Synchronization

Han Wang
CS 3410, Spring 2012
Computer Science
Cornell University

P&H Chapter 2.11
Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- Assume physical addresses (ignore virtual memory)
- Assume uniform memory access (ignore NUMA)
Synchronization

The need for synchronization arises whenever there are concurrent processes in a system. (even in a uni-processor system)

Forks and Joins: In parallel programming, a parallel process may want to wait until several events have occurred.

Producer-Consumer: A consumer process must wait until the producer process has produced data

Exclusive use of a resource: Operating system has to ensure that only one process uses a resource at a given time
All you need to know about OS (for today)

<table>
<thead>
<tr>
<th>Process</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS abstraction of a running computation</td>
<td>OS abstraction of a single thread of control</td>
</tr>
<tr>
<td>• The unit of execution</td>
<td>• The unit of scheduling</td>
</tr>
<tr>
<td>• The unit of scheduling</td>
<td>• Lives in one single process</td>
</tr>
<tr>
<td>• Execution state</td>
<td></td>
</tr>
<tr>
<td>+ address space</td>
<td>From thread perspective</td>
</tr>
<tr>
<td>From process perspective</td>
<td>• one virtual CPU core on a virtual multi-core machine</td>
</tr>
<tr>
<td>• a virtual CPU</td>
<td></td>
</tr>
<tr>
<td>• some virtual memory</td>
<td>Thread is much more lightweight.</td>
</tr>
<tr>
<td>• a virtual keyboard, screen, ...</td>
<td></td>
</tr>
</tbody>
</table>
Thread A
   for(int i = 0, i < 5; i++) {
      x = x + 1;
   }

Thread B
   for(int j = 0; j < 5; j++) {
      x = x + 1;
   }
Thread A

```java
for(int i = 0, i < 5; i++) {
    LW $t0, addr(x)
    ADDI $t0, $t0, 1
    SW $t0, addr(x)
}
```

Thread B

```java
for(int j = 0; j < 5; j++) {
    LW $t0, addr(x)
    ADDI $t0, $t0, 1
    SW $t0, addr(x)
}
```
Possible interleaves:
Atomic operation

To understand concurrent processes, we need to understand the underlying indivisible operations.

**Atomic operation**: an operation that always runs to the end or not at all.

- Indivisible. It cannot be stopped in the middle.
- Fundamental building blocks.
- Execution of a single instruction is atomic.

**Examples**:
- Atomic exchange.
- Atomic compare and swap.
- Atomic fetch and increment.
- Atomic memory operation.
Agenda

- Why cache coherency is not sufficient?
- HW support for synchronization
- Locks + barriers
Shared Memory Multiprocessor (SMP)
What could possibly go wrong?

```
x = x+1
while (x==5) {
    // wait
}
```

Diagram:
- Core0
- Core1
- Core3
- Interconnect
- I/O
Cache coherence defined...

Informal: **Reads** return most recently *written* value

Formal: For concurrent processes $P_1$ and $P_2$

- $P$ writes $X$ before $P$ reads $X$ (with no intervening writes)
  $\Rightarrow$ read returns written value

- $P_1$ writes $X$ before $P_2$ reads $X$
  $\Rightarrow$ read returns written value

- $P_1$ writes $X$ and $P_2$ writes $X$
  $\Rightarrow$ all processors see writes in the same order
    - all see the same final value for $X$
Recall: **Snooping** for Hardware Cache Coherence

- All caches monitor bus and all other caches
- **Bus read**: respond if you have dirty data
- **Bus write**: update/invalidate your copy of data
Example with cache coherence:

\[ P_1 \]
\[ x = x + 1 \]

\[ P_2 \]
\[ \text{while } (x == 5) ; \]
Example with cache coherence:

P1: \[ x = x + 1 \]

P2: \[ x = x + 1 \]

Is cache coherence sufficient?
Hardware Primitive: Test and Set

Test-and-set is a typical way to achieve synchronization when only one processor is allowed to access a critical section.

Hardware atomic equivalent of...

```c
int test_and_set(int *m) {
    old = *m;
    *m = 1;
    return old;
}
```

- If return value is 0, then you succeeded in acquiring the test-and-set.
- If return value is non-0, then you did not succeed.
- How do you "unlock" a test-and-set?

Test-and-set on Intel:

```c
xchg dest, src
```

- Exchanges destination and source.
- How do you use it?
Using test-and-set for mutual exclusion

Use test-and-set to implement mutex / spinlock / crit. sec.

```c
int m = 0;
...

while (test_and_set(&m)) { /* skip */ };

m = 0;
```
Snoop Storm

mutex acquire:

LOCK BTS var, 0
JC mutex acquire

mutex release:

MOV var, 0

- mutex acquire is very tight loop
- Every iteration stores to shared memory location
- Each waiting processor needs var in E/M each iteration
Test and test and set

mutex acquire:

TEST var, 1
JNZ mutex acquire
LOCK BTS var, 0
JC mutex acquire

mutex release:

MOV var, 0

• Most of wait is in top loop with no store
• All waiting processors can have var in $ in top loop
• Top loop executes completely in cache
• Substantially reduces snoop traffic on bus
Hardware Primitive: LL & SC

- LL: load link (sticky load) returns the value in a memory location.
- SC: store conditional: stores a value to the memory location ONLY if that location hasn’t changed since the last load-link.
- If update has occurred, store-conditional will fail.

- LL $rt$, immed($rs$) ("load linked") — $rt \leftarrow \text{Memory}[rs+immed]$
- SC $rt$, immed($rs$) ("store conditional") —
  if no writes to $\text{Memory}[rs+immed]$ since LL:
    $\text{Memory}[rs+immed] \leftarrow rt; rt \leftarrow 1$
  otherwise:
    $rt \leftarrow 0$

- MIPS, ARM, PowerPC, Alpha has this support.
- Each instruction needs two registers.
Operation of LL & SC.

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

Any time a processor intervenes and modifies the value in memory between the ll and sc instruction, the sc returns 0 in R3, causing the code to try again.
mutex from LL and SC

Linked load / Store Conditional

```c
fmutex_lock(int *m) {
again:
    LL t0, 0(a0)
    BNE t0, zero, again
    ADDI t0, t0, 1
    SC t0, 0(a0)
    BEQ t0, zero, again
}
```
try:    ll   R2, 0(R1) ; load linked
addi   R3, R2, #1
sc      R3, 0(R1) ; store condi
beqz   R3, try ; branch store fails

This has a name!
Hardware Primitive: CAS

- Compare and Swap
- Compares the contents of a memory location with a value and if they are the same, then modifies the memory location to a new value.
- CAS on Intel:
  \[ \text{cmpxchg loc, val} \]
- Compare value stored at memory location `loc` to contents of the Compare Value Application Register.
  - If they are the same, then set `loc` to `val`.
  - ZF flag is set if the compare was true, else ZF is 0
- X86 has this support, needs three registers (address, old value, new value). CISC instruction.
Alternative Atomic Instructions

Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional
  (MIPS, ARM, PowerPC, DEC Alpha, ...)

24
Spin waiting

Also called: spinlock, busy waiting, spin waiting, ...

- Efficient if wait is short
- Wasteful if wait is long

Possible heuristic:

- spin for time proportional to expected wait time
- If time runs out, context-switch to some other thread
Spin Lock

The single winning processor will read a 0 - all others processors will read the 1 set by the winning processor.

The diagram illustrates the process of acquiring a spin lock. The steps are as follows:

1. Read the lock variable.
2. Check if the lock variable is unlocked (0). If yes, proceed with the lock operation. If no, spin and check again.
3. Try to lock the variable using an atomic operation (ll&sc): read the lock variable and set it to the locked value (1).
4. Check if the lock operation succeeded (0). If yes, spin and check again. If no, unlock the variable.
5. If the lock operation succeeded, begin the update of shared data.
6. Finish the update of shared data.
7. Unlock the variable: set the lock variable to 0.
Example

_itmask # enter critical section

# lock acquisition loop
LL r1, 0(r4)  # r1 <= M[r4]
BNEZ r1, loop # retry if lock
already taken (r1 != 0)
ORI r1, r0, 1 # r1 <= 1
SC r1, 0(r4)  # if atomic (M[r4] <= 1 / r1 <= 1) else (r1 <= 0)
BEQZ r1, loop # retry if not atomic (r1 == 0) ...

# lock release
ORI r1, r0, 0 # r1 <= 0
SW r1, 0(r4)  # M[r4] <= 0

_itunmask # exit critical section
How do we fix this?

Thread A
for(int i = 0, i < 5; i++) {
    x = x + 1;
}

 acquire_lock(m);
 release_lock(m);

Thread B
for(int j = 0; j < 5; j++) {
    x = x + 1;
}

 acquire_lock(m);
 release_lock(m);
Guidelines for successful mutexing

Insufficient locking can cause races

- Skimping on mutexes? Just say no!

