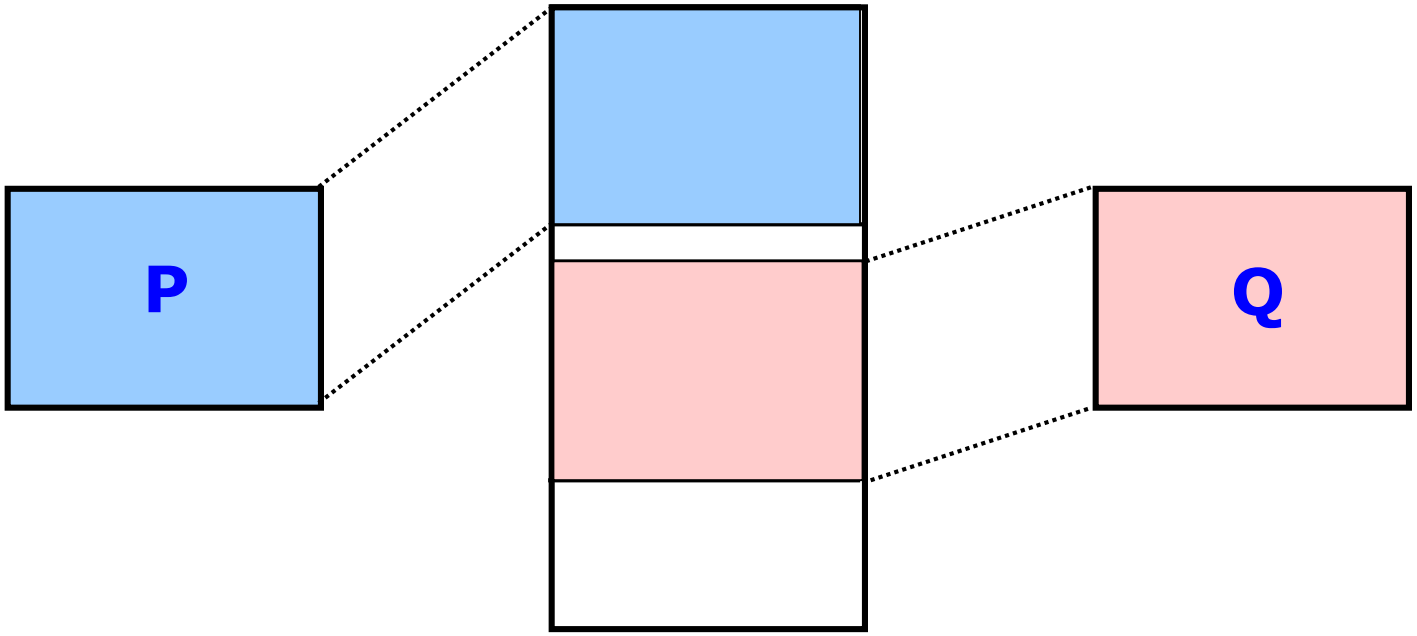
Memoria condivisa

Normalmente il meccanismo di gestione della memoria, qualunque esso sia, garantisce di allocare in RAM l'immagine di più processi in indirizzi *non sovrapposti* per evitare che uno danneggi l'altro per errore o per "malizia"

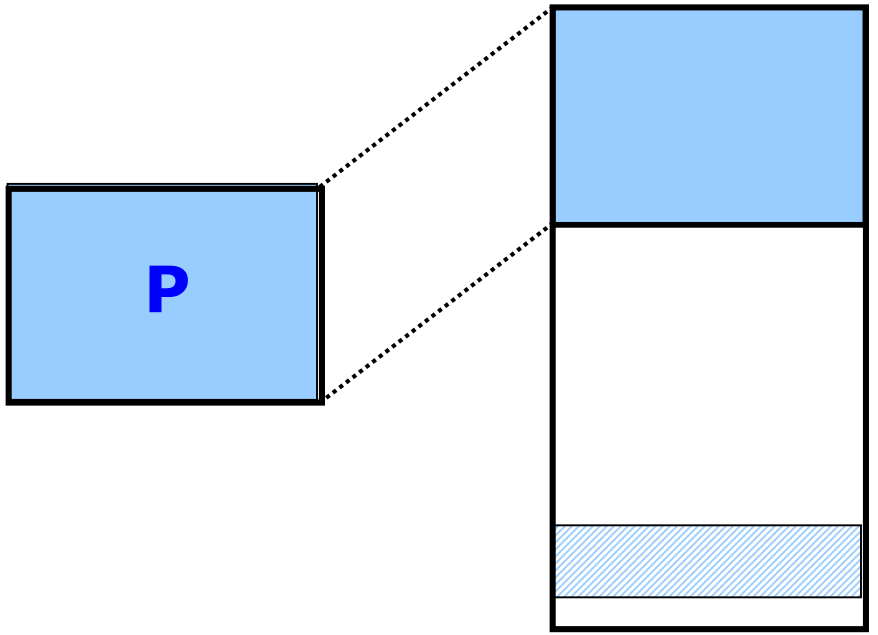
Per semplicità di illustrazione assumiamo, come meccanismo *di base*, uno semplice in cui l'immagine di ogni processo è allocata *interamente* in memoria, ed è allocata *in modo contiguo*, cioè in un *intervallo* di indirizzi della RAM:



Memoria condivisa

Nello standard POSIX si può creare un'area di memoria condivisibile fra processi con:

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, flags);
ftruncate(shm_fd,size);
```



si ha uno «shared memory object»

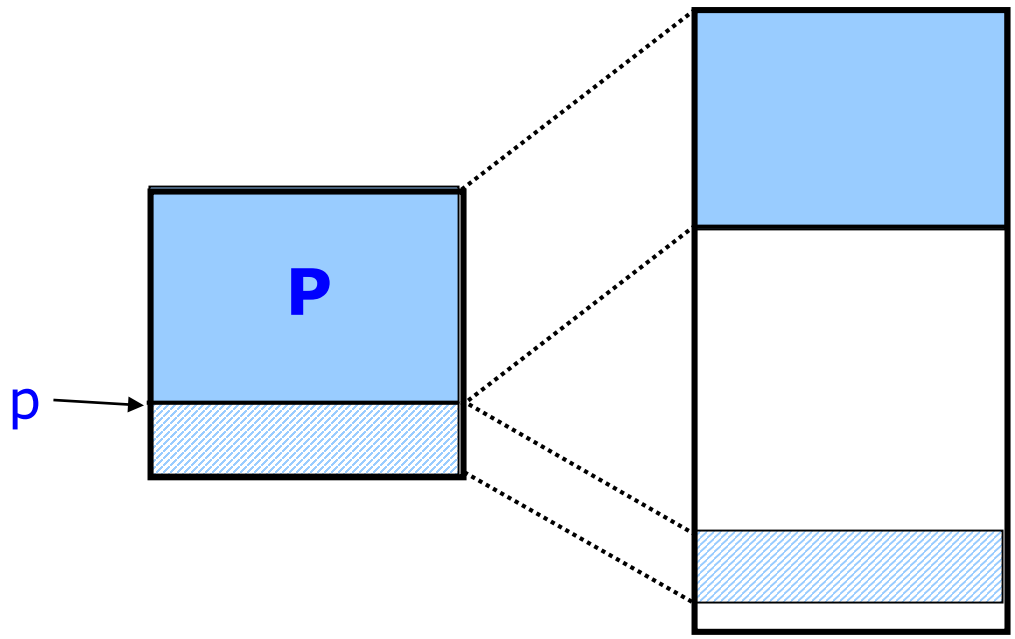
- identificato nel sistema da **`name`**, stringa che inizia con uno "/" e non ne contiene altri
- nel processo, identificato da **`shm_fd`** che è un file descriptor (!)
- usando **`O_CREAT`** viene creato se non esiste, se no solo «aperto»
- con diritti di accesso dati da **`flags`**
- di dimensione **`size`**

Memoria condivisa

L'area di memoria condivisibile può poi essere "collegata" allo spazio di indirizzamento di P con:

```
p = mmap(..., size, ..., MAP_SHARED, shm_fd, ...)
```

restituisce un puntatore, come malloc



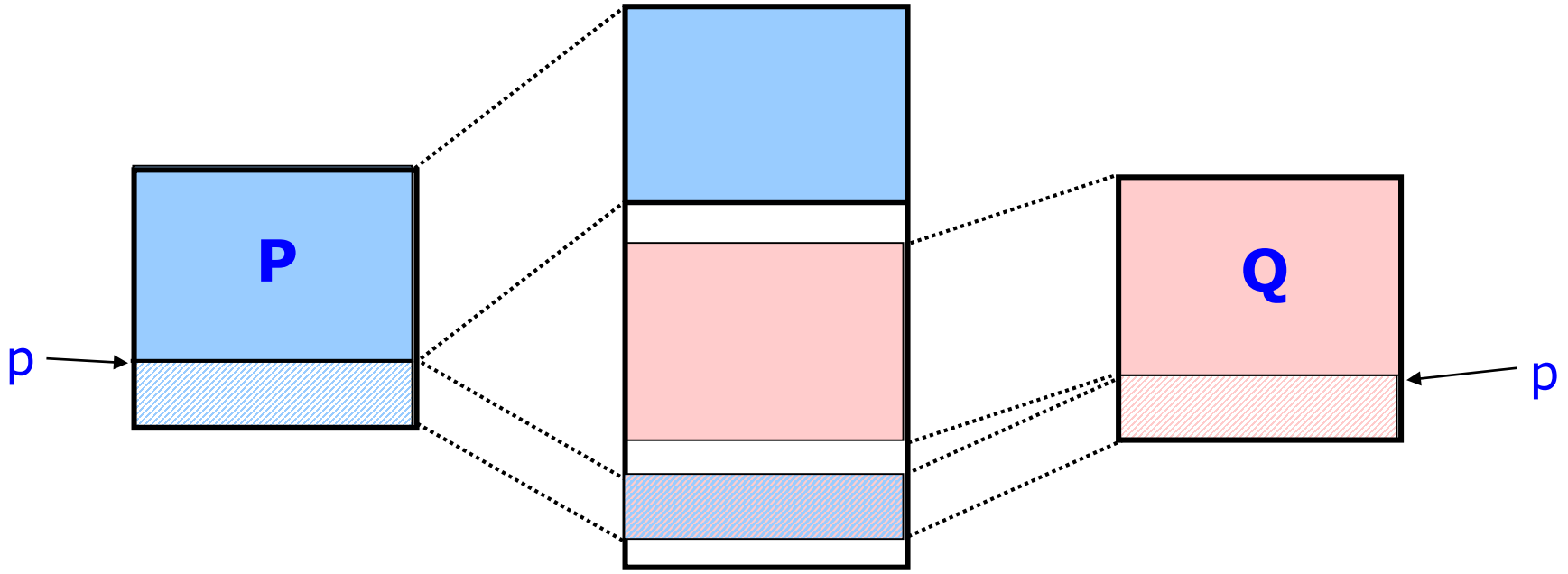
(**`mmap`** è nata per fare «I/O mappato in memoria» cioè leggere/scrivere su una porzione di file leggendo/scrivendo in memoria;

la sua interfaccia viene riusata qui, visto che la memoria condivisa è identificata da un file descriptor)

Memoria condivisa

creando nuovi processi con **`fork()`** dopo **`mmap`** abbiamo effettivamente memoria condivisa tra il processo e i suoi discendenti

processi non «parenti» in questo senso possono invece ripetere le operazioni usando lo stesso **`name`** (il primo che arriva crea l'oggetto, gli altri lo aprono solo)



Semafori (POSIX)

Nello standard POSIX sono presenti semafori:

- in versione *named,* nel senso che hanno un nome a livello di sistema, possono essere usati da (thread di) processi diversi
- in versione *unnamed*, possono essere usati da threads nello stesso processo (anche o tra processi diversi, se collocati in una regione di memoria condivisa tra i processi)

Semafori (POSIX)

Semafori POSIX *named* si ottengono con:

```
sem_t *sp;
sp = sem_open(name,O_CREAT,O_RDWR,val);
```

dove **name** è come per **shm_open** e **val** è il valore a cui viene inizializzato

L'equivalente dell'operazione "up" è

```
sem_post(sp);
```

e l'equivalente dell'operazione "down" è

```
sem_wait(sp);
```

Semafori (POSIX)

Semafori POSIX *unnamed* per threads nello stesso processo:

si dichiara un semaforo con:

```
sem_t s;
```

si inizializza con

```
sem_init(&s,0,val);
```

dove "val" è il valore a cui viene inizializzato (il secondo parametro 0 serve ad indicare che il semaforo può essere usato solo dai threads del processo che l'ha creato)

Si fa "up" con:

```
sem_post(&s);
```

e "down" con:

```
sem_wait(&s);
```

Semafori (XSI)

L'estensione XSI mette a disposizione (**`semget`**) dei semafori (originariamente della versione System V) che offrono con la chiamata **`semop`** funzioni più generali di **`down/up`** :

- aumentare/decrementare un semaforo di 2,3,4...
- operazioni su più semafori, eseguite in modo atomico (se tutte possono essere eseguite senza sospendere il processo, le si esegue, se no si attende che possano essere eseguite tutte)

Sono un po' più «costosi» (come tempo di esecuzione delle operazioni) rispetto agli altri, quindi se non serve questa maggiore generalità, meglio usare quelli POSIX

Mutex e condizioni (Pthreads)

In UNIX i Phtreads si possono sincronizzare con altri paradigmi fra cui uno ispirato a quello dei monitor:

- usando esplicitamente dei *mutex* per la mutua esclusione (non c'è gestione automatica come per le procedure dei monitor o i metodi *synchronized)*
- Variabili condizione con operazioni *wait*, *signal* e *broadcast* (segnala una condizione a tutti i thread in attesa su quella)

In particolare, per i *mutex* (tipo **`pthread_mutex_t`**) si hanno a disposizione le funzioni:

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

per prendere/lasciare la mutua esclusione su (le variabili condivise associate a) una variabile *mutex*

Mutex e condizioni (Pthreads)

Un thread, tipicamente dopo avere conquistato l'accesso esclusivo a variabili condivise con

```
pthread_mutex_lock(&m);
```

può sospendersi su una variabile condizione `cond` (di tipo `pthread_cond_t`) con:

```
pthread_cond_wait(&cond,&m);
```

questo comporta implicitamente anche *l'unlock* di `m`, cioè il rilascio della mutua esclusione, e il nuovo *lock* di `m` quando il thread verrà sbloccato dall'attesa della condizione.

Mutex e condizioni (Pthreads)

Un *thread* può "segnalare" una condizione con:

```
pthread_cond_signal(&cond);
```

che "risveglia" uno dei *threads* bloccati su **cond** (se non ce ne sono, non ha effetto), oppure con:

```
pthread_cond_broadcast(&cond);
```

che "risveglia" tutti i threads bloccati sulla condizione.

Attenzione al noto problema: per riprendere l'esecuzione, i *thread* che avevano fatto *wait* devono riprendere l'accesso mutuamente esclusivo sul *mutex* che avevano passato come secondo argomento a *wait*.

Tipicamente, la segnalazione viene fatta da un *thread* che opera sulle stesse variabili condivise e quindi ha il *lock* sullo stesso *mutex* (anzi è raccomandato che questo avvenga sempre, per evitare il problema della "perdita della segnalazione" come per *sleep* e *wakeup*).
Il *thread* sbloccato, o i *thread* sbloccati in caso di *broadcast*, potrà (potranno) riprendere il lock (uno per volta) quando il *thread* segnalante lo rilascerà.

Mutex e condizioni (Pthreads)

Questo è un tipico schema di programma per un thread t1 che opera su variabili condivise: attende, anche usando una var. condizione, che una condizione booleana ***test*** sulle variabili sia falsa, poi modifica le variabili condivise ed eventualmente segnala (a uno/tutti) una condizione:

```
pthread_mutex_lock(&m);
while (test) pthread_cond_wait(&cond1,&m);
/* modifica variabili condivise */
/* eventuale pthread_cond_signal(&cond2) */
pthread_mutex_unlock(&m);
```

Qui è veramente opportuno un **while** e non un **if** (all'uscita della **wait** viene di nuovo eseguito il ***test***) perché:

- chi segnala **cond1** non necessariamente conosce quale ***test*** t1 attende che diventi falso
- se anche fosse così, non è detto che t1 conquisti la mutua esclusione immediatamente dopo la segnalazione della condizione; un altro thread t2 che esegue ad es. lo stesso codice di t1 può superare **lock** e modificare le var. condivise rendendo nuovamente vero **test**

Mutex e condizioni (Pthreads)

Ad esempio per un “produttore” :

```
pthread_mutex_lock(&m);
while (buf.count==N)pthread_cond_wait(&empty,&m);
/* deposita elemento in buf */
if (buf.count==1) pthread_cond_signal(&full);
pthread_mutex_unlock(&m);
```

E per un “consumatore”:

```
pthread_mutex_lock(&m);
while (buf.count==0)pthread_cond_wait(&full,&m);
/* preleva elemento da buf */
if (buf.count==N-1) pthread_cond_signal(&empty);
pthread_mutex_unlock(&m);
```