Esercitazione:
UNIX IPC, Shared Memory e Semafori

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

Interprocess Communication (IPC)

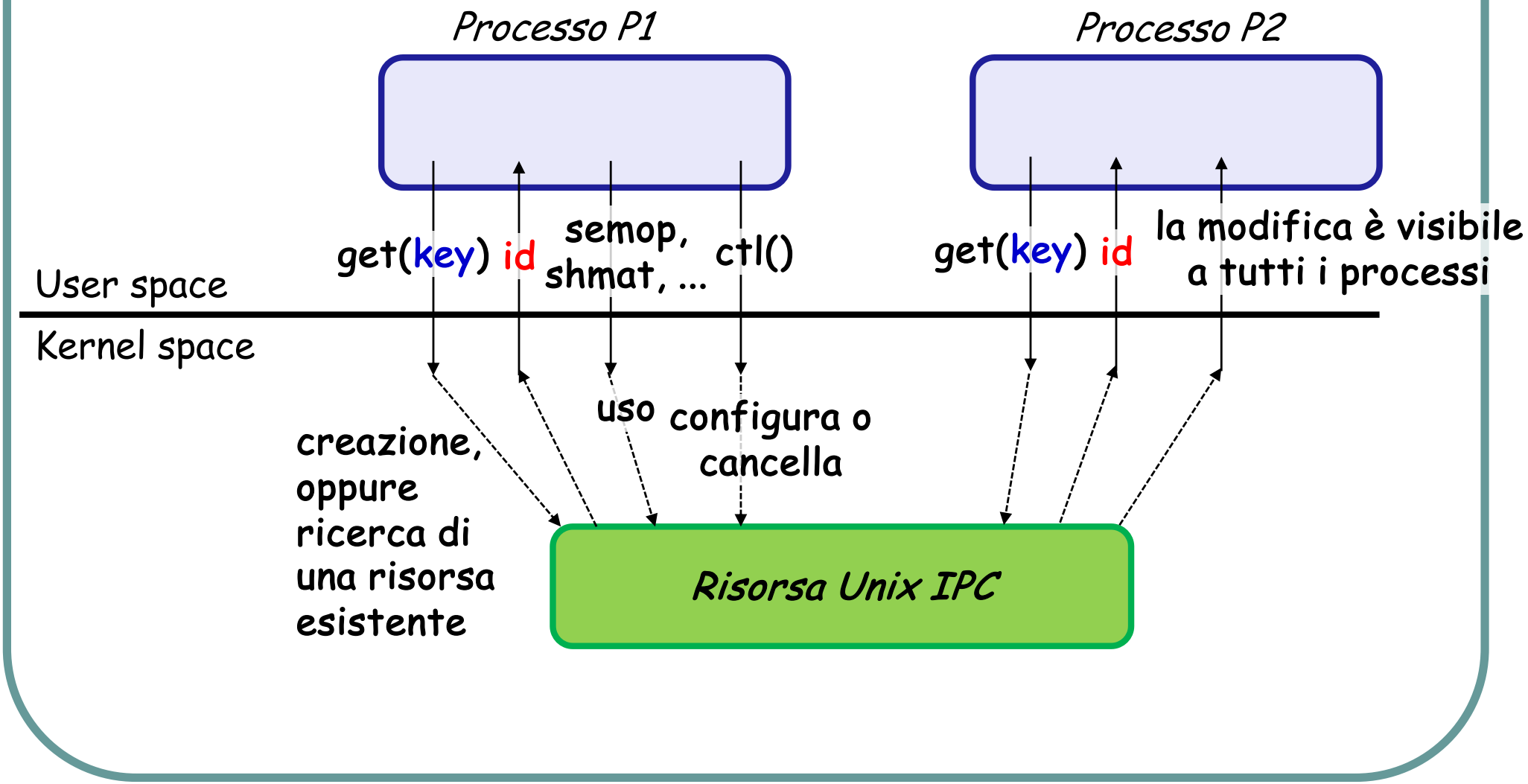
- Linux/UNIX permette la **comunicazione tra processi** mediante primitive e strutture dati fornite dal kernel

- Oggetti (o risorse) IPC in UNIX:
  - Memoria condivisa (SHM : shared memory segments)
  - Semafori (SEM: semaphore arrays)
  - Code di messaggi (MSG: queues)

