Lập trình IPC và thread

Bộ môn Hệ thống và Mạng máy tính
Khoa Khoa học và kỹ thuật máy tính

Lập trình trên Linux

- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread
  - Cơ bản về lập trình POSIX pthread
  - Giải quyết tranh chấp trên POSIX thread

Lập trình trên Linux

- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread
  - Cơ bản về lập trình POSIX pthread
  - Giải quyết tranh chấp trên POSIX thread

Giới thiệu về IPC

- Mục tiêu của IPC
  - IPC: Inter-Process Communication
  - Cho phép phối hợp hoạt động giữa các quá trình trong hệ thống
  - Giải quyết đụng độ trên vùng tranh chấp
  - Truyền thông điệp từ quá trình này đến các quá trình khác
  - Chia sẻ thông tin giữa các quá trình với nhau

Giao tiếp và đồng bộ

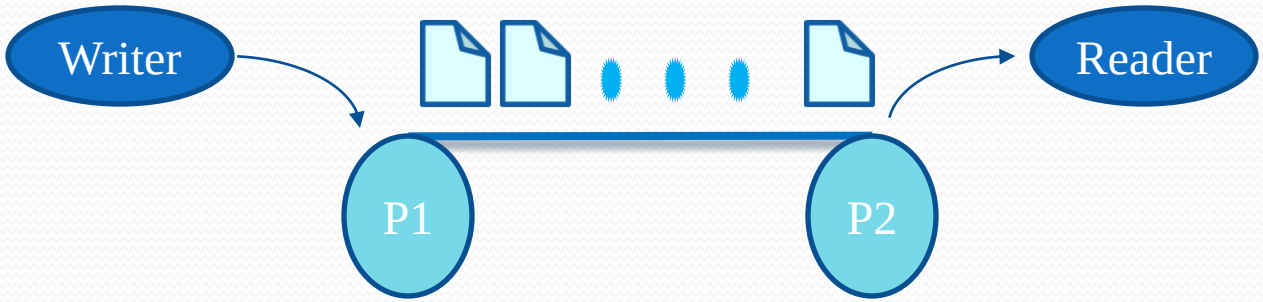
- Communication
  - Truyền dữ liệu
  - Chia sẻ thông tin
  - Các cơ chế:
    - Pipe
    - Signal
    - Message queue
    - Shared memory
    - Socket
    - RPC/RMI
- Synchronization
  - Giải quyết tranh chấp
  - Đảm bảo thứ tự xử lý
  - Các cơ chế:
    - Lock file
    - Semaphore
    - Mutex (pthread)

Lập trình trên Linux

- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread bằng pthread

Giao tiếp thông qua PIPE

- Là kênh truyền dữ liệu giữa các process với nhau theo dạng FIFO



Các tác vụ trên pipe

- Write:

  #include <unistd.h>

  ssize_t write(int fd, const void *buf, size_t count)

- Read:

  #include <unistd.h>

  ssize_t read(int fd, const void *buf, size_t count)

Hai loại pipe

- Unnamed pipe
  - có ý nghĩa cục bộ
  - chỉ dành cho các process có quan hệ bố con với nhau
- Named pipe (còn gọi là FIFO)
  - có ý nghĩa toàn cục
  - có thể sử dụng cho các process không liên quan bố con

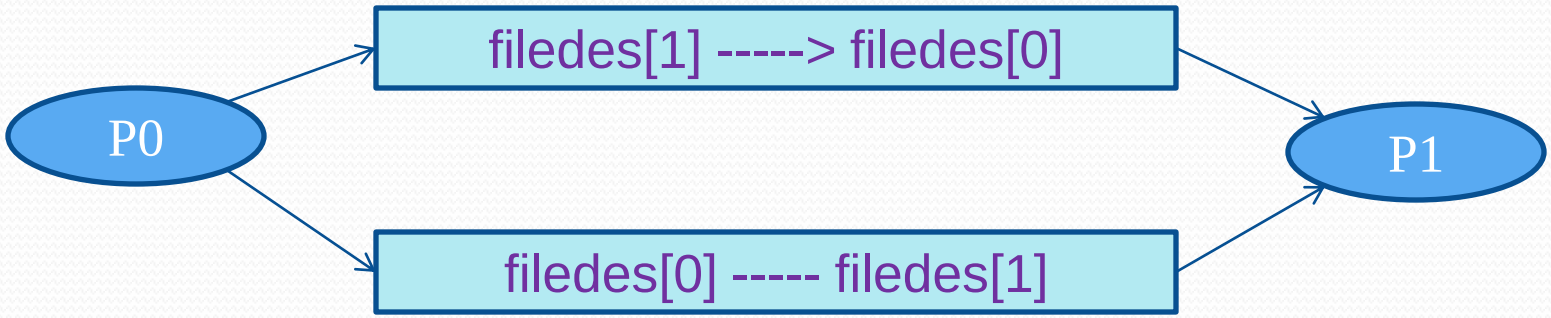
Unnamed pipe

- Tạo unnamed pipe:

