```

Soluzione 2 – es2_command.c (frammento)

```c
fgets(line, sizeof(line), stdin);
line[strcspn(line, "\n")] = '\0';

/* Ritenta ogni 5s finche' executor non e' pronto */
int fd_cmd;
while (1) {
    fd_cmd = open(FIFO_CMD, O_WRONLY | O_NONBLOCK);
    if (fd_cmd != -1) break;
    if (errno == ENXIO || errno == ENOENT) {
        printf("[command] Attendo executor... 5s\n"); sleep(5);
    } else { perror("open"); exit(1); }
}
write(fd_cmd, line, strlen(line));
write(fd_cmd, "\n", 1);
close(fd_cmd);

/* Apertura bloccante: sincronizza con l'apertura O_WRONLY di executor */
int fd_in = open(FIFO_OUT, O_RDONLY);
char buf[512]; ssize_t n;
while ((n = read(fd_in, buf, sizeof(buf)-1)) > 0) {
    buf[n] = '\0'; printf("%s", buf);
}
close(fd_in);
```

Esercizio 3 – Fruit Picker 2 (Shared Memory)

Fruit Picker 2

Simulare la raccolta della frutta con shared memory e thread.

- Il processo principale crea un **processo figlio** e genera randomicamente frutti in una **variabile condivisa** (shared memory).
- Il figlio crea **N thread**: ognuno prova a raccogliere la frutta e tiene traccia di quanta ne raccoglie.
- **Un solo thread** alla volta può raccogliere.
- Il padre **non genera** un nuovo frutto finché quello corrente non è stato raccolto.

TOR VERGATA

Soluzione 3 – Struttura condivisa e padre

```c
typedef struct {
    char  fruit[32];
    int   done;          /* 1 = nessun altro frutto verra' generato */
    sem_t sem_ready;     /* padre posta (1x) -> UN thread si sveglia */
    sem_t sem_picked;    /* thread posta     -> padre genera prossimo */
} SharedMem;

int main(void) {
    SharedMem *shm = mmap(NULL, sizeof(SharedMem),
        PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
    sem_init(&shm->sem_ready,  1, 0); /* pshared=1 */
    sem_init(&shm->sem_picked, 1, 0);

    if (fork() == 0) { run_child(shm); /* lancia N_THREADS */ }

    for (int i = 0; i < TOTAL_FRUIT; i++) {
        strcpy(shm->fruit, FRUITS[rand() % N_TYPES]);
        printf("[padre] generato: %s\n", shm->fruit);
        sem_post(&shm->sem_ready);   /* segnala UN thread */
        sem_wait(&shm->sem_picked);  /* aspetta raccolta   */
    }
    shm->done = 1;
    for (int i = 0; i < N_THREADS; i++) sem_post(&shm->sem_ready);
    wait(NULL);
}
```

Soluzione 3 – Thread picker

```c
typedef struct { SharedMem *shm; int id; int count; } ThreadArg;

void *picker(void *arg) {
    ThreadArg *a = arg;
    while (1) {
        sem_wait(&a->shm->sem_ready);   /* attende frutto */
        if (a->shm->done) {
            /* Shutdown: risveglia il prossimo thread in attesa */
            sem_post(&a->shm->sem_ready);
            break;
        }
        /* Sezione critica: un solo thread arriva qui perche'
         * sem_ready viene postato una volta sola per frutto. */
        printf("[thread %d] raccolto: %s\n", a->id, a->shm->fruit);
        a->count++;
        sem_post(&a->shm->sem_picked);  /* notifica padre */
    }
    return NULL;
}

/* run_child: crea N_THREADS, aspetta il join e stampa i conteggi */
void run_child(SharedMem *shm) {
    ThreadArg args[N_THREADS]; pthread_t tids[N_THREADS];
    for (int i = 0; i < N_THREADS; i++) {
        args[i] = (ThreadArg){shm, i, 0};
        pthread_create(&tids[i], NULL, picker, &args[i]);
    }
    for (int i = 0; i < N_THREADS; i++) { pthread_join(tids[i], NULL);
        printf("[figlio] thread %d -> %d frutti\n", i, args[i].count); }
    exit(0);
}
```

Esercizio 4 – Produttore-Consumatore (Semafori + Mutex)

Produttore-Consumatore a buffer circolare

- Un **buffer circolare** di `K=5` slot è condiviso tra thread.
- **1 thread produttore** inserisce gli interi da 1 a 20, poi invia un sentinel (`-1`) per ogni consumatore.
- **2 thread consumatori** prelevano gli item e stampano quanti ne hanno consumati al termine.
- Nessun busy-wait: usare `sem_empty` + `sem_full` e un **mutex** per proteggere `head`/`tail`.

TOR VERGATA

Soluzione 4 – Produttore e setup

