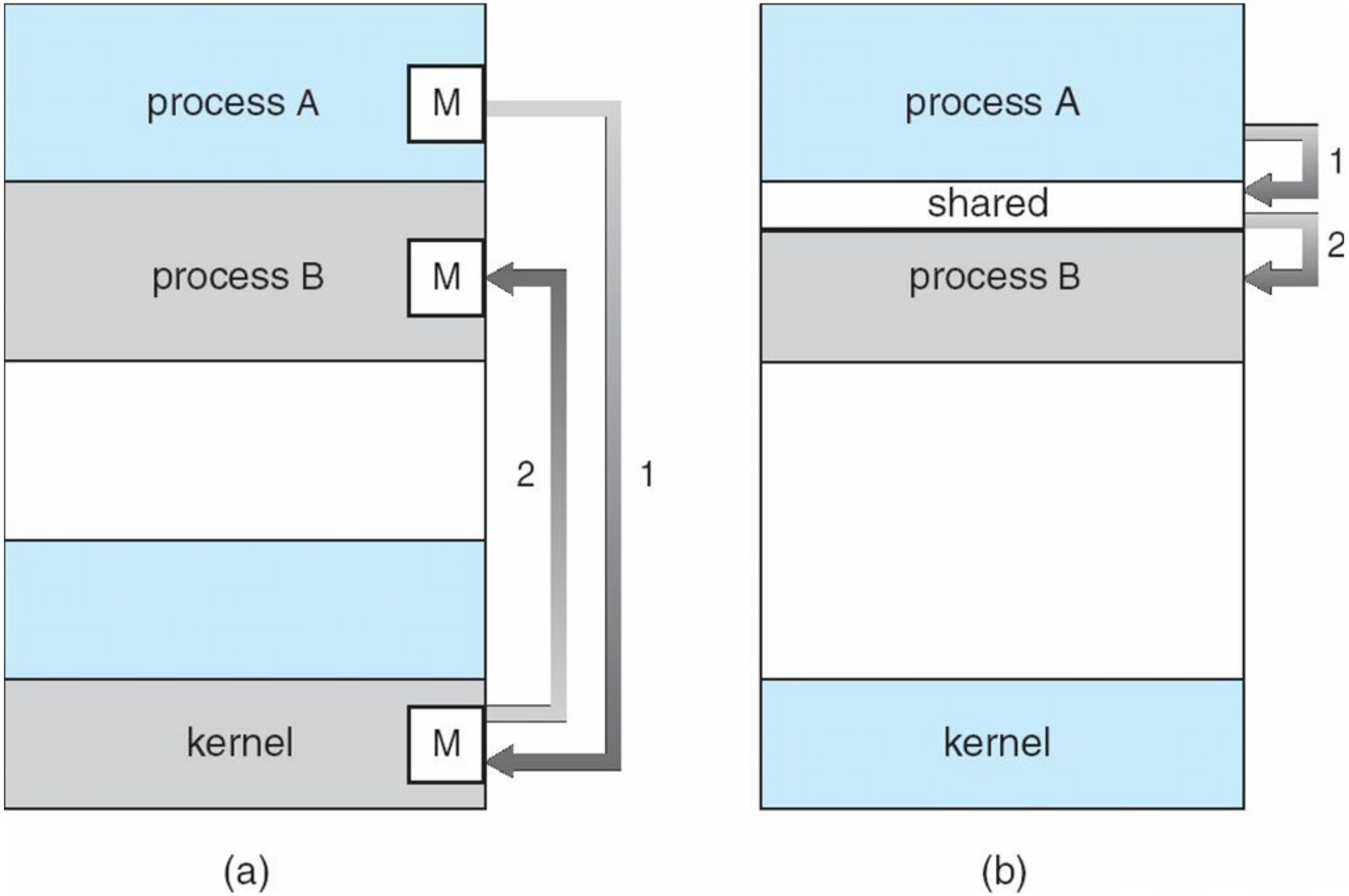
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models