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- Kết quả
  - Thành công, kết quả thực thi hàm pipe() là 0, có hai file descriptor tương ứng sẽ được trả về trong filedes[0], filedes[1]
  - Thất bại: hàm pipe() trả về -1, mã lỗi trong biến ngoại errno

Unnamed pipe (2)



- Duplex
  - Linux: unidirectional/half-duplex, i.e. filedes[0] chỉ được dùng để đọc còn filedes[1] chỉ được dùng để ghi dữ liệu
  - Solaris: full-duplex, i.e. nếu ghi vào filedes[0], thì filedes[1] được dùng để đọc và ngược lại

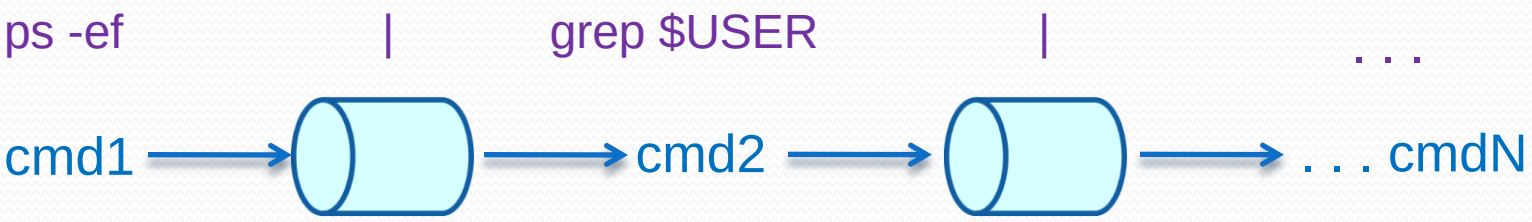
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main() {
    int fp[2];
    char s1[BUFSIZ], s2[BUFSIZ];
    pipe(fp);
    if (fork()==0) {            /* Child Write */
        printf("\nInput: ");
        fgets(s1,BUFSIZ,stdin);
        s1[strlen(s1)]=0;
        close(fp[0]);
        write(fp[1],s1,strlen(s1)+1);
    } else {                    /* Parent Read */
        close(fp[1]);
        read(fp[0],s2,BUFSIZ);
        printf("\nFrom pipe> %s\n", s2);
    }
    return 0;
}
```

Dịch, thực thi

