Karlsruhe Institute of Technology

Betriebssysteme

07. Inter Process Communication

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP

Where we ended last lecture

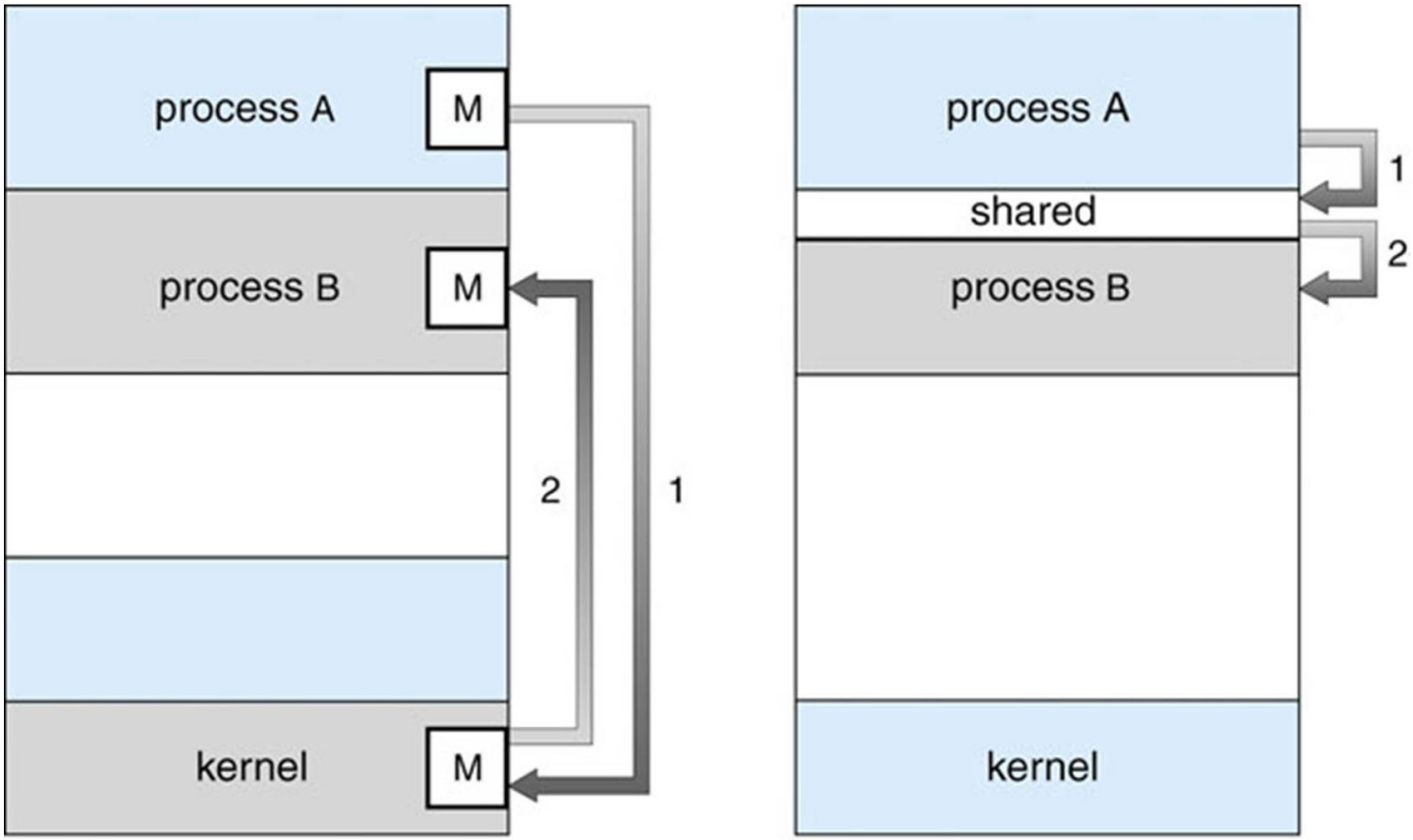
- Programs define data and algorithms
  - Processes encapsulate the resources that program instances need to run
  - Threads represent the execution state of program instances
- We use processes and threads to model concurrency and parallelism
  - Processes and KLTs can run in parallel on separate cores
  - Processes, KLTs, and ULTs can run concurrently on the same core
- Collaboration
  - Threads: Implicitly shared memory (same AS)
  - Processes: explicitly shared memory (map AS regions to same pmem)
- Today: How do processes communicate exactly, what problems arise, and how to overcome them

Inter Process Communication

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., POSIX message passing, pipes, sockets
  - 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



Message Passing Shared Memory

- We will discuss the implementation of shared memory later in this lecture, for now just assume that it is possible.

