The Process in unix like OS

Inter Process Communication

Apurba Sarkar

Department of Computer Science and Tech.
Bengal Engg. and Sc. University Shibpur.

August 7, 2012

fork() again

- The only way in which a new process is created by Unix system is by executing the `fork()` system call.
- the `fork()` system call creates a copy of the the process that was executing.
- The process that calls the `fork()` is called the *parent process* and the new process is called the *child process*.
- Realize that if the text segment can be shared, then both the parent and child can share the text segment after `fork()`.

fork() again

- Realize that the child's copy of the data segment is the copy of the parent's data segment when the `fork()` operation takes place. It is not a copy from the program's disk file.
- The files that were open in the parent process before the `fork()` are shared by the child process after the `fork()`.
- This provides an easy way for the parent process to open specific files or devices and pass these open files to the child process.
- After the `fork()` parent closes the files that it opened for the child, so that both the processes are not sharing the same file.

fork() again

- Regarding the process variable that we were discussing in last couple of classes, the values of the following in the child process are copied from the parent
  - `real user ID`
  - `real group ID`
  - `effective user ID`
  - `effective group ID`
  - `process group ID`
  - `root directory`
  - `current working directory`
  - `signal handling settings`
  - `file mode creation mask`

fork() again

- The child process differs from the parent process in the following ways
  - the child process has a new, unique process ID
  - the child proces has a diffrent parent process ID
  - the child process has its own copies of the parent's file descriptors
  - the time left untill an alarm clock signal is set zero in the child

uses of fork()

- There are two uses of `fork()`
  1. A process wants to make a copy of itself so that one copy can handle an operation while the other copy does another task. This is typical for network servers.
  2. A process wants to execute another program. Since the only way to create a new process is with the `fork()` operation, the process must `fork()` to make a copy of itself, then one copy issues an `exec()` to execute the new program. this is typical for programs such as shells.

feature of exec()

- The program invoked by `exec()` system call inherits the following attributes from the process that calls `exec()`
  - `real user ID`
  - `real group ID`
  - `process ID`
  - `parent process ID`
  - `process group ID`
  - `time left untill am alarm clock signal`
  - `root directory`
  - `current working directory`
  - `file mode creation mask`
  - `file locks`

feature of exec()

- two attributes that can change when a new program is `execed` are
  - `effective user ID`
  - `effective group ID`
- if the set-user-id bit is set for the program being `execed`, the effective user ID is changed to the user ID of the owner of the program file.
- similarly, if the ser-group-id bit is set for the program being `execed`, the effective group ID is changed to the group ID of the owner of the program file.

dup() and dup2() system call

dup() An exixting file descriptor is duplicated by
`int dup(int fildes);`

- This returns a new file descriptor that shares the following with original *fildes*:
  - both refers to the same file or pipe
  - the access mode of the new file descriptor is the same as that of the original: read, write, or read/write.
  - both the file descriptors share the same file position.
  - it is guranteed that the new file descriptor returned by `dup()` is the lowest numbered available file descriptor.

dup() and dup2() system call

dup2() 4.3BSD provides another variant of this system call

`int dup2(int oldfildes,int newfildes);`

- here the *newfildes* argument specifies the new descriptor value that is desired. If this new descriptor value is already in use, it is first released, as if a `close()` had been done.

fork() and file sharing

- The sharing of files across a `fork()` system call merits further review, as it nicely describe the sharing of files between unix processes.
- There are three kernel data structures(actly tables) used to access a file
  - Every process has a process table entry. one portion of the process table is an array of file pointers. the file descriptors are just indexes into this array.
  - The file pointers in the process table point to entries in the file table. The file table is where the current file position for all open files in the system is maintained.
  - Another table maintained by kernel is *i-node table*. Every open file has an entry in the *i-node table*. An inode table entry contains all the inforamtion read from i-node on disk along with some other fields that the kernel needs to maintain.

fork() and file sharing

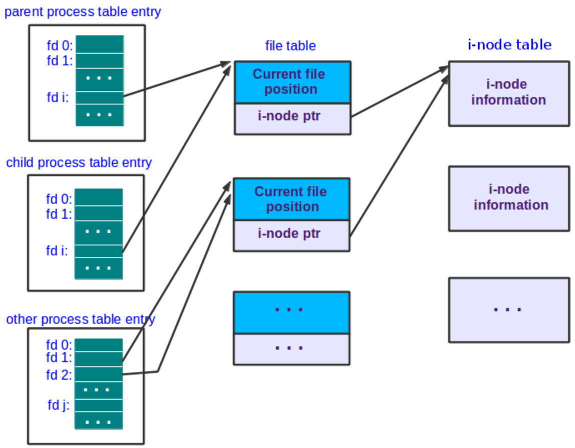

Figure 1: The sharing of files between processes

What is pipe?