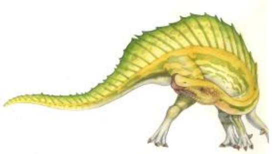


Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size



Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

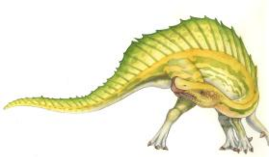
Bounded-Buffer – Producer

```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE count)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

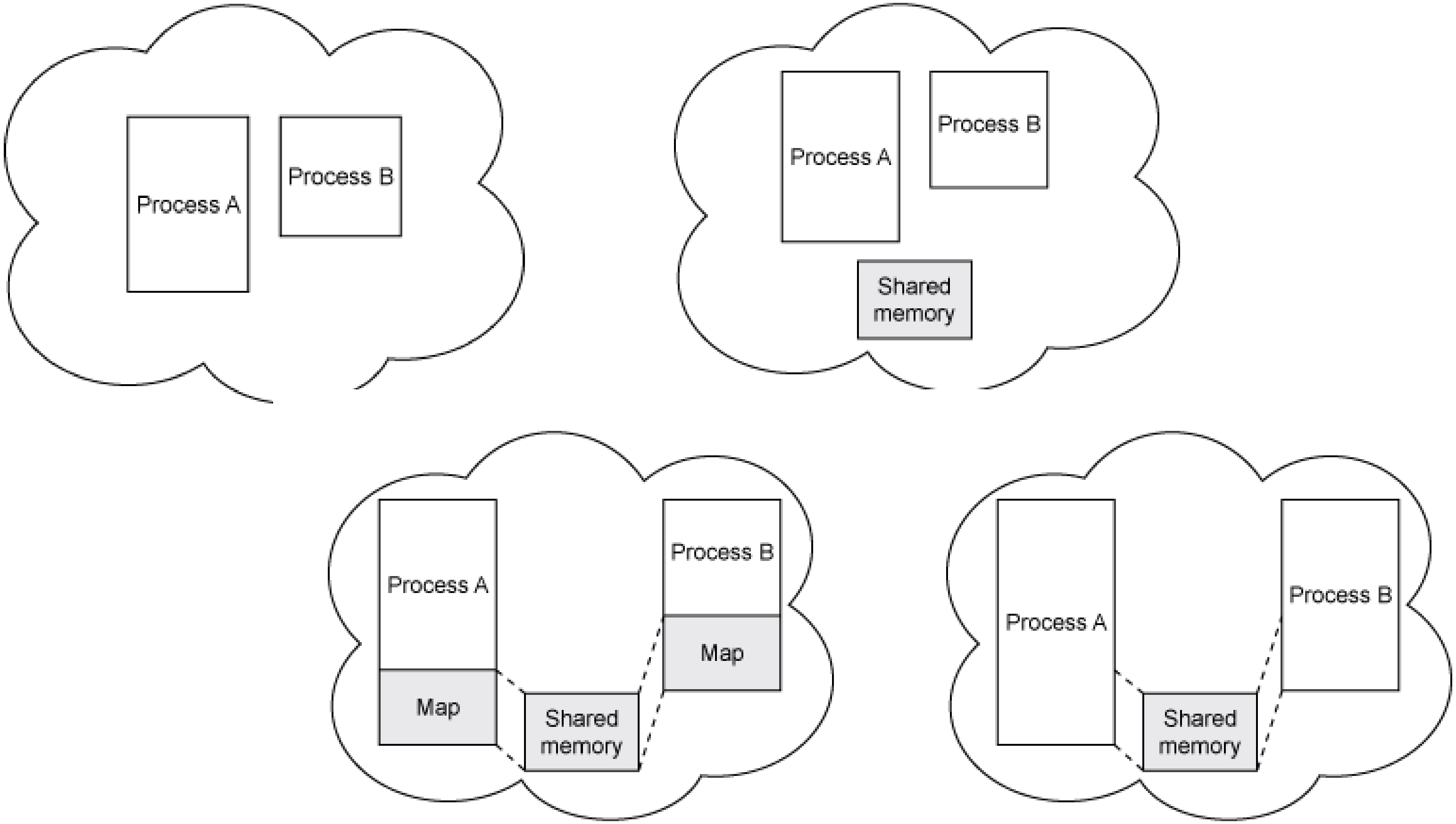
```
while (true) {
     while (in == out)
             ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
return item;
}
```



How shared/mapped memory works

- Two processes running on a host, executing different code
- One process requests a shared memory segment
- Both processes annex, or map, the shared memory segment
- Two or more processes can now share data via common memory



Shared/Mapped Memory in Linux

- Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S IRUSR | S
   IWUSR);
```

  - Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

  - When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

shmget parameters

- shmget
  - IPC_CREAT—a new segment should be created. This permits creating a new segment while specifying a key value.
  - IPC_EXCL—This flag, which is always used with IPC_CREAT, causes shmget to fail if a segment key is specified that already exists. If this flag is not given and the key of an existing segment is used, shmget returns the existing segment instead of creating a new one.
  - Mode flags—This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. For example, S_IRUSR and S_IWUSR specify read and write permissions for the owner.
- shmat
  - The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address.

Examples of IPC Systems - POSIX

- POSIX provides five entry points to create, map, synchronize, and undo shared memory segments:
  - **shm_open()**: Creates a shared memory region or attaches to an existing, named region. Returns a file descriptor.
  - **shm_unlink()**: Deletes a shared memory region given a file descriptor (returned from shm_open()). Once shm_unlink() is called (typically by the originating process), no other processes can access the region.
  - **mmap()**: Maps a shared memory region into the process's memory. This system call requires the file descriptor from shm_open() and returns a pointer to memory.
  - **munmap()**: The inverse of mmap().
  - **msync()**: Used to synchronize a shared memory segment with the file system—a technique useful when mapping a file into memory.