CS 24000 - Programming In C

Week 11: Processes

Zhiyuan Li
Department of Computer Science
Purdue University, USA

Review of Endianess

- What does “od –t X[size]” command do?
- When an integer a = 1 is dumped to a file
  - What is the sequence of the bytes?
    - On a little endian machine
    - On a big endian machine

Type conversion

- From unsigned char to int
- From signed char to int
- The result of byte addition and subtraction
  - Unsigned versus signed
  - Assigning to a char versus writing to an int
- We review these next Tuesday

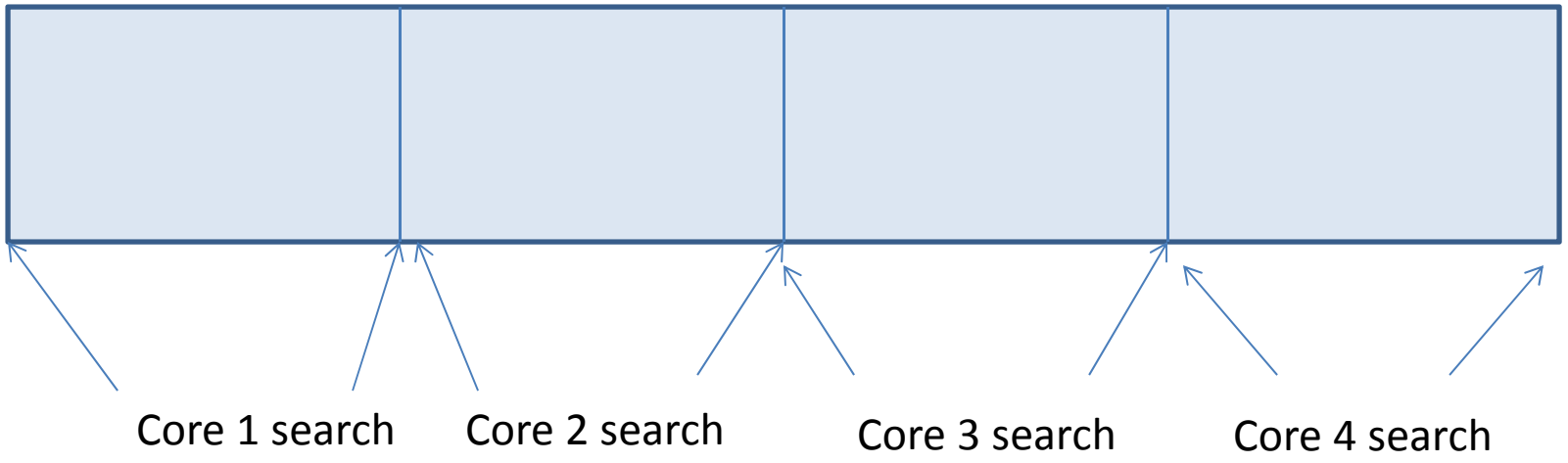
Lectures for the rest of the semester

- Programming with Unix processes in C
  - Process creation (fork)
  - Process state inheritance
    - Copy on write mechanism
  - Process termination (kill)
  - Process synchronization (wait)
  - Process communication
    - Pipe
    - Mutex
    - Shared memory

Project 3: Using multiple processes to search a file simultaneously

- Today's microprocessor has multiple "cores" (that are identical) on the same processor chip
- Different cores can perform operations simultaneously and independently
- They can also communicate through memory or through files
- There is a great incentive to coordinate multiple cores for solving a single computation problem
  - In Project 3, we coordinate multiple cores to search the same file in different places

Simultaneously search a file



- How do we activate multiple cores in the search?
  - Spawn multiple processes in the program
  - Spawn multiple "threads" in the program
  - The OS can "schedule" different processes (or threads) to different cores (depending on how busy the cores are)

- Conceptually, threads share the same address space
- Processes have separate address spaces
- Depending on the nature of the computation
  - There are pros and cons for processes and for threads
    - Easiness to program
    - Performance
- We will study the use of processes
  - Because the follow-up courses depend on processes more
    - System programming
    - Operating systems

What is a process?

- An execution instance of a program
- How a process is launched (i.e. spawned)?
- Often, it is launched when the operating system executes the command that we enter
  - The corresponding program is loaded
  - Ready to be executed as long as
    - Resource is available
    - No blocking conditions
- The operating system also launch “daemons” or service programs when the system is booted
  - We can use “ps” command to see how many processes are running in the background

- A new process is spawned by the system call fork()
- The fork() can be called in a user program also
  - That is how we are going to start multiple processes to perform task simultaneously
  -

Process state

- Roughly speaking, the process may be in one of the following states
  - Running
  - Ready (to be scheduled to run)
  - Sleep (blocked by some conditions to be resolved)
    - Sleep at a lock
    - Sleep and waiting for a timer wakeup
    - Sleep while waiting for a page fault to be served
    - … …
  - Zombie
    - Terminated but the resources are not released yet

Process context

- program code
- machine registers
- Program data
- System data
- User/system stack
- open files (file descriptors)
- environment variables

- Process ID (`pid`) — unique integer
- Parent process ID (`ppid`)
- Real User ID — ID of user/process which started this process
- Effective User ID — ID of user who wrote the process' program
- Current directory
- File descriptor table
- Seek position for each open file
- … …

continued

The fork() call

- Creates a child process by making a copy of the parent process --- an **exact** duplicate.
  - Inherit the program context (incl. the current instruction address, registers, stack, data, files, etc)
- Both the child *and* the parent continue running.

The copy on write (COW) mechanism

- Obviously to physically copy the entire context can be very time consuming
- Therefore the modern OS does “copy on write”
- At the time of fork(), the child gets a pointer to the entire context of the parent process
  - No physical copying
  - Read only (but marked as COW)
  - As soon as any “page” is modified (by child or parent) **after** the fork(),
    - a distinct copy of the written page is created
    - given to the writing process

Separate address space

- From this point of view, we say different processes have separate address spaces
  - Except for the regions that are explicitly designated as "shared" (we will use this for Project 4)
  - Some system level data structures are shared
    - **E.g. the global file table (with the seek position for each entry)**
- How do processes communicate data then?
  - Use the shared memory mentioned above
  - Use files (files are persistent objects and are shared)
    - A special kind of file called **pipe** is uniquely useful for data communication
    - We use pipe to communicate data in Project 3

- First, let us examine how the fork() call impacts the execution of the program

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int global_v=0;
int main( )
{
   pid_t child_PID;
   int local_v = 0;

   child_PID = fork();

   if(child_PID < 0) { // error
      printf("\n Fork failed\n");
      return 1;
   }
```

```
 // forkp.c
if(child_PID == 0)  // child process
        printf("In Child Process, global_v is
\t%d local_v is \t%d\n", global_v, local_v);

   else { //Parent process
        global_v = 1;
        local_v = 1;
        printf("In Parent Process, global_v is
\t%d local_v is \t%d\n", global_v, local_v);
   }
   return 0;
}
```

In Parent Process, global_v is 1 local_v is 1

In Child Process, global_v is 0 local_v is 0

```
int global_v=0;

int main(void)
{
   pid_t child_PID;
   int local_v = 0;
   int i=0;

   FILE * fp;

   fp = fopen("./output", "a");
   if (fp == NULL) {
      printf("failed to open ./output \n");
      return(1);
   }
 child_PID = fork();

   if(child_PID < 0) { // error
      printf("\n Fork failed\n");
      return 1;
   }
```

```
//forkfile.c
 if(child_PID == 0) {  // child process
          for (i=0; i<1000; i++) {
            fprintf(fp, "child re-write \n");
            usleep(1000);
          }
   }
   else { //Parent process
          for (i=0; i<1000; i++) {
               fprintf(fp, “parent re-write \n”);
               usleep(1000);
          }
   }
   return 0;
}
```

```
parent re-write
child re-write
parent re-write
parent re-write
parent re-write
…  …   …
child re-write
```

Notice how the printout are interleaved in an unpredictable way

Because we didn’t synchronize them

The wait and waitpid calls

- These are one way to synchronize between parent process and child processes
  - Only useful for the parent process to wait on the termination of (or certain conditions happening to) child processes
  - In the previous example, we can use such calls to let the parent process print after the child process

- ```
  #include <sys/types.h>
  #include <sys/wait.h>
  pid_t wait(int *statloc);
  ```
- Suspends the calling process until any of its child processes has finished.
- If successful, the return value is the process ID of the terminated child process, -1 on error.
- `*statloc` will have the status information about the child.
- We can make statloc a null pointer if we don’t need to status info.
  ```

```
int global_v=0;

int main(void)
{
   pid_t child_PID;
   int local_v = 0;
   int i=0;

   FILE * fp;

   fp = fopen("./output", "a");
   if (fp == NULL) {
      printf("failed to open ./output \n");
      return(1);
   }
 child_PID = fork();

   if(child_PID < 0) { // error
      printf("\n Fork failed\n");
      return 1;
   }
```

```
//forkwait.c
 if(child_PID == 0) {  // child process
         for (i=0; i<1000; i++) {
           fprintf(fp, "child re-write \n");
           usleep(1000);
         }
   }
   else { //Parent process
         wait((int *) 0);
         for (i=0; i<1000; i++) {
              fprintf(fp, “parent re-write \n”);
              usleep(1000);
         }
   }
   return 0;
}
```

