Lập trình tương tranh

Tương tranh (concurrency) xảy ra khi các tiến trình chạy đồng thời và cạnh tranh sử dụng tài nguyên chung. Trong phần này chúng ta sẽ đề cập tới các mô hình lập trình tương tranh sử dụng tiến trình và luồng, cách thức chúng liên lạc với nhau. Các biện pháp giải quyết vấn đề tương tranh sẽ được thảo luận trong một phần tiếp theo.

Cơ chế liên lạc liên tiến trình

Mỗi tiến trình thực thi được HĐH quản lý thông qua 1 số hiệu duy nhất là PID. Các tiến trình có thể chạy độc lập với nhau hoặc một tiến trình có thể là cha của các tiến trình con khác.

Có các cơ chế liên lạc sau giữa các tiến trình

Cơ chế	Mô hình	Mục tiêu	Chú thích
Pipe / Named pipe	Truyền thông điệp	Trao đổi dữ liệu	Pipe tạo tệp đường ống nối output của một tiến trình với input của tiến trình khác có quan hệ cha/con. Named pipe dùng cho các tiến trình độc lập. Dữ liệu trao đổi là phi cấu trúc.
Message queue	Truyền thông điệp	Trao đổi dữ liệu	Tạo tệp có cấu trúc cho việc trao đổi dữ liệu theo nguyên tắc FIFO giữa các tiến trình độc lập.
Socket	Truyền thông điệp	Trao đổi dữ liệu	Tạo tệp socket mạng trao đổi dữ liệu giữa các tiến trình liên mạng qua thư viện lập trình socket.
Memory mapped file	Vùng nhớ chung	Trao đổi dữ liệu	Ánh xạ nội dung 1 tệp vào bộ nhớ và chia sẻ nội dung tới nhiều tiến trình.
Shared memory	Vùng nhớ chung	Trao đổi dữ liệu	Tạo vùng nhớ chia sẻ chung cho nhiều tiến trình.
Signal	Truyền thông điệp	Đồng bộ	Tạo 1 thông điệp báo cho tiến trình khác về 1 sự kiện đã xảy ra cần xử lý (cơ chế ngắt – interrupt).
Semaphore	Truyền thông điệp	Đồng bộ	Tạo một "đèn hiệu" chung liên tiến trình để đồng bộ xử lý tài nguyên chung.

Để phục vụ cho việc mô phỏng các bài toán giải quyết tương tranh, chúng ta sẽ sử dụng cơ chế Shared memory và Semaphore để chia sẻ dữ liệu và đồng bộ hoạt động giữa các tiến trình. Cơ chế Shared memory cho phép chia sẻ dữ liệu lớn và tức thời giữa các tiến trình. Sử dụng cơ chế này, các chương trình tương tranh theo mô hình đa tiến trình hoặc đa luồng sẽ có cấu trúc tương tự khi triển khai.

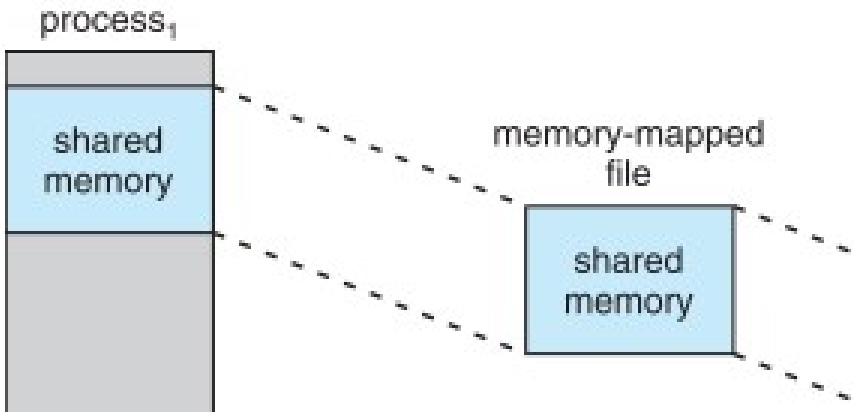
Shared memory

Hai phương pháp Shared memory và Memory mapped file có cùng một cách thức hoạt động