- A pipe is used for one-way communication of a stream of bytes.
- The command to create a pipe is `pipe()`, which takes an array of two integers.
- It fills in the array with two file descriptors that can be used for low-level I/O. The following code shows how to crate a pipe,

```
int pfd[2];
pipe(pfd);
```

What is pipe? contd

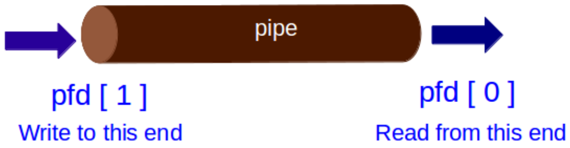

Figure 2: A pipe

I/O with pipe

- File descriptors created by pipe system call can be used for block I/O.
- Following code snippet shows how to write to and read from a pipe. `write(pfd[1], buf, SIZE);`
  `read(pfd[0], buf, SIZE);`

pipe and fork

- A single process would not use a pipe.
- They are used when two processes wish to communicate in a one-way fashion.
- A process splits in two using fork().
- A pipe opened before the fork becomes shared between the two processes.

I/O with pipe

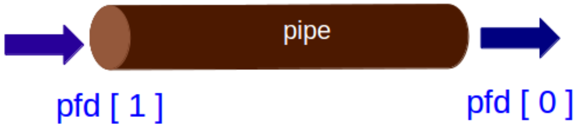

Figure 3: pipe before fork()

pipe and fork contd

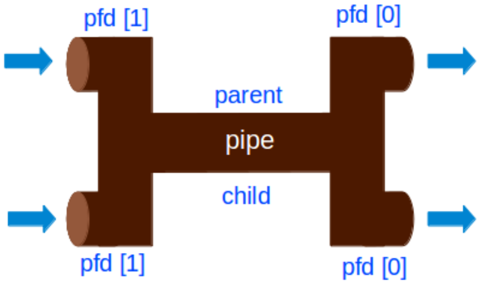

Figure 4: pipe after fork()

pipe and fork contd

- This gives two read ends and two write ends.
- The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
- Either process can write into the pipe, and either can read from it.Which process will get what is not known.
- For predictable behaviour, one of the processes must close its read end, and the other must close its write end to make it a simple pipeline again.

pipe and fork contd

- Suppose the parent wants to write down a pipeline to a child.
- The parent closes its read end, and writes into the other end.
- The child closes its write end and reads from the other end.
- When the processes have ceased communication, the parent closes its write end.
- This means that the child gets eof on its next read, and it can close its read end. This is shown in the next slide.

pipe and fork contd

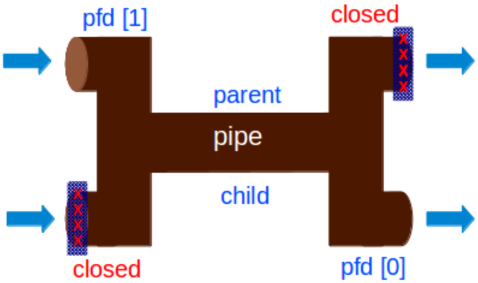

Figure 5: simple pipe line

An example

```
#include <stdio.h>
#define SIZE 1024
int main(int argc, char **argv)
{
  int pfd[2],nread,pid;
  char buf[SIZE];
  if (pipe(pfd) == -1){
  perror("pipe failed");
    exit(1);
  }
  if ((pid = fork()) < 0){
    perror("fork failed");
    exit(2);
  }
```

An example

```
    if (pid == 0) /* child */
    {close(pfd[1]);
      while ((nread = read(pfd[0], buf, SIZE))!= 0)
        printf("child read %s\n", buf);
        close(pfd[0]);
    } else {
      close(pfd[0]); /* parent */
      strcpy(buf, "hello...");
      /* include null terminator in write */
      write(pfd[1], buf,strlen(buf)+1);
      close(pfd[1]);
    }
    exit(0);
  }
```

dup system call

- A pipeline works because the two processes know the file descriptor of each end of the pipe.
- Each process has a stdin (0), a stdout (1) and a stderr (2).
- The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.
- If one of the processes replaces itself by an "exec". The new process will have files for descriptors 0, 1, 2, 3 and 4 open.

