Inter-Process Communication

Inter-Process Communication Overview

1. IPC: OS-supported mechanisms for interaction among processes (coordination and communication)
    - Message passing (sockets, pipes, message queues)
    - Memory-based (shared memory, memory mapped files)
    - Higher-level semantics (files, RPC)
    - Synchronization primitives
2. Processes share memory
    - Data they both need is in shared memory
3. Process exchange messages
    - Message passing via sockets
4. Requires synchronization (mutexes, waiting…)

Message-Based IPC

1. Processes create messages and send/recv them
2. OS creates and maintains a channel (buffer, FIFO queue…)
3. OS provides interface to processes (a port)
    - Processes send/write messages to a port
    - Processes recv/read messages from a port
4. Kernel required to establish communication and perform each IPC operation
    - send: system call + data copy
    - recv: system call + data copy
    - Request/Response: 4 user/kernel crossings + 4 data copies
5. Pros:
    - Simplicity: kernel does channel management and synchronization
6. Cons:
    - Overhead

Forms of Message Passing

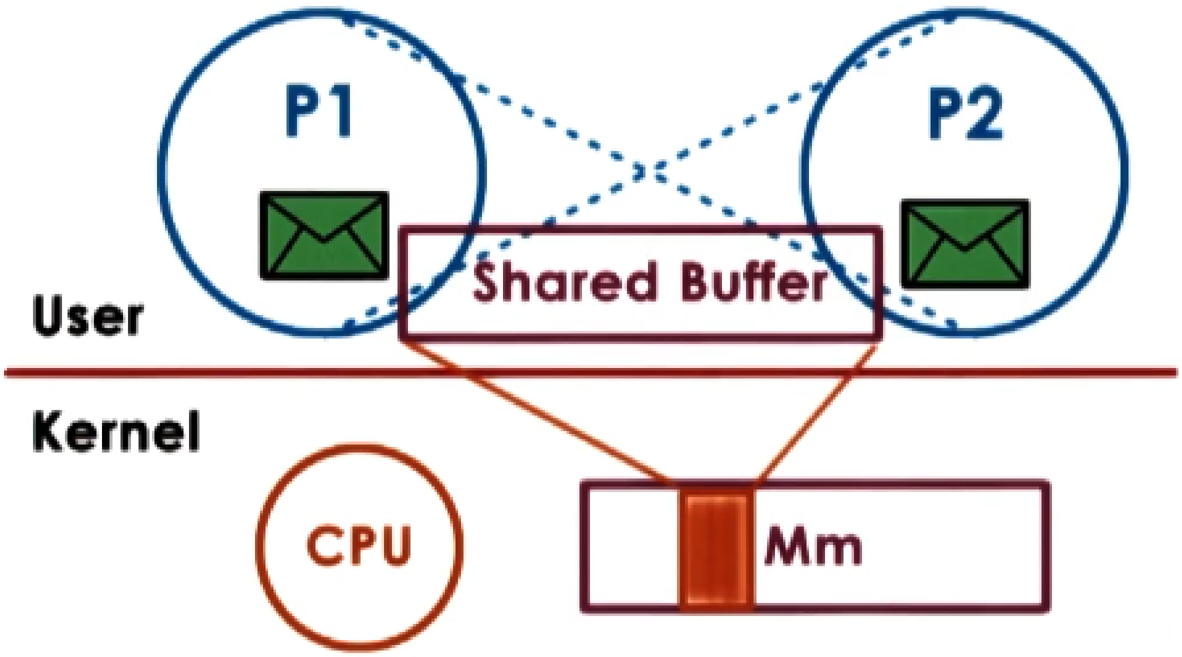
1. Pipes: Carry byte stream between 2 processes
    - Connect output from one process to input of another
2. Message queues: Carry "messages" among processes
    - OS managament includes priorities, scheduling of message delivery…
    - APIs: SysV and POSIX
3. Sockets: Ports are the socket abstraction supported by the OS
    - send(), recv() == pass message buffers
    - socket() == create kernel-level socket buffer
    - Associate necessary kernel-level processing (TCP/IP, …)
    - If different machines, channel between process and network device
    - If same machine, bypass full protocol stack

Shared Memory IPC

1. Processes read and write to shared memory region
    - OS establishes shared channel between the processes
        1. Physical pages mapped into virtual address space
        2. VA for processes 1 and 2 map to the same physical address
        3. VA(P1) != VA(P2)
        4. Physical memory doesn't need to be continuous
    - Pros:
        - System calls only for setup

- Data copies potentially reduced (but not eliminated)
- Cons:
  - Explicit synchronization
  - Communication protocol, shared buffer management are the programmer's responsibility

2. APIs:
   - SysV API
   - POSIX API
   - Memory mapped files
   - Android ashmem



Shared Memory IPC

Copy vs Map

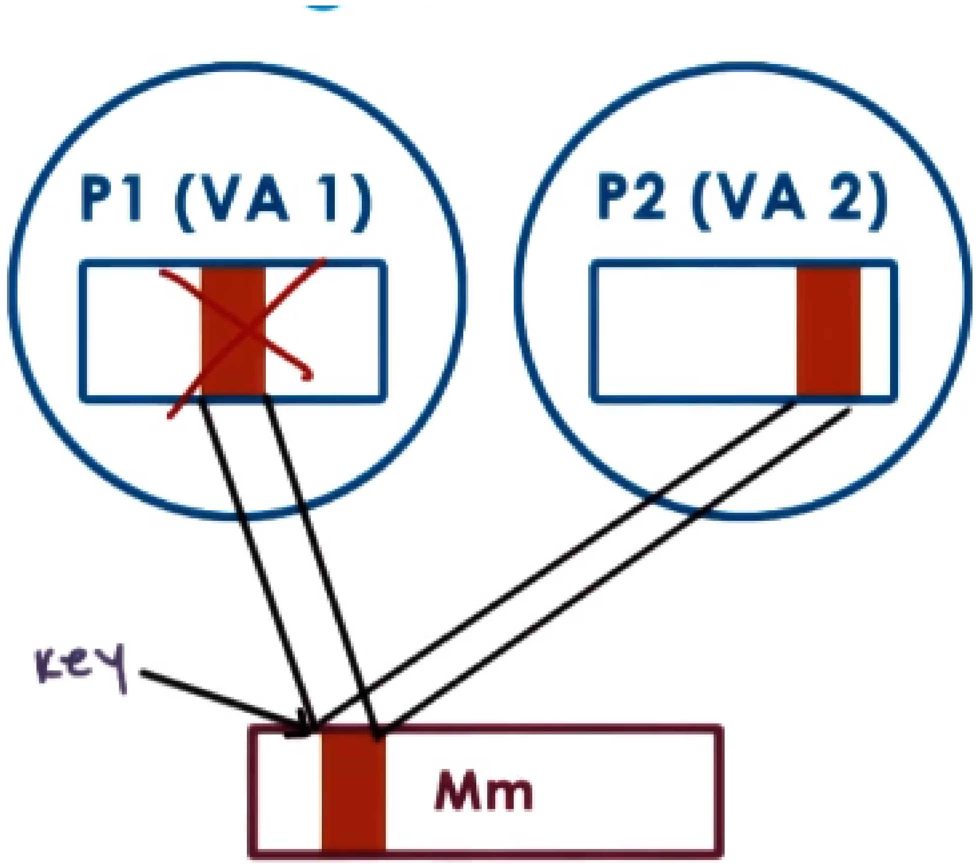
1. Goal: Transfer dat from one address space into the target address space
   - Copy (Messages)
     - CPU cycles to copy data to/from port
     - For large data t(copy) » t(map)
   - Map (Shared Memory)
     - CPU cycles to map memory into address space
     - CPU used to copy data to channel
     - Set up once to use many times -> good payoff; can still perform well for one-time use
2. Local Procedure Calls (LPC): Windows kernel dynamically decides whether to use copy or map depending on the size of the data for optimal efficiency

SysV Shared Memory

1. "Segments" of shared memory -> not necessrily contiguous physical pages
2. Shared memory is system-wide -> system limits on number of segments and total size
3. Process:
   - Create: OS assigns unique key

- Attach: Map virtual -> physical addresses
- Detach: Invalidate address mappings
- Destroy: Only remove data when explicitly deleted (or reboot)

4. API:
   - shmget(shmid,size,flag) to create or open ftok(pathname,proj_id) Same args return the same key
   - shmat(shmid,addr,flags) addr = NULL -> arbitrary (cast addr to appropriate type)
   - shmdt(shmid) virtual to physical memory mappings are no longer valid
   - shmctl(shmid,cmd,buf) destroy with IPC_RMID



Shared Memory SysV

POSIX Shared Memory API

1. Uses files instead of segments and file descriptors instead of keys
   - Not part of actual file system, only exist in tmpfs
2. API:
   - shm_open() returns file descriptor in "tmpfs"
   - mmap() and unmmap() mapping virtual -> physical addresses
   - shm_close()
   - shm_unlink()

Shared Memory and Synchronization

1. Similar to threads accessing shared state in a single address space, but for processes
2. Methods:

    - Mechanisms supported by process threading library (pthreads)
    - OS-supported IPC for synchronization
3. Either method must coordinate…
    - Number of concurrent accesses to shared segment
    - When data is available and ready for consumption (condition variables)

PThreads Synchronization for IPC

1. pthread_mutexattr_t and pthread_condattr_t describe whether the object is private to a process or shared among processes
    - PTHREAD_PROCESS_SHARED keyword
    - Synchronization variables must be shared (global to all processes)

```
// ...make shm data struct
typedef struct {
    pthread_mutex_t mutex;
    char *data;
} shm_data_struct, *shm_data_struct_t;

// ...create shm segment
seg = shmget(ftok(arg[0], 120), 1024, IPC_CREATE|IPC_EXCL));
shm_address = shmat(seg, (void *) 0, 0);
shm_ptr = (shm_data_struct_t_)shm_address;

// ...create and init mutex
pthread_mutexattr_t(&m_attr);
pthread_mutexattr_set_pshared(&m_attr,PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&shm_prt.mutex, &m_attr);
```

PThreads Synchronization for IPC

Alternative IPC Synchronization

1. Message queues: Implement "mutual exclusion" via send/recv
    - Example protocol:
        - P1 writes data to shmem, sends "ready" to queue
        - P2 receives message, read data, and sends "ok" message back
2. Semaphores: Binary semaphore <-> mutex
    - If value == 0 -> stop/blocked
    - If value == 1 -> decrement (lock) and proceed

IPC Command Line Tools

1. ipcs: list all IPC facilities
    - -m displays info on shared memory IPC only
2. ipcrm: delete IPC facility
    - -m [shmid] deletes shm segment with given id

Shared Memory Design Considerations

1. Different APIs/mechanisms for synchronization

2. OS provides shared memory and gets out of the way
3. Data passing and synchronization protocols are up to the programmer
4. How many segments will two processes need to communicate?
    - 1 large segment -> manager for allocating/freeing memory from shared segments
    - Many small segments -> Use pool of segments, queue segment ids
        - One for each pairwise communication
        - Communicate segment IDs among processes
5. How large should a segment be?
    - Segment size == Data size works for well-known static sizes
        - Limits max data size
    - Segment size < message size
        - Transfer data in rounds; include protocol to track progress