CMPT 300 D100
Operating System I
Threads – Chapter 4

Dr. Hazra Imran

Admin

- Assignment 1 grade released
- Assignment 1 average is 96% **(Good Job!)**
- Solution available on Canvas
- https://canvas.sfu.ca/courses/70193/pages/assignment-sample-solution?module_item_id=2510052

Thread Creation

```
#include <pthread.h>

int
pthread_create(       pthread_t*          thread,
                const pthread_attr_t*     attr,
                      void*               (*start_routine)(void*),
                      void*               arg);
```

optional

thread: Reference (or pointer) to the ID of the thread

attr: used to specify any attributes this thread might have. E.g. Stack size, Scheduling priority….

start_routine: The name of the function that the thread starts to execute.

if the function's return type is void *, then its name is simply written; otherwise, it has to be type-cast.

arg: Points to the single argument to be passed to the start_routine routine.

To pass multiple arguments, send a pointer to a structure.

void pointer allow us to pass in any type of argument

Returns 0 on success, or a positive error number on error

EAGAIN The process lacks the resources to create another thread.
EINVAL A value specified by attr is invalid.

Example

```c
#include <stdio.h>
#include <pthread.h>

void *print_msg(void *ptr) {
    printf("%s \n", (char *)ptr);
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, print_msg, "Going to thread function");
    printf("main: End \n");
    return 0;
}
```

```
himran@TABLET-FCBQI4FM:/mnt/c/Users/hazra/CMPT300demo/ch4_Threads$ gcc -pthread PthreadCreate1.c -o PthreadCreate1
```

main End
Going to thread

→ main: End

Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

struct args {
    char* EmpName;
    int id;
};

void *print_msg(void *input) {
    printf("Empname: %s\n", ((struct args*)input)->EmpName);
    printf("id: %d\n", ((struct args*)input)->id);
}

int main() {
    struct args *Test = (struct args *)malloc(sizeof(struct args));
    char name[] = "Chris";
    Test->EmpName = name;
    Test->id = 22;

    pthread_t tid;
    pthread_create(&tid, NULL, print_msg, (void *)Test);
    return 0;
}
```

Thread attributes

- At the time a thread is created, we can optionally set some attributes e.g., pthread setscope, pthread setschedpolicy, pthread setdetachstate

- pthread setscope – Either PTHREAD SCOPE SYSTEM or PTHREAD SCOPE PROCESS, indicating how the scheduler should deal with CPU contention between this thread and other processes

- pthread setschedpolicy – Can set a scheduling policy for a thread

fifo
RR
Batch ...

pthread setdetach state – Changes the behaviour when a thread finishes.

One of:

PTHREAD CREATE JOINABLE – the default state.

- This means that the thread will remain in a zombie state until another thread joins it to collect its return value

- PTHREAD CREATE DETACHED – start a thread in a detached state. When this thread finishes, it automatically disappears. It cannot be joined and its return value is ignored

Wait for a thread to complete

EINVAL → tid is invalid

```
int pthread_join(pthread_t thread, void **value_ptr);
```

thread: Specify the ID of the thread that the parent thread waits for.

ESRCH

value_ptr: The value returned by the exiting thread is caught by this pointer.

If the value is **NULL**, the termination status is not returned.

If value_ptr is not NULL, then the value passed to pthread_exit() by the terminating thread will be available in the location referenced by value_ptr, provided pthread_join() succeeds.

① wait for thread to finish

② clean up resources

Returns a zero when it completes successfully.
Any other returned value indicates that an error occurred.

Example : Clicker

```c
#include <stdio.h>
#include <pthread.h>

void *print_msg(void *ptr) {
    printf("%s \n", (char *)ptr);
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, print_msg, "Going to thread function");
    pthread_join(tid, NULL);
    printf("main: End \n");
    return 0;
}
```

A)
main:End

B)
main:End
Going to thread function

C)
Going to thread function
main:End

Example : Clicker

```c
#include <stdio.h>
#include <pthread.h>

void *print_msg(void *ptr) {
    printf("%s \n", (char *)ptr);
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, print_msg, "Going to thread function");
    pthread_join(tid, NULL);
    printf("main: End \n");
    return 0;
}
```