Poorly designed locking can cause deadlock

```
P1: lock(m1);  P2: lock(m2);
lock(m2);     lock(m1);
```

- know why you are using mutexes!
- acquire locks in a consistent order to avoid cycles
- use lock/unlock like braces (match them lexically)
  - lock(&m); ...; unlock(&m)
  - watch out for return, goto, and function calls!
  - watch out for exception/error conditions!
sum[Pn] = 0;
for (i = 1000*Pn; i< 1000*(Pn+1); i = i + 1) {
    sum[Pn] = sum[Pn] + A[i];
    /* each processor sums its */
    /* subset of vector A */
}

repeat /* adding together the */
    /* partial sums */
    synch(); /* synchronize first */

if (half%2 != 0 && Pn == 0) {
    sum[0] = sum[0] + sum[half-1];
    half = half/2
}
if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* final sum in sum[0] */
/* each processor sums its */
/* subset of vector A */
A[i];
Barrier Synchronization

Diagram showing a barrier synchronization scenario with processes labeled P0 to P9.
lock();
if(count==0) release=FALSE; /* First resets release */
count++;
unlock();
if(count==total) /* All arrived */
{
    count=0;
    release = TRUE;
}
else /* Wait for more to come */
{
    while (!release);
}

Problem: deadlock possible if reused
• Two processes: fast and slow
• Slow arrives first, reads release, sees FALSE
• Fast arrives, sets release to TRUE, goes on to execute other code, comes to barrier again, resets release to FALSE, starts spinning on wait for release
• Slow now reads release again, sees FALSE again
• Now both processors are stuck and will never leave
Correct Barrier Synchronization

Initially `localSense = True, release = FALSE`

```c
localSense=!localSense;        /* Toggle local sense */
lock();
    count++;                   /* Count arrivals */
    if(count==total){          /* All arrived */
        count=0;               /* Reset counter */
        release=localSense;    /* Release processes */
    }
unlock();
while(release==localSense);    /* Wait to be released */
```

Release in first barrier acts as reset for second

- When fast comes back it does not change release, it just waits for it to become FALSE
- Slow eventually sees release is TRUE, stops waiting, does work, comes back, sets release to FALSE, and both go forward.
Barrier with many processors

- Have to update counter one by one – takes a long time
- Solution: use a combining tree of barriers
  - Example: using a binary tree
  - Pair up processors, each pair has its own barrier
    - E.g. at level 1 processors 0 and 1 synchronize on one barrier, processors 2 and 3 on another, etc.
  - At next level, pair up pairs
    - Processors 0 and 2 increment a count a level 2, processors 1 and 3 just wait for it to be released
    - At level 3, 0 and 4 increment counter, while 1, 2, 3, 5, 6, and 7 just spin until this level 3 barrier is released
    - At the highest level all processes will spin and a few “representatives” will be counted.
  - Works well because each level fast and few levels
    - Only 2 increments per level, \( \log_2(\text{numProc}) \) levels
    - For large numProc, \( 2\log_2(\text{numProc}) \) still reasonably small
Beyond Mutexes

Language-level synchronization
• Conditional variables
• Monitors
• Semaphores
Software Support for Synchronization and Coordination: Programs and Processes
How do we cope with lots of activity?

Simplicity? Separation into **processes**

Reliability? **Isolation**

Speed? Program-level **parallelism**
Process

OS abstraction of a running computation
• The unit of execution
• The unit of scheduling
• Execution state
  + address space

From process perspective
• a virtual CPU
• some virtual memory
• a virtual keyboard, screen, ...

Program

“Blueprint” for a process
• Passive entity (bits on disk)
• Code + static data
Role of the OS

Context Switching

• Provides illusion that every process owns a CPU

Virtual Memory

• Provides illusion that process owns some memory

Device drivers & system calls

• Provides illusion that process owns a keyboard, ...

To do:

How to start a process?

How do processes communicate / coordinate?
Creating Processes:
Fork
Q: How to create a process?
A: Double click
After boot, OS starts the first process
...which in turn creates other processes
• parent / child  ➔ the process tree
$ pstree | view -
init++-NetworkManager++-dhclient
   | -apache2
   | -chrome++-chrome
   |   `--chrome
   | -chrome---chrome
   | -clementine
   | -clock-applet
   | -cron
   | -cupsd
   | -firefox---run-mozilla.sh---firefox-bin++-plugin-cont
   | -gnome-screensaver
   | -grep
   | -in.tftpd
   | -ntpd
   `--sshd---sshd---sshd---bash++-gcc---gcc---cc1
   |   `-pстree
   |     `-vim
   `--view
Init is a special case. For others...

Q: How does parent process create child process?
A: fork() system call

Wait. what? int fork() returns TWICE!
main(int ac, char **av) {
    int x = getpid(); // get current process ID from OS
    char *hi = av[1]; // get greeting from command line
    printf("I'm process %d\n", x);
    int id = fork();
    if (id == 0)
        printf("%s from %d\n", hi, getpid());
    else
        printf("%s from %d, child is %d\n", hi, getpid(), id);
}

$ gcc -o strange strange.c
$ ./strange "Hey"
I'm process 23511
Hey from 23512
Hey from 23511, child is 23512
Parent can pass information to child

• In fact, *all parent data* is passed to child
• But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?
A: Invent OS “IPC channels” : send(msg), recv(), ...
Parent can pass information to child

- In fact, *all parent data* is passed to child
- But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?

A: Shared (Virtual) Memory!
Processes and Threads
Parallel programming with processes:

• They share almost everything code, shared mem, open files, filesystem privileges, ...

• Pagetables will be *almost* identical

• Differences: PC, registers, stack

Recall: process = execution context + address space
**Process**

OS abstraction of a running computation

- The unit of execution
- The unit of scheduling
- Execution state
  + address space

From process perspective

- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, ...

**Thread**

OS abstraction of a single thread of control

- The unit of scheduling
- Lives in one single process

From thread perspective

- one virtual CPU core on a virtual multi-core machine
Multithreaded Processes

Single-threaded process vs. multithreaded process:
- In a single-threaded process, there is only one thread of execution.
- In a multithreaded process, there are multiple threads of execution, each with its own set of registers and stack.
#include <pthread.h>

int counter = 0;

void PrintHello(int arg) {
    printf("I'm thread %d, counter is %d\n", arg, counter++);
    ... do some work ...
    pthread_exit(NULL);
}

int main () {
    for (t = 0; t < 4; t++) {
        printf("in main: creating thread %d\n", t);
        pthread_create(NULL, NULL, PrintHello, t);
    }
    pthread_exit(NULL);
}
Threads versus Fork

in main: creating thread 0
I’m thread 0, counter is 0

in main: creating thread 1
I’m thread 1, counter is 1

in main: creating thread 2

in main: creating thread 3
I’m thread 3, counter is 2
I’m thread 2, counter is 3

If processes?
Example: Apache web server

void main() {
    setup();
    while (c = accept_connection()) {
        req = read_request(c);
        hits[req]++;
        send_response(c, req);
    }
    cleanup();
}
Example: Apache web server
Each client request handled by a separate thread (in parallel)

- Some shared state: hit counter, ...

Thread 52
read hits
addi
write hits

Thread 205
read hits
addi
write hits

(look familiar?)

Timing-dependent failure $\implies$ race condition

- hard to reproduce $\implies$ hard to debug
Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

Problem: hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency isn’t sufficient...

Need explicit synchronization to make sense of concurrency!
Managing Concurrency
Races, Critical Sections, and Mutexes
Concurrency Goals

Liveness
• Make forward progress

Efficiency
• Make good use of resources

Fairness
• Fair allocation of resources between threads

Correctness
• Threads are isolated (except when they aren’t)
Race Condition

Timing-dependent error when accessing shared state

- Depends on scheduling happenstance
  ... e.g. who wins “race” to the store instruction?

Concurrent Program Correctness = all possible schedules are safe

- Must consider *every possible* permutation
- In other words...
  ... the scheduler is your adversary
What if we can designate parts of the execution as **critical sections**

- Rule: only one thread can be “inside”

<table>
<thead>
<tr>
<th>Thread 52</th>
<th>Thread 205</th>
</tr>
</thead>
<tbody>
<tr>
<td>read hits</td>
<td>read hits</td>
</tr>
<tr>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td>write hits</td>
<td>write hits</td>
</tr>
</tbody>
</table>
Q: How to implement critical section in code?
A: Lots of approaches....

Disable interrupts?

CSEnter() = disable interrupts (including clock)
CSExit() = re-enable interrupts

read hits
addi
write hits

Works for some kernel data-structures
Very bad idea for user code
Q: How to implement critical section in code?
A: Lots of approaches....

Modify OS scheduler?
CSEnter() = syscall to disable context switches
CSExit() = syscall to re-enable context switches

read hits
addi
write hits

Doesn’t work if interrupts are part of the problem
Usually a bad idea anyway
Q: How to implement critical section in code?
A: Lots of approaches....

**Mutual Exclusion Lock (mutex)**

*acquire(m): wait till it becomes free, then lock it*

*release(m): unlock it*

```c
apache_got_hit() {
    pthread_mutex_lock(m);
    hits = hits + 1;
    pthread_mutex_unlock(m)
}
```
Q: How to implement mutexes?