Inter Process Communication

Daniel Hagimont (INPT)

hagimont@enseeiht.fr

http://hagimont.perso.enseeiht.fr

I/O redirection

- A file is addressed through a descriptor
  - 0, 1 et 2 correspond to standard input, standard output, and standard error
  - The file descriptor number is returned by the open system call
- Basic operation
  - int open(const char *pathname, int flags);
    - O_RDONLY, O_WRONLY, O_RDWR …
  - int creat(const char *pathname, mode_t mode);
  - int close(int fd)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);

I/O redirection

- Descriptor duplication
  - dup(int oldfd); dup2(int oldfd, int newfd);
  - Used to redirect standard I/O

```
#include <stdio.h>
#include <unistd.h>
int f;
/* redirect std input */

…
close(0);          // close std input
dup(f) ;           // dupliquate f on the first free descriptor (i.e. 0)
close(f);          // free f
…
```

```
dup2(f,0);
close(f);
```

Exercice

- Copy with cat

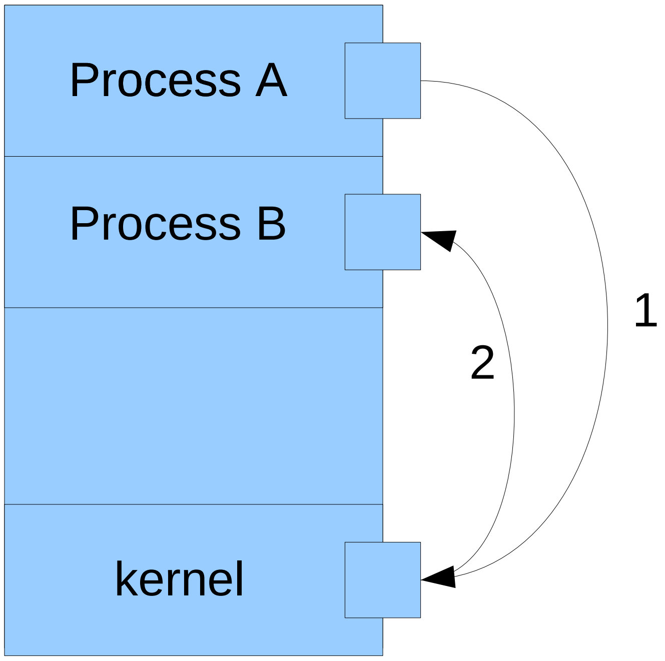
```
int main (int argc, char *argv[]) {
    char *src, *dest;
    int fd;
    if (argc != 3) {
        printf("usage: copy <src> <dest>\n");
        return 0;
    }
    src = argv[1];
    dest = argv[2];
    fd = open(dest, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR |
                              S_IWUSR | S_IRGRP | S_IROTH);
    if (fd == -1) {perror("error open");exit(0);}
    If (dup2(fd,1) == -1) {perror("error dup2");exit(0);}
    if (execlp("cat","cat", src, NULL) == -1) {perror("error execl");exit(0);}
}
```

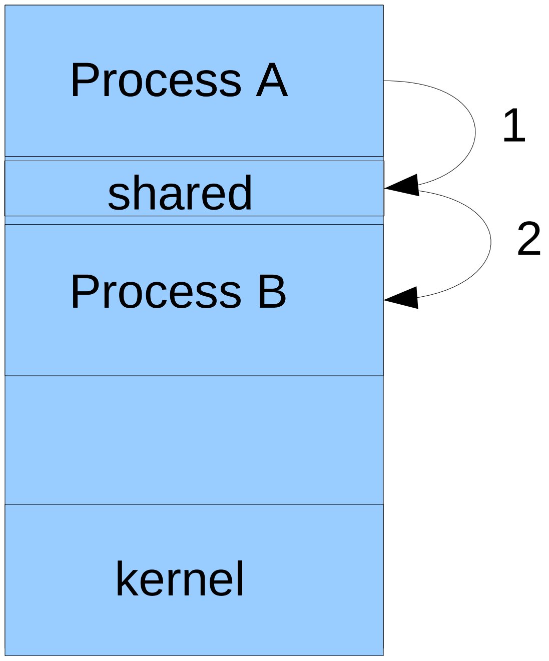
Cooperation between processes

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process. Advantages:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Process Interaction

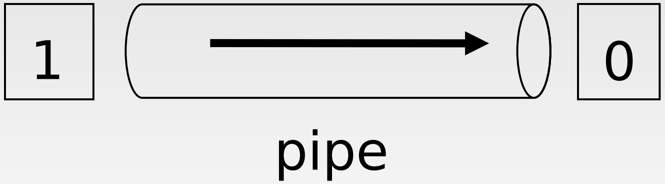
- How can processes interact in real time?
  - Through files but it’s not really “real time”.
  - Through asynchronous signals or alerts
  - By sharing a region of physical memory
  - By passing messages through the kernel/network





Pipe

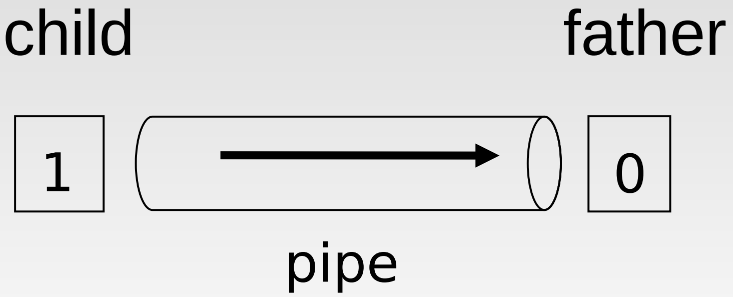
- Communication mechanism between processes
  - Fifo structure
  - Limited capacity
  - Producer/consumer synchronization



- int pipe (int fds[2]);
  - Returns two file descriptors in fds[0] and fds[1]
  - Writes to fds[1] will be read on fds[0]
  - Returns 0 on success, -1 on error
- Operations on pipes
  - read/write/close – as with files
  - When fds[1] closed, read(fds[0]) returns 0 bytes (EOF)
  - When fds[0] closed, write(fds[1]): kill process with SIGPIPE

Exercise

```
int main (int argc, char *argv[]) {
    int pipefds[2];
     pipe (pipefds);
       switch (fork ()) {
            case -1:      perror ("fork"); exit (1);
            case 0:
                        dup2 (pipefds[1], 1);
                        close (pipefds[0]); close (pipefds[1]);
                        execlp("ps", "ps", "-ef", NULL);
           default:
                        dup2 (pipefds[0], 0);
                        close (pipefds[0]); close (pipefds[1]);
                        execlp("grep", "grep", "firefox", NULL);
       }
}
```



Asynchronous notification (Signal)

- A process may send a SIGSTOP, SIGTERM, SIGKILL signal to suspend (CTRL-Z), terminate or kill a process using the kill function:
    - int kill (int pid, int sig);
    - A lot of signals … see man pages
    - Some signals cannot be blocked (SIGSTOP and SIGKILL)
- Upon reception of a signal, a given handler is called. This handler can be obtained and modified using the signal function:
    - typedef void (*sighandler_t)(int); // handler
    - sighandler_t signal(int signum, sighandler_t handler); // set a handler

Signal example

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void handler(int signal_num) {
    printf("Signal %d => ", signal_num);
    switch (signal_num) {
    case SIGTSTP:
        printf("pause\n");
        break;
    case SIGINT:
    case SIGTERM:
        printf("End of the program\n");
        exit(0);
        break;
    }
}
```

```
int main(void) {
  signal(SIGTSTP, handler);
  /* if control-Z */
  signal(SIGINT, handler);
  /* if control-C */
  signal(SIGTERM, handler);
  /* if kill process */
  while (1) {
    sleep(1);
    printf(".\n");
   }
printf("end");
exit(0);
}
```

- Signal handling is vulnerable to race conditions: another signal (even of the same type) can be delivered to the process during execution of the signal handling routine.
- The sigprocmask() call can be used to block and unblock delivery of signals.

Exercise

- Without signals
    - Try control-C, control-Z
- With signals (previous slide)
    - Try control-C, control-Z

```
int main(void) {
  while (1) {
    sleep(1);
    printf(".\n");
   }
}
```

Message queue

- Creation of a message queue
  - int msgget(key_t key, int msgflg);
- Control of the message queue
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
- Emission of a message
  - int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
- Reception of a message
  - int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

Exercise

creator

```
int main() {
   int msgid;
   key_t key = 1234;

   /* Create the queue */
   if ((msgid = msgget(key,
        IPC_CREAT | 0666)) < 0) {
        perror("msgget failedt");
        exit(1);
   }
}
```

sender

```
struct message {
        long mtype;
        char mtext[20];
};

int main() {
   int msgid;
   key_t key = 1234;
   struct message msg;

   /* get the queue */
   if ((msgid = msgget(key, 0666)) < 0) {
         perror("msgget failedt");
         exit(1);
   }

   /* send a message */
   msg.mtype=1;
   strcpy(msg.mtext, "hello vietnam");
   if ((msgsnd(msgid, (void *)&msg,
            20,0)) == -1) {
            perror("msgsnd failed");
            exit(1);
   }
}
```

receiver

```
struct message {
        long mtype;
        char mtext[20];
};

int main() {
   int msgid;
   key_t key = 1234;
   struct message msg;

   /* get the queue */
   If ((msgid = msgget(key, 0666)) < 0) {
          perror("msgget failedt");
          exit(1);
   }

   /* receive a message */
   if ((msgrcv(msgid, (void *)&msg,
         20,0,0)) == -1) {
         perror("msgsnd failed");
         exit(1);
   }
   printf("received : %s\n", msg.mtext);
}
```

Shared memory segment

- A process can create/use a shared memory segment using:
  - int shmget(key_t key, size_t size, int shmflg);
  - The returned value identifies the segment and is called the shmid
  - The key is used so that process indeed get the same segment.
- The owner of a shared memory segment can control access rights with shmctl()
- Once created, a shared segment should be attached to a process address space using
  - void *shmat(int shmid, const void *shmaddr, int shmflg);
- It can be detached using int shmdt(const void *shmaddr);
- Can also be done with the mmap function
- Example

Exercise

creator

```
int main() {
   int shmid;
   key_t key = 1234;
   /* Create the segment */
   if ((shmid = shmget(key, 10,
            IPC_CREAT | 0666)) < 0) {
            perror("shmget failed");
            exit(1);
   }
}
```

writer

```
int main() {
   int shmid, i, t;
   char *shm;
   key_t key = 1234;

   /* Get the segment */
   if ((shmid = shmget(key, 10,
                        0666))< 0) {
          perror("shmget failed");
          exit(1);
   }

   /* Attach the segment */
   if ((shm = shmat(shmid, NULL,
                   0)) == (void *) -1) {
          perror("shmat failed");
          exit(1);
   }

   t = 0;
   while (1) {
      sleep(1);
      for (i=0;i<5;i++) shm[i] = 'a'+t;
      shm[i] = 0;
      printf("wrote : %s\n",shm);
      t++;
   }
}
```

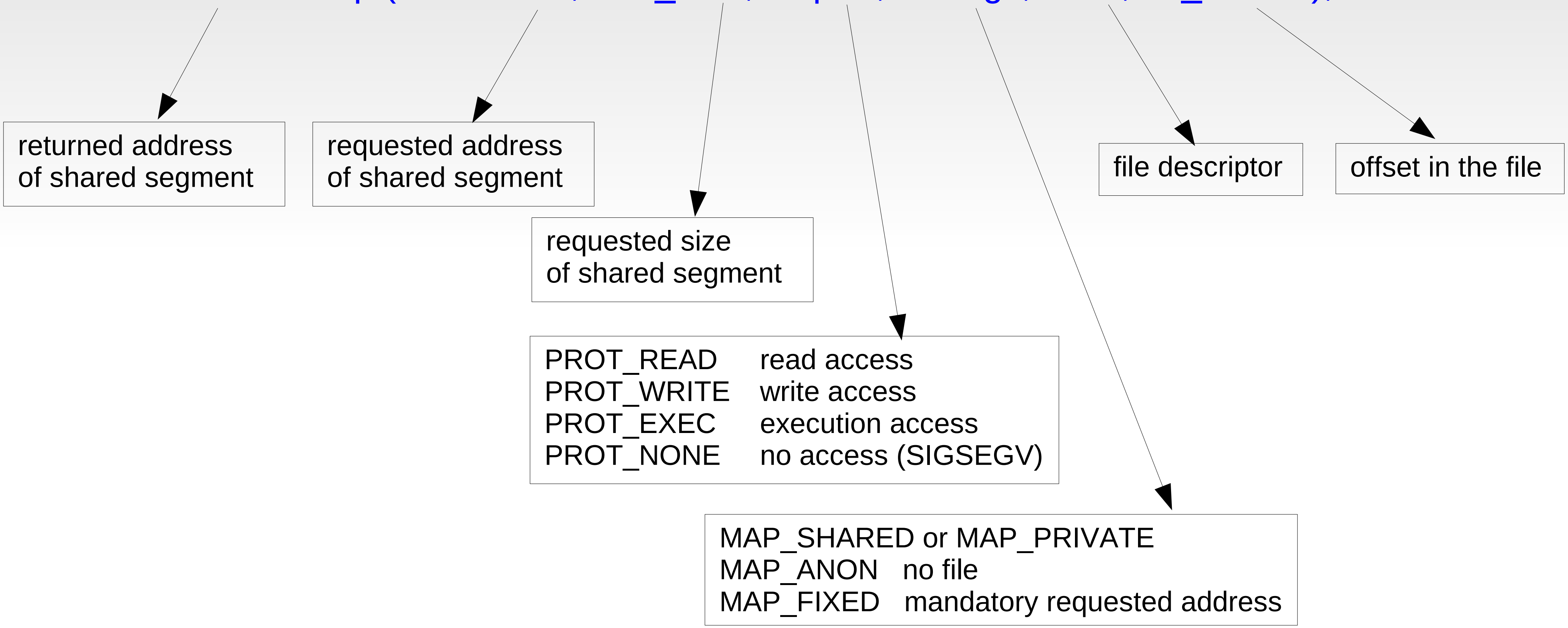
reader

```
int main() {
   int shmid, i, t;
   char *shm;
   key_t key = 1234;

   /* Get the segment */
   if ((shmid = shmget(key, 10,
                        0666))< 0) {
          perror("shmget failed");
          exit(1);
   }

   /* Attach the segment */
   if ((shm = shmat(shmid, NULL,
                      0)) == (void *) -1) {
          perror("shmat failed");
          exit(1);
   }

   while (1) {
      sleep(1);
      printf("read : %s\n",shm);
   }
}
```

Mmap

- Another interface for sharing memory

```
void * mmap (void * addr, size_t len, int prot, int flags, int fd, off_t offset);
```



returned address
of shared segment

requested address
of shared segment

requested size
of shared segment

PROT_READ	read access
PROT_WRITE	write access
PROT_EXEC	execution access
PROT_NONE	no access (SIGSEGV)

MAP_SHARED or MAP_PRIVATE
MAP_ANON   no file
MAP_FIXED   mandatory requested address

file descriptor

offset in the file

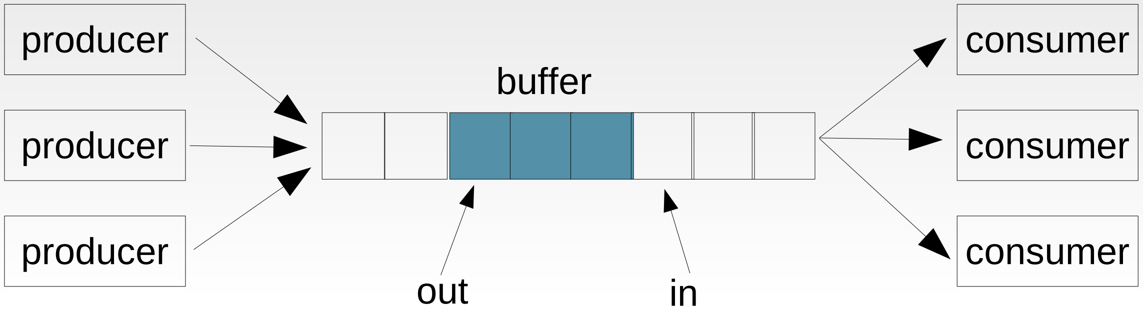
Mmap examples

```
long pagesize = sysconf(_SC_PAGESIZE);
int cf = open("content",O_RDWR);
char* base = mmap(0,pagesize, PROT_WRITE|PROT_READ,MAP_SHARED,cf,0);
```

```
char* b = mmap(0,pagesize,PROT_WRITE|PROT_READ,
                                        MAP_SHARED|MAP_ANON,-1,0);
```

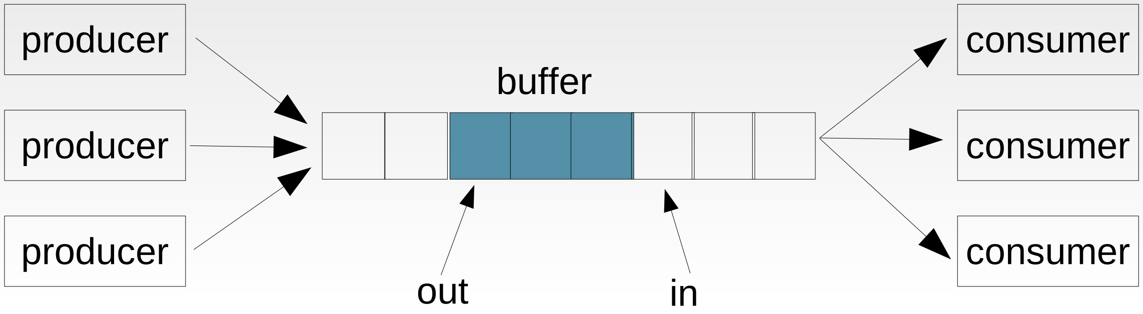
/* adresses [base,base+pagesize[accessible in read/write mode */

Use-case: producer-consumer



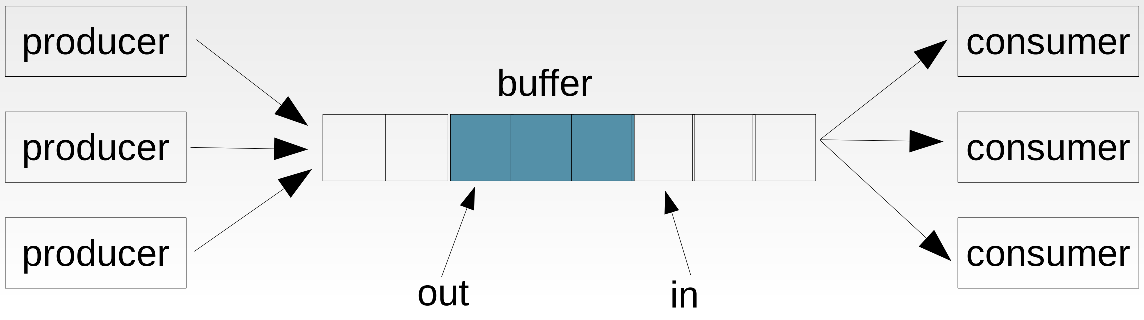
```
#define BUFFER_SIZE 10

typedef struct {
        char product;
        int amount;
} item;

item buffer [BUFFER_SIZE];
int in = 0; // where to produce
int  out = 0; // where to consume
int nb = 0; // number of items
```

Use-case: producer-consumer



```
void produce(item *i) {
        while (nb == BUFFER_SIZE) {
                // do nothing – no free place in buffer
        }
        memcopy(&buffer[in], i, sizeof(item));
        in = (in+1) % BUFFER_SIZE;
}
```

Use-case: producer-consumer



producer
producer
producer
buffer
out
in
consumer
consumer
consumer

```
item *consume() {
        item *i = malloc(sizeof(item));
        while (nb == 0) {
                // do nothing – nothing to consume
        }
        memcopy(i, &buffer[out], sizeof(item));
        out = (out+1) % BUFFER_SIZE;
        return i;
}
```

Socket

- A socket is defined as an endpoint for communication
- Used for remote communication
- Basic message passing API
- Identified by an IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication between a pair of sockets and bidirectionnal
- => second part of Teaching Unit (networking)

Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 3
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.3)