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 bus
  - shared memory
  - kernel memory
  - network interface card (NIC)

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? (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 received
  - 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
- Can non-blocking sender communicate with non-blocking receiver?
  → depends on buffering scheme

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
  - Sender must wait if link full (see UNIX “pipe” and “named pipe”)
- Unbounded capacity
  - Sender never waits
  - Memory may overflow
    → potentially very large latency/jitter between send and receive

Example: Message Boxes in Mach (e.g., 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
- Three system calls for messaging: **`msg_send`**, **`msg_receive`**, **`msg_rpc`**
- Further mailboxes can be allocated for process-to-process communication via **`port_allocate()`**
- Flexible synchronization options: blocking, time-out, non-blocking
- Maximal buffer capacity is 65536 messages
- 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
  `port_status()` reports the number of messages in a mailbox

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 to that memory region is now visible to all other processes
  - Hardware gurarantees that readers always read the most recent write

- Communication semantics via shared memory are application specific
  - Can 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 addres 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
- Without SC, multiple processes on multiple cores behave “worse” than preemptive threads on a single core
  - May see different results than when interleaving on one core

Memory Consistency Model

- Unfortunately life is hard and 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

Don't try to access the same memory location with multiple threads at the same time without proper synchronization!

x86 Memory Consistency

- x86 suports multiple consistency and caching models
  - 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

Synchronization

Race Conditions

- For now, let's assume that we have sequential memory consistency
  - We still don't have atomic memory transactions → we're not done yet
- Assume the following two code fragments are executed by two threads

Thread 1

```
count++;
```

Thread 2

```
count--;
```

- After both threads finish, **`count`** should still have the same value as before, right? What can possibly go wrong?

Thread 1 instructions

```
mov count A
add A 1
mov A count
```

Thread 2 instructions

```
mov count A
sub A 1
mov A count
```

Race Conditions

Thread 1 instructions

```
mov count A
add A 1
mov A count
```

Thread 2 instructions

```
mov count A
sub A 1
mov A count
```

Possible execution order (assume we initialized **count** to **0**)

```
mov count A    ; count = 0
sub A 1        ; decrement register, count still 0
mov count A    ; count = 0
add A 1        ; increment register, count still 0
mov A count    ; write -1 back to count
mov A count    ; write 1 back to count
```

- Both threads have private registers
  - → separate **A** register exists for every thread (no problem here)
- However: **count** is now **1** instead of the expected **0**
  - We call this problem data race or race condition

What about single-instruction add/subtract?

- x86 allows single instruction **`add count 1`**
  - Are we safe now? No! Same race condition
- Only interlocked operations will save the day!
  - But only if there is a single interlocked operation for your problem
  - Moreover they are more expensive than regular operations
    → Your compiler will not generate them if you write **`count++`**
- We need a general solution for the critical section (CS) problem!
  - Place **`count++`** and **`count--`** inside of a critical section **`CS`**
  - Protect critical section from concurrent execution

Thread 1

```
enter_critical_section( &CS );
count++;
leave_critical_section( &CS );
```

Thread 2

```
enter_critical_section( &CS );
count--;
leave_critical_section( &CS );
```

Desired Properties for Solution to Critical-Section Problem

- Mutual Exclusion

  At most one thread can be in the CS at any time

- Progress

  No thread running outside of the CS may block another thread from getting in
  - If no thread is in the CS, a thread trying to enter will eventually get in
  - If no thread can enter CS → don't have progress

- Bounded Waiting

  Once a thread starts trying to enter the CS, there is a bound on the number of times other threads get in
  - You cannot make assumptions concerning relative speeds of threads
  - Don't have bounded waiting if thread A waits to enter CS while B repeatedly leaves and re-enters CS infinitely

Disabling Interrupts

- The kernel only switches threads on interrupts
  - Usually on the timer interrupt
- Have per-thread “do not interrupt” (DNI) bit
- On a **single-core system** we can just implement
  - `enter_critical_section()` sets DNI bit
  - `leave_critical_section()` clears DNI bit
  - With interrupts disabled, the scheduler is never called
    → the thread runs until it reaches `leave_critical_section()`

\+ Easy and convenient in the kernel

– Only works in single-core systems: Disabling interrupts on one CPU does not affect other CPUs

– Only feasible in kernel: Don’t want to give user the power to turn off interrupts. What if he never turns them on again?

