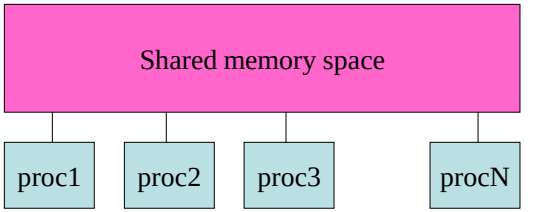
Shared Memory: Synchronization, Coherence, and Consistency

Shared Memory: A Look Underneath



Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
  - ensuring that modifications made by a processor propagate to all copies of the data
  - Program order preserved
  - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors

Synchronization

- Basic types
  - Mutual exclusion
    - Primitive: locks
  - Events
    - Global event-based
      - Primitive: Barriers
    - Point-to-point event-based
      - Semaphores (blocking)
      - Condition variables
      - Flags (busy-waiting/spinning)
      - Full-empty bits (hardware implementation; also considered message passing – produce-consumer)
    - Interrupts

Components of a Synchronization Event

– Acquire method (enter critical section, proceed past event)
– Waiting algorithm (busy waiting, blocking)
– Release method (enable others to proceed)

The Critical-Section Problem

- Problem context:
  - *n* processes all competing to use some shared data
  - Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Find a solution that satisfies the following:
  1. **Mutual Exclusion**. No two processes simultaneously in the critical section.
  2. **Progress**. No process running outside its critical section may block other processes.
  3. **Bounded Waiting/Fairness**. Given the set of concurrent processes, a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Bounded Buffer

- Shared data

```
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```

- Producer process

```
item nextProduced;
while (1) {
   while (counter==BUFFER_SIZE)
        ; /* do nothing */
   buffer[in] = nextProduced;
   in = (in+1) % BUFFER_SIZE;
   counter++;
}
```

- Consumer process

```
item nextConsumed;
while (1) {
   while (counter==0)
        ; /* do nothing */
   nextConsumed = buffer[out];
   out = (out+1) % BUFFER_SIZE;
   counter--;
}
```


Bounded Buffer

- The following statements must be performed *atomically*:

```
counter++;
counter--;
```

- Atomic operation means an operation that completes in its entirety without interruption
- The statement "`counter++`" may be compiled into the following instruction sequence:

```
register1 = counter;
register1 = register1 + 1;
counter   = register1;
```

- The statement "`counter--`" may be compiled into:

```
register2 = counter;
register2 = register2 - 1;
counter   = register2;
```

Race Condition

- **Race condition**:
  - The situation where several processes access and manipulate shared data concurrently
  - The final value of the shared data and/or effects on the participating processes depends upon the order of process execution – nondeterminism
- To prevent race conditions, concurrent processes must be **synchronized**

Eliminating Concurrency

- First idea: eliminating the chance of context switch when a process runs in the critical section.
  - effective as a complete solution only on a single-processor machine
  - only for short critical sections
- How to eliminate context switch?
  - software exceptions
  - hardware interrupts
  - system calls
- Disabling interrupts?
  - not feasible for user programs since they shouldn't be able to disable interrupts
  - feasible for OS kernel programs

Critical Section

- General structure of process P_i

```
do {
    entry section
       critical section
    exit section
       remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions
- Assumption: instructions are atomic and no re-ordering of instructions

Algorithm 3

- Combine shared variables of algorithms 1 and 2.
- Process P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn==j) ;
     critical section
   flag[i] = false;
     remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes. ⇒ called Peterson's algorithm.