Primitive GET e CTL

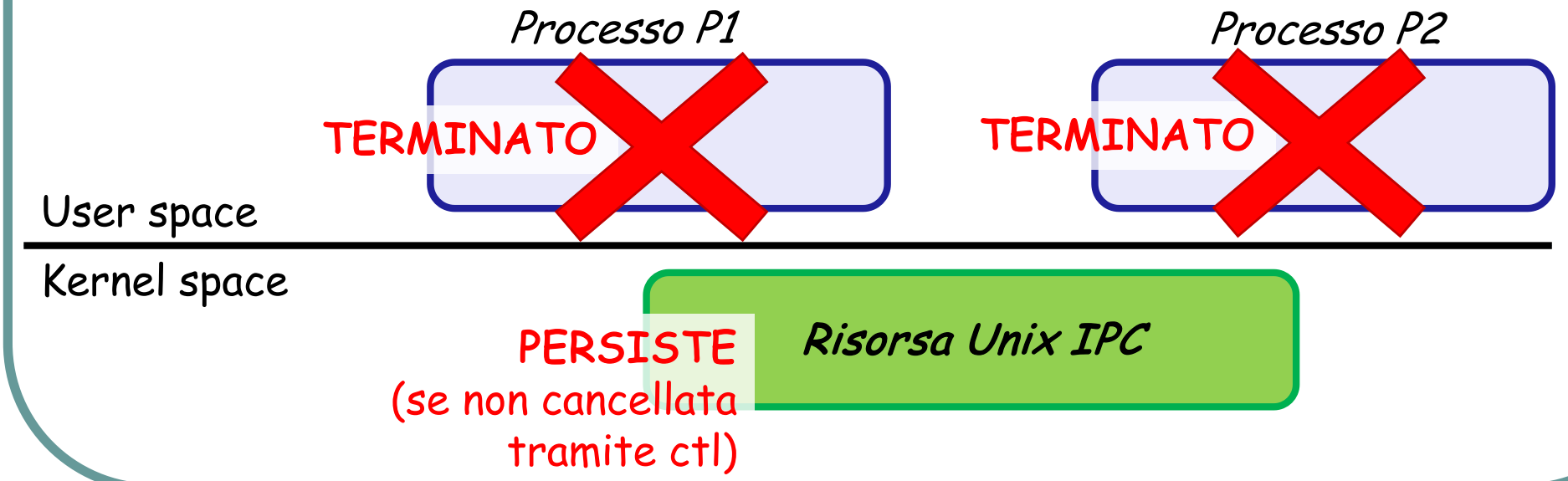
- Ogni risorsa IPC è gestita da due primitive get/ctl
- La primitiva **get** utilizza una "chiave" (IPC key), ed opportuni parametri, per restituire al processo un **descrittore della risorsa**
- La primitiva **ctl** (control) permette, dato un descrittore, di:
  - Verificare lo stato di una risorsa
  - Cambiare lo stato di una risorsa
  - Rimuovere una risorsa

Primitive GET e CTL



Primitive GET e CTL

- Le risorse IPC sono permanenti
  - se un processo esce, non sono eliminate automaticamente
  - è necessaria una chiamata esplicita alla primitiva ctl



Primitiva get

```
int …get (key_t key, …, int flag);
```

- **key**: chiave dell'oggetto
  - valore intero arbitrario «cablato», oppure
  - valore generato da ftok(), oppure
  - la costante IPC_PRIVATE

Primitiva get

```
int …get (key_t key, …, int flag);
```

- **flag**: indica la modalità di acquisizione della risorsa
  - Una o più costanti, passate insieme in "OR" (carattere pipe "|")

Esempio tipico:

```
int id = …get(…, IPC_CREAT | 0644);
```

descrittore della risorsa

si richiede di creare la risorsa, se non esiste già

permessi da assegnare alla creazione

Primitiva get

```
int …get (key_t key, …, int flag);
```

- **flag**: indica la modalità di acquisizione della risorsa
  - Una o più costanti, passate insieme in "OR" (carattere pipe "|")
  - nessun valore: utilizza la risorsa ***se già esiste***, altrimenti dà errore
  - **IPC_CREAT**: Crea una nuova risorsa se non ne esiste già una con la chiave indicata. Se già esiste, viene riusata (il flag è ininfluente)
  - **IPC_EXCL**: da usare in combinazione con IPC_CREAT, permette di gestire i due casi di risorsa già esistente e risorsa appena creata (utile per gestire il valore iniziale della risorsa)
  - Permessi di accesso (es. 0644)

Primitiva ctl

```
int …ctl (int desc, …, int cmd, …);
```

- **desc**: indica il descrittore della risorsa
- **cmd**: il comando da eseguire:
  - IPC_RMID: rimozione della risorsa
  - IPC_STAT: richiede informazioni sulla risorsa
  - IPC_SET: richiede la modifica di attributi della risorsa (es. permessi di accesso)

IPC keys (chiave cablata nel codice)

- Ogni risorsa IPC è identificata da un **valore univoco nel sistema**, denominato chiave (IPC key)

Esempio:

```
key_t mykey = 123;
int id = ....get(mykey, ...);
```

Il modo più semplice di scegliere una chiave è usare un valore a piacere del programmatore, "cablato" nel programma

Problema: esiste il rischio che più sviluppatori di programmi diversi **scelgano (per caso!) lo stesso valore**!

