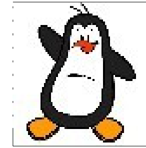
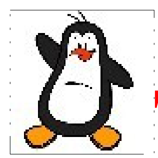


Lecture # 2.6
UNIX IO Models

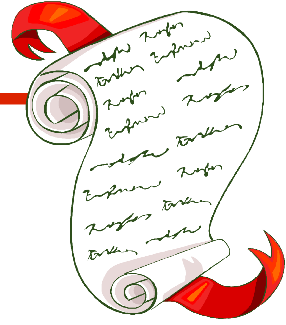
Course: Advanced Operating System

Instructor: Arif Butt

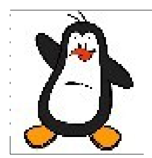
Punjab University College of Information Technology (PUCIT)
University of the Punjab



Today's Agenda



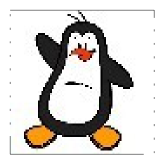
- UNIX I/O Models
  - Blocking I/O model
  - Non-Blocking I/O model
  - Multiplexed I/O model
  - Signal Driven I/O model
  - Asynchronous I/O model



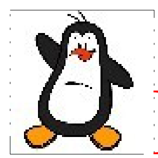
UNIX I/O Models

Before we proceed understanding each model, remember there are normally, two distinct phases for an input operation:

- Waiting for data to be ready in the kernel buffer.
- Copying the data from kernel buffer to process buffer.



Blocking I/O Model



Blocking I/O Model

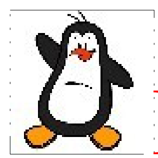
- Most prevalent model for I/O is the Blocking I/O model. When an I/O operation is initiated in this model, the calling process is blocked until the I/O is completed
- The process/application making the system call is put into a waiting state by the kernel. The application sleeps until the I/O operation is complete or has generated an error at which point it is scheduled to run again
- The most common system calls having a default blocking behavior are `read()`, `write()`, `recv()`, `send()`, `recvfrom()`, `sendto()`, `accept()`, `readv()`, `writev()`, `recvmsg()`, `sendmsg()`,

Pros:

- Easy to use and well understood
- Ubiquitous
- Beneficial if many fast I/O operations are to be made

Cons:

- Does not maximize I/O throughput
- Mostly causes all threads in a process to block



Blocking I/O Model

Application/Process | Kernel

`recvfrom()` — system call → No datagram ready

Wait for data to be available in kernel buffer

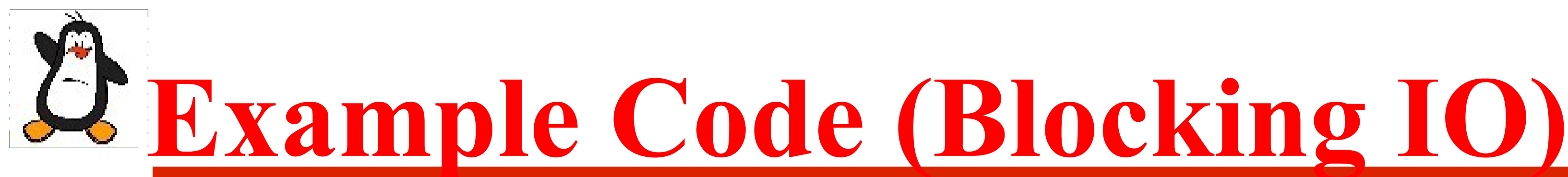
Process blocks and wait for
Data to be ready in kernel buffer
Data to be copied

Datagram ready

Copy datagram

Copy data From Kernel buffer to Process buffer

Copy complete → return OK → Process datagram



Example Code (Blocking IO)

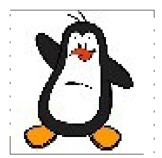
To create a large size file of 1GiB, you can use following command:

```
$ dd if=/dev/urandom of=bigfile1 count=1048576 bs=1024
```

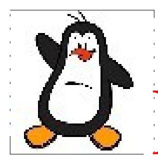
```
int main(){
   int fd1 = open("bigfile1",O_RDONLY);
   int fd2 = open("bigfile2",O_RDONLY);
   char buf[50];
   while(read(fd1,buf,50) != 0);

   printf("\nI have read Big File 1\n" );
   while(read(fd2,buf,50) != 0);

   printf("\nI have read Big File 2\nBye...\n" );
   return 0;
}
```



Non-Blocking I/O Model



Non Blocking I/O Model

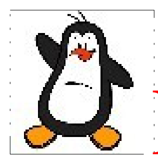
- By making an I/O call non blocking, we are telling the kernel that when the requested I/O operation cannot be completed w/o putting the process to sleep. Do not put the process to sleep, but return an error instead
- **Polling:** Polling is an operation in which an application sits in a loop on a non blocking descriptor to see if the operation can be performed
- We can make an I/O call non blocking, using **`fcntl()`** system call

Pros:

- Prevents a process from sleeping if the data is not available
- Parallel programming is possible

Cons:

- Difficult to code
- Polling wastes CPU cycles
- At times a partial read or write may be performed

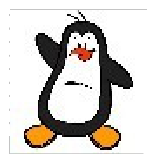


Non Blocking I/O Model

Application/Process		**Kernel**	
`recvfrom()`	system call → ← EWOULD BLOCK	No datagram ready	Wait for data to be ready in the kernel buffer
`recvfrom()`	system call → ← EWOULD BLOCK	No datagram ready	
• • •		• • •	
`recvfrom()`	system call →	Datagram ready Copy datagram	Copy data From Kernel buffer to Process buffer
Process datagram	← return OK	Copy complete	

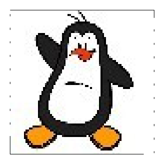
Process repeatedly calls read waiting for an OK

Process blocks while data is copied from kernel to application buffer



Example Code (Non-blocking IO)

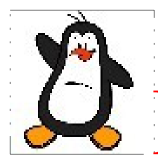
```
int main(){
        char buf1[] ="\nPlease enter your name:\n";
        write(1, buf1, sizeof buf1);
        fcntl(0, F_SETFL, O_NONBLOCK);
        char name[1024]; int n; int i=0;
        while (i++ < 3){
            n = read(0, name, 1023);
            if(n < 0)
                perror("");
            if(n > 0)
                break;
            sleep(5);
        }
        if (n > 0){
            char buf2[] = "\nWelcome Mr. ";
            write(1, buf2, sizeof buf2);
            write(1, name, n);
        }
        return 0;
}
```



Multiplexed I/O Model

Multiplexed I/O Model

- With I/O Multiplexed model, a process can call `select()` or `poll()` and block on this system call, instead of blocking in the actual I/O system call
- Comparing it with blocking I/O, there seems no advantage. It is because over here as well the process is either blocked on `select()` or on `recvfrom()`
- However, the advantage of using `select()` is that a process can wait for more then one descriptor to be ready. So this model allows a programmer to simultaneously monitor multiple file descriptors to see if I/O is possible on any of them



Multiplexed I/O Model

Application | **Kernel**

Process blocks in call to select, waiting for possibly many descriptors to become readable

`select()` —— system call ——> No datagram ready

Wait for data to be ready in kernel buffer

Datagram ready

<—— Return readable ——

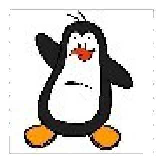
`recvform()` —— system call ——> Copy datagram

Process blocks while data is copied from kernel to application buffer

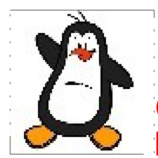
Copy data From Kernel Buffer to Process buffer

Copy Complete

Process datagram <—— return OK ——

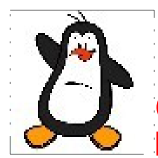


Signal Driven I/O Model

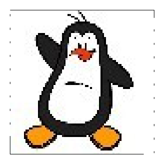


Signal Driven I/O Model

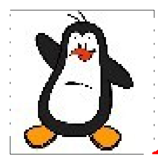
- In this model, the application program use signals, telling the kernel to notify with the `SIGIO<29>` signal when the descriptor is ready
- The programmer need to enable the descriptor for signal driven I/O and register the SIGIO signal handler using `sigaction()` system call. The return from this system call is immediate and our process continues i.e it is not blocked
- When the datagram is ready to be read, the operating system generates the `SIGIO<29>` signal for our process. The process can then read the datagram from the signal handler by calling `recvfrom()` and then notify the main loop that the data is ready to be processed



Signal Driven I/O Model



Asynchronous I/O Model



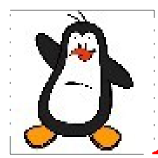
Asynchronous I/O Model(...)

POSIX defines synchronous and asynchronous I/O as follows:

- **Synchronous I/O** operation causes the requesting process to be blocked until the I/O operation completes. All previously discussed four models are synchronous
- **Asynchronous I/O** does not cause the requesting process to be blocked. In general, these functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of data from kernel to process buffer) is complete

Difference between Signal Driven and Asynchronous I/O model:

- With signal driven I/O model, the kernel tells us when an I/O operation can be initiated (i.e. when data is there in the kernel buffer)
- With asynchronous I/O model, the kernel tells us when an I/O operation is complete (i.e. when data is there in the process buffer)



Asynchronous I/O Model

Application | **Kernel**

`aio_read()` —system call→ No datagram ready

←return—

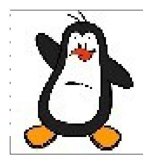
Process Continuous executing

No datagram ready → Datagram ready: Wait for data to be ready in kernel buffer

Copy datagram → Copy complete: Copy data from Kernel buffer to process buffer

Copy complete —Deliver signal→ **Signal handler process datagram**

Specified in `aio_read()`



Things To Do

If you have problems visit me in counseling hours. . . .