```
Going to thread function
main: End
```

Issues with thread implementation

Some issues that cause problems include:

- Multiple simultaneous calls to pthread_join() specifying the same target thread have undefined results.
- Failing to join with a thread that is joinable produces a "zombie thread".
  - Each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

Thread exit

- pthread_exit() is used to exit a thread.

- This function is usually written at the end of the starting routine

```
void pthread_exit(void *retval);
```

retval - Return value of thread.

- This routine kills the thread.

- The pthread_exit function never returns

safety + termination status.

Thread cancellation

- Thread cancellation - Terminating a thread before it has finished

- General approaches:
  - 1 Asynchronous cancellation terminates the target thread immediately
  - 2 Deferred cancellation allows the target thread to periodically check if it should be cancelled

Sharing memory

- A child thread can access its parent thread's stack, but not the other way around

- In general, threads cannot access their siblings' stacks

- Any data passed back and forth between threads should be done using the heap

Linux note

- Unlike other operating system kernels, Linux does not make a real distinction between threads and processes
- Both fork and pthread create are implemented in terms of the Linux-specific clone system call
- fork requests that the new process/thread be given a totally new memory image
- pthread create requests that the new process/thread share the heap and static memory segments with the parent
- Threads in Linux have their own distinct PID, PCB, etc.

Summary

- Thread: a single execution stream within a process
- Switching between user-level threads is faster than between kernel threads since a context switch is not required.
- User-level threads may result in the kernel making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.

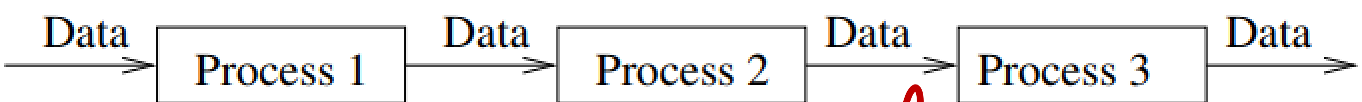
✓ ch-4

Pipes

P1 P2 unamed

- Acts as a channel abstraction allowing two processes to communicate

- Pipes are used to specify that the output of one process is to be used as the input to another process



unnamed

- **Issues:**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

stream pipes

fifos

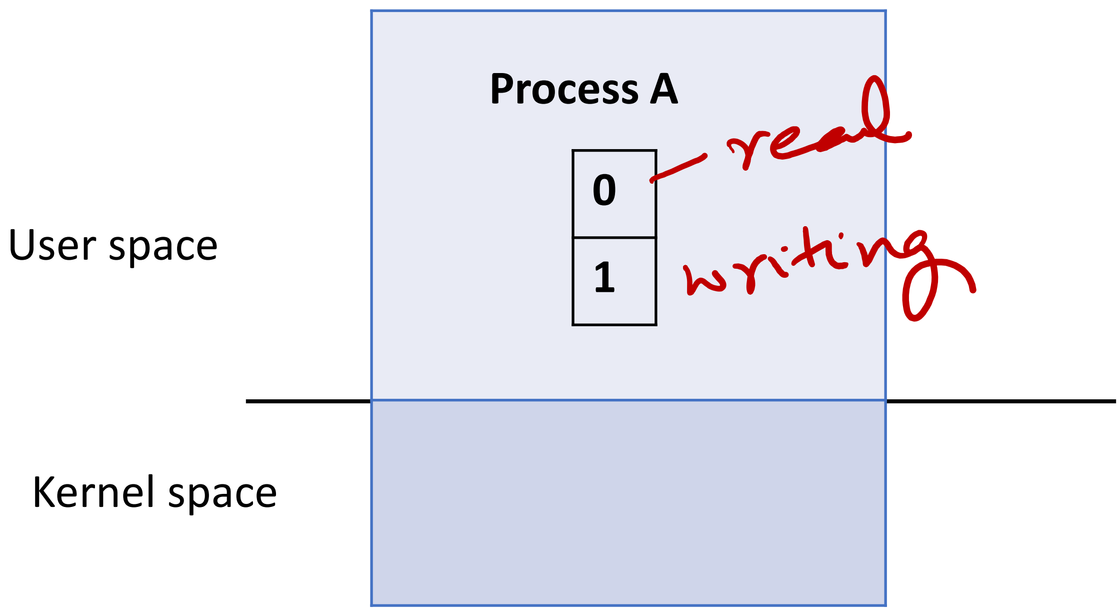
Example

- Example: who | sort
- Symbol ‘|’ is called ‘pipe’
- Related to every process having three default I/O channels:
  - stdin (standard input)
  - stdout (standard output)
  - stderr (standard error output)

Pipe()

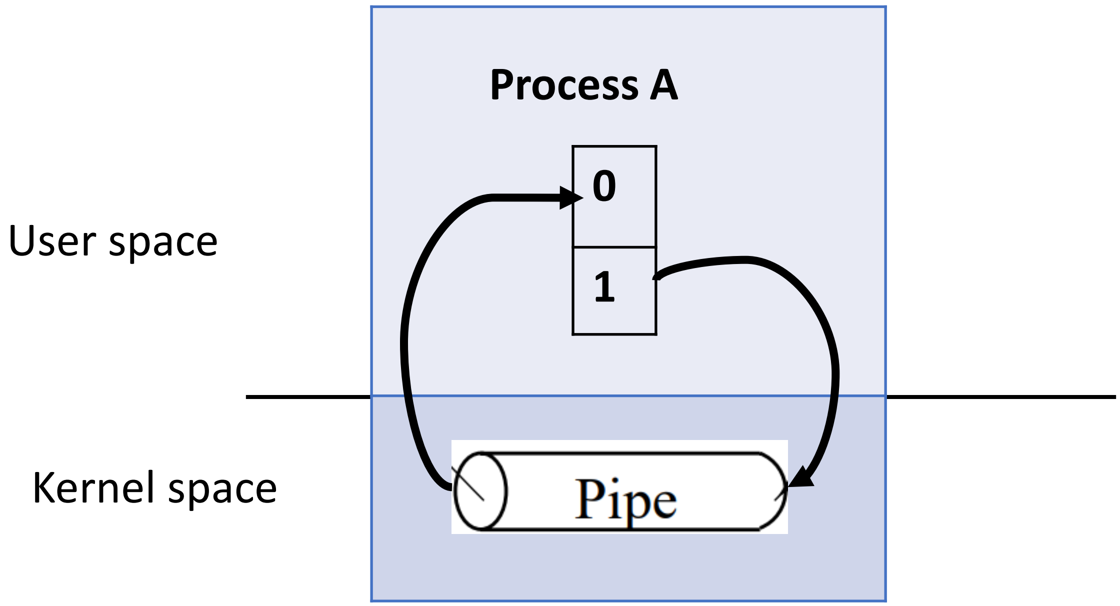
- A pipe is a pair of connected descriptors, normally declared as an array.

```
int fds[2];
```



Pipe()

```
pipe(fds);
```



Pipe()

- $ man 2 pipe

```
PIPE(2)                     Linux Programmer's Manual                    PIPE(2)

NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates  a pipe, a unidirectional data channel that can be used for interprocess communication.
       The array pipefd is used to return two file descriptors referring to the ends of the  pipe.   pipefd[0]
       refers  to  the  read end of the pipe.  pipefd[1] refers to the write end of the pipe.  Data written to
       the write end of the pipe is buffered by the kernel until it is read from the read  end  of  the  pipe.
       For further details, see pipe(7).

       If  flags  is 0, then pipe2() is the same as pipe().  The following values can be bitwise ORed in flags
       to obtain different behavior:

       O_CLOEXEC
              Set the close-on-exec (FD_CLOEXEC) flag on the two new file descriptors.  See the description of
              the same flag in open(2) for reasons why this may be useful.

       O_DIRECT (since Linux 3.4)
              Create  a pipe that performs I/O in "packet" mode.  Each write(2) to the pipe is dealt with as a
              separate packet, and read(2)s from the pipe will read one packet at a time.  Note the  following
              points:
```

Pipe()

Creates a pipe.

- Include(s): < unistd.h>
- Syntax: ***int pipe (int pipefd[2]);***
- Return: Success: 0; Failure: -1; Sets errno: Yes
  - What does it mean to return errno?
- 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.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    int fds[2];

    if(pipe(fds) == -1) {
      perror("pipe"); //prints a descriptive error message to stderr.
      exit(EXIT_FAILURE);
    }

    printf("Read File Desc value: %d\n", fds[0]);
    printf("Write File Desc value: %d\n", fds[1]);
    return 0;;
}
```

```
Read File Desc value: 3
Write File Desc value: 4
```

Write()

- To write ***n bytes (count)*** to the write end of a pipe.
  - If a write process attempts to ***write*** to a pipe, the default action is for the system to block the process until the data is able to be received.