```
$gcc unpipe.c -o unpipe
$./unpipe
Input: I Love Penguin
From pipe> I Love Penguin
$
```

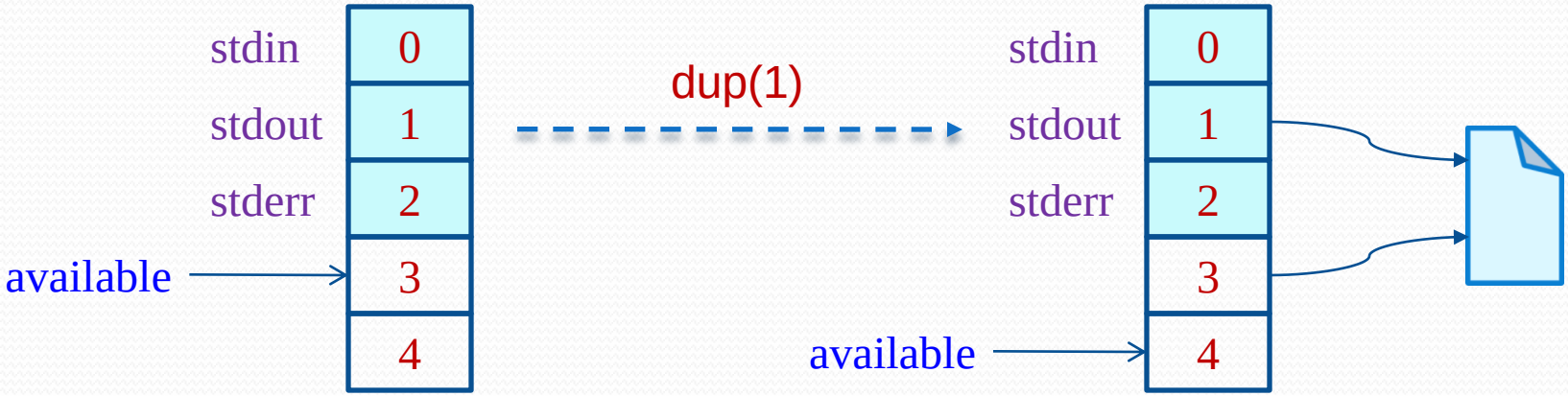
Dùng pipe để tái định hướng

- Pipe có thể được dùng để kết nối các lệnh với nhau (do chương trình shell thực hiện)
  - Ví dụ: $ ps -ef | grep a01 | sort
    $ ls | more
- Đối với chương trình người dùng, có thể dùng một trong hai system call sau kết hợp với pipe đểthực hiện:
  - dup()
  - dup2()



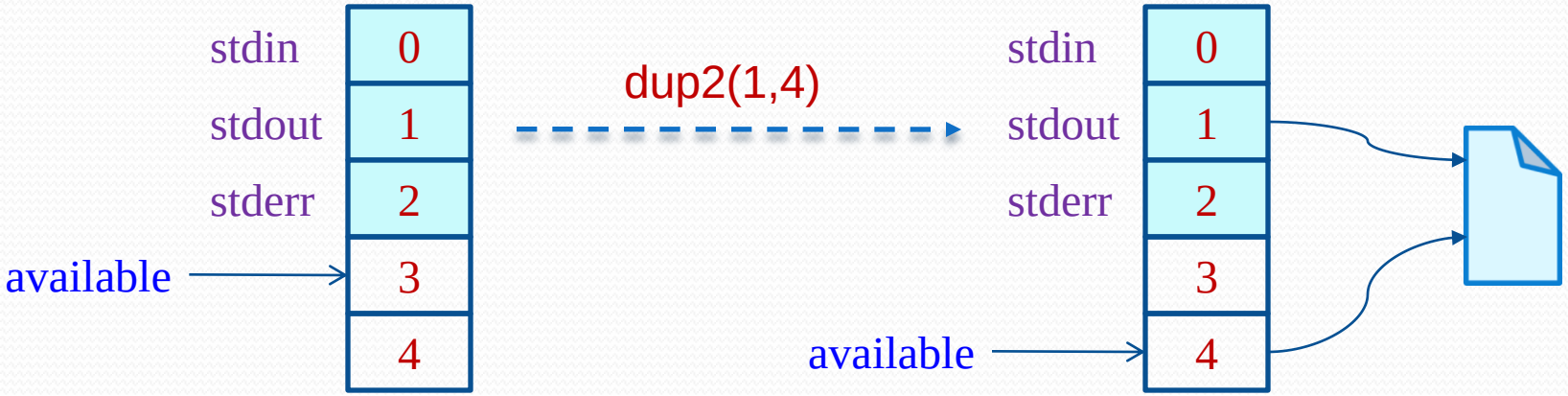
dup()

```
#include <unistd.h>
int dup(int oldfd);
```



dup2()

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```



```
#include <unistd.h>
int main() {        // ps -ef | sort | grep
    int pipe1[2], pipe2[2];
    pipe(pipe1);
    if (fork()) { /* Parent */
       pipe(pipe2);
       if(fork()) { /* Parent */
          close(0); // Close standard input
          dup(pipe2[0]); // standard input -> Read Pipe2
          close(pipe1[0]);
          close(pipe1[1]);
          close(pipe2[0]);
          close(pipe2[1]);
          execl("/bin/grep", "grep", NULL);
       }
```

```
    else {        /* Child 2 */
      close(0);           // Close standard input
      dup(pipe1[0]);      // standard input -> Read Pipe1
      close(1);           // Close standard output
      dup(pipe2[1]);      // standard output -> Write Pipe2
      close(pipe1[0]); close(pipe1[1]);
      close(pipe2[0]); close(pipe2[1]);
      execl("/bin/sort", "sort", NULL);
    }
  } else {  /* Child 1 */
    close(1);   // Close standard output
    dup(pipe1[1]);        // standard output -> Write Pipe1
    close(pipe1[0]); close(pipe1[1]);
    execl("/bin/ps", "ps", "-ef", NULL);
  }
  exit(0);
}
```

Named pipe

- Tương tự như unnamed pipe
- Một số tính năng cần chú ý:
  - Được ghi nhận trên file system (directory entry, file permission)
  - Có thể dùng với các process không có quan hệ bố con
  - Có thể tạo ra từ dấu nhắc lệnh shell (bằng lệnh mknod)

Tạo named pipe - mknod()

- System call

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod(const char *path, mode_t mode, dev_t dev);
```

- Trong đó
  - path: đường dẫn đến pipe (trên file system)
  - mode: quyền truy cập trên file = S_IFIFO kết hợp với trị khác
  - dev: dùng giá trị 0
- C/C++ library call

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS 0666
int main(){
      char s1[BUFSIZ], s2[BUFSIZ];
      int childpid, readfd, writefd;
```

Dịch và thực thi

$gcc fifo.c -o fifo

$./fifo

Parent writes to FIFO1: Test1

Child reads from FIFO1: Test1

Child feedbacks on FIFO2: Test2

Feedback data from FIFO2: Test2

```
if ((mknod(FIFO1, S_IFIFO | PERMS, 0)<0) &&
    (errno!=EEXIST)) {
      printf("can't create fifo1: %s", FIFO1);
      exit(1);
}
if ((mknod(FIFO2, S_IFIFO | PERMS, 0)<0) &&
    (errno!=EEXIST)) {
      unlink(FIFO1);
      printf("can't create fifo2: %s", FIFO2);
      exit(1);
}
if ((childpid=fork())<0) {
      printf("can't fork");
      exit(1);
}
```

```
else if (childpid>0) { /* parent */
      if ((writefd=open(FIFO1,1))<0)
         perror("parent: can't open writefifo");
      if ((readfd=open(FIFO2,0))<0)
         perror("parent: can't open readfifo");
      printf("\nParent writes to FIFO1: ");
      gets(s1);
      s1[strlen(s1)]=0;
      write(writefd,s1,strlen(s1)+1);
      read(readfd,s2,BUFSIZ);
      printf("\nFeedback data from FIFO2: %s\n",s2);
      while (wait((int*)0)!=childpid); /*wait for child finish*/
      close(readfd);
      close(writefd);
      if (unlink(FIFO1)<0) perror("Can't unlink FIFO1");
      if (unlink(FIFO2)<0) perror("Can't unlink FIFO2");
      exit(0);
}
```

```
    else { /* child */
        if ((readfd=open(FIFO1,0))<0)
          perror("child: can't open readfifo");
        if ((writefd=open(FIFO2,1))<0)
          perror("child: can't open writefifo");
        read(readfd,s2,BUFSIZ);
        printf("\nChild read from FIFO1: %s\n",s2);
        printf("\nInput string from child to feedback: ");
        gets(s1);
        s1[strlen(s1)]=0;
        write(writefd,s1,strlen(s1)+1);
        close(readfd);
        close(writefd);
        exit(0);
    }
}
```

Lập trình trên Linux

- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread
  - Cơ bản về lập trình POSIX pthread
  - Giải quyết tranh chấp trên POSIX thread

SystemV IPC

- Gồm: message queue, shared memory, semaphore
- Có một số thuộc tính chung như
  - Người tạo, người sở hữu (owner), quyền truy cập (perms)
- Có thể theo dõi trạng thái các IPC bằng lệnh ipcs

```
$ipcs
------Shared Memory Segments --------
key          shmid  owner  perms  bytes    nattch  status
0x00000000  65536  root    644   110592     11     dest

------Semaphore Arrays --------
key         semid  owner  perms   nsems

------Message Queues --------
key         msqid  owner  perms   used-bytes    messages
```

Semaphore

- Đồng bộ các process theo giải thuật của semaphore
- Biến semaphore
  - số nguyên, truy cập qua các hàm do hệ điều hành cung cấp: P (wait), V (signal)
- Đảm bảo loại trừ tương hỗ
- Trong UNIX System V, semaphore được dùng theo set – danh sách các semaphore.

Lệnh IPC trong Linux

- Theo dõi trạng thái các IPC (gồm message queue, semaphore, shared memory)
  - ipcs hoặc ipcs -a
- Theo dõi trạng thái các semaphore của hệ thống
  - ipcs -s
- Loại bỏ một semaphore (phải đủ quyền hạn)
  - ipcrm sem semid hoặc ipcrm -s semid

Các thao tác chủ yếu trên đối tượng IPC

- semget()
- semop()
- semctl()

Hàm semget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- key: giá trị key cho IPC object, nếu key=IPC_PRIVATE thì semaphore tạo ra chỉ được sử dụng trong nội bộ process.
- nsems: số lượng semaphore trong semaphore set, thông thường chỉ cần dùng 1 semaphore.
- semflag: IPC_CREAT, IPC_EXCL và có thể OR với giá trị ấn định quyền truy cập (tương tự quyền hạn trên một file).

- Ví dụ

```
sset1=semget(IPC_PRIVATE,1,IPC_CREAT|IPC_EXCL|0600);
sset2=semget(12345,1,IPC_CREAT|IPC_EXCL|0666);
```

Tạo key cho IPC object

```
#include <sys/types.h>
#include <sys/ipc.h>
keyt_t key;
char *path;
int id=123;
...
key=ftok(path,id);
```

⇒ các process khác nhau chỉ cần cung cấp path và id giống nhau là có thể tạo đúng key truy cập đến cùng một IPC object.

Hàm semop()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

- semid: semaphore set ID do hàm semget() trảvề
- sops: là danh sách gồm nsops cấu trúc sembuf định ra các thao tác cho từng semaphore trong tập semaphore.
- nsops: số semaphores trong semaphore cần thao tác

Cấu trúc sembuf

```
struct sembuf {
    ushort sem_num;          /*semaphore thứ #*/
    short sem_op;     /*operation*/
    short sem_flg;    /*operation flags*/
}
```

- sem_num: chỉ số của semaphore trong semaphore set, chỉ số này bắt đầu từ 0
- sem_op: là số nguyên
  - >0: tăng giá trị semaphore
  - <0: giảm giá trị semaphore
- sem_flg:
  - IPC_NOWAIT: non-blocking mode
  - SEM_UNDO: undo operation

Hành vi của hàm semop()

- semop<0
  - semval ≥abs(semop)
    semval=semval-abs(semop)
  - semval ≥abs(semop) & SEM_UNDO
    semval-=abs(semop) AND semadj+=abs(semop)
  - semval<abs(semop)
    block until semval ≥abs(semop)
  - semval<abs(semop) & IPC_NOWAIT
    return -1, errno=EAGAIN
- semop>0
  - semval+=semop
  - SEM_UNDO
    - semval+=semop AND semadj-=semop

Hàm semctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid,int semnum,int cmd);
int semctl(int semid,int semnum,int cmd,union semun arg);

union semun{
    int val;
    struct semid_ds *buf;
    ushort *array;
};
```

Hàm semctl() - tham số cmd

- Các thao tác thông thường
  - IPC_SET : thiết lập quyền truy cập
  - IPC_STAT: lấy thông tin thuộc tính
  - IPC_RMID: xoá semaphore set
- Các thao tác trên từng semaphore riêng lẻ
  - GETVAL: lấy thông tin thuộc tính
  - SETVAL: thay đổi thuộc tính
  - GETPID: lấy PID của process vừa truy cập semaphore
  - GETNCNT: lấy số process đang đợi **semval** tăng lên
  - GETZCNT: lấy số process đang đợi **semval** về 0
- Các thao tác trên toàn semaphore set
  - SETALL: thay đổi thuộc tính
  - GETALL: lấy thông tin thuộc tính

