CSCI 3753
Operating Systems

Interprocess Communication

Lecture Notes By

Shivakant Mishra

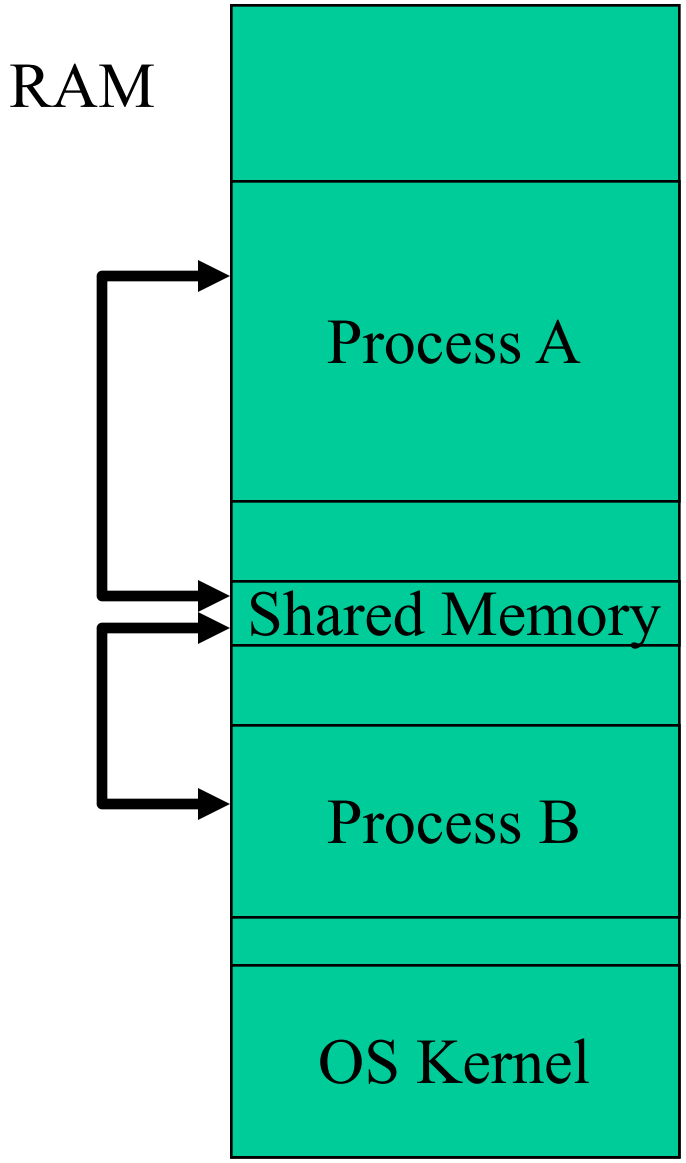
Computer Science, CU-Boulder

Last Update: 09/21/17

Communicating Between Processes

- Inter-Process Communication (IPC): Want to communicate between two processes because
  - An application may split its tasks into two or more processes for reasons of convenience and/or performance
    - e.g. Web server
  - Sharing information
  - Improved fault isolation
- Two types of IPC
  - Shared memory
  - Message passing

IPC Shared Memory



- OS provides mechanisms for creation of a shared memory buffer between processes
- applies to processes on the same machine
- Problem: shared access introduces complexity
  - need to synchronize access

IPC Shared Memory (Linux)

- shmid = *shmget* (key name, size, flags) is part of the POSIX API that creates a shared memory segment, using a name (key ID)
  - All processes sharing the memory need to agree on the key name in advance.
  - Creates a new shared memory segment if no such shared memory with the same name exists and returns handle to the shared memory. If it already exists, then just return the handle.

IPC Shared Memory (Linux)

- shm_ptr = *shmat* (shmid, NULL, 0) to attach a shared memory segment to a process's address space
  - This association is also called binding
  - Reads and writes now just use shm_ptr
- *shmctl*() to modify control information and permissions related to a shared memory segment, & to remove a shared memory segment