How will it know which are the ones belonging to the pipe?

pipe A real example “ls|wc”

- To implement “ls | wc” the shell will have created a pipe and then forked.
- The parent will exec to be replaced by “ls”, and the child will exec to be replaced by “wc”
- The write end of the pipe may be descriptor 3 and the read end may be descriptor 4.
- “ls” normally writes to 1 and “wc” normally reads from 0.

pipe A real example “ls|wc”

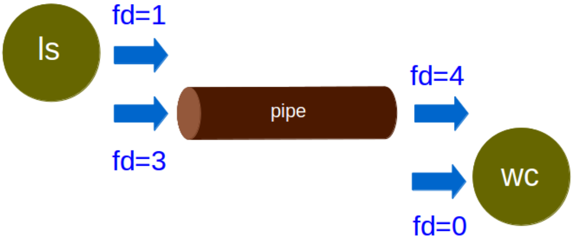

Figure 6: Before dup system call

pipe A real example "ls|wc"

- The dup2() system call takes an existing file descriptor, and another one that it "would like to be".
- Here, fd=3 would also like to be 1, and fd=4 would like to be 0.
- So we dup2 fd=3 as 1, and dup2 fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

pipe A real example “ls|wc”

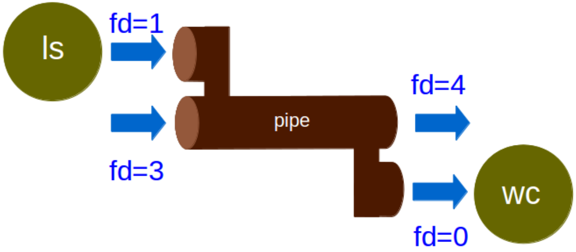

Figure 7: After dup system call

pipe A real example “ls|wc”

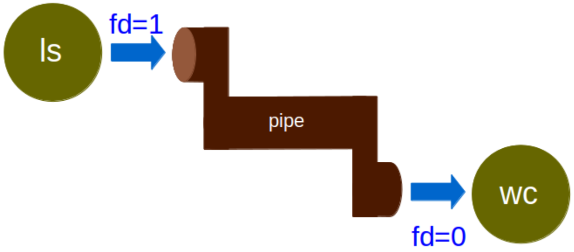

Figure 8: After close system call

pipe and fork example

Without any error checks, the program to do this is

```
int main(void)
{ int pfd[2],pipe(pfd);
  if (fork() == 0) {
    close(pfd[1]);
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("wc", "wc",(char *) 0);
  } else {
    close(pfd[0]);
    dup2(pfd[1], 1);
    close(pfd[1]);
    execlp("ls", "ls",(char *) 0);
  }
}
```

Summery

- pipe system call is always followed by a fork system call.
- As an effect of the `fork` the ends of the pipe are shared.
- We generally close one read end and one write end of the pipe to create a one way channel.
- pipe can be used for one way communication between two processes only when they have parent-child relationship.
- A pipeline may consist of three or more process (such as a C version of `ps | sed 1d | wc -l`).
- A process may want to both write to and read from a child. In this case it creates two pipes.

FIFOs

- FIFO stands for First In, First Out.
- A Unix FIFO is similar to a pipe.
- It is a one way flow of data, with the first byte written to it being the first bye read from it.
- Unlike pipes, however, a FIFO has a name associated with it, allowing unrelated processes to access a single FIFO.
- FIFOs are sometimes called *named pipes*

Creating FIFOs

- A FIFO is created by the `mknod` system call.
  `int mknod(char *`*pathname*`, int `*mode*`, int `*dev*`);`
- The *pathname* is a normal Unix pathname, and this is the name of the FIFO.
- The *mode* argument specifies the file access mode (read and write permissions for owner, group and other)and is logically or'ed with the `S_FIFO` flag from `<sys/stat.h>` to specify that a FIFO is being created.
- the *dev* value is ignored for a FIFO.
- A FIFO can also be created by the `mknod` command.
  `/etc/mknod `*name*` p`

Creating FIFOs

- once a FIFO is created, it must be opened for reading or writing. (`open()`, `fopen()`).
- Note that it takes three system calls to crate a FIFO and open it for reading or writing (`mknod`, `open`, and `open`) while the single `pipe` system call does the same thing.
- The `O_NDELAY` flag for a file, which is set either on the call to `open` or by calling `fcntl` after the file has been opened, sets the "no delay" flag for either a pipe or a FIFO. The effects of this flag is shown in the figure below.

Creating FIFOs

Condition	Normal	O_NDELAY
open FIFO, read-only with no process having the FIFO open for writing	wait until a process opens the FIFO for writing	return immediately, no error
open FIFO, write-only with no process having the FIFO open for reading	wait until a process opens the FIFO for reading	return an error immediately, `errno` set to `ENXIO`
read pipe or FIFO, no data	wait until there is data in the pipe or FIFO; return a value of zero if no process have it open for writing, otherwise return the count of data	return immediately, return value of zero
write, pipe or FIFO is full	wait until space is available, then write data	return immediately, return value of zero

FIFOs Rules for reading and writing

- A read requesting less data than is in the pipe or FIFO returns only the requested amt of data. The remainder is left for subsequent read.
- If a process asks to read more data than is currently available in the pipe or FIFO, only the data available is returned. The process must be prepared to handle a return value from read that is less than requested requested byte.
- If there is no data in the pipe or FIFO, and if no process have it open for writing, a read returns zero, signifying the end of file.

FIFOs Rules for reading and writing

- If a process writes less than the capacity of the pipe(which is atleast 4096 bytes) the write is guaranteed to be atomic. If, However, the write specifies more data than the pipe can hold, there is no guarantee that the write operation is atomic.
- If a process writes to a pipe or FIFO, but there are no process in existence that have it open for reading, the `SIGPIPE` signal is generated and the `write` returns zero with `errno` set to `EPIPE`.

Why Shared memory?

- pipe is used for communication between two related process.
- What if the they are not related and still want to communicate?
- shared memory comes handy in such situation.

Why Shared memory?

Consider a client server file copying program.

- The server reads from the input file. Typically the data is read by the kernel into one of its internal block buffers and copied from there to the server's buffer(the second argument to the `read` request)
- the server writes this data in a message, using one of(`pipe`, `FIFO or message queue`). Any of this IPC channel require the data to be copied from user's buffer into the kernel.

Why Shared memory?

Consider a client server file copying program.

- the client reads the data from from the IPC channel, again requiring the data be copied from kernel's IPC buffer to the client's buffer
- finally the data is copied from the client's buffer, the second argument to `write` system call, to the output file. This might involve just copying the data into a kernel buffer and returning, with the kernel doing the actual write operation to the device at some later time.

Why Shared Memory

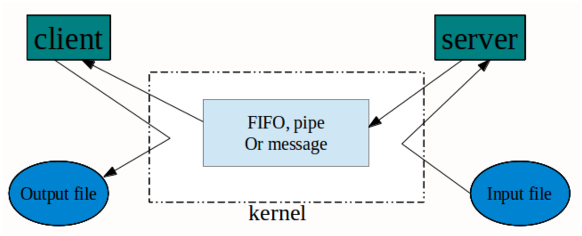

Figure 9: file copy program

Shared Memory

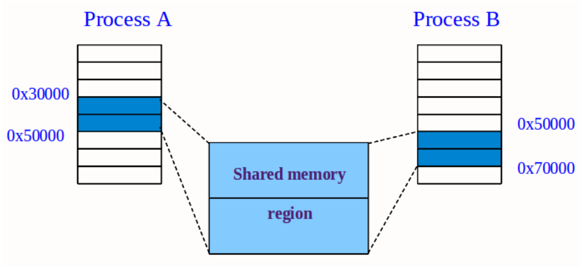

Figure 10: attaching shm to process virtual address space

Why shared memory

Consider the same client server file copying program.

- The server gets access to shared memory segment(semaphore may be used.)
- The server reads from the input file into the shared memory segment. The address to read into, the second argument to the `read` system call, points into shared memory
- when the read is complete the the server notifies the client, again using a semaphore
- the client writes the data from the shared memory segment to the output file.

shared memory

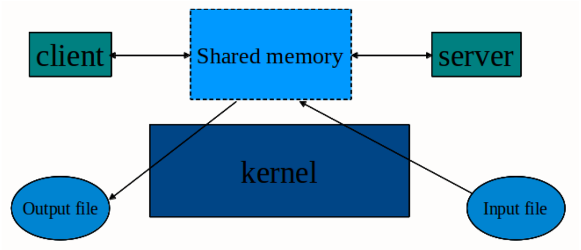

Figure 11: client server copy

Shared memory advantage

- The data is only copied twice from the input file into shared memory and from shared memory to the output file, unlike total of four copies with pipe, FIFO or message queues.

Procedure for Using Shared Memory

- Find a `key`. Unix uses this key for identifying shared memory segments.
- Use `shmget()` to allocate a shared memory.
- Use `shmat()` to attach a shared memory to anaddress space.
- Use `shmdt()` to detach a shared memory from an address space.
- Use `shmctl()` to deallocate a shared memory.

Keys

- To use shared memory, include the following:
  ```
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/shm.h>
  ```
- A key is a value of type `key_t`. There are three ways to generate a key:
  - Do it yourself
  - Use function ftok()
  - Ask the system to provide a private key.

Keys contd

- Do it yourself: use

  ```
   key_t SomeKey;
  SomeKey = 1234;
  ```

- Use ftok() to generate one for you:

  ```
   key_t = ftok(char *path, int ID);
  ```

  - path is a path name (e.g., "./")
  - ID is an integer (e.g., 'a')
  - Function `ftok()` returns a key of type `key_t`:

    ```
    SomeKey = ftok(‘‘./’’, ‘x’);
    ```

- Keys are global entities. If other processes know your key, they can access your shared memory.
- Ask the system to provide a private key using `IPC_PRIVATE`.

Asking for a Shared Memory

- Include the following:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Use `shmget()` to request a shared memory:

```
shm_id = shmget(
key_t key,   /* identity key */
int size,   /* memory size */
int flag);   /* creation or use *
```

- The flag, for our purpose, is either 0666 (rw) or `IPC_CREAT | 0666`

Asking for a Shared Memory

- The following creates a shared memory of size `struct Data` with a private key `IPC_PRIVATE`. This is a creation (`IPC_CREAT`) and permits read and write (0666).

```
struct Data { int a; double b; char x; };

int ShmID;

ShmID = shmget(
IPC_PRIVATE,                       /* private key */
sizeof(struct Data),                      /* size */
IPC_CREAT | 0666);                     /* cr & rw */
```

Asking for a Shared Memory

- The following creates a shared memory with a key based on the current directory:

```
struct Data { int a; double b; char x; };
int ShmID;
key_t key;
Key = ftok(``./'', `h');
ShmID = shmget(
Key,    /* a key */
sizeof(struct Data),
IPC_CREAT | 0666);
```

Asking for a Shared Memory

- When asking for a shared memory, the process that creates it uses `IPC_CREAT | 0666` and the process that accesses a created one uses `0666`.
- If the return value is negative (Unix convention), the request was unsuccessful, and no shared memory is allocated.
- Create a shared memory before its use!

shmget()

- It creates or locates a piece of shared memory.
- The pieces are identified by a key.
- The key is an integer agreed upon by all the communicating processes.
- It also specifies the size of the memory and the permission flags (eg -rwx - - - - - -).

shmget()

- The value returned by shmget is the shared memory identifier, *shmid*, or -1 if an error occurs.
- The *shmflag* argument is a combination of the constants shown below.