Zombies and child processes

- The wait() call returns immediately (with error) if no child processes exist
- If a child process, c, terminates *before* the parent process does, c goes to the zombie state
  - until the parent (or a foster parent) does a wait() on c or explicitly kills c.
- If the parent process exits before a child process does, a grandparent process (e.g. init) becomes the foster parent
- Init will issue wait() for all those foster child processes
- For our project, we really do not want the system to be inundated by zombies (No new processes can be spawned then.)
- Also, if any child process has an infinite loop or blocked indefinitely, they will continue to exist even after the main program terminates
  - We must be careful in our project not to accidentally inundate the system with orphan processes

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main(void)
{
   pid_t child_PID;
   int i=0;


   child_PID = fork();

   if(child_PID < 0) { // error
      printf("\n Fork failed\n");
      return 1;
   }
```

```
 if(child_PID == 0) {  // child process
        printf("In Child Process, infinite loop\n");
        while (1) {}   // infinite loop
   }
   else { //Parent process
        printf("In Parent Process after
fork%d\n", child_PID);
        exit(0);
   }
}
```

After the main program returns, do "**ps**" to see the existing processes in the current login session

wait() is a special case of waitpid()

```
pid_t waitpid( pid_t pid, int *status, int
  opts )
```

- *pid* < -1
  - Wait for any child process whose process group ID is equal to the absolute value of *pid*. *(We won't worry about this in this course)*
- *pid* == -1
  - Wait for any child process.
  - wait( ) can be viewed as this case
- *pid* == 0
  - Wait for any child process whose process group ID is equal to that of the calling process. *(we won't worry about this in this course)*

- *pid* > 0
  - Wait for the child whose process ID is equal to the value of *pid*.
  - *options*
    - Zero or more of the following constants can be ORed.
      - WNOHANG
        - » Return immediately if no child has exited.
      - WUNTRACED
        - » Also return for children which are stopped, and whose status has not been reported (because of signal).
  - Return value
    - The process ID of the child which exited.
    - -1 on error; 0 if WNOHANG was used and no child was available.

Effect of fork() on file I/O

- The file descriptors and file pointers are all part of the process context
- When fork() is executed, the child process inherits these
  - Afterwards, these can be changed independently
    - Note that the files are not duplicated by fork()
    - Neither is the global file table (which is OS kernel's data structure)
    - Hence the seek position is shared between parent and child processes

```
int main(void)
{
   pid_t child_PID;
   int i=0;

   FILE * fp;

   fp = fopen("./sharedfile", "w+");
   if (fp == NULL) {
      printf("failed to open ./sharedfile
\n");
      return(1);
   }

   for (i=0; i<100; i++)
      fprintf(fp, "%d\n", i);

   child_PID = fork();

   if(child_PID < 0) { // error
      printf("\n Fork failed\n");
      return 1;
   }
```

```
 if(child_PID == 0) {  // child
process
            rewind(fp);
          fprintf(fp, "%d\n", -1);
            return (0);
   }
   else { //Parent process
          fprintf(fp, "%d\n",
1000);

   }
   return 0;
}
```

-1

2
3
4
5
… … …
96
97
98
99
1000

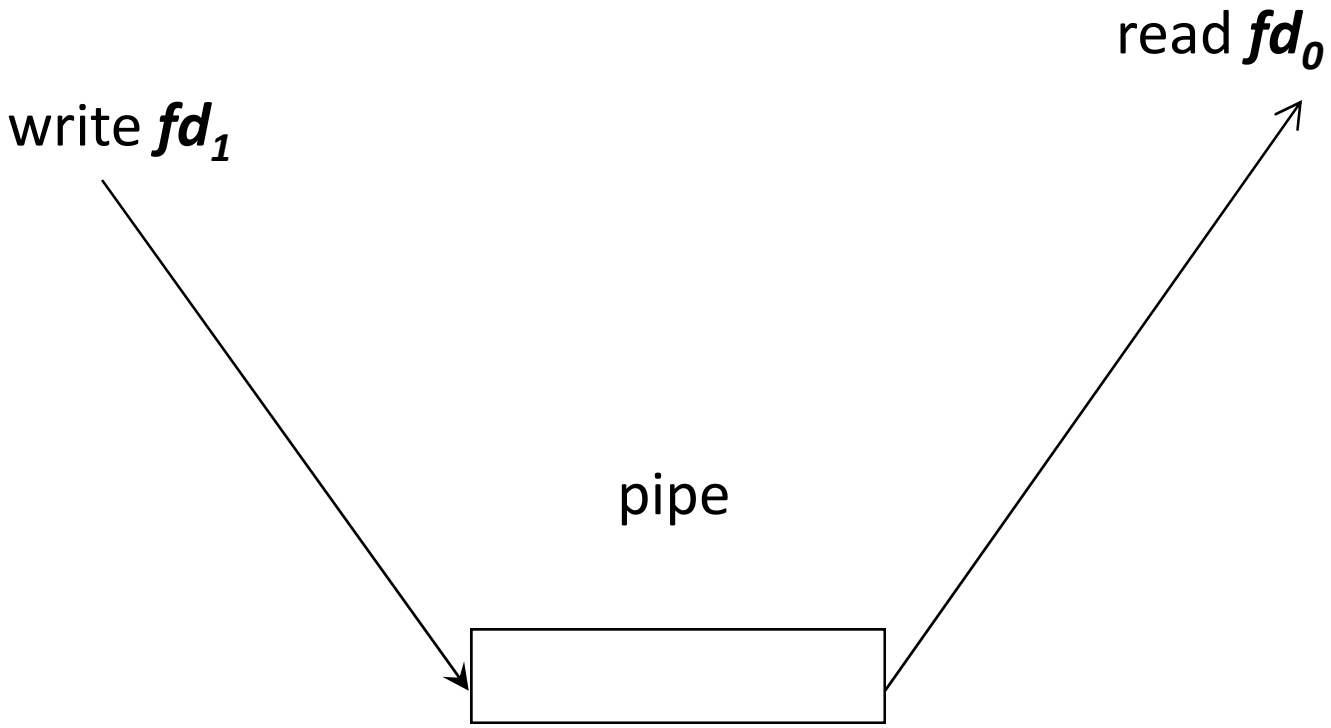
Using pipe for synchronization and communication

- We have seen that the regular file I/O is not a reliable way to communicate between processes (writing and reading order is unpredictable)
- Pipe is a more reliable way

pipe System Call (unnamed)

Creates a half-duplex pipe.

- Return: **Success: 0**; **Failure: -1**; Will set errno when fail.
- If successful, the ***pipe*** system call will return two integer file descriptors, pipefd[0] and pipefd[1].
  - pipefd[1] is the write end to the pipe.
  - pipefd[0] is the read end from the pipe.
- Parent/child processes communicating via unnamed pipe.
  - One writes to the write end
  - One reads from the read end
- A pipe exists **until both** file descriptors are **closed** in **all processes**