IPC keys (ftok)

```
key_t ftok(char * path, char id);
```

Esempio:

```
key_t mykey = ftok("./percorso", 'a');
```

- Genera una chiave automaticamente
- Parametri di ingresso:
  - Il percorso di un file/cartella appartenente al programma
  - Un carattere (scelto a piacere)

Se due processi (es. di uno stesso programma) usano gli **stessi parametri**, ottengono le **stesse chiavi** (la funzione è deterministica)

Processi di programmi diversi useranno percorsi diversi, evitando il problema della collisione delle chiavi

IPC keys (IPC_PRIVATE)

- **IPC_PRIVATE** (equivale a 0), è un valore costante, che può essere usato per creare una risorsa senza chiave
- Semplifica la gestione
- Problema: come fanno altri processi ad accedervi?

Esempio:

```
key_t mykey = IPC_PRIVATE;
int id = ....get(mykey, ...);

pid = fork();
if(pid == 0) {
    /* figlio, utilizza il descrittore "id" */
}
else if(pid > 0) {
    /* padre, utilizza anche lui "id" */
}
```

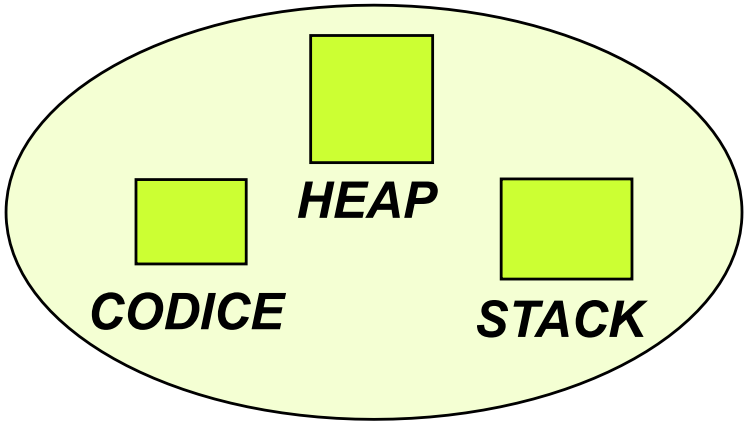
L'unico modo per condividere questo tipo di risorsa è tramite il meccanismo di fork

ipcs e ipcrm

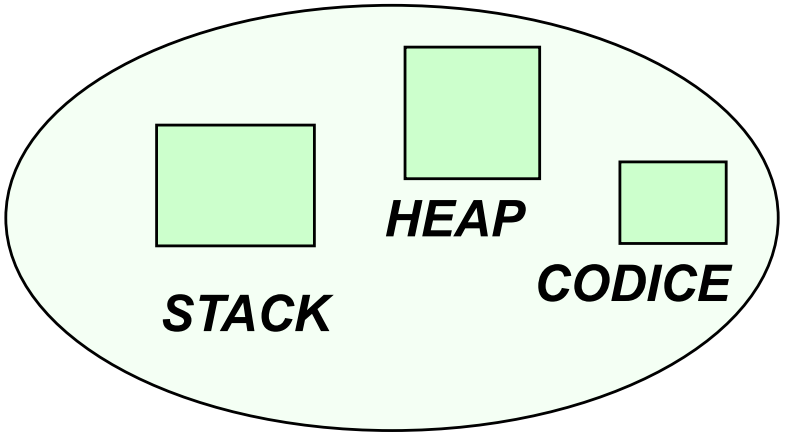
- **`ipcs <-m|-s|-q>`**
  - visualizza le risorse IPC allocate nel sistema
  - ne mostra l'identificatore e l'utente proprietario
- **`ipcrm <shm|sem|msg> <IPC_ID>`**
  - rimuove una risorsa IPC
  - utilizzabile qualora i processi non abbiano rimosso le risorse (es. in caso di terminazione anomala)

Memoria condivisa

- I processi Unix (a differenza dei thread) non condividono alcuna area di memoria



Memoria del processo P1



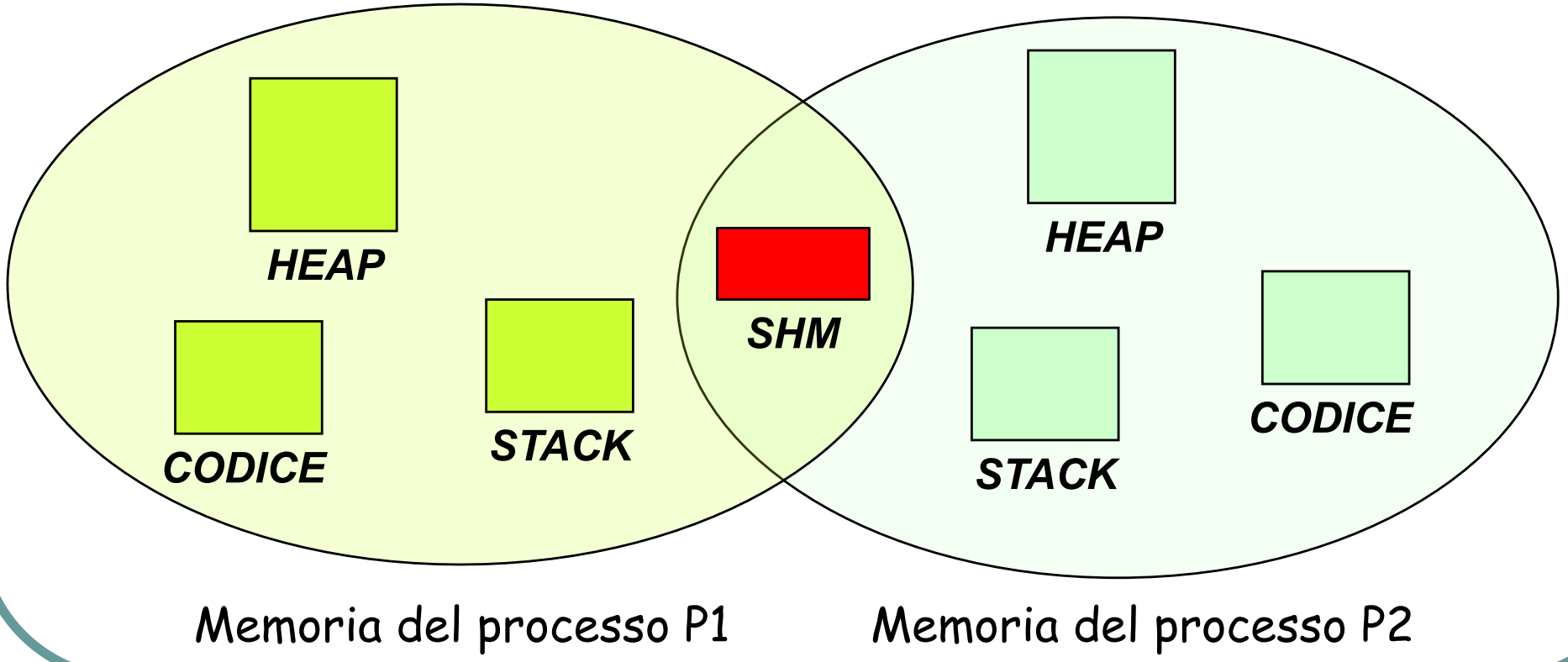
Memoria del processo P2

Nota: questo vale anche per i processi "parenti":

- Il figlio ottiene una copia dei dati del padre
- Ogni processo modifica la propria copia

Memoria condivisa

- Una memoria condivisa (SHM) è una porzione di memoria accessibile da più processi



Memoria condivisa: Utilizzo

- Creazione della SHM
- Collegamento alla SHM
- Uso della SHM
- Scollegamento della SHM
- Eliminazione della SHM

Creazione

```
int shmget(key_t key, int size, int flag)
```

- **size**: dimensione in byte della memoria condivisa

- restituisce un identificatore numerico (descrittore) in caso di successo
- oppure, -1 in caso di fallimento

Esempio 1 – shmget senza flags

```
key_t chiave = 40;
int ds_shm;
ds_shm = shmget(chiave, 1024, 0);

if(ds_shm < 0) {
  // la risorsa non esiste! esci dal programma
  perror("errore shmget!");
  exit(1);
}
... Utilizza la shared memory già esistente...
```

- Cerca un segmento di memoria condivisa con chiave 40 e dimensione 1KB
- Se la SHM non è stata **già creata** da un altro processo, la shmget() produce un errore (valore di ritorno negativo)

Esempio 2 – shmget con IPC_CREAT

```
key_t chiave = 40;
int ds_shm;
ds_shm = shmget(chiave, 1024, IPC_CREAT | 0664);

if(ds_shm < 0) {
  // qualcosa è andato storto (memoria esaurita, etc.)
  perror("errore shmget!");
  exit(1);
}
... Utilizza la shared memory (esistente o nuova)...
```

- Se non esiste già, IPC_CREAT **crea una nuova shared memory**
- Se esiste già, **si ri-utilizza** il segmento esistente
- Solo l'utente ed il gruppo proprietari hanno permessi di accesso RW (66), mentre gli altri hanno solo accesso in lettura (4)

Esempio 3 – ftok()

```
key_t chiave = ftok("./eseguibile", 'k');
int ds_shm;
ds_shm = shmget(chiave, 1024, IPC_CREAT | 0664);

if(ds_shm < 0) {
  // qualcosa è andato storto (memoria esaurita, etc.)
  perror("errore shmget!");
  exit(1);
}
... Utilizza la shared memory (esistente o nuova)...
```

- Se due programmi usano gli stessi parametri in ftok() ottengono la stessa chiave