- Include(s): <unistd.h>

- Syntax: **`ssize_t write(int`** *`fd`*`,` **`const void *`***`buf`*`,` **`size_t`** *`count`***`);`**
  - writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

- Returns
  - success: Number of bytes written; Failure; -1; Sets errno:Yes.

Read()

- To read ***n bytes (count)*** from the read end of a pipe.
  - if ***read*** is attempted on an empty pipe, the process will block until data is available.

- Includes: <unistd.h>

- Syntax: `ssize_t read(int fd, void *buf, size_t count);`
  - **read**() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

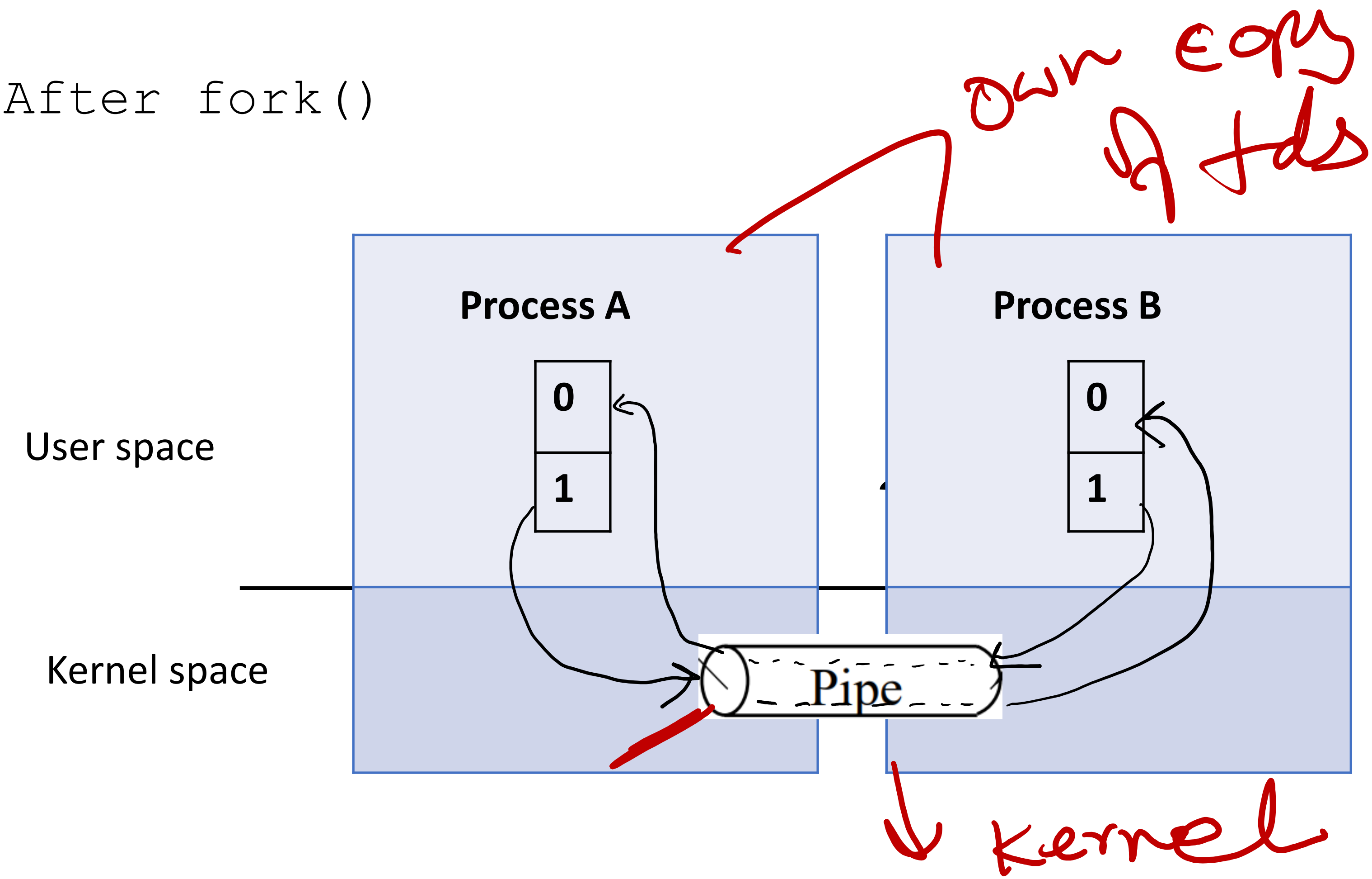
- Return
  - success: Number of bytes read;
  - Failure; -1; Sets errno:Yes.
  - EOF (0): write end of pipe closed

```c
#include<sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char* argv[])
{
    int fds[2];
    pid_t pid;
    char buff[30];
    if(pipe(fds) == -1){ // create pipe
        perror("Error: ");
        return(-1);
    }
    memset(buff,0,30);
    pid = fork();
    if (pid > 0) {
      printf(" Parent process using  pipe to write \n");
      close(fds[0]);//parent process close the read end
      write(fds[1], "Hello World!", 12); //parent process write in the pipe write end
     close(fds[1]); // parent process closes the write end after finishing writing
    wait(NULL); //parent process wait for the child process

    }else {
      //child process closes the write end
      close(fds[1]);
      //child process read from the pipe read end until the pipe is empty
      while(read(fds[0], buff, 1)==1)
        printf("Child process reading from pipe -- %s\n", buff);
        close(fds[0]); // Child process closes the read end after finishing reading
      printf("Child process: Exiting....!");
      exit(EXIT_SUCCESS);
    }
    return 0;
}
```

```
 Parent process using  pipe to write
Child process reading from pipe -- H
Child process reading from pipe -- e
Child process reading from pipe -- l
Child process reading from pipe -- l
Child process reading from pipe -- o
Child process reading from pipe --
Child process reading from pipe -- W
Child process reading from pipe -- o
Child process reading from pipe -- r
Child process reading from pipe -- l
Child process reading from pipe -- d
Child process reading from pipe -- !
Child process: Exiting....!
```