Numeric	Symbolic	Description
0400	SHM_R	Read by owner
0200	SHM_W	Write by owner
0040	SHM_R$\gg 3$	Read by group
0020	SHM_W$\gg 3$	Write by group
0004	SHM_R$\gg 6$	Read by other
0002	SHM_W$\gg 6$	Write by other

shmat()

- shmget call creates or opens a shared memory segment, but does not provide access to the segment for the calling process.
- It must be attached to the address space of the calling proces by shmat system call.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat(int shmid, char *shmaddr, int shmflg);
```

shmat()

- It returns the starting address of the shared memory segment
- If the *shmaddr* argument is zero, the system selects tthe address for the caller.
- If the *shmaddr* argument is nonzero, the returned address depends whether the caller specifies the SHM_RND value for the *shmflag*:
    - if the SHM_RND value is not specified, the shared memory segment is attached at the address specified by the *shmaddr* argument.
    - if the SHM_RND value is specified, the shared memory segment is attached at the address specified by the *shmaddr* argument, rounded down by the constant SHMLBA.(LBA stands for lower boundary address).

`shmat()`

- For all practical purpose, the only portable calls to *shmat* specify the *shmaddr* of zero.
- By default, the shared memory segment is attached for both reading and writing by the calling process. The `SHM_RDONLY` value can also be specified for the *shmflag* argument, specifying "read-only" access.

shmdt()

- When a process is finished with a shared memory segment, it detaches the segment by calling the `shmdt()` system call
  `int shmdt(char *shmaddr);`
- This call does not delete the shared memory segment, it just detaches the segment from its address space.
- To remove it completely, the *shmctl* system call is used.
  `int *shmctl(int shmid, int cmd, struct shmid_ds *buf);`
- A *cmd* of `IPC_RMID` removes a shared segment from the system.

SHM Server

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
main()
{ char c;
  int shmid;
  key_t key;
  char *shm, *s;

  /*We'll name our shared memory segment "5678".*/
  key = 5678;

  /* Create the segment.*/
  if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

SHM Server contd

```
/* Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the memory for the other process to read.*/
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /* Finally, we wait until the other process changes the first
      character of our memory indicating that it has read what we put there. */

    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

SHM Client

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /* We need to get the segment named "5678",
    created by the server. */
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

SHM Client Contd

```
/* Now we attach the segment to our data space.  */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*  Now read what the server put in the memory. */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /* Finally, change the first character of the segment to '*',
    indicating we have read the segment. */

    *shm = '*';

    exit(0);
}
```

Semaphore

- Semaphores are synchronization primitive.
- As a form of IPC, they are not used for exchanging large amts of data, as pipes, FIFOs, and message queues.
- Mainly intended to let multiple process synchronize their operations.
- Main use is to synchronize the access to shared memory segments.

Background

- Concurrent access to shared data may result in data inconsistency.
- A situation where several process accesses and manipulate the same shared data concurrently and the outcome of the execution depends on the particular order in which the acesses takes place, is called race condition.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Background

- Consider the following bonded buffer solution to the producer consumer problem.
- This scheme allows atmost `n-1` items in the buffer.

Producer

```
while (true) {
     /*Produce an item */
     while ((in+1)mod n == out)
     ; /* do nothing --
     no free buffers */
     buffer[in] = item;
    in = (in + 1) mod n;
}
```

Consumer

```
while (true) {
      while (in == out)
      ; /* do nothing --
      nothing to consume*/

      /*remove an item from
      the buffer*/
      item = buffer[out];
      out = (out + 1) mod n;
      return item;
}
```

Background

- Suppose we want to modify the above algorithm to remedy the deficiency.
- One posibility is to add an integer variable *counter* initialized to 0
- *counter* is incremented each time an item is added and decremented each time an item is removed.

Background

- The code for the producer consumer problem now...

Producer

```
while (true) {
     /*Produce an item */
     while (counter == n)
     ; /* do nothing --
     no free buffers */
     buffer[in] = item;
     in = (in + 1) mod n;
     counter = counter + 1;
}
```

Consumer

```
while (true) {
      while (counter == 0)
      ; /* do nothing --
      nothing to consume*/

      /*remove an item from
      the buffer*/
      item = buffer[out];
      out = (out + 1) mod n;
      counter = counter - 1;

      return item;
}
```

Background

- both the producer and consumer routines are correct separately but may not function correctly when executed concurrently.
- suppose that current value of *counter* is 5 and that the producer and consumer processes executes `counter = counter + 1` and `counter = counter - 1` concurrently.
- the value of the variable counter may be $4, 5$ or 6! The only correct result is 5.