read fd_0
write fd_1
pipe

- However, when we fork a process after a pipe is open
  - Both the parent and the child will have the same write and read ends
  - It is a good idea to let each process close one of the ends
    - To avoid confusion and errors

If we want two way communication

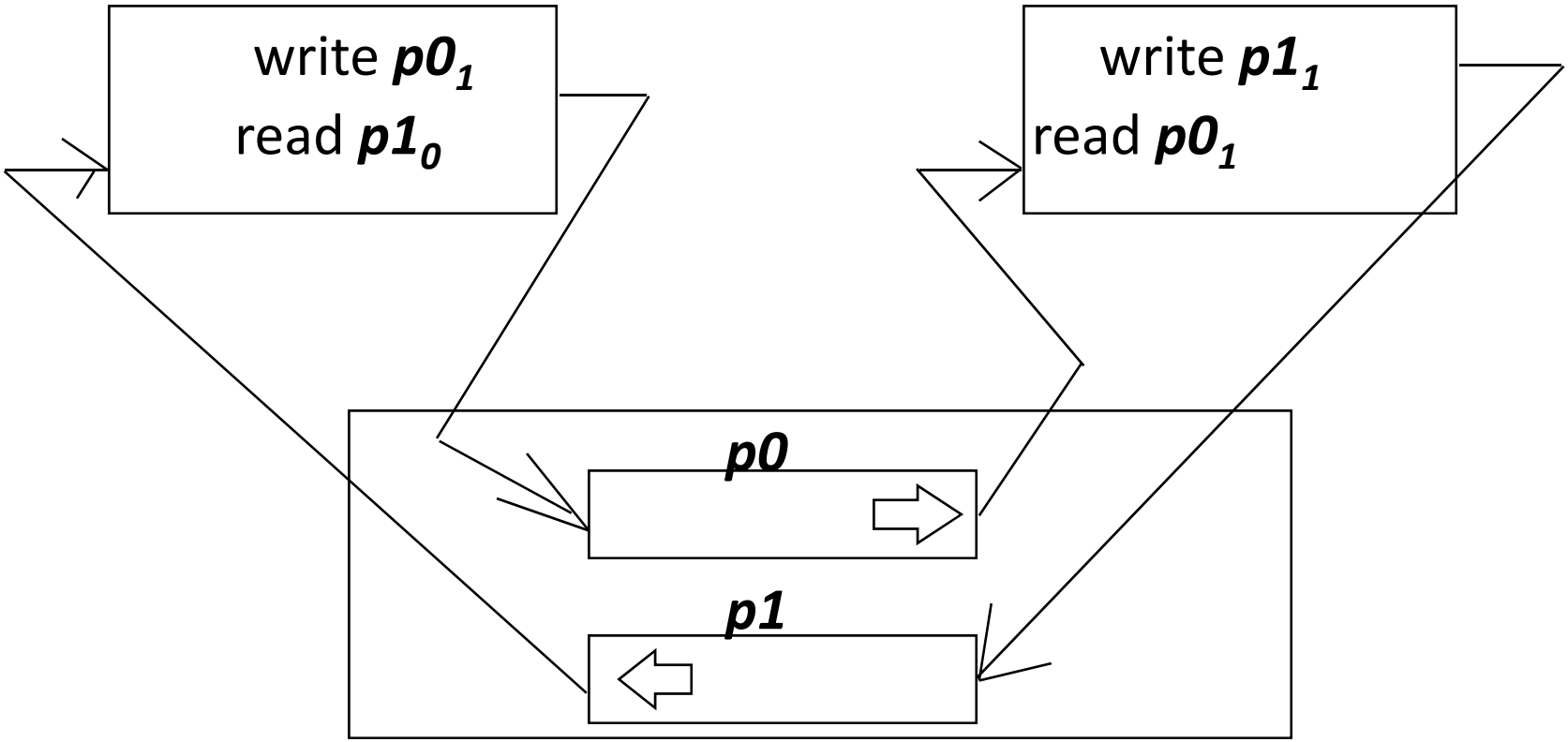
- We create two pipes
  - One for each direction

Full Duplex Communication via Two Pipes

Two separate pipes, say ***p0*** and ***p1***

Process A

Process B



Unnamed Pipes

- only used between related process, such as parent/child, or child/child process.
- exist only as long as the processes using them.
- Next, we examine an example of unnamed pipes

Named Pipes

We will let the system programming course and the OS course to tell you.