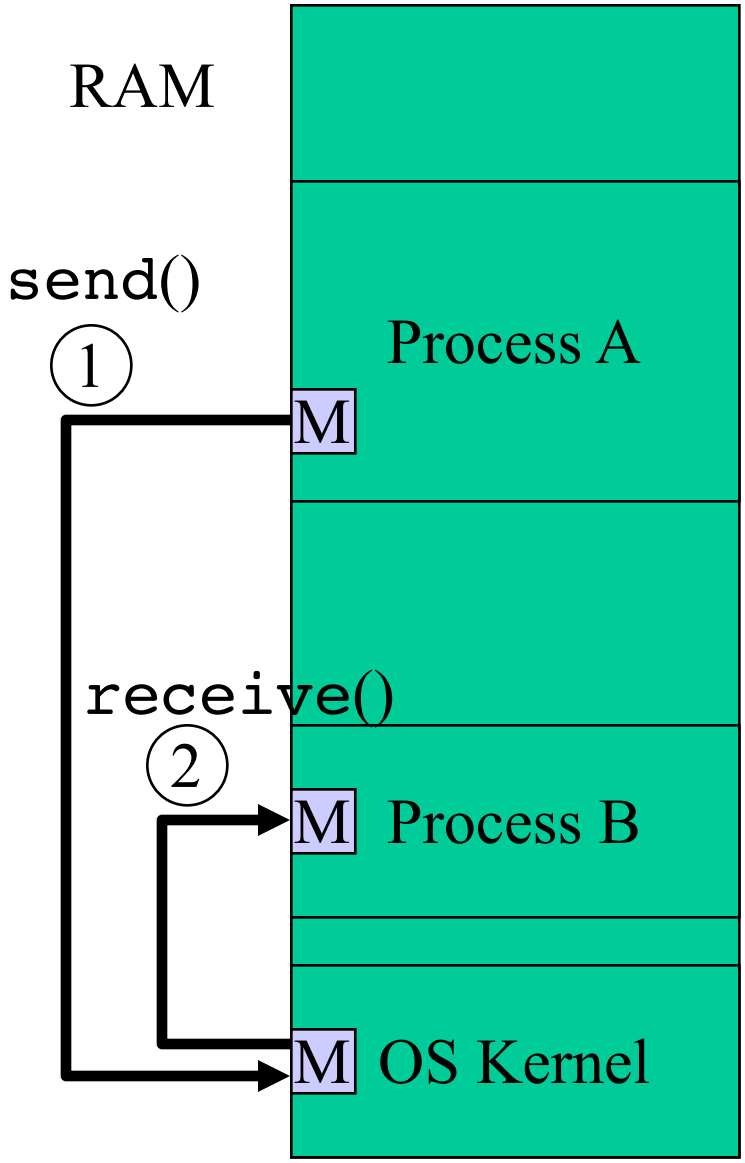
Details about Linux support for shared memory IPC will be covered in recitation

IPC Message Passing

OS provides mechanisms for communication via buffers

- Basic primitives are `send()` and `receive()`
- Typically implemented via system calls, and is hence slower than shared memory
- Sending process has a buffer to send/receive messages, as does the receiving process and OS
- In direct message-passing, processes send directly to each other's buffers
- In indirect message-passing, sending process sends a message to OS by calling send(). OS acts as a buffer or mailbox for relaying the message to the receiving process, which calls receive() to retrieve message

IPC Message Passing



Example of indirect message-passing

- `send()` and `receive()` can be blocking/synchronous or non-blocking/asynchronous
- used to pass small messages
- Advantage: doesn't require synchronization
- Disadvantage: Slow - OS is involved in each IPC operation for control signaling and possibly data as well
- Message Passing IPC types: pipes, UNIX-domain sockets, Internet domain sockets, message queues, and remote procedure calls (RPC)

IPC via Pipes

- Process 1 writes into one end of the pipe, & process 2 reads from other end of the pipe, e.g. "ls | more"
  - Form of IPC similar to message-passing but data is viewed as a stream of bytes rather than discrete messages
  - was one of UNIX's original forms of IPC
  - essentially FIFO buffers accessed like file I/O API, so standard read() and write() for files can be used
  - Asynchronous/non-blocking send() and blocking/synchronous receive()
- This is a one-way pipe
- This also called an *anonymous* pipe in Windows
  - Parent process uses `pipe()` system call to create pipe

IPC via Pipes (Linux)

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

- piped[0] is file descriptor of the read end of the pipe
- piped[1] is file descriptor of the write end of pipe
- Use read( ) and write( ) to communicate using using pipe

Details about Linux support for pipes will be covered in recitation

Named Pipes

- Traditional one-way or anonymous pipes only exist transiently between the two processes connected by the pipe
  - As soon as these processes complete, the pipe disappears
- Named pipes persist across processes
  - Operate as FIFO buffers or files, e.g. created using mkfifo(unique_pipe_name) on Unix
  - Different processes can attach to the named pipe to send and receive data
  - Need to explicitly remove the named pipe
  - *See textbook for more info on named pipes*