Lock Variables

- Lets define a global variable **`lock`**
  - Only enter CS if **`lock`** is 0 and then set it to 1 when entering
  - Wait for lock to become 0 otherwise (busy waiting)

```
void enter_critical_section( volatile bool *lock )
{
   while( *lock != 0 )
      ; // wait for lock to become 0

   *lock = 1;
}

void leave_critical_section( volatile bool *lock )
{
   *lock = 0;
}
```

- CS problem solved?

Lock Variables

- Lets define a global variable **`lock`**
  - Only enter CS if **`lock`** is 0 and then set it to 1 when entering
  - Wait for lock to become 0 otherwise (busy waiting)

```
void enter_critical_section( volatile bool *lock )
{
   while( *lock != 0 )
      ; // wait for lock to become 0

   *lock = 1;
}

void leave_critical_section( volatile bool *lock )
{
   *lock = 0;
}
```

- CS problem solved?
- NO! Same problem! Reading lock and setting lock to 1 not atomic!

Atomic Operations

- To make the lock variables approach work we need a way to **test and set** the lock variable at the same time (atomically)
- x86: **`XCHG`** can atomically exchange memory content with a register. Lets assume this C interface for xchg:
  - **`bool xchg( volatile bool *lock, register bool A );`**
  - Exchanges register content with memory content
  - Returns previous memory content of lock
- Now we can implement our critical section as a spinlock:

```
void enter_critical_section( volatile bool *lock )
{
   while( xchg(lock, 1) == 1 )  // lock = 1 and return old value
      ;                         // repeat until old value is not 1
}

void leave_critical_section( volatile bool *lock )
{
   *lock = 0;
}
```

Spinlocks with Atomic Operations

- Most modern CPUs provide atomic instructions with such semantics
  - Test memory word And Set value (TAS) (e.g., LDSTUB on SPARC V9)
  - Fetch and Add (e.g., XADD on x86)
  - Exchange contents of two memory words (SWAP, XCHG)
  - Compare content of one memory word and set to new value
    - Compare and Swap (e.g., CAS on SPARC V9 and Motorola 68k)
    - Compare and Exchange (e.g., CMPXCHG on x86)
  - Load-Link/Store-Conditional (LL/SC) (e.g., ARM, PowerPC, MIPS)

- CS Problem solved?

Spinlocks with Atomic Operations

- Most modern CPUs provide atomic instructions with such semantics
  - Test memory word And Set value (TAS) (e.g., LDSTUB on SPARC V9)
  - Fetch and Add (e.g., XADD on x86)
  - Exchange contents of two memory words (SWAP, XCHG)
  - Compare content of one memory word and set to new value
    - Compare and Swap (e.g., CAS on SPARC V9 and Motorola 68k)
    - Compare and Exchange (e.g., CMPXCHG on x86)
  - Load-Link/Store-Conditional (LL/SC) (e.g., ARM, PowerPC, MIPS)

- CS Problem solved?

✓ **Mutual Exclusion** Only one thread can enter CS

✓ **Progress** Only the thread within the CS hinders others to get in

✗ **Bounded Waiting** No upper bound

Spinlock Limitations

- Spinlocks don't work well if the lock is congested
  - If most of the time there is no thread in the CS when another tries to enter then spinlocks are very easy and efficient
  - If the CS is large (always keep it small!) or many threads try to enter, spinlocks might not be a good choice as all threads actively wait spinning
- Spinlocks don't work well if threads on different cores use the lock
  - The memory address is written at every atomic swap operation
    → memory is kept coherent between cores which is expensive
    (see tutorials for a MESI recapitulation and how to improve this issue)
- Spinlocks can behave unexpectedly when processes are scheduled with static priorities such as priority inversion
  - Assume two threads share a lock and are scheduled with static priorities
  - If the low-priority thread holds the lock, it will never be able to release it, because it will never be scheduled!