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value
- Transaction support (e.g., Intel's TSX)

Simple Spinlock

type lock = (unlocked, locked)

procedure acquirelock (L : lock)

while (testandset (L) == locked) // returns old value

procedure releaselock (L : lock)

lock = unlocked

Implementing Critical Sections Using Busy Waiting

- In all our solutions today, a process enters a loop until the entry is granted ⇒ busy waiting.
- Problems with busy waiting:
  - Waste of CPU time
  - Potential for extra traffic/communication
  - If a process is switched out of CPU during critical section
    - other processes may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling (priority inversion problem)

Synchronization Using Special Instruction: TSL (test-and-set)

```
entry_section:
   TSL R1, LOCK          | copy lock to R1 and set lock to 1
   CMP R1, #0            | was lock zero?
   JNE entry_section     | if it wasn't zero, lock was set, so loop
   RET                   | return; critical section entered

exit_section:
   MOV LOCK, #0          | store 0 into lock
   RET                   | return; out of critical section
```

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

```
_Lock_acquire:
    ldstub [%o0], %o1
    addcc %g0, %o1, %g0
    bne _Lock
    nop
fin:
    jmpl %r15+8, %g0
    nop
_Lock_release:
    st %g0, [%o0]
    jmpl %r15+8, %g0
    nop
```

Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

```
try: mov R3, R4      ; mov exchange value
     ll   R2, 0(R1)  ; load linked
     sc R3, 0(R1)    ; store conditional
     beqz R3, try    ; branch if store fails
     mov  R4, R2     ; put load value in R4
```

Spinlock Algorithms

- Test&test&set (w, w/o exponential backoff)
- Ticket lock (w, w/o proportional backoff)
- Array based queue locks
- MCS linked-list based queue locks

MCS Lock Acquire

```
mcs_lock_acquire:
   st   %g0, [%o1+4]
   mov %o1, %g3
   swap [%o0], %g3
   cmp %g3, 0
   be   .LL4
   mov 1, %g2
   st   %g2, [%o1]
   st   %o1, [%g3+4]
.LL9:
   ld   [%o1], %g2
   cmp %g2, 0
   bne .LL9
   nop
.LL4:
   retl
   nop
```

MCS Lock Release

```
mcs_lock_release:
   ld  [%o1+4], %g2
   cmp %g2, 0
   bne .LL11
   nop
   cas [%o0], %o1, %g2
   cmp %g2, %o1
   be  .LL10
   nop
.LL17:
   ld  [%o1+4], %g2
   cmp %g2, 0
   be  .LL17
   nop
.LL11:
   st  %g0, [%g2]
.LL10:
   retl
   nop
```

Barrier Algorithms

- Centralized sense-reversing barrier
- Software combining tree
- Tournament barrier
- Dissemination barrier
- Combining tree with improved locality

A Simple Barrier

```
mycount: local variable; counter, flag, lock: shared variables
p = number of processors

lock(&lock);
if counter == 0
      flag = 0
mycount = ++counter
unlock(&lock);
if (mycount == p) {
      counter = 0
      flag = 1
}
else
      while (flag == 0) {};
```

Will this work?

A Sense-Reversing Barrier

```
local_sense=0: local variable; counter, flag, lock: shared variables
p = number of processors;

local_sense = !(local_sense);
lock(&lock)
counter++
if (counter == p) {
      unlock(&lock)
      counter = 0
      flag = local_sense
}
else {
      unlock (&lock)
      while (flag != local_sense) {};
}
```

Performance Goals

- Low latency, short critical path
- Low traffic
- Scalability
- Low storage cost
- Fairness

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Non-blocking algorithms

Failure or suspension of any thread cannot cause failure or suspension of another thread (no indefinite delay due to mutual exclusion)

- Operations defined on it do not require mutual exclusion over multiple instructions (use atomic primitives)

- Obstruction-free algorithm
  - One that guarantees that a thread running in isolation will make progress (although livelock is possible)
- Lock-free algorithm
  - Operations guarantee that some process will complete its operation a finite amount of time, even if other processes halt
- Wait-free algorithm
  - Operations can guarantee that EVERY non-faulting process will complete its operation in a finite amount of time

Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
  - ensuring that modifications made by a processor propagate to all copies of the data
  - Program order preserved
  - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors