Synchronization

OS Lecture 3

UdS/TUKL WS 2015
Announcements

1. First assignment out today. Start working on it early.
   
   »  http://courses.mpi-sws.org/os-ws15/

2. Send email to course mailing list if you are still looking for a partner

3. Slides available on course homepage a day or so after lecture.

   »  This does not replace attendance. Not all discussed topics will be reflected in the slides.

   »  Take your own notes and ask questions.
Review: Processes

- sphere of isolation (*protection domain*) and computation in progress (*thread*)

- *independent* processes
  - perfectly isolated
  - deterministic

- *cooperating* processes
  - possibly non-deterministic
  - require proper *synchronization*

- Why cooperate?
Cooperating Processes

How can processes cooperate?
Cooperating Processes

» through shared files

» explicitly via communication channels

» send() / receive() — message passing

» read() / write() — pipelines

» Ex: grep bar /tmp/foo | sort -n | head 12

» share memory

» some, but not all memory: shared segments (e.g., mmap())

» all memory: multithreaded process
Review: Threads

» multithreaded processes: can have more than one computation in progress in a sphere of isolation

» absolutely no isolation between threads of the same process

» each thread has its own program counter (PC), register contents, and stack

» Why have threads?

  » Why not just communication channels?

  » Why not just shared memory segments?
Review: Race Condition

Processes “racing” to carry out their conflicting operation. Example:

\[ A = 0x1 | | A = 0x10000 \]

Outcome depends on…

» interleaving of operations and relative speed of processes

» on what exactly constitutes an atomic operation

While there can be benign races, a race condition is typically indicative of buggy or missing synchronization.
Review: Atomic Operations

» Cannot be interrupted / interleaved “in the middle” of execution.

» Fixed set of primitive atomic ops provided by hardware.

» On a uniprocessor, anything between two interrupts is atomic: \(\rightarrow\) interrupts masked / disabled = atomic.

» For now, suppose we have only atomic reads and atomic writes.
The “too much milk” problem

Motivational example to illustrate challenges of proper synchronization.

Setting:

» You and a roommate (two processes). Buy new milk (action) if none left in fridge (condition).

Protocol:

» Whoever notices that there’s no milk left goes shopping.

What could go wrong?
The “too much milk” problem

Person A
3:00 Look in fridge. Out of milk.
3:05 Leave for store.
3:10 Arrive at store.
3:15 Leave store.
3:20 Arrive home, put milk away.
3:25
3:30

Person B
Look in fridge. Out of milk.
Leave for store.
Arrive at store.
Leave store.
Arrive home. OH, NO!

>> What does correct mean?
Specification

» don’t buy more than one bottle of milk at the same time

» somebody needs to go shopping

Refined:

» at most one person goes shopping at the same time
  (→ *mutual exclusion*)

» if one person has gone shopping (→ *critical section*), the other should await the outcome

» if there is no milk left, somebody should “eventually” go shopping (→ *progress*)
Terminology

**Mutual exclusion** / **mutex**: a mechanism that ensures that, from a set of operations, at most one happens at the same time (all others are excluded)

**Critical section**: a section of code (or a collection of operations) which only one process may be executing at the same time

How accomplished?
Locks

A common way to realize mutual exclusion is to use a locking mechanism:

» real-world equivalent: leave a note "hey, I’m getting milk; will be back soon"

» lock() before a critical section (= leave a note)

» unlock() after a critical section (= remove note)

» must wait if locked (= don’t shop if note on fridge)
Computerized Too Much Milk — Attempt 1

Idea: before shopping, leave a note on the refrigerator (= lock the shopping operation)
Computerized Too Much Milk — Attempt 1

Processes A & B:
1:   if (NoMilk) {
2:       if (NoNote) {
3:           Leave Note;
4:           Buy Milk;
5:           Remove Note;
6:       } 
7:   }

>> Does this work?
Attempt 1 — Why it fails

❗ Trace: A1-B1-A2-B2-A3-B3-...

➢ We have made the problem less likely, but we haven’t fixed it: → typical of broken synchronization

➢ Root cause: A and B observe exactly the same state (no milk, no note), so reach the same conclusion

➢ Why does attempt 1 work for humans, but not computers?

➢ Can we fix it by leaving the note first? Before checking for milk?
Computerized Too Much Milk — Attempt 2

Idea: break the symmetry

» A buys if there is no note
» B buys if there is a note

Effectively, take turns to buy milk and only go if it’s your turn.
Computerized Too Much Milk — Attempt 2

Processes A:
1: if (NoNote) {
2:  if (NoMilk) {
3:      Buy Milk;
4:  }
5:  Leave Note;
6: }

Process B:
1: if (Note) {
2:  if (NoMilk) {
3:      Buy Milk;
4:  }
5:  Remove Note;
6: }

» Does this work?
Claim: at most one process will buy milk.

How can you tell?
Claim: at most one process will buy milk.