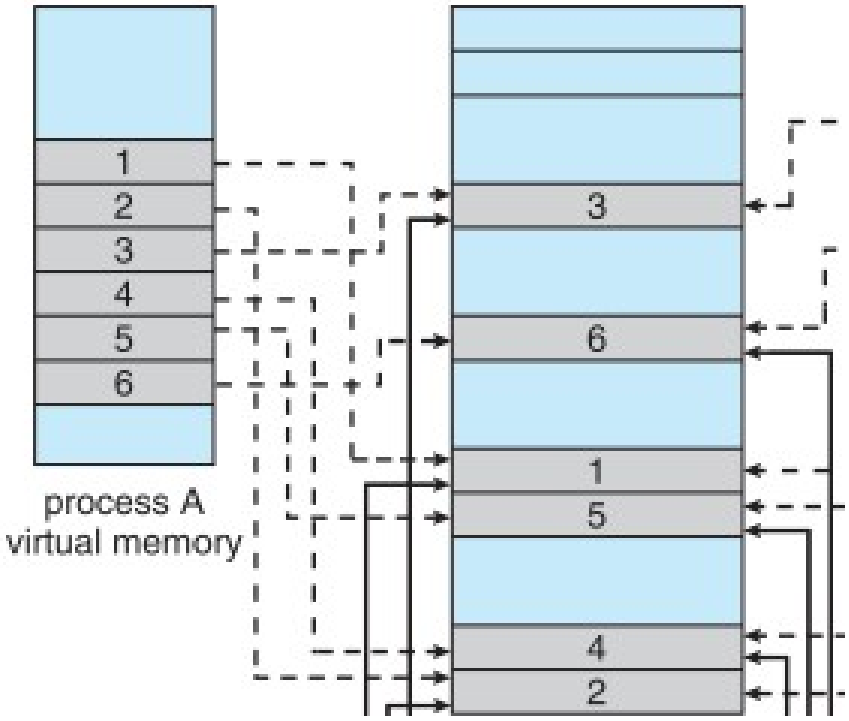
- Đầu tiên là mở hoặc tạo mới một tệp và tải nội dung một tệp vào trong bộ nhớ vật lý RAM. Shared memory dùng hàm shm_open() còn Memory mapped file sử dụng hàm open() thông thường.
- Tiếp theo mỗi tiến trình sẽ ánh xạ vùng nhớ riêng của mình tới vùng nhớ chia sẻ chung đó, sử dụng hàm mmap().
- Thay đổi ở vùng nhớ chung có thể được đồng bộ lại về tệp ban đầu qua hàm msyn().
- Khi chia sẻ xong, tiến trình có thể ngắt ánh xạ tới vùng nhớ chung qua hàm munmap()
- Cuối cùng tệp sẽ được đóng lại hoặc xóa. Memory mapped file dùng hàm close() thông thường, còn Shared memory xóa tệp bằng hàm shm_unlink()

Điểm khác biệt là với Shared memory, chúng ta không quan tâm tới sự tồn tại của tệp cũng như việc cập nhật nội dung thay đổi xuống tệp, do vậy cần ít thao tác hơn. Trong cài đặt thực tế, hàm shm_open() thực chất cũng chỉ là thêm phần đường dẫn tới một thư mục gắn với RAM, chẳng hạn /dev/shm và gọi hàm open() để mở/tạo một tệp trong đó.

Ở mức logic, chúng ta thấy các tiến trình ánh xạ vùng nhớ riêng của mình tới một vùng nhớ được chia sẻ chung. Vùng nhớ này thực chất là ảnh nội dung của một tệp.



Ở mức cài đặt, việc ánh xạ vùng nhớ chung vào vùng nhớ của từng tiến trình được thực hiện theo nguyên tắc ánh xạ trang bộ nhớ ảo sang trang bộ nhớ thực như trong hình dưới đây. Điều này dẫn tới thực tế là vùng nhớ của mỗi tiến trình được ánh xạ vào cùng một vùng nhớ thực chung nhưng sẽ có các địa chỉ ảo khác nhau. Do vậy trong shared memory chúng ta phải tránh sử dụng các con trỏ, chẳng hạn như danh sách móc nối. Con trỏ trong danh sách móc nối chỉ có thể trỏ tới 1 phần tử tiếp theo, tuy nhiên vấn đề là phần tử tiếp theo lại có địa chỉ ảo khác nhau trong mỗi tiến trình nên sẽ gây lỗi thực thi.



Các hàm được sử dụng trong shared memory

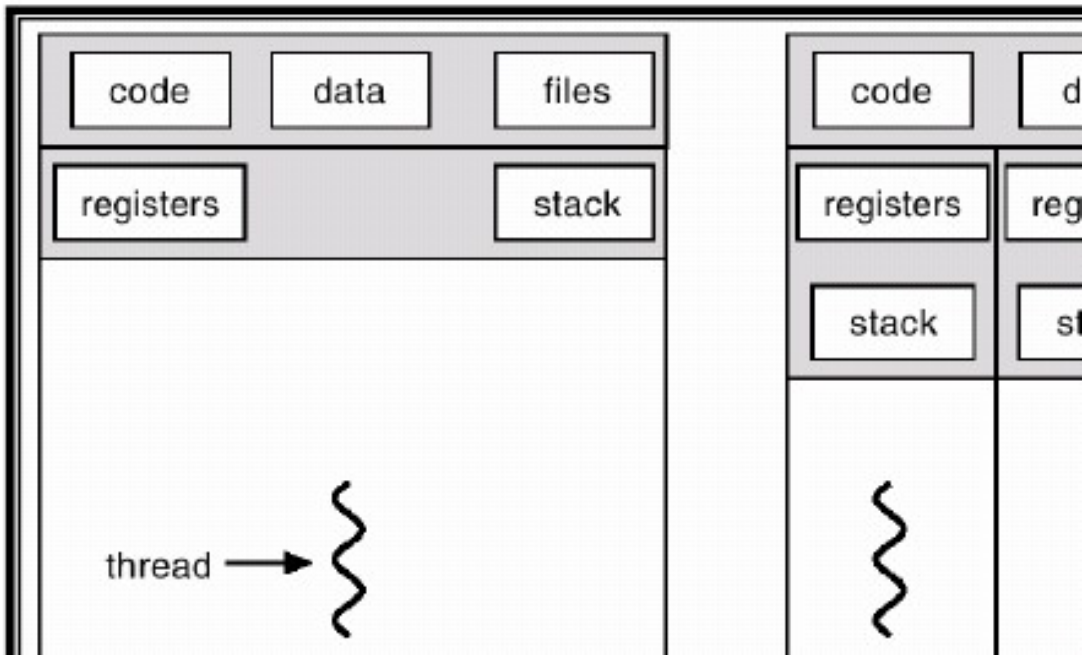
Hàm	Tham số	Mô tả chức năng
`int shm_open (const char *name, int oflag, mode_t mode);`	`name`: tên của đối tượng `oflag`: các chế độ mở đọc/ghi/tạo `mode`: chế độ thao tác vùng nhớ đọc/ghi	Tạo / mở một tệp làm đối tượng shared memory
`int shm_unlink (const char *name);`	`name`: tên đối tượng shared mem	Xóa tệp đối tượng shared memory
`int ftruncate(int fd, off_t length);`	`fd`: file descritor của đối tượng `length`: độ dài thiết lập	Điều chỉnh độ dài vùng nhớ chia sẻ
`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`	`addr`: địa chỉ vùng nhớ của tiến trình; `length`: độ dài vùng nhớ; `prot`: chế độ protection vùng nhớ; `flags`: chia sẻ hay dùng riêng; `fd`: file descriptor của đối tượng shared mem; `offset`: địa chỉ thành phần của file	Tạo ánh xạ từ vùng nhớ ảo của tiến trình tới đối tượng bộ nhớ shared memory
`int munmap(void *addr, size_t length);`	`addr`: địa chỉ vùng nhớ tiến trình; `length`: độ dài vùng nhớ	Xóa ánh xạ tới đối tượng shared memory. Khi tiến trình kết thúc, các ánh xạ cũng được tự động xóa.

Ví dụ: Đoạn mã dưới đây chia sẻ một cấu trúc dữ liệu struct permission giữa tiến trình cha và tiến trình con. Tiến trình cha chờ tới khi tiến trình con kết thúc và in nội dung các trường được gán bởi tiến trình con.

```
/* Code Listing 3.10:
   Using POSIX shared memory to exchange data between processes
*/

/* Create unsized shared memory object;
   return value is a file descriptor */
int shmfd = shm_open ("/OpenCSF_SHM", O_CREAT | O_EXCL | O_RDWR, S_IRUSR |
S_IWUSR);
assert (shmfd != -1);

/* Resize the region to store 1 struct instance */
assert (ftruncate (shmfd, sizeof (struct permission)) != -1);

/* Map the object into memory so file operations aren't needed */
struct permission *perm = mmap (NULL, sizeof (struct permission),
                                PROT_READ | PROT_WRITE, MAP_SHARED, shmfd,
0);
assert (perm != MAP_FAILED);

/* Create a child process and write to the mapped/shared region */
pid_t child_pid = fork();
if (child_pid == 0)
  {
    perm->user = 6;
    perm->group = 4;
    perm->other = 0;

    /* Unmap and close the child's shared memory access */
    munmap (perm, sizeof (struct permission));
    close (shmfd);
    return 0;
  }

/* Make the parent wait until the child has exited */
wait (NULL);

/* Read from the mapped/shared memory region */
printf ("Permission bit-mask: 0%d%d%d\n", perm->user, perm->group, perm-
>other);

/* Unmap, close, and delete the shared memory object */
munmap (perm, sizeof (struct permission));
close (shmfd);
shm_unlink ("/OpenCSF_SHM");
```

Lập trình đa luồng

Hình dưới đây mô tả về khái niệm luồng.



Một tiến trình gồm có:

- đoạn mã chương trình (code);
- dữ liệu của tiến trình (data) gồm các biến tổng thể của chương trình trong phạm vi khai báo;
- các tài nguyên đang sử dụng (vd files đang mở);
- ngăn xếp (stack) chứa các tham số gọi hàm, các giá trị biến cục bộ khi gọi hàm/thủ tục con;
- các giá trị của tập thanh ghi (register) mà CPU đang thao tác trong đó có thanh ghi con trỏ lệnh (instruction pointer) chỉ ra vị trí của lệnh tiếp theo sẽ được CPU tải vào thực hiện.

Luồng là một môi trường con trong một tiến trình gồm có:

- Môi trường chung của các luồng trong tiến trình gồm đoạn mã chương trinh, dữ liệu, tệp đang mở
- Môi trường riêng:
    - các thanh ghi chứa các kết quả tính toán của CPU riêng đối với mỗi luồng gồm có con trỏ lệnh riêng, tức là mỗi luồng có chung 1 đoạn mã chương trình nhưng đang thực hiện tại các vị trí khác nhau
    - ngăn xếp cất giữ các tham số, biến cục bộ của hàm/thủ tục con

Với cơ chế đa luồng, một tiến trình có thể dùng cùng một mã chương trình để xử lý song song với các yêu cầu trên nhiều bộ dữ liệu khác nhau; các luồng chia sẻ chung dữ liệu tổng thể, tệp thao tác. Mỗi luồng vẫn xử lý công việc theo thuật toán tuần tự. Ví dụ một tiến trình trình web server có thể tạo 10 luồng để đáp ứng 10 yêu cầu duyệt web từ các trình duyệt; người lập trình không phải viết thuật toán xử lý song song để thực hiện công việc này.