Esempio 4 – IPC_PRIVATE

```
int ds_shm = shmget(IPC_PRIVATE, 1024, IPC_CREAT|0664);

if(ds_shm < 0) {
  // qualcosa è andato storto (memoria esaurita, etc.)
  perror("errore shmget!");
  exit(1);
}
... Utilizza la shared memory ...
```

- Crea un nuovo segmento **senza assegnare** una chiave (in ipcs, la chiave apparirà «0»)
- La risorsa è **utilizzabile solo dal padre e i suoi figli**

Esempio 5 – IPC_CREAT + IPC_EXCL

```
key_t chiave = 40;
int ds_shm = shmget(chiave,1024,IPC_CREAT|IPC_EXCL|0664);
if(ds_shm >= 0) {




} else {




}
 ... Utilizza la shared memory ...
```

La combinazione di IPC_CREAT+IPC_EXCL consente di gestire separatamente i due casi di **risorsa già esistente** e di **risorsa appena creata**.

Esempio 5 – IPC_CREAT + IPC_EXCL

```
key_t chiave = 40;
int ds_shm = shmget(chiave,1024,IPC_CREAT|IPC_EXCL|0664);
if(ds_shm >= 0) {
   // La risorsa non esisteva, ed è stata appena creata.
   // Il programma ha l'opportunità di inizializzarne
   // il contenuto (ad esempio, «0» o una stringa vuota).
   ...
   memcpy(shared_mem, 0, 1024);
} else {
   // Se negativo, indica che la risorsa già esiste
   // Non rappresenta un "problema", ma è un modo per
   // capire che la risorsa non va inizializzata

   ds_shm = shmget(chiave, 1024, 0);
   if(ds_shm<0) { /* errore */ }
}
 ... Utilizza la shared memory ...
```

ds_shm è negativo. Occorre chiamare una seconda volta la shmget, senza usare IPC_CREAT + IPC_EXCL

Collegamento

```
void* shmat(int shmid, const void *shmaddr,
                                    int flag)
```

Collega ("attach") il segmento di memoria allo spazio di indirizzamento del chiamante

- **shmid**: identificatore del segmento di memoria
- Restituisce un puntatore all'area di memoria collegata, oppure -1 in caso di fallimento

Parametri opzionali (si possono porre a NULL e 0):

- **shmaddr** (opzionale): indirizzo al quale collegare il segmento di memoria condivisa. Se NULL, l'indirizzo viene automaticamente scelto dal SO.
- **flag** (opzionale): 0 per lettura/scrittura, IPC_RDONLY per collegare in sola lettura

Utilizzo

- Leggere e scrivere una memoria condivisa non richiede di usare chiamate di sistema
- L'area può essere acceduta **come una qualsiasi variabile** dello spazio di indirizzamento del processo

Esempio:

```
mystruct * p = shmat(ds_shm, NULL, 0);

p->variabile = 123;  // la modifica è visibile
                     // anche ad altri processi
```

Esempio 6 – esempio completo

```
key_t chiave = IPC_PRIVATE;
int ds_shm = shmget(chiave, sizeof(int),IPC_CREAT|0664);
if(ds_shm<0) { perror("errore shmget!"); exit(1); }

int * p = (int*) shmat(ds_shm, NULL, 0);
if(p==(void*)-1) { perror("errore shmget!"); exit(1); }

key_t pid = fork();
if(pid==0) {
    // processo figlio
    *p = 123;
    exit(0);
} else if(pid>0) {
    // processo padre
    wait(NULL);
    printf("Contenuto SHM: %d\n", *p);
}
```

NOTE:

- Sia il processo padre sia il figlio ottengono una copia del puntatore a memoria condivisa
- La dimensione della memoria condivisa è atta a contenere una variabile intera "sizeof(int)"

Esempio 7 – esempio avanzato (stringhe)

```
key_t chiave = ftok(“./eseguibile”, ‘k’);
int ds_shm = shmget(chiave,100,IPC_CREAT|IPC_EXCL|0664);
char * p;

if(ds_shm < 0) {
  // la risorsa già esiste (ma occorre una seconda shmget)
  ds_shm = shmget(chiave, 100, 0664);
  p = (char*) shmat(ds_shm, NULL, 0);
} else {
  // la risorsa è stata appena creata
  p = (char*) shmat(ds_shm, NULL, 0);
  strncpy(p, "Ciao", sizeof("Ciao")); // inizializza
}

...
printf("Contenuto SHM: %s\n", p);
```

Operazione di controllo

```
int shmctl(int ds_shm, int cmd,
                       struct shmid_ds * buff)
```

Esegue un comando su una SHM

- **ds_shm**: descrittore della memoria condivisa su cui si vuole operare
- **cmd**: specifica del comando da eseguire

Esempio tipico:

```
shmctl(ds_shm, IPC_RMID, NULL);
```

Nota: la memoria condivisa non viene cancellata subito, ma solo quando nessun processo vi è più collegato