Pipe()

After fork()

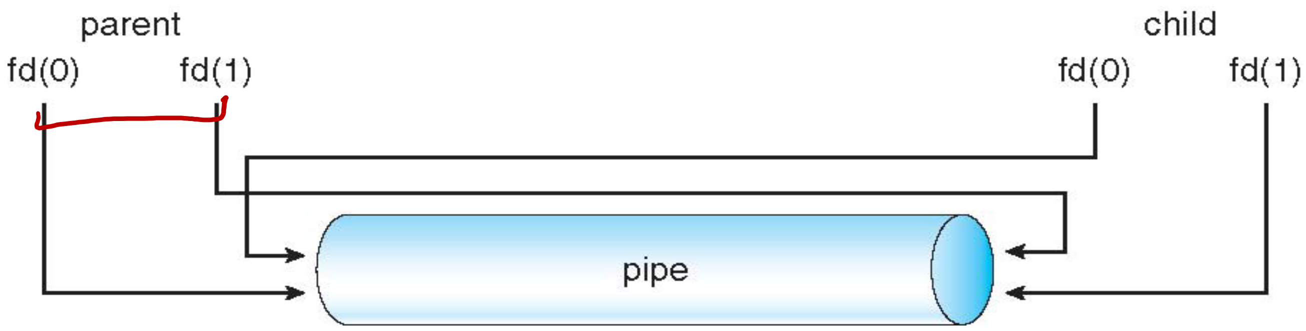


Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
  - Producer writes to one end (the *write-end* of the pipe)
  - Consumer reads from the other end (the *read-end* of the pipe)

- Ordinary pipes are therefore unidirectional

- Require parent-child relationship between communicating processes



Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Features of Pipes

- On many systems, pipes are limited to 10 logical blocks, each block has 512 bytes.
- As a general rule, one process will write to the pipe (as if it were a file), while another process will read from the pipe.
- Data is written to one end of the pipe and read from the other end.
- A pipe exists until both file descriptors are closed in all processes

Homework (no submission on canvas … just for practice)

Write a program

- Process A – has a secret code "50"
- Process A will pass secret code to child process B through pipe
- Child Process B has a secret code "10"
- Child Process will add the secret code and pass sum to the Process A
- Process A will display the sum
- Sample output as follows:

```
From parent...the sum is : 60
```

Try!!