Operating Systems

IPC

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

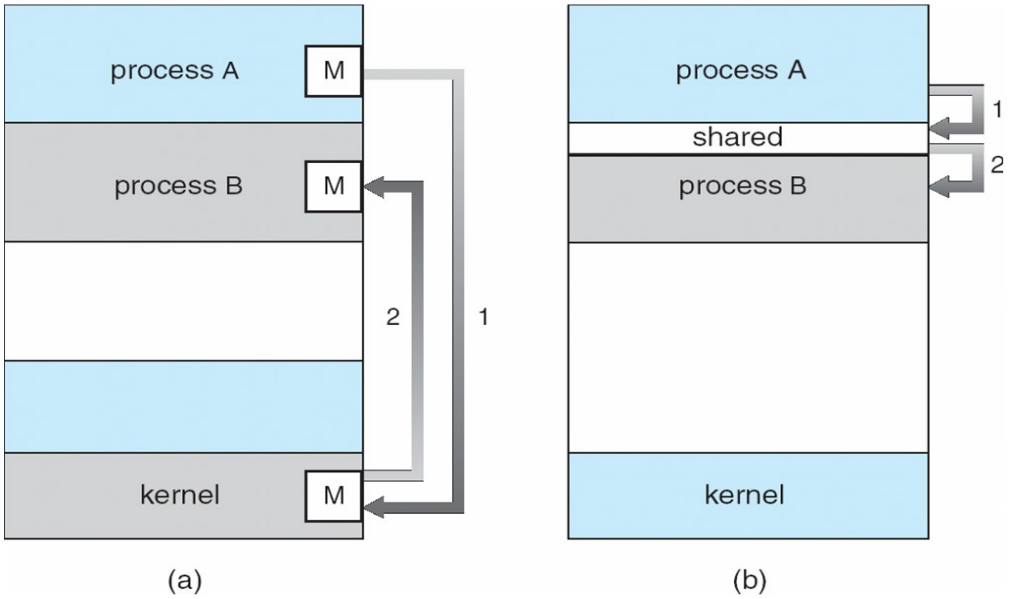
Interprocess Communication

In a complex system, processes might need to communicate with other processes

- sharing information
- enhancing the efficience of the computation
- improving modularity
- cross check

Basic IPC

- message passing (a)
- shared memory (b)



Message Passing

Compared to shared memory, passing a message requires copying memory

- Pro: easier implementation
- Cons: decreased efficiency

Three operations

- send(sender/mailbox, message)
- receive(sender/mailbox, message)

Communication

- synchronous (blocking) vs asynchronous (non-blocking)
- direct (process) vs indirect (mailbox)
- limited vs unlimited

Message Passing

A message queue is a system-managed object that implements a mailbox

Processes need to know an identifier of the queue to operate on it

Functionalities

- open (creates or attaches to a message queue)
- close (detaches from a message queue)
- unlink (destroy)
- get/set attr (inspects the status of a queue/sets operation flags)
- put a message
- wait for a message/ notify when a message is ready

Posix message interface

- mq_close() close a message queue
- mq_getattr() get the current attributes of a message queue
- mq_notify() notify the calling process when the queue becomes nonempty
- mq_open() open or create a message queue
- mq_receive() receive a message from a queue
- mq_send() put a message into a message queue
- mq_setattr() set the flags for a message queue
- mq_unlink() unlink (i.e. delete) a message queue

Message Queue Example

```
#define QUEUE_NAME  "/test_queue"
#define MAX_SIZE    1024
#define MSG_STOP    "exit"
int run=1;
void * queueServer(void * args) {
  mqd_t mq;
  ssize_t bytes_read;
  struct mq_attr attr;
  char buffer[MAX_SIZE + 1];
  /* initialize the queue attributes */
  attr.mq_flags = 0;
  attr.mq_maxmsg = 10;
  attr.mq_msgsize = MAX_SIZE;
  attr.mq_curmsgs = 0;
  /* create the message queue */
  mq = mq_open(QUEUE_NAME,
               O_CREAT | O_RDONLY | O_NONBLOCK,
               0644, &attr);
  while(run) {
    memset(buffer, 0x00, sizeof(buffer));
    bytes_read = mq_receive(mq, buffer,
                            MAX_SIZE, NULL);
    if(bytes_read >= 0) {
      printf("SERVER: Received message: %s\n", buffer);
    } else {
      printf("SERVER: None \n");
    }
    fflush(stdout);
    usleep(0.725 * 1e6);
  }
  mq_close(mq);
  mq_unlink(QUEUE_NAME);
  return NULL;
}
```

```
void * queueClient(void * args) {
 mqd_t mq;
  char buffer[MAX_SIZE];
  mq = mq_open(QUEUE_NAME, O_WRONLY);
  int count = 0;
  while(run) {
    snprintf(buffer, sizeof(buffer),
             "MESSAGE %d", count++);
    printf("CLIENT: Send message... \n");
    mq_send(mq, buffer, MAX_SIZE, 0);
    fflush(stdout);
    usleep(7.33 * 1e6);
  }
  mq_close(mq);
  return NULL;
}

int main(int argc, char** argv) {
  pthread_t client, server;
  printf("Start...\n");
  pthread_create(&server, 0, &queueServer, 0);
  pthread_create(&client, 0, &queueClient, 0);
  char c = getchar();
  run=0;
  pthread_join(server, NULL);
  pthread_join(client, NULL);

  printf("Done...\n");
  return (EXIT_SUCCESS);
}
```

Checking the queues

- From a shell, when the program is running you should see in the following directory

  /dev/mqueue all queues created by the running processes and also those not unlinked

- A queue survives the creating process
- Doing cat <queue_file> displays the status of the queue

Shared Memory

- Is a memory area shared between processes
- Similar to the message queues or the semaphores, multiple shared memories can be present
- Functionalities are similar to those of processes
  - opening
  - closing
  - destroying
- A shared memory, once opened needs to be "mapped" to the memory area of a process through mmap

Shared Memory Writer

```
int main(int argc, char** argv) {
  // create a shared memory object at key argv[1]

  char* resource_name=argv[1];
  int fd=shm_open(resource_name, O_RDWR|O_CREAT, 0666);
  if (fd<0){
    printf("cannot create shared memory object, error: %s \n", sys_errlist[errno]);
    exit(-1);
  }
  int ftruncate_result = ftruncate(fd, SHMEM_SIZE);
  if (ftruncate_result<0) {
    printf("cannot truncate shared memory object, error: %s \n", sys_errlist[errno]);
    exit(-1);
  }
  // map the descriptor to the memory of the process
  void * my_memory_area = mmap(NULL, SHMEM_SIZE, PROT_WRITE, MAP_SHARED,
                      fd, 0);
  int num_rounds=100;
  for (int i=0; i<num_rounds; ++i) {
    char* buffer=(char*) my_memory_area;
    sprintf(buffer, "This is the message %d", i);
    printf("writing [%s]\n", buffer);
    sleep(1);
  }

  int unlink_result=shm_unlink(resource_name);
  if (unlink_result<0) {
    printf("cannot unlink shared memory object, error: %s \n", sys_errlist[errno]);
    exit(-1);
  }

}
```

Shared Memory Reader

```
int main(int argc, char** argv) {
  // create a shared memory object at key argv[1]

  char* resource_name=argv[1];
  int fd=shm_open(resource_name, O_RDONLY, 0666);
  if (fd<0){
    printf("cannot link to shared memory object, error: %s \n", sys_errlist[errno]);
    exit(-1);
  }
  int SHMEM_SIZE=0;

  struct stat shm_status;
  int fstat_result = fstat(fd, &shm_status); /* To obtain file size */
  if (fstat_result<0){
    printf("cannot get stats from memory object, error: %s \n", sys_errlist[errno]);
    exit(-1);
  }

  SHMEM_SIZE = shm_status.st_size;

  void * my_memory_area = mmap(NULL, SHMEM_SIZE, PROT_READ, MAP_SHARED,
                       fd, 0);
  while (1) {
    char* buffer=(char*) my_memory_area;
    printf("%s\n", buffer);
    usleep(100000);
  }
  int unlink_result=shm_unlink(resource_name);
  if (unlink_result<0) {
    printf("cannot unlink shared memory object, error: %s \n", sys_errlist[errno]);
    exit(-1);
  }
}
```

Checking the Shmem

- Similar to /dev/mqueue the shared memory is accessible through a virtual file in /dev/shm
- reading/writing to that file results in reading/weiting to the shmem

Considerations

- We have seen the POSIX IPC API
- In conjunction to this API there is another set of functions that implement the SystemV primitives. These were prior the POSIX standardization
- In general the api is rather coherent for all objects
  - creation/opening: requires a string (filename) and returns an int (file descriptor)
  - specific functions operate on the object, solely based on the file descriptor
  - little things to remember, you can guess a lot

Exercises

- Implement a message queue by using a shared memory and semaphores