```c
#define BUF_SIZE 5
#define N_ITEMS 20
#define N_CONS  2
int buf[BUF_SIZE], head=0, tail=0;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
sem_t empty, full; /* init: BUF_SIZE e 0 */

void *producer(void *arg) {
    for (int i = 1; i <= N_ITEMS; i++) {
        sem_wait(&empty);    /* attende slot libero */
        pthread_mutex_lock(&mtx);
        buf[tail] = i;  tail = (tail+1) % BUF_SIZE;
        pthread_mutex_unlock(&mtx);
        sem_post(&full);     /* segnala item pronto */
    }
    for (int i = 0; i < N_CONS; i++) { /* sentinel x consumatore */
        sem_wait(&empty);
        pthread_mutex_lock(&mtx);
        buf[tail] = -1; tail = (tail+1) % BUF_SIZE;
        pthread_mutex_unlock(&mtx);
        sem_post(&full);
    }
    return NULL;
}
```

TOR VERGATA

Soluzione 4 – Consumatore

```c
void *consumer(void *arg) {
    int id = *(int *)arg, count = 0;
    while (1) {
        sem_wait(&full);     /* attende item pronto */
        pthread_mutex_lock(&mtx);
        int val = buf[head]; head = (head+1) % BUF_SIZE;
        pthread_mutex_unlock(&mtx);
        sem_post(&empty);    /* libera slot */
        if (val == -1) break; /* sentinel: termina */
        count++;
        printf("[consumer %d] prelevato %d\n", id, val);
    }
    printf("[consumer %d] fine -- %d elementi\n", id, count);
    return NULL;
}
int main(void) {
    sem_init(&empty, 0, BUF_SIZE);
    sem_init(&full,  0, 0);
    pthread_t prod, cons[N_CONS]; int ids[N_CONS];
    pthread_create(&prod, NULL, producer, NULL);
    for (int i=0; i<N_CONS; i++) {
        ids[i]=i;
        pthread_create(&cons[i], NULL, consumer, &ids[i]);
    }
    pthread_join(prod, NULL);
    for (int i=0; i<N_CONS; i++) pthread_join(cons[i], NULL);
}
```

TOR VERGATA

Esercizio 5 – Lettori-Scrittori (Mutex + Cond. Var.)

Readers-Writers

Un intero condiviso è acceduto da thread lettori e scrittori.

- **3 thread lettori**: leggono il valore più volte; più lettori possono leggere **contemporaneamente**.
- **2 thread scrittori**: modificano il valore; richiedono **accesso esclusivo** (nessun altro attivo).
- Implementare **manualmente** il lock con `pthread_mutex_t` + `pthread_cond_t` (vietato `pthread_rwlock_t`).
- Stampare ogni operazione con l'ID del thread e il numero di lettori attivi.

Soluzione 5 – Struttura e lock lettura

```c
typedef struct {
    int value, readers, writer;
    pthread_mutex_t mtx;
    pthread_cond_t  cond; /* unica CV per lettori e scrittori */
} RWData;

void read_lock(RWData *rw) {
    pthread_mutex_lock(&rw->mtx);
    while (rw->writer)          /* WHILE: spurious wakeup */
        pthread_cond_wait(&rw->cond, &rw->mtx);
    rw->readers++;              /* mi registro come lettore attivo */
    pthread_mutex_unlock(&rw->mtx);
}
void read_unlock(RWData *rw) {
    pthread_mutex_lock(&rw->mtx);
    if (--rw->readers == 0)
        pthread_cond_broadcast(&rw->cond); /* sveglia scrittori */
    pthread_mutex_unlock(&rw->mtx);
}
```

Soluzione 5 – Lock scrittura e thread

```c
void write_lock(RWData *rw) {
    pthread_mutex_lock(&rw->mtx);
    while (rw->readers > 0 || rw->writer) /* accesso esclusivo */
        pthread_cond_wait(&rw->cond, &rw->mtx);
    rw->writer = 1;
    pthread_mutex_unlock(&rw->mtx);
}
void write_unlock(RWData *rw) {
    pthread_mutex_lock(&rw->mtx);
    rw->writer = 0;
    pthread_cond_broadcast(&rw->cond); /* sveglia tutti i waiter */
    pthread_mutex_unlock(&rw->mtx);
}

void *reader(void *arg) { /* ... */
    read_lock(rw);
    printf("[reader %d] value=%d (lettori: %d)\n",
           id, rw->value, rw->readers);
    read_unlock(rw);
}
void *writer(void *arg) { /* ... */
    write_lock(rw);
    rw->value += (id == 0) ? +1 : -1;
    printf("[writer %d] value -> %d\n", id, rw->value);
    write_unlock(rw);
}
```

TOR VERGATA