→ Nevertheless, spinlocks are widely used, especially in kernels

Sleeping While Waiting

- We have identified the busy part of busy waiting as an important spinlock limitation
- Busy waiting. . .
  - wastes resources when threads wait for locks
  - stresses the cache coherence protocol when used across cores
  - can cause the priority inversion problem
- Idea for improvement
  - Threads should sleep on locks if they are occupied
  - Wake up threads one at a time when lock becomes free

Semaphore

- Introduce two syscalls that operate on integer variables that we call semaphore in this context
  - **`wait( &s )`**: if **`s`** $>$ 0: **`s--`** and continue. Otherwise let caller sleep.
  - **`signal( &s )`**: if no thread is waiting: **`s++`**. Otherwise wake one up.
- Initialize **`s`** to the maximum number of threads that may enter the CS at any given time
  - **`wait`** corresponds to **`enter_critical_section()`**
  - **`signal`** corresponds to **`leave_critical_section()`**
- A semaphore that is initialized to 1 is called binary semaphore, mutex semaphore or just mutex
  - A mutex only admits one thread into the CS at a time
- If you want to be specific about your semaphore allowing more than one thread in the CS, you can call it counting semaphore

Semaphore Implementation Considerations

- The **`wait`** and **`signal`** calls need to be carefully synchronized
- Otherwise using semaphores could result in a race condition between checking and decrementing **`s`**
- Moreover, signal loss can occur when waiting and waking threads up at the same time
  - Consider a thread T1 checking **`s`**, which is 0
  - Before the thread goes to sleep, another thread T2 within the CS finishes
  - T2's **`signal`** does not wake up any threads, as no thread is sleeping
  - After T2's **`signal`** call finishes, T1 continues and begins sleeping

  → One thread less than expected can now enter the CS

  → If no other thread comes along, T1 will sleep forever

Semaphore Implementation Considerations

- Each semaphore is associated with a wake-up queue
  - Weak semaphores Wake up a random waiting thread on **`signal`**
  - Strong semaphores Wake up thread strictly in the order in which they started **`wait`**ing
- CS problem finally solved?

Semaphore Implementation Considerations

- Each semaphore is associated with a wake-up queue
  - Weak semaphores Wake up a random waiting thread on **`signal`**
  - Strong semaphores Wake up thread strictly in the order in which they started **`wait`**ing

- CS problem finally solved?

✓ **Mutual Exclusion** Only one thread can enter CS for semaphores initialized to 1

✓ **Progress** Only the thread within the CS hinders others to get in

✓ **Bounded Waiting** Strong semaphores guarantee bounded waiting

Semaphore Implementation Considerations

- Each semaphore is associated with a wake-up queue
  - Weak semaphores Wake up a random waiting thread on **`signal`**
  - Strong semaphores Wake up thread strictly in the order in which they started **`wait`**ing

- CS problem finally solved?

✓ **Mutual Exclusion** Only one thread can enter CS for semaphores initialized to 1

✓ **Progress** Only the thread within the CS hinders others to get in

✓ **Bounded Waiting** Strong semaphores guarantee bounded waiting

- Now every **`enter_critical_section()`** and **`leave_critical_section()`** is a syscall
  - Syscalls are an order of magnitude slower than function calls
  - Can we do better?

Fast User Space Mutex (futex)

- Spinlocks
  - \+ Quick when the wait-time is short
  - – Waste resources when the wait-time is long
- Semaphores
  - \+ Efficient when the wait-time is long
  - – Syscall overhead at every operation
- Idea of Linux' Fast User Space Mutex (futex)
  - There is a userspace and kernel component
  - Try to get into the CS with a userspace spinlock
  - If the CS is busy use a syscall to put thread to sleep
  - Otherwise just enter the CS with the now locked spinlock completely in userspace

Summary

- There is often the need for processes or threads to communicate
  - Message passing facilities provide explicit send and receive functions to exchange messages
  - Implicitly shared memory between threads or explicitly shared memory between processes allows exchanging information by modifying shared state
- When communicating, data races need to be taken into account
- Common techniques to synchronizes access to shared data include
  - Interlocked atomic operations
  - Spinlocks
  - Semaphores
  - Futexes

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition: Pages 119–134
- On Consistency
  - Lamport: “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”
  - Adve, Gharachorloo: “Shared Memory Consistency Models: A Tutorial”
  - Intel 64 and IA-32 Architectures Developer’s Manual: Vol. 3A, §8.2
  - Owens, Sarkar, Sewell: “A Better x86 Memory Model: x86-TSO”
- On Synchronization
  - Boehm, Adve: “You don’t know Jack about Shared Variables or Memory Models”
  - David, Guerraoui, Trigonakis: “Everything you always wanted to know about synchronization but were afraid to ask” (SOSP 2013)
  - Drepper: “Futexes are Tricky”