Lập trình với luồng sử dụng tối thiểu các hàm sau

Hàm	Tham số	Chức năng
int pthread_create (pthread_t *thread_id, const pthread_attr_t *attr, void* (*thread) (void *), void *arg);	thread_id – số hiệu của luồng attr – đặt thuộc tính luồng, NULL mặc định thread – hàm được khởi chạy khi luồng tạo ra arg – tham số cho hàm khởi tạo luồng	Tạo 1 luồng
int pthread_join(pthread_t thread_id, void **ret_value);	thread_id – số hiệu của luồng con ret_value – giá trị trả lại của luồng con qua pthread_exit()	Chờ đồng bộ luồng con rồi mới kết thúc

Ví dụ: Tiến trình sau tạo một luồng in ra xâu tham số Hello World, đợi cho luồng con hoàn tất rồi mới kết thúc.

```
#include <pthread.h>
#include <stdio.h>

char *pHello="Hello world";

void *myThread(void* pStr) { // Đây là mã của luồng
   printf("%s\n", (char*) pStr);
}

void main() {
   pthread_t tid;
   // tạo luồng *myThread, truyền tham số pHello
   pthread_create(&tid, NULL, myThread, (void*) pHello);
   // cha chờ *myThread kết thúc
   pthread_join(tid, NULL);
}
```

Dữ liệu chung của các luồng và tiến trình là biến tổng thể char* pHello
Mã thực hiện luồng là hàm *myThread, chạy song song với tiến trình cha trong main()
Tham số void* pStr là dữ liệu thuộc phần không gian riêng của luồng *myThread, được cất và lấy ra từ ngăn xếp.

Lập trình với semaphore

Khái niệm semaphore đã được đề cập tới trong nội dung phần tiến trình và tương tranh tài nguyên. Trong mục này chúng ta chỉ đề cập tới việc cài đặt sử dụng tới semaphore.

Các khai báo và hàm cần thiết liên quan tới semaphore

Hàm / Khai báo	Tham số	Chức năng
sem_t sem;	Khai báo biến semaphore *sem*	Khai báo biến semaphore
int sem_init (sem_t *sem, int pshared, unsigned int value);	sem – biến semaphore pshared – nếu =0 thì sem chỉ dùng trong nội bộ tiến trình (dành cho lập trình đa luồng), nếu <>0 thì chia sẻ giữa các tiến trình (phải khai báo dạng shared memory, dành cho lập trình đa tiến trình) value – giá trị khởi tạo	Khởi tạo giá trị semaphore

int sem_post(sem_t *sem);	sem – biến semaphore	Up, tăng giá trị semaphore lên 1 đơn vị, đánh thức 1 tiến trình đang ngủ chờ semaphore
int sem_wait(sem_t *sem);	sem – biến semaphore	Down, giảm giá trị semaphore đi 1 đơn vị nếu còn > 0, nếu down một semaphore bằng 0 thì thao tác down sẽ bị trì hoãn và tiến trình chuyển sang trạng thái ngủ (blocked)

Ví dụ về lập trình tương tranh cho bài toán producers – consumers

Giải pháp sử dụng tiến trình

Chúng ta sẽ để tiến trình cha sinh ra hai tiến trình con producer và consumer. Hai tiến trình con sẽ chia sẻ một vùng nhớ chung qua cơ chế Shared memory, chứa semaphore liên tiến trình đặt trong vùng nhớ chung đó. Sau khi 2 tiến trình con hoàn tất thì tiến trình cha sẽ kết thúc.

<insert source code>

Giải pháp sử dụng luồng

Chúng ta sẽ để tiến trình tạo ra 2 luồng con producer và consumer chạy 2 hàm khác nhau nhưng thao tác trên các biến tổng thể chung. Semaphore được sử dụng trong nội bộ tiến trình nhưng là chung giữa tất cả các luồng. Sau khi 2 luồng con hoàn tất thì tiến trình mới kết thúc.

<insert source code>