Ví dụ

- Hiện thực 4 hàm cơ bản của semaphore
  - seminit: tạo binary semaphore
  - p (wait)
  - v (signal)
  - semrel: xoá semaphore
- Viết chương trình giải quyết tranh chấp dùng semaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
#include <unistd.h>
union {
        int val;
        struct semid_ds *buf;
        ushort *array;
} carg;
int seminit() {
        int i, semid;
        if (semid=semget(IPC_PRIVATE,1,0666|IPC_EXCL)==-1)
            return(-1);
        carg.val=1;
        if (semctl(semid,0,SETVAL,carg)==-1) return(-1);
        return semid;
}
```

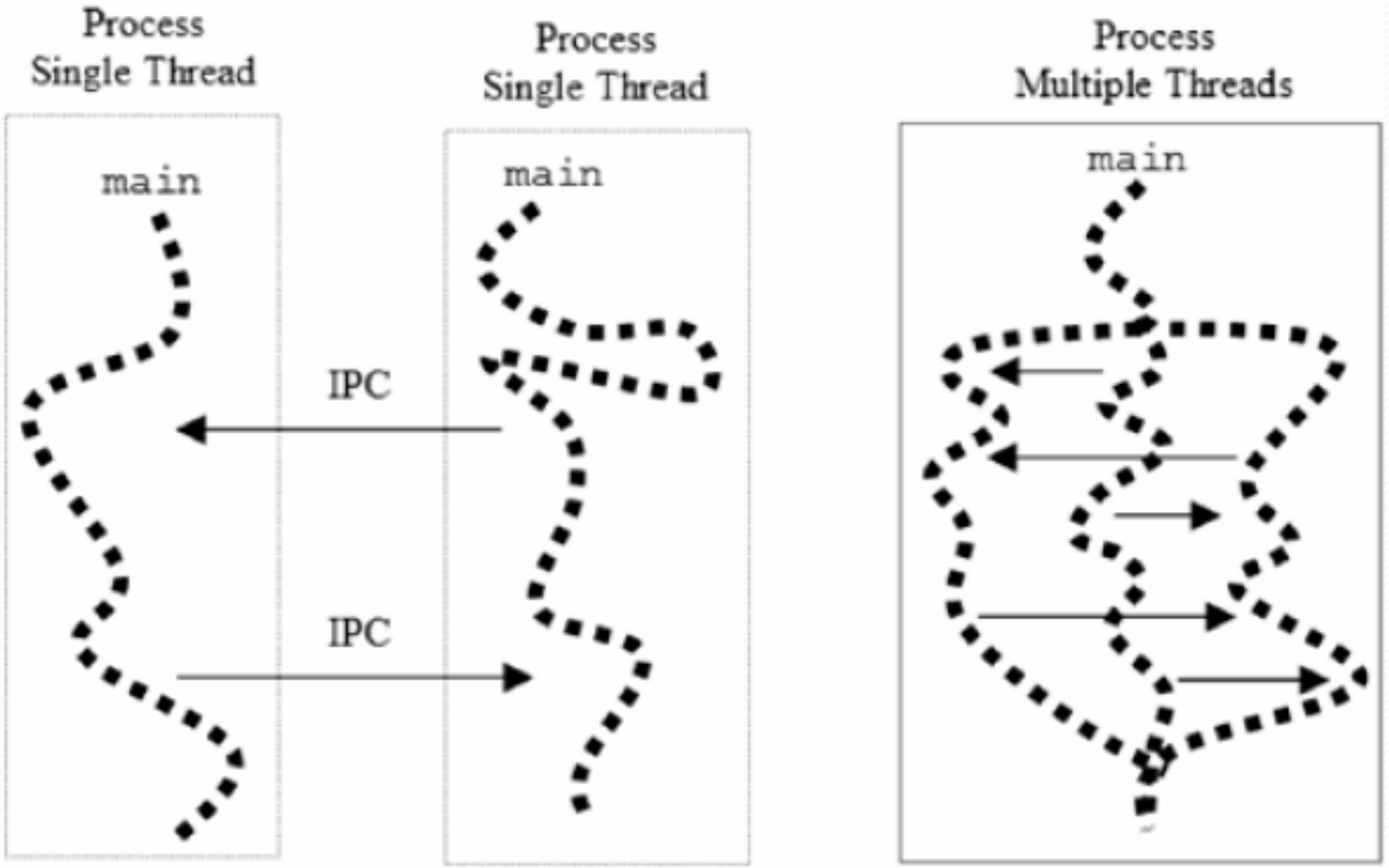
```
void p(int sem){
        struct sembuf pbuf;
        pbuf.sem_num=0;
        pbuf.sem_op=-1; /* giảm giá trị semaphore */
        pbuf.sem_flg=SEM_UNDO;
        if (semop(sem,&pbuf,1)==-1) {
                perror("semop"); exit(1);
        }
}
void v(int sem){
        struct sembuf vbuf;
        vbuf.sem_num=0;
        vbuf.sem_op=1;
        vbuf.sem_flg=SEM_UNDO;
        if (semop(sem,&vbuf,1)==-1) {
                perror("semop"); exit(1);
        }
}
```

