Process Synchronization

Azzam Mourad

www.encs.concordia.ca/~mourad

mourad@encs.concordia.ca

COEN 346

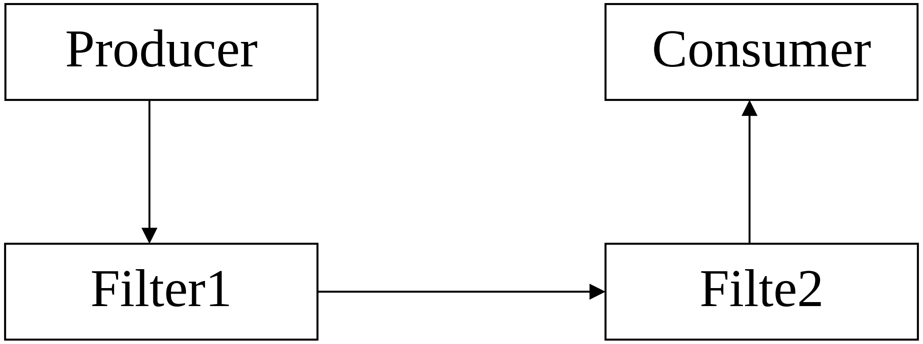
Agenda

- Process Synchronization using Semaphore
- Process Synchronization using Pipes

Part I   Process Synchronization using Pipes

- This problem is to write a multiple process program called *Assign4-1.c/Assign4-1.cpp* to manipulate information in a Pipeline fashion. The processes will communicate through UNIX pipes. You will use the pipe system call to create the pipes and the write (or send) and read (or recv).

- You will write a program with four processes, structure like:



Part I Process Synchronization using Pipes

- The Producer process will read an input file, one line at a time. Producer will take the each line of the input and pass it to process Filter1.
- Filter1 will scan the line and replace each blank character with an asterisk ("*") character. It will then pass the line to process Filter2.
- Filter2 will scan the line and convert all lower case letters to upper case (e.g., convert "a" to "A"). It will then pass the line to process Consumer.
- Consumer will write the line to an output file.

Part II Process Synchronization using Semaphores

- In this part, you will write a program with two processes, one producer and the other consumer, called *Assign4-1.c/Assign4-2.cpp*. Your program should implement the following scenario:

- The Producer process will read an input file, one line at a time, put its content in a buffer called *SharedBuffer*, and then write the content of *SharedBuffer* in a file called *SharedFile.txt*.

- The Consumer process will read the content of *SharedFile.txt*, put its content in the buffer *SharedBuffer*, and then write the content of *SharedBuffer* to an output file.

- The Producer and Consumer operate on the same *SharedBuffer* and *SharedFile.txt* and should synchronize on them using semaphores. The Consumer should be blocked from accessing them when the Producer is operating and vice versa.

Unix Pipes

- *Pipes are one of most used Unix process communication mechanisms, and can be classified as indirect communication.*
- *Pipes are half duplex, i.e. data flows only in one direction.*
- *A pipe can be used only between processes that have a common ancestor that created the pipe.*
- *A pipe is created by calling the pipe function:*

  pipe(int fd[2]);
- *Returns: 0 if OK, -1 on error.*

Unix Pipes

- *Two open file identifiers are returned by the pipe system call through the fd argument. fd[0] is open for reading, while fd[1] is open for writing and the output of fd[1] is the input for fd[0]*
- *A pipe in a single process is useless*
- Normally the process that calls *pipe* then creates child process.
- For a pipe in direction from the parent to the child, the parent closes the read end of the pipe (with *close*(*fd[0]*)), while the child closes the write end of the pipe (with *close*(*fd[1]*)). For the reasons to be clear later it is essential to do those closings.

Unix Pipes

- The parent then can use the standard *write* system call with fd[1] as *openFileID*, while the child can use the standard *read* system call with fd[0] as *openFileID*.

- After reading all data from the pipe whose write end has been closed, the next *read* returns 0. If there is no data in a pipe whose write end is not closed, the process that issues pipe *read* will be blocked until data is written in the pipe.

- Pipe *write* into the pipe whose read end has been closed returns negative value. If a pipe is full, the process that issues *write* will be blocked until there is enough room in the pipe for write data to be stored.

Example of Unix Pipes

```
int ends[2];
if (pipe(ends))
{ perror ("Opening pipe");
exit (-1); }

if (pid > 0) { // //---Parent process is consumer //
    close(ends[1]);
    Consumer (ends[0]); exit (0); }
else if (pid == 0) { // // Child process is producer //
    close(ends[0]);
    Producer (ends[1]); exit (0);
```

Example of Unix Pipes

Consumer

while ((count = read(fd, buff, MAXBUFF)) > 0) { cout << buff << endl; }

Producer

write(fd, buff, strlen(buff))

Process Synchronization

- Processes interact directly through some way of process cooperation.
- We shall study the following related topics:
  - memory sharing, also referred as critical section problem, situations when two or more processes access shared data
  - synchronization, situations in which progress of one process depends upon the progress of another process

Process Synchronization

- The synchronization problem can be solved using semaphore
- The semaphore will allow one process to control the shared resource, while the other process waits for for the resource to be released
- We consider here the Dijkstra semaphore

Process Synchronization

- A semaphore s is a non negative integer variable changed and tested only by one of two routines
  - V(S): [s=s+1]
  - P(S): [while(s==0) {wait}; s=s-1]

Example

```
Proc_A{
while(TRUE){
    ....
    write(x)
    V(s1);
    ....
    P(s2)
    read(y);
}
Proc_b{
While(TRUE){ ... P(s1); read(x)...write(y);... V(s2)...}
```