Operazione di controllo

```
int shmctl(int ds_shm, int cmd,
                        struct shmid_ds * buff)
```

Esegue un comando su una SHM

- **ds_shm**: descrittore della memoria condivisa su cui si vuole operare
- **cmd**: specifica del comando da eseguire:
  - IPC_STAT
  - IPC_SET
  - **IPC_RMID**: marca da eliminare, rimuove solo quando non vi sono più processi attaccati
  - **SHM_LOCK**: impedisce che il segmento venga swappato o paginato
- **buff**: puntatore ad una struttura di tipo **shmid_ds** (parametro di ingresso nel caso del comando IPC_SET, parametro di uscita nel caso del comando IPC_STAT)
- Restituisce -1 in caso di errore

Operazione di controllo

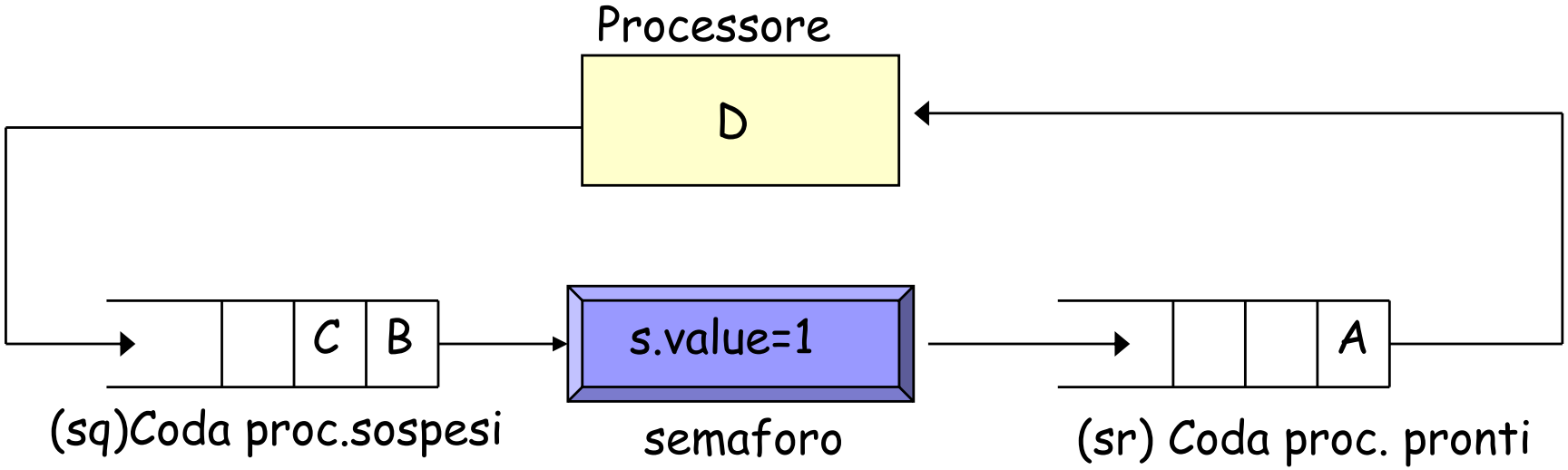
shmid_ds è definito nell' header <sys/shm.h>

```
struct shmid_ds {
  struct ipc_perm shm_perm;/* permessi */
  size_t shm_segsz;        /* dim del segm. in bytes */
  __time_t shm_atime;      /* tempo ultimo shmat() */
  __time_t shm_dtime;      /* tempo ultimo shmdt() */
  __time_t shm_ctime;      /* tempo ultimo shmctl() */
  __pid_t shm_cpid;        /* pid del creatore */
  __pid_t shm_lpid;        /* pid dell'ultimo operatore */
  shmatt_t shm_nattch;     /* # di attach. correnti */
};
```

