LECTURE 15: IPC

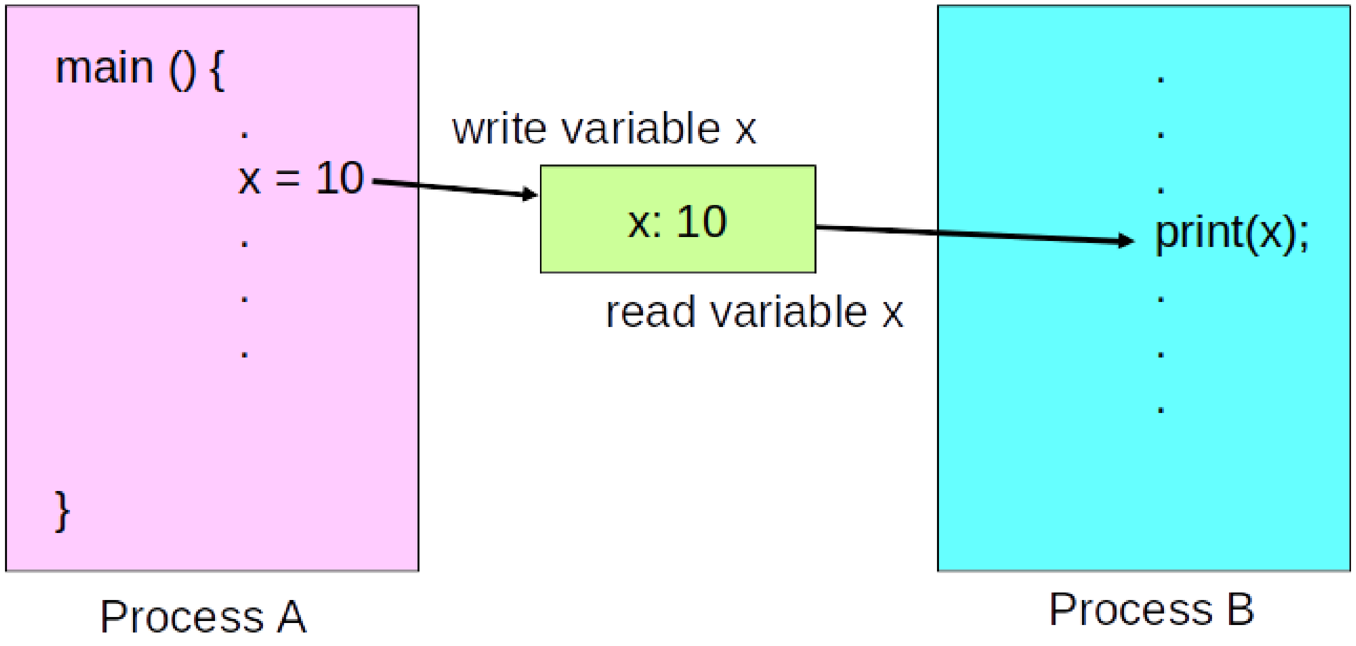
Inter Process Communication (IPC)

- Processes do not share any memory with each other
- Some processes might want to work together for a task, so need to communicate information
- IPC mechanisms to share information between processes

Shared Memory

- Processes can both access same region of memory via shmget()system call
- int shmget (key_t key, int size, int shmflg)
- By providing same key, two processes can get same segment of memory
- Can read/write to memory to communicate
- Need to take care that one is not overwriting other’s data: how?

Shared Memory IPC Diagram



Signals

- A certain set of signals supported by OS
  - Some signals have fixed meaning (e.g., signal to terminate process)
  - Some signals can be user-defined
- Signals can be sent to a process by OS or another process (e.g., if you type Ctrl+C, OS sends SIGINT signal to running process)
- Signal handler: every process has a default code to execute for each signal
  - Exit on terminate signal
  - Some signal handlers can be overridden to do other things

Sockets

- Sockets can be used for two processes on same machine or different machines to communicate
  - TCP/UDP sockets across machines
  - Unix sockets in local machine
- Communicating with sockets
  - Processes open sockets and connect them to each other
  - Messages written into one socket can be read from another
  - OS transfers data across socket buffers

Pipes

- Pipe system call returns two file descriptors
  - Read handle and write handle
  - A pipe is a half-duplex communication
  - Data written in one file descriptor can be read through another
- Regular pipes: both fd are in same process (how it is useful?)
  - Parent and child share fd after fork
  - Parent uses one end and child uses other end
- Named pipes: two endpoints of a pipe can be in different processes
- Pipe data buffered in OS buffers between write and read

Pipes Cont

#include <unistd.h>

int pipe(int fildes[2]);

fildes[0] is open for reading and

fildes[1] is open for writing

The output of fildes[1] is the input for fildes[0]

Understanding Pipes

- Within a process
  - Writes to fildes[1] can be read on fildes[0]
  - Not very useful
- Between processes
  - After a fork()
  - Writes to fildes[1] by one process can be read on fildes[0] by the other

Understanding Pipes (cont.)

- Even more useful: two pipes, fildes_a and fildes_b
- After a fork()
- Writes to fildes_a[1] by one process can be read on fildes_a[0] by the other, and
- Writes to fildes_b[1] by that process can be read on fildes_b[0] by the first process

Using Pipes

- Usually, the unused end of the pipe is closed by the process
- If process A is writing and process B is reading, then process A would close fildes[0] and process B would close fildes[1]
- Reading from a pipe whose write end has been closed returns 0 (end of file)
- Writing to a pipe whose read end has been closed generates SIGPIPE
- PIPE_BUF specifies kernel pipe buffer size

Example

```
int main(void) {
    int n, fd[2];
    pid_t pid;
    char line[maxline];

    if(pipe(fd) < 0)  err_sys(“pipe error”);
    if( (pid = fork()) < 0)  err_sys(“fork error”);
    else if(pid > 0) {
            close(fd[0]);
            write(fd[1], “hello\n”, 6);
    } else {
            close(fd[1]);
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
    }
```

Message Queues

- Mailbox abstraction
- Process can open a mailbox at a specified location
- Processes can send/receive messages from mailbox
- OS buffers messages between send and receive

Blocking vs. non-blocking communication

- Some IPC actions can block
  - Reading from socket/pipe that has no data, or reading from empty message queue
  - Writing to a full socket/pipe/message queue
- The system calls to read/write have versions that block or can return with an error code in case of failure
  - A socket read can return error indicating no data to be read, instead of blocking

Thank You