```
int semrel(int semid){
    return semctl(semid,0,IPC_RMID,0);
}
void func(int sem) {
      while(1) {
            p(sem);              /* enter section */
            printf("%d Do something in CS\n",getpid());
            sleep(5);
            v(sem);              /* exit section */
            printf("%d Out of CS\n",getpid());
            sleep(1);
      }
}
void main() {
      int sem=seminit();;
      if (fork()==0) func(sem);
      else func(sem);
      semrel(sem);
}
```

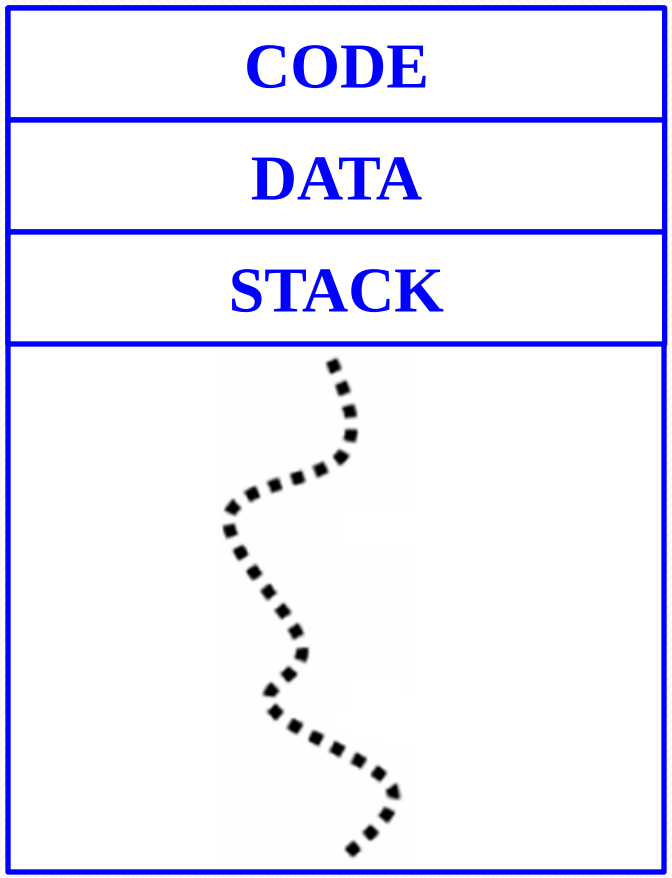
Lập trình trên Linux

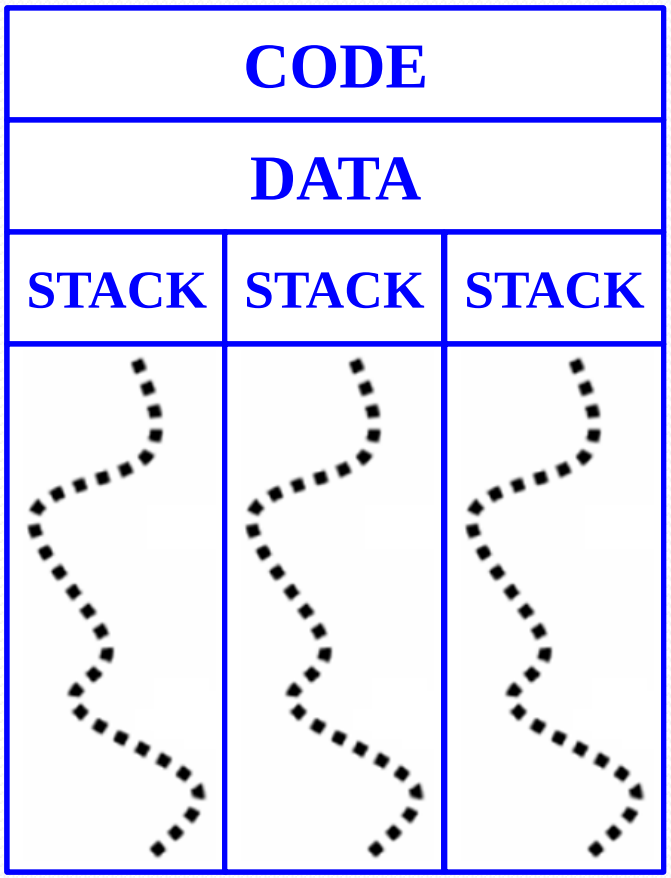
- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread
  - Cơ bản về lập trình POSIX pthread
  - Giải quyết tranh chấp trên POSIX thread

Giới thiệu về thread



Giới thiệu về thread





Các chuẩn về thread

- POSIX (Portable Operating System Interface) thread hay còn gọi là IEEE 1003.1, 1003.1c
  - phổ biến trong các hệ thống *NIX hiện tại
  - đặc tả các giao diện lập trình API và thư viện user-level thread
- Sun Thread

Lập trình trên Linux

- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread
  - Cơ bản về lập trình POSIX pthread
  - Giải quyết tranh chấp trên POSIX thread

Khởi tạo thread mới

```
#include <pthread.h>
int pthread_create(
            pthread_t *thread,
            pthread_attr_t *attr,
            void *(*start_routine)(void *),
            void *arg);
```

- Kết quả trả về
  - 0: Thành công, tạo thread mới, tham số *thread* chứa thread ID
  - <>0: Thất bại (mã lỗi trả về chứa trong biến ngoài *errno*)

Lập trình POSIX thread

- Lưu ý về tham số thứ 3 *start_routine*
  - nên có kiểu trả về là con trỏ kiểu void, nếu không thì phải có type casting khi gọi pthread_create().
  - nên có một tham số kiểu con trỏ void. Tham số của hàm *start_routine* sẽ được truyền vào thông qua tham số thứ 4 của hàm *pthread_create()*.
- Lưu ý về tham số thứ 4 *arg*
  - là tham số truyền vào cho hàm *start_routine*
  - nếu cần truyền nhiều hơn 1 tham số thì nên định nghĩa *arg* là kiểu cấu trúc *struct*

Lập trình POSIX thread

- Thread kết thúc thực thi khi
  - hàm `start_routine` kết thúc
  - có lời gọi hàm `pthread_exit()` tường minh.
  - thread bị ngắt bởi lời gọi hàm `pthread_cancel()`
  - process chính kết thúc
  - một trong các thread gọi system call `exec()`
- Lời gọi hàm kết thúc thread tường minh

```
void pthread_exit(void * retval);
```

Ví dụ

```
#include <pthread.h>
#include <stdio.h>
void* func(void* arg)
{
  int i;
  for (i = 0; i < 2; i++) {
    printf ("This is thread %d\n",
                                *((int*)arg));
    sleep (1);
  }
}
```

Ví dụ (tt)