Header file

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/shm.h>

Semafori: modello concettuale



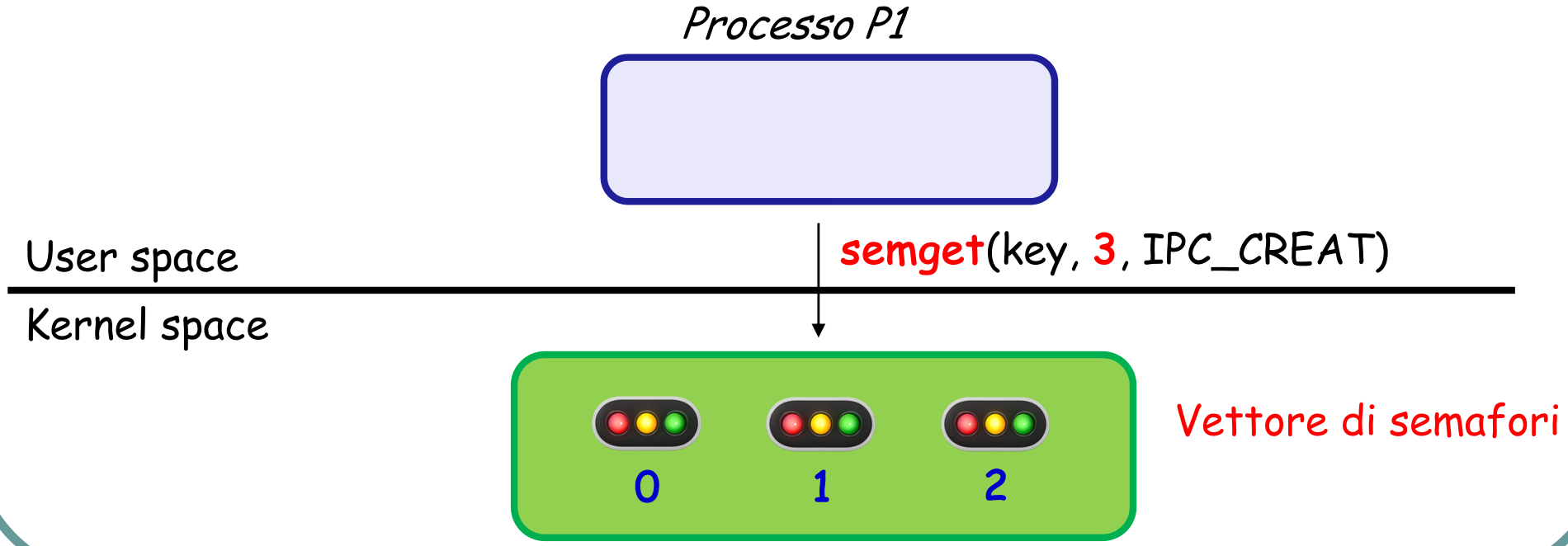

```
void wait(semaphore s) {
    s.value--;
    if (s.value<0){
            s.queue.insert(Process);
            suspend(Process);
    }
}
```

```
void signal(semaphore s) {
    s.value++;
    if (s.value<=0){
            s.queue.remove(Process);
            wake-up(Process);
    }
}
```

Creazione ed inizializzazione di semafori

```
int semget(key_t key, int nsems, int semflg);
```

```
int semctl(int semid, int semnum, int cmd, …);
```



Creazione ed inizializzazione di semafori

```
int semget(key_t key, int nsems, int semflg);
```

```
int semctl(int semid, int semnum, int cmd, …);
```

```
key_t chiave_sem = IPC_PRIVATE;

//richiesta di un array di 2 semafori, ed
  inizializzazione
sem = semget(chiave_sem,2,IPC_CREAT|0664);

//Inizializzazione dei due semafori
semctl(sem,0,SETVAL, 1);    // mutex
semctl(sem,1,SETVAL, 5);    // semaforo N-ario
```

Stesso descrittore "sem", diversa posizione nel vettore di semafori ("0" e "1")

Struttura dati semaforo

- La semget definisce un ARRAY di più semafori (es. 2 nel caso precedente)

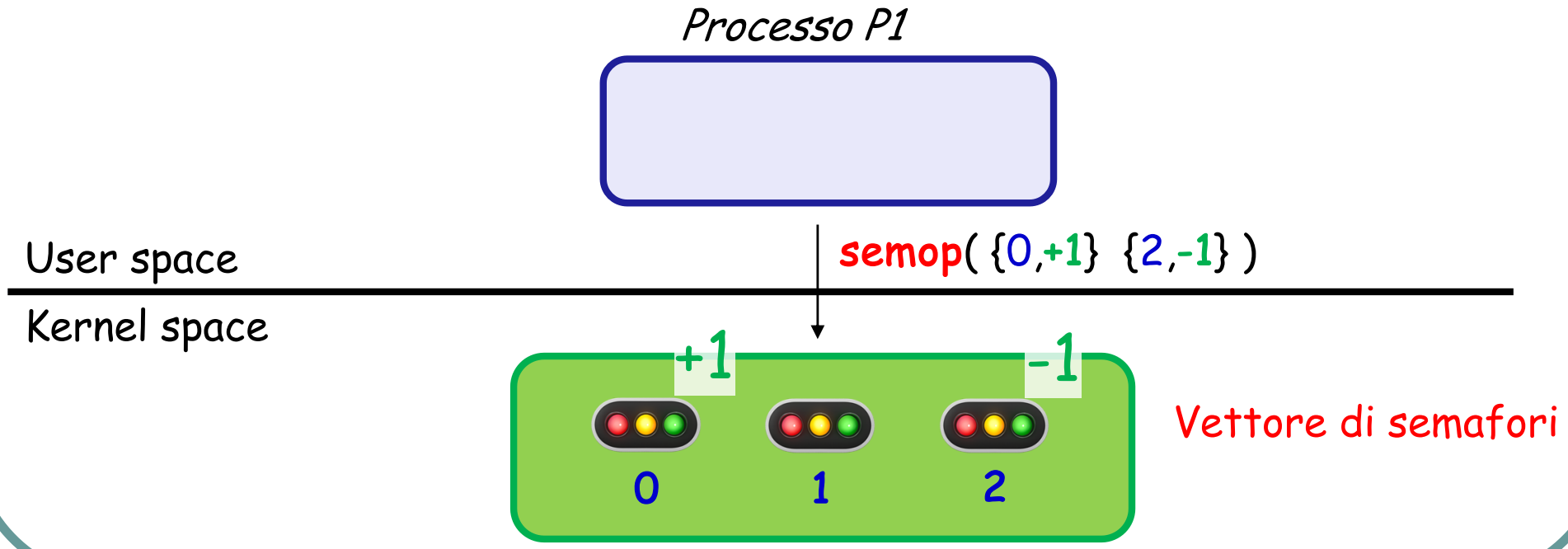
- Ogni semaforo dell'array è rappresentato internamente nel kernel da una struttura dati, che comprende tra i suoi campi i seguenti:

```
unsigned short semval;  /* valore semaforo*/
unsigned short semzcnt; /* # proc che aspettano 0 */
unsigned short semncnt; /* # proc che aspettano incr.*/
```

semop - semaphore operations

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

La primitiva semop effettua una operazione (o un gruppo di operazioni) su un vettore di semafori



semop - semaphore operations

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

La primitiva semop effettua una operazione (o un gruppo di operazioni) su un vettore di semafori

Prende in ingresso un vettore di operazioni (***sops**, di dimensione *nsops*)

Ogni operazione è rappresentata da una struttura, ***struct sembuf***:

```
struct sembuf {
  unsigned short sem_num;  /* numero di semaforo */
  short sem_op;            /* operazione da compiere */
  short sem_flg;           /* flags (IPC_NOWAIT, SEM_UNDO) */
}
```

Implementazione di una wait

```
void Wait_Sem (int id_sem, int numsem)    {

    struct sembuf sem_buf;

    sem_buf.sem_num = numsem;
    sem_buf.sem_op = -1;                    // Decrementa di -1
    sem_buf.sem_flg = 0;

    semop(id_sem, &sem_buf, 1);
}
```

Nota: il terzo parametro di semop() è "1"
(eseguiamo una sola operazione, con una sola struct sembuf)

Implementazione di una signal

```
void Signal_Sem (int id_sem, int numsem)    {

    struct sembuf sem_buf;

    sem_buf.sem_num = numsem;
    sem_buf.sem_op = +1;                    // Incrementa di +1
    sem_buf.sem_flg = 0;

    semop(id_sem, &sem_buf, 1);
}
```

Nota: il terzo parametro di semop() è "1"
(eseguiamo una sola operazione, con una sola struct sembuf)

semop - semaphore operations

Le operazioni nel vettore ***sops*** sono eseguite in maniera **atomica**

- La primitiva si sospende (o ritorna un errore) se qualcuna delle operazioni non può essere svolta (es. decremento di un semaforo già negativo)
- Quando tutte le operazioni sono possibili, avvengono simultaneamente (mutua esclusione con altre semop)

Ogni operazione è eseguita sul semaforo indicato dalla variabile ***sem_num***

- Nota: Il conteggio dei semafori nell'array parte dall'indice 0

Tramite ***sem_op***, è possibile indicare tre tipi di operazioni su un semaforo:

- ***sem_op < 0*** : *wait*
- ***sem_op > 0*** : *signal*
- ***sem_op==0*** : *wait_for_zero*

sem_op > 0: “signal”

Se *sem_op* è un intero positivo, l'operazione consiste nell'addizionare il valore di *sem_op* al valore del semaforo (*semval*).

semval += sem_op

Questa operazione non causa in alcun caso il blocco del processo.

sem_op < 0: “wait”

Se *sem_op* ha valore negativo, l'operazione si articolerà in due modi:

1. se (semval ≥ |sem_op|), l'operazione procede immediatamente (senza sospendere il processo), e il valore del semaforo sarà modificato come:

   semval -= |sem_op|

1. se (semval < |sem_op|)

   - Il processo si sospende, finché semval ≥ |sem_op|
   - Quando questa condizione sarà verificata, il processo è sbloccato e si modifica come al punto 1

sem_op == 0: “wait_for_zero”

Se *sem_op* ha valore nullo, l’operazione specificata ("wait-for-zero") è articolata nei seguenti passi:

1. se il valore *semval* è zero, l’operazione procede immediatamente (il processo non si sospende)

2. Altrimenti (semval ≠ 0), il processo si sospende finché *semval* non ritorna ad essere 0

Rimozione di una struttura dati semaforo

```
semctl(id_sem, num_sem ,IPC_RMID);
```

La variabile num_sem in questo caso viene ignorata
(es. può essere posta a "0")

Header file

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/sem.h>