```
int  main (void)
   {
     pid_t pid;
     int mypipe[2];

     /* Create the pipe. */
     if (pipe (mypipe)) {
        fprintf (stderr, "Pipe failed.\n");
        exit(1); }
/* Create the child process. */
     pid = fork ();
     if (pid == (pid_t) 0)
      {
        /* This is the child process.
           Close other end first. */
        close (mypipe[1]);
        read_from_pipe (mypipe[0]);
        return EXIT_SUCCESS;
      }
```

```
else if (pid < (pid_t) 0)
      {
        /* The fork failed. */
        fprintf (stderr, "Fork failed.\n");
        return EXIT_FAILURE;
      }

else  {/* This is the parent process.
           Close other end first. */
        close (mypipe[0]);
        write_to_pipe (mypipe[1]);
        return EXIT_SUCCESS;
      }
   }
```

```
/* Read characters from the pipe and echo them to stdout. */

void read_from_pipe (int file)
  {
    FILE *stream;
    int c;
    stream = fdopen (file, "r");
    while ((c = fgetc (stream)) != EOF)
      putchar (c);
    fclose (stream);
  }
```

```
/* Write some random text to the pipe. */

void  write_to_pipe (int file)
  {
    FILE *stream;
    stream = fdopen (file, "w");
    fprintf (stream, "hello, world!\n");
    fprintf (stream, "goodbye, world!\n");
    fclose (stream);
  }
```

NOTE: fdopen() simply map a **file descriptor** (from unix file open call) to a **file pointer** (which is a C concept)

The mode must match

How does this read/write even work?

- A pipe behaves like a queue (*first-in, first-out*).
  - The first thing written to the pipe is the first thing read from the pipe.
  - For efficiency, the pipe has a limited size (we won't go into system details).
- Writes to the pipe will **block** when the pipe is full.
  - They block until another process reads some data from the pipe
  - The write call returns when all the data given to write have been written to the pipe.
- Reads from the pipe will **block** if the pipe is empty
  - until at least a byte is written at the other end.
  - The read call (when successfully read some data) then returns without waiting for the number of bytes requested to be all available.
  - Hence the most reliable way to read from a pipe is to continue reading a byte until the pipe is closed (reading will return with zero byte read)

- When all the descriptors on a pipe's write end are closed,
  - a call to read from its read end returns zero (for UNIX read call) or EOF (for C lib call) .

  From Posix and Linux man pages:

  "If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read(2) from the pipe will see end-of-file (read(2) will return 0)."

  NOTE: This statement is highly ambiguous. It implies that even if the buffer is nonempty, the read attempt would simply see end-of-file and read(2) will return 0. This is not true at least for the current implementation of Linux. (See next page)

What happens when trying to read from a pipe whose write end has been closed by all processes

- The Posix and Linux documents have been ambiguous on this issue, but according to a description of the Linux kernel description, the reading process will read all bytes left in the pipe by previous writing processes before they closed the write end.
- This is summarized by the following table from *Understanding the Linux Kernel,* Third Edition. By: Daniel P. Bovet; Marco Cesati, Publisher: O'Reilly Media, 2005

Table 19-3. Reading n bytes from a pipe

<table>
<tr><th rowspan="2">Pipe Size p</th><th colspan="3">At least one writing process</th><th rowspan="2">No writing process</th></tr>
<tr><th colspan="2">Blocking read</th><th>Nonblocking read</th></tr>
<tr><th></th><th>Sleeping writer</th><th>No sleeping writer</th><th></th></tr>
<tr><td>p = 0</td><td rowspan="2">Copy n bytes and return n, waiting for data when the pipe buffer is empty.</td><td>Wait for some data, copy it, and return its size.</td><td>Return -EAGAIN.</td><td>Return 0.</td></tr>
<tr><td>0 < p < n</td><td colspan="3">Copy p bytes and return p: 0 bytes are left in the pipe buffer.</td></tr>
<tr><td>p ≥ n</td><td colspan="4">Copy n bytes and return n: p–n bytes are left in the pipe buffer.</td></tr>
</table>

A Scheme for our project

- So we can have a cascading sequence of
  - Processes
  - A cascading sequence of pipes to accumulate the count
  - Wait()
- Each process
  - Remembers its only search range
  - Determines the range for the child process
  - Fork() a child process
  - Open the file again
  - Search through its own range
  - Wait for its child process to exit
  - Read from a pipe (upstream) the accumulated number or matches from child
  - Update the count
  - Send the count upstream
  - exit

Overhead

- Obviously, the data communication and synchronization incurs overhead
- Weighing the parallelism against the overhead is one of the dominating issues in parallel processing

- Pipe() has many other uses in operating systems
  - For standard file redirection
  - For pipelining two commands

Quiz 7 #1

- The following C library routine call

```
FILE *fp = fopen("./text.txt", "r+");
```

(a) will return a null pointer if file ./text.txt does not exist

(b) If ./text.txt exists, then this fopen call will allow the file to be read or over-written from the first byte position

(c) Both (a) and (b)

(d) Neither (a) nor (b)

Quiz 7 #1

- The following C library routine call

```
FILE *fp = fopen("./text.txt", "r+");
```

(a) will return a null pointer if file ./text.txt does not exist

(b) If ./text.txt exists, then this fopen call will allow the file to be read or over-written from the first byte position

(c) Both (a) and (b)

(d) Neither (a) nor (b)

- Answer (c): both (a) and (b)

Quiz 7 #2

- The following C library routine call

  FILE *fp = fopen("./text.txt", “w+");

(a) will return a null pointer if file ./text.txt does not exist

(b) If ./text.txt exists, then this fopen call will allow the file to be read or over-written from the first byte position

(c) Both (a) and (b)

(d) Neither (a) nor (b)

- Answer (b)

Quiz 7 #3

- Decimal number 2.25 is binary number

- (a) 10.11001
- (b) 10.1
- (c) 10.11
- (d) 10.01
- (e) none of the above

- Answer (d) 10.01