How can you tell? Prove it!

A proof sketch:

1. A note will be left only by A, and only if there isn’t already a note.
2. A note will be removed only by B, and only if there is a note.
3. Thus, there is either one note, or no note.
4. If there is a note, only B will buy milk.
5. If there is not a note, only A will buy milk.
6. Thus, only one process will buy milk.
But does it really work?

» What if process B goes on vacation? (= doesn’t run for some time, e.g., blocked on I/O)

» Process A will not be able to buy milk more than once. ➞ *starvation*!

» Root cause: for A, no difference between "you’re buying" and "not my turn"
Computerized Too Much Milk — Attempt 3

Idea: use 2 separate notes to tell apart who is buying
Computerized Too Much Milk — Attempt 3

Processes A:
1: Leave NoteA;
2: if (NoNoteB) {
3:      if (NoMilk) {
4:          Buy Milk;
5:      }
6:  }
7: Remove NoteA;

Process B:
Leave NoteB;
if (NoNoteA) {
  if (NoMilk) {
    Buy Milk;
  }
}
Remove NoteB;

» Does this work?
Attempt 3 — Does it work?

» at most one process will buy milk ✔

» if one process “goes on vacation,” the other will still buy milk ✔

❗ Trace: A1–B1–A2–B2–A7–B7

» If both processes leave note at the same time: nobody will buy milk. ➔ starvation!
Computerized Too Much Milk — Attempt 4

Idea: *explicit tie-break rule*

process $B$ buys the milk if both try
Computerized Too Much Milk — Attempt 4

Processes A:
1: Leave NoteA;
2: if (NoNoteB) {
3:   if (NoMilk) {
4:     Buy Milk;
5:   }
6: }
7: Remove NoteA;

Process B:
Leave NoteB;
while (NoteA) DoNothing;
if (NoMilk) {
   Buy Milk;
}
Remove NoteB;

» Does this work?
Attempt 4 — Does it work?

Finally, yes!

» at most one process will buy milk ✔
» somebody will buy milk in all cases ✔

But:

» asymmetric & complex code
» Difficult to extend: what happens if a third roommate joins? What happens if there are multiple fridges & a pin board?
» Process B is busy-waiting (line 2), which wastes resources (especially on a uniprocessor).
The OS Approach: Abstraction

Problem:

» Piecing together a synchronization solution from low-level hardware primitives (like atomic read/write) is too cumbersome and error-prone.

Solution:

» A higher-level abstraction at the OS level: semaphores

» Flexible, portable semantics, easier to reason about
Higher-Level Synchronization Primitive: Goals

What are desirable properties for a general, high-level synchronization primitives?
Higher-Level Synchronization Primitive: Goals

- **Correctness**: allow at most one process in critical section at a time

- **Progress**: processes must be able to stall (“go on vacation”) for arbitrary amounts of time outside critical section

- **Fairness**: if multiple processes are waiting, don’t let anyone wait “forever”

- **Efficiency**: don’t waste large amounts of resources on waiting processes

- **Simplicity**: should be easy to use
Semaphores

A **semaphore** is a *counter* with two *atomic* operations:

- **P()**: wait for counter to exceed zero, then atomically decrement by 1
  - after operation returns, we know counter was positive
- **V()**: increment counter by 1
  - allows *exactly one*, already waiting or future, P() operation to proceed

Proposed by *Edsger Dijkstra* in 1962.
Semaphore Operation Names

- **P()**: Dutch *proberen* (to test), *passeren* (to pass), or *pakken* (to grab)
  - Common alternative: *wait()*
  - Linux kernel: *down()*
  - Java: *acquire()*

- **V()**: Dutch *verhogen* (to increase) or *vrijgave* (release)
  - Common alternative: *signal()*
  - Linux kernel: *up()*
  - Java: *release()*
Computerized Too Much Milk — Attempt 5

Idea: *use a semaphore named OKToBuyMilk*
Processes A & B:

1:  P(OKToBuyMilk);

2:  if (NoMilk) {

3:      Buy Milk;

4:  }

5:  V(OKToBuyMilk);

>> Does this work? What is right right initial value for OKToBuyMilk?
Binary semaphore

Important special case: a **binary semaphore** that takes on only the values **zero** and **one** can be used to provide *mutual exclusion*.

» initialize to one

» `lock() = P()`
   → counter becomes zero, no other `P()` can pass

» `unlock() = V()`
   → lock released, next critical section can start
Proper use of (Binary) Semaphores

What to do and what to avoid when dealing with locks or semaphores?
Proper use of (Binary) Semaphores

» Always lock with \( P() \) before manipulating shared data

» Always unlock with \( V() \) after manipulating shared data

» Do not lock again if already locked (\( \rightarrow \) requires reentrant locks)

» Do not unlock if it was not locked by the same process
  » but special cases exists where it’s ok to break this rule — can you think of an example?

» