CS2106 Inter-process communication(IPC)

Mu Changrui

May 10, 2021

1 The Goal

Communication between processes in well-structured way(less participation of interrupt)
Three isssues
1. how one orecess can pass information to another
2. the SecondL: make sure two or more processes do not get in each other's way (if two precess have the same request)
3. proper sequencing when dependencies are present (eg. if A produce data and process B store down them)
ps: Murphy's law: anything that can go wrong will go wrong

2 Shared memory

2.1 General idea

1. Process P1 **creates(OS)** a shared memory region M
2. Process P2 **attaches(OS)** memory region M to its own memory space
3. P1 and P2 can now communicate using memory region M(like like normal memory r/w); IMPORTANT: Any writes to the region can be seen by all other parties

Only the (OS) tagged involves OS

2.2 Shared memory and race conditions/data race

Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
To avoid race condition, should prohibit more than one precess from reading and writing the shared data at the same time

Mu Changrui

Coz the common prob is that B starts to use one of the shared variables before process A finishes
The critical region: the part of the program where the shared memory is accessed
Four conditions to hold

1. No two processes may be simultaneously inside their critical regions.
2. No assunmptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

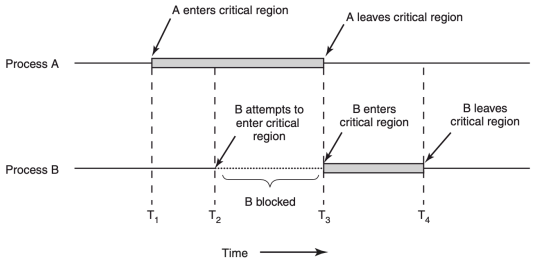


Figure 2-22. Mutual exclusion using critical regions.

2.3 A example of race condition

```
incCounter()
  load  reg1,counter
  add   reg1,reg1,1
  store reg1,counter
```

```
decCounter()
  load  reg2,counter
  sub   reg1,reg1,1
  store reg1,counter
```

- The final outcome depends on the interleaving of read and write operations to shared variable counter
- How many interleavings are possible between two processes? 4!/2*2 = 6
  - inc-LD, inc-ST, dec-LD, dec-ST → counter = *5*
  - inc-LD, dec-LD, inc-ST, dec-ST → counter = *4*
  - inc-LD, dec-LD, dec-ST, inc-ST → counter = *6*
  - dec-LD, dec-ST, inc-LD, inc-ST → counter = 5
  - dec-LD, inc-LD, dec-ST, inc-ST → counter = 6
  - dec-LD, inc-LD, inc-ST, dec-ST → counter = 4

Are they all OK? No!

The most probable value: 5(this program is too tiny, may no inter leaving involved)

2.4 Advantage

1. Efficient: Only the initial steps (e.g., Create and Attach shared memory region) involves OS
2. Ease of use:Shared memory region behaves the same as normal memory space

2.5 Disadvantages

1. Requires Synchronization:Shared resource(race condition); lock synchronization is needed
2. Limited to a single machine

2.6 POSIX Shared Memory in *nix

2.6.1 Basic steps

1. Create/locate a shared memory region M(OS)
2. Attach M to process memory space(OS)
3. Read from/Write to M(Written values visible to all process that share M)
4. Detach M from memory space after use(OS)
5. Destroy M(OS) (Only one process need to do this; Allowed only if M is not attached to any process)

Mu Changrui

2.6.2 Master program

The master program create the shared memory region and wait for the "worker" program to produce values before proceeding.

```
//The master program create the shared memory region and
//wait for the "worker" program to produce values before proceeding.
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main()
{
    int shmid, i, *shm;

    shmid = shmget( IPC_PRIVATE, 40, IPC_CREAT | 0600); //Create Shared Memory region.
    /*
    int shmget(key_t key, size_t size, int shmflg);
    shmget() returns the identifier of the System V shared memory segment associated with the
        value of the argument key.
     Permission bits can be set to allow other processes to use a particular
shared memory region. If you use the most permissive setting, any process can access. (e.g.
     a permission bits of 0666).
    */

    if (shmid == -1){
        printf("Cannot create shared memory!\n"); exit(1);
    } else
        printf("Shared Memory Id = %d\n", shmid);

    shm = (int*) shmat( shmid, NULL, 0 ); //Attach Shared Memory region.
    // Important: shmid is memory id but not memory address
    if (shm == (int*) -1){
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
    shm[0] = 0;//The first element in the shared memory region is used as "control" value
    //(0: values not ready, 1: values ready).
    //The next 3 elements are values produced by the worker program.
    //This is HLL synchronization
    //In the synchronization part, it will be found that it may fail in practice
    while(shm[0] == 0)
```

Mu Changrui

```
    {
    sleep(3);
    }
    for (i = 0; i < 3; i++){
        printf("Read %d from shared memory.\n", shm[i+1]);
    }
    //Detach and destroy Shared Memory region.
    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);
    /*shmctl() performs the control operation specified by cmd on the
      System V shared memory segment whose identifier is given in
      shmid.
    IPC_RMID(cmd)
    Mark the segment to be destroyed.
    The caller must ensure that a segment is eventually destroyed; otherwise its pages that
        were faulted in will remain in memory or swap.
    IMPORATNT: Without an explicit delete command(shmctl(..IPC_RMID...)), the shared memory
        region stays around.
    */
    return 0;
}
```

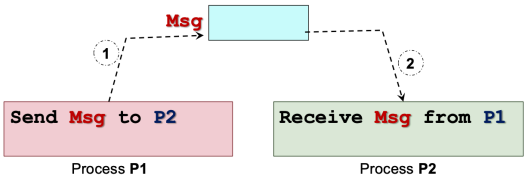
```
//similar header files to master program
 //By using the shared memory region id directly, we skip shmget() in this case.
int main()
{
    int shmid, i, input, *shm;
    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);
    shm = (int*)shmat( shmid, NULL, 0); //Attach to shared memory region.
    if (shm == (int*)-1)
        {
        printf("Error: Cannot attach!\n");
        exit(1);
        }
    for (i = 0; i < 3; i++)
    {
        scanf("%d", &input);
        shm[i+1] = input; //Write 3 values into shm[1 to 3]
    }
    shm[0] = 1;//Let master program know we are done!
    shmdt( (char*)shm );//Detach Shared Memory region.
    return 0;
```

```
}
```

3 Message Passing

Explicit Communication through exchange of messages
The method is consisted of two primitives: send and receive
is SYSTEM CALL

3.1 General Idea



The Msg have to be stored in kernel memory space
All send/receive operations must go through OS (i.e., a system call) (This is why it's much slower than shared memory)

1. Process P1 prepares a message M and sends it to Process P2
2. Process P2 receives the message M
3. Message sending and receiving are usually provided as system calls

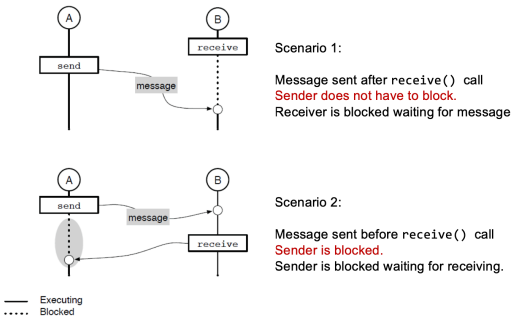
3.2 Additional issues to deal with

1. Naming: How to identify the other party in the communication
2. Synchronization: through the blocking behavior of the sending/receiving operations.

3.3 Blocking Primitives (Synchronous Message Passing)

Also known as rendezvous
IMPORTANT: no intermediate buffering required

Mu Changrui



3.3.1 Blocking receive

Receive(): Receiver is blocked until a message has arrived Most common
Receiver must wait for a message if it is not already available
Usually, receive is blocking(optimization by very experienced programmer)

3.3.2 Blocking sender

Send(): Sender is blocked until the message is received

3.3.3 Advantages

1. Portable: Can be easily implemented on different processing system(e.g. distributed system, wide area network)
2. Easier synchronization

3.3.4 Disadvantage

1. Inefficient: sender and receive should wait each other; usally needs intervention of OS,which makes it slower
2. Hard to use: massage are usually limited in format and size(other wise, there is no signal for the sender to stop block)

3.4 Non-blocking primitives(Asynchronous Message Passing)

3.4.1 Non-blocking receive

Receive(): Receiver either receive the message if available or some indication(can also not indicate) that message is not ready yet

Mu Changrui

1. Checks whether a message is available
2. If message is available, retrieves it and moves on
3. If message not available, continues without a message

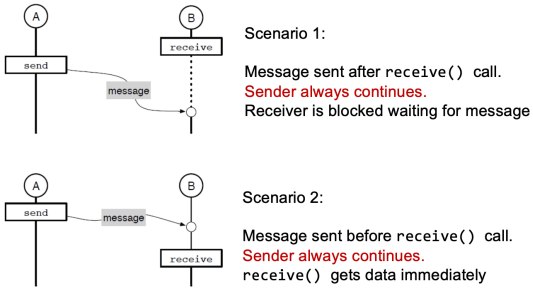
3.4.2 Non-Blocking sender

Send(): Sender resume operation immediately

1. Sender is never blocked even if the receiver has not yet executed the matching receive()
2. System buffers the message (up to certain capacity).
3. receive() performed by the receiver later will be completed immediately.

Disadvantages of non-blocking sender

1. Too much freedom for the programmer; program is complex to understand.
2. Finite buffer size means system is not truly asynchronous



IMPORTANT: in Asynchronous message passing, the receive is still blocked usually(non-blocking is just optimization by some experienced programmer)

3.4.3 Message Buffers

In asynchronous communication, there is an intermediate buffers between sender and receiver
The buffer is under the control of OS, so there is no need for program to do synchronization
Large buffer decouples sender and receiver making them less sensitive to variations in execution; they do not wait for each other unnecessarily
Small buffer would the communication indeed synchronous(receive and send should wait each other)
No amount of buffering helps when the sender is always faster than the receiver

3.5 The problem of Message Passing

1. In different machine, the messages may be lost by the network(ACK is needed), ACK may be lost(sequence numbers are needed)
2. In different machine Question of how processes are named
3. In different machine Authentication
4. In the same machine, performance copying messages from one process to another is always slower than shared memory
5. The producer consumer problem: Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one, in this waym the total number of message in the system remains constant(can be a pre-seted num)


```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with *N* messages.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI** (**Message-Passing Interface**). It is widely used for scientific computing. For more information about it, see for example Gropp et al. (1994), and Snir et al. (1996).

Mu Changrui

However, if the producer works faster than consumer, the message will be fulled and hence the producer will be blocked. On the contrary, the consumer will be blocked

3.6 Indirect Communication

3.6.1 mailbox/port

A malibox or port is a place to buffer a certain number of messages, typically specified in the send and receive calls are mailboxes(like a transfer station, but different to shared memory, this memory does not belong to any of the processes.

Mailbox Problem: both the producer and consumer would create mailboxes (send/wait to/from no one) e.g.

```
Send(MB, Msg); //Send Message Msg to Mailbox MB
Receive (MB, Msg); Receive Message Msg from Mailbox MB
```

3.6.2 rendezvous

if the receive is done first, the receiver is blocked until a send happens. This strategy is often known as a rendezvous.

3.7 MPI (Message-Passing Interface)

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is MPI (Message-Passing Interface).

3.8 Summary: logically implemention of send() and receive()

3.8.1 Direct or indirect communication(name perspective)

1. Direct communication: each process that wanna send message to another process must explicitly name the recipient or sender of the communication
chracters: 1. one-to-one pairs; 2. the address is in symmetry manner(there is also asymmetric vision where only the sender should name the receivers);
2. Indirect communication: the messages are sent to and received from mailboxes, or ports.
characters: 1. A link may have more than one processes; 2. one link one mailbox
In indirect communication, the message in mail box may not be received by every other processes, it depends on the setting in the next sub-section

3.8.2 setting of indirect communication

1. Allow a link to bea ssociated with two processes at most.
2. Allow atmost one process at a time to execute a receive() operation.
3. Allow the system to select arbitrarily which process will receive the message

3.8.3 Synchronous(blocking) or asynchronous(non-blocking) communication

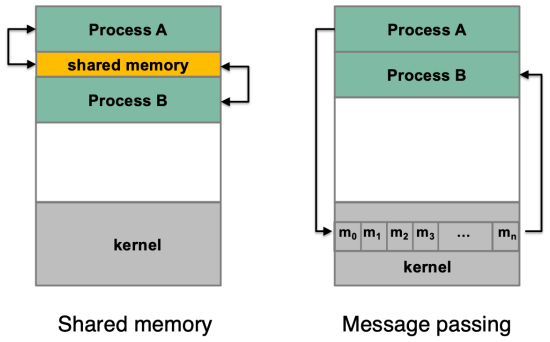
1. Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox.

2. Non blocking send. The sending process sends the message and resumes operation.
3. Blocking receive.The receiver blocks until a message is available.
4. Nonblocking receive. The receiver retrieves either a valid message or a null.

3.8.4 Automatic or explicit buffering(the queue perspectives)

1. Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
2. Bounded capacity.The queue has finite length n;thus,at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link' s capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
3. Unbounded capacity.The queue' s length is potentially infinite;thus,any number of messages can wait in it. The sender never blocks.

4 Comparation between shared memory and message passing



ps: for the shared memory, the memory space could be in the kernel space but does not have to nowadays(it could be in the data, heap or anywhere memory else)

4.1 Message passing

1. useful for exchanging smaller amounts of data, because no conflicts need be avoided
2. easier to implement than shared memory for intercomputer communication.

4.2 Shared memory

1. maximum speed and convenience of communication.
2. faster than message passing, as message- passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. shared- memory systems, system calls are required only to establish shared-memory regions. all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

3. IMPORTANT: it's processes' responsibility(rather than OS) for ensuring that they are not writing to the same location simultaneously. 4. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

4.2.1 unbounded buffer and bounded buffer

unbounded buffer: places no practical limit on the size of the buffer.
bounded buffer :assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

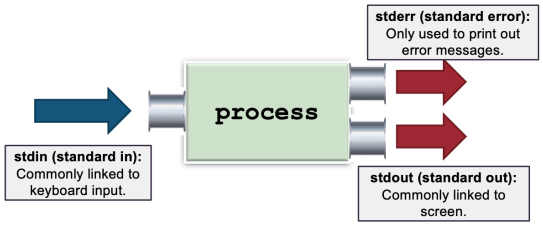
4.2.2 in and out

Two logical pointers;

1. in: the variable in points to the next free position in the buffer
2. out: the variable out points to the first full position in the buffer

5 Pipe

5.1 Three default communication channels of a process in Unix



Mu Changrui

Unix shell provides the "|" symbol to link the input/output channels of one process to another, this is known as piping
A communication channel is created with 2 ends:
1 end for reading,
2 the other for writing
A pipe can be shared between two processes

5.2 Behavior

1. Like an anonymous file
2. FIFO → must access data in order
3. functions as circular bounded byte buffer with implicit synchronization(that is Writers wait when buffer is full; Readers wait when buffer is empty)

5.3 Variants of pipe

1. Can have multiple readers/writers(The normal shell pipe has 1 writer and 1 reader)
2. half-duplex pipe(unidirectional: with one write end and one read end)
3. full-duplex(bidirectional: any end for read/write)

5.4 System call

```
#include <unistd.h>
int pipe( int fd[] );
//0 to indicate success; !0 for errors
//An array of file descriptors is returned:
//fd[0] == reading end
//fd[1] == writing end
```

Example code

```
#define READ_END 0 #define WRITE_END 1
int main()
{
int pipeFd[2], pid, len;
char buf[100], *str = "Hello There!";
pipe( pipeFd );
if ((pid = fork()) > 0)
```

Mu Changrui

```
{ /* parent */
close(pipeFd[READ_END]);
write(pipeFd[WRITE_END], str, strlen(str)+1);
close(pipeFd[WRITE_END]);
}
else
{ /* child */
close(pipeFd[WRITE_END]);
len = read(pipeFd[READ_END], buf, sizeof buf);
printf("Proc %d read: %s\n", pid, buf);
close(pipeFd[READ_END]);
}
}
```

In pracrtice dup() and dup2() provides a stronger function

6 Signal

A form of inter-process communication
asynchronous notification
Sent to a process/thread

6.1 The recipient of the signal must handle the signal by

1. A default set of handlers OR
2. User supplied handler (only applicable to some signals)

6.2 Reliable and unreliable signal

In some old version of Unix system, some signals are classified as unreliable signal
Unreliable signals are those for which the signal handler does not remain installed once called. These "one-shot" signals must re-install the signal handler within the signal handler itself, if the program wishes the signal to remain installed. Because of this, there is a race condition in which the signal can arrive again before the handler is re-installed, which can cause the signal to either be lost or for the original behavior of the signal to be triggered (such as killing the process). Therefore, these signals are "unreliable" because the signal catching and handler re-installation operations are nonatomic.
Those signals based on the early phase of the signal is called "unreliable signal", the signal value

Mu Changrui

is less than SIGRTMIN (Red hat 7.2 in, SIGRTMIN = 32, SIGRTMAX = 63) wereUnreliable signal

6.2.1 Handling unrealiable signals

e.g SIGINT(control+c) is a unreliable signal

```
#include <stdio.h>
#include <fcntl.h>   //For stat()
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h> //for waitpid()
#include <unistd.h>  //for fork(), wait()
#include <string.h>  //for string comparison etc
#include <stdlib.h>  //for malloc()
#include <signal.h>//for signal
void myOwnHandler(int signo)
   {

  signal(SIGINT, myOwnHandler);// each time when the handler is called should call
      themselves to make the self defined handler again

   if (signo == SIGINT ){
     if (currPid!=0)
     {
       //printf("kill %d!\n",currPid);
       int status=kill (currPid, SIGUSR1);
       see++;
       if (status==-1) printf("Nothing to kill\n");
       else printf("Someone killed me!\n");
       }
  else printf("Nothing to kill\n");

    }
}
signal(SIGINT, myOwnHandler);
```

Mu Changrui