```
int main (int argc, char **argv) {
   int i;
   pthread_t tid[3];
   for (i=0; i<3; i++){
      pthread_create(&tid[i], NULL, func,
                                 (void*)&tid[i]);
  }
   sleep (5);
   return 0;
}
```

Ví dụ (tt)

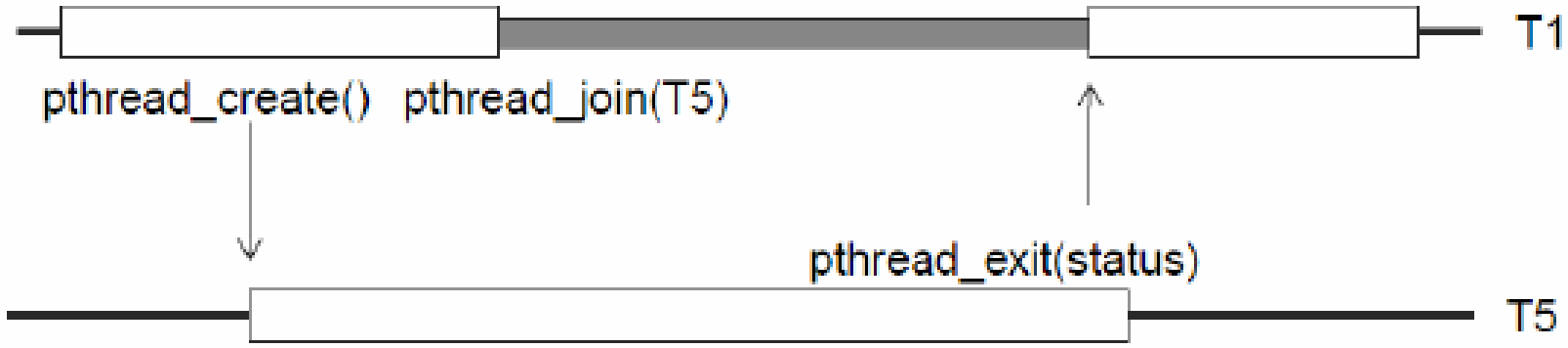
- Biên dịch và thực thi

```
$gcc pthcreate.c -o pthcreate -lpthread
$./pthcreate
This is thread -1208886368
This is thread -1219376224
This is thread -1229866080
This is thread -1208886368
This is thread -1219376224
This is thread -1229866080
```

Các hàm lập trình khác

- PTHREAD_JOIN

```
#include <pthread.h>
int pthread_join(pthread_t th,
                 void **thread_return);
```



Ví dụ

```
#include <pthread.h>
#include <stdio.h>
void* func(void* arg)
{
   int i;
   for (i = 0; i < 2; i++) {
    printf("This is thread %d\n",
                          *((int*)arg));
     sleep(1);
  }
}
```

Ví dụ (tt)

```
int main(int argc, char **argv) {
   int i;
   pthread_t tid[3];
   for (i=0; i<3; i++){
     pthread_create(&tid[i], NULL, func,
                                  (void*)&tid[i]);
     pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Ví dụ (tt)

- Biên dịch và thực thi

```
$gcc pthjoin.c -o pthjoin -lpthread
$./pthjoin
This is thread -1208710240
This is thread -1208710240
This is thread -1208710240
This is thread -1208710240
This is thread -1208710240
This is thread -1208710240
```

Truyền dữ liệu cho thread

```
#include <pthread.h>
#include <stdio.h>
struct char_print_parms {
   char character;
   int count;
};
```

Truyền dữ liệu cho thread (2)

```
void* char_print (void* args) {
   struct char_print_parms* p =
      (struct char_print_parms*) args;
   int i;
   for (i=0; i<p->count; i++)
     printf ("%c\n", p->character);
   return NULL;
}
```

Truyền dữ liệu cho thread (3)

```
int main () {
   pthread_t tid;
   struct char_print_parms th_args;
   th_args.character = 'X';
   th_args.count = 5;
   pthread_create(&tid,NULL,&char_print,
                            &th_args);
   pthread_join (tid, NULL);
   return 0;
}
```

Truyền dữ liệu cho thread (4)

- Biên dịch và thực thi

```
$gcc charprint.c -o charprint -lpthread
$./pthjoin
X
X
X
X
X
```

Lập trình trên Linux

- Lập trình IPC
  - Dùng pipe
  - Dùng semaphore
- Lập trình thread
  - Cơ bản về lập trình POSIX pthread
  - Giải quyết tranh chấp trên POSIX thread
    - MUTEX
    - Conditional variable
    - POSIX semaphore

Questions???