Karlsruhe Institute of Technology

Operating Systems

11. Inter Process Communication

Prof. Dr. Frank Bellosa | WT 2020/2021

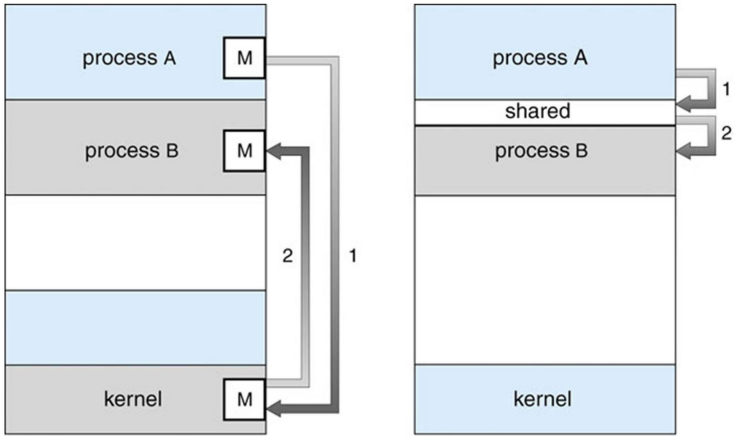
Interprocess Communication (IPC)

- Processes/Threads frequently need to communicate with one another
- Reasons for cooperating processes
  - **Information sharing**
    share file/data-structure in memory
  - **Computation speed-up**
    break larger task into subtasks → executed in parallel
  - **Modularity**
    divide system into collaborating modules with clean interfaces
- Interprocess Communication (IPC) allows exchanging data
  - Message passing explicitly send and receive information using system calls
    - e.g.,pipes, sockets, POSIX message queues
  - Shared memory establishes a physical memory region that multiple processes/threads can access
    - e.g., POSIX shared memory, shared memory-mapped files

Interprocess Communication (IPC)

- Processes/Threads frequently need to communicate with one another
- Reasons for cooperating processes
  - **Information sharing**
    share file/data-structure in memory
  - **Computation speed-up**
    break larger task into subtasks → executed in parallel
  - **Modularity**
    divide system into collaborating modules with clean interfaces
- Interprocess Communication (IPC) allows exchanging data
  - Message passing explicitly send and receive information using system calls
    - e.g.,pipes, sockets, POSIX message queues
  - Shared memory establishes a physical memory region that multiple processes/threads can access
    - e.g., POSIX shared memory, shared memory-mapped files

Message Passing vs. Shared Memory[SGG12]



Message Passing

Shared Memory

Interprocess Communication – Message Passing

- Mechanism for processes to communicate/synchronize their actions
- Message passing facilities generally provide operations to
  - **`send`**
  - **`receive`**
- Implementation of communication link
  - hardware interconnect (bus, point-to-point)
  - network interface card (NIC) e.g., Ethernet, InfiniBand
  - kernel memory
  - shared memory

Direct vs. Indirect Messages

- Processes name each other explicitly when exchanging direct messages
  - **`send`**(P, message) – send a message to process P
  - **`receive`**(Q, message) – receive a message from process Q
- Indirect messages can be sent to and received from mailboxes
  - Each mailbox has a unique id
  - First communicating process creates mailbox, last destroys mailbox
  - Processes can communicate only if they share a mailbox
- Mailbox sharing
  - P1, P2, and P3 share mailbox A → P1, **`send`**s; P2 and P3 **`receive`**
  - Who gets the message?
  - Allow a link to be associated with at most two processes?
  - Allow only one process at a time to execute a receive operation?
  - Allow the system to arbitrarily select the receiver? (maybe the sender is notified who received the message)

Direct vs. Indirect Messages

- Processes name each other explicitly when exchanging direct messages
  - **`send`**(P, message) – send a message to process P
  - **`receive`**(Q, message) – receive a message from process Q
- Indirect messages can be sent to and received from mailboxes
  - Each mailbox has a unique id
  - First communicating process creates mailbox, last destroys mailbox
  - Processes can communicate only if they share a mailbox
- Mailbox sharing
  - P1, P2, and P3 share mailbox A → P1, **`send`**s; P2 and P3 **`receive`**
  - Who gets the message?
  - Allow a link to be associated with at most two processes?
  - Allow only one process at a time to execute a receive operation?
  - Allow the system to arbitrarily select the receiver?
    (maybe the sender is notified who received the message)

Sender/Receiver Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is copied.
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

Buffering

Messages are queued using different capacities while they are in-flight

- Zero capacity – 0 messages/no queuing
  - Sender must wait for receiver (rendezvous)
  - message is transferred as soon as receiver becomes available
    ⟶ no latency/no jitter
- Bounded capacity – finite number and length of messages
  - Sender can send before receiver waits for messages
  - Sender can send while receiver still processes previous message
    ⟶ Variable latency between send and receive
  - Sender must wait if link full (see UNIX “pipe” and “named pipe”)
- Unbounded capacity
  - Sender never waits
  - Memory may overflow

Example: Message Boxes in Mach Microkernel(e.g. in Mac OS X)

- All communication is message based (even system calls are messages)
- Every task gets two initial mailboxes (ports) at creation:
  Kernel and Notify
- Further mailboxes can be allocated for process-to-process communication via
  `port_allocate()`
- Maximal buffer capacity is 65536 messages
  `port_status()` reports the number of messages in a mailbox
- Three system calls for messaging: `msg_send`, `msg_receive`, `msg_rpc`
- Flexible synchronization options: blocking, time-out, non-blocking
- Every port is owned by a single process which is allowed to receive messages
  (privilege can be transferred)
- Mailbox-Set allows to receive from multiple mailboxes

Example: POSIX Message Queues

- Create or open an existing message queue

```
mqd_t mq_open( const char *name, int oflag );
```

  - `name` is a path in the file system
  - Access permission is controlled through file system access permission

- Send a message to the message queue

```
int mq_send( mqd_t md, const char *msg, size_t len,
             unsigned priority );
```

- Receive the message with the highest priority in the message queue

```
int mq_receive( mqd_t md, char *msg, size_t len,
                unsigned *priority );
```

- Register callback handler on message queue to avoid polling

```
int mq_notify( mqd_t md, const struct sigevent *sevp );
```

- Remove message queue

```
int mq_unlink ( const char *name );
```

IPC through Shared Memory

- Communicate through a region of shared memory
  - Between processes: Create shared region in one address space
    → share with other processes
  - Threads “naturally” share address space
  - Every write (store operation) to that memory region is now visible to all other processes
  - Hardware guarantees that readers always read the most recent write
- Communication semantics via shared memory are application specific
  - Can also implement message passing via shared memory region
- Using shared memory in a safe way and with high performance is tricky
  - Especially if many processes and many CPUs are involved
    → due to cache coherency protocol
  - Especially if there are multiple writers
    → due to race conditions

Example: POSIX Shared Memory

- Open or create a new POSIX shared memory object (returns handle)

```
int shm_open( const char *name, int oflag, mode_t mode );
```

- Set size of shared memory region

```
ftruncate( smd, size_t len );
```

- Map shared memory object to address space

```
void* mmap( void* addr, size_t len, [...], smd, [...] );
```

- Unmap shared memory object from address space

```
int munmap( void* addr, size_t len );
```

- Destroy shared memory object

```
int shm_unlink( const char *name );
```

Sequential Memory Consistency

- When communicating via shared memory we tend to assume sequential consistency
- Sequential consistency (SC) “The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program.” (Lamport)
- Boils down to
  - All memory operations occur one at a time in program order
  - Ensures write atomicity
- In reality, the compiler and the CPU re-order instructions to execution order for more efficient execution

Memory Consistency Model

- Modern CPUs are generally **not** sequentially consistent because it would:
  - Complicate write buffers
  - Complicate non-blocking reads (speculative prefetch)
  - Make cache coherence more expensive
- Compilers also do not generate code in program order
  - Re-arrange loops for better performance
  - Common subexpression elimination
  - Software pipelining
- As long as a single thread accesses a memory location at a time this is not a problem

Accessing the same memory location concurrently (with multiple threads) requires proper synchronization

x86 Memory Consistency

- x86 supports multiple consistency and caching models [int20]
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory
  - Page Attribute Table (PAT) allows control on 4K page granularity
- Caching model and memory consistency are tied together tightly
  - e.g., certain store instructions such as **`movnt`** bypass the cache and can be re-ordered with other writes that do go through the cache
- **`lock`** prefix make memory instructions atomic
  - All lock instructions are totally ordered
  - Other instructions cannot be re-ordered with locked ones
- **`xchg`** instruction is always locked (although it doesn't have the prefix)
- Special fence instructions prevent re-ordering
  - **`lfence`** can't be re-ordered with reads
  - **`sfence`** can't be re-ordered with writes
  - **`mfence`** can't be re-ordered with reads or writes

References I

[int20] *Four-Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals*, 2020. retrieved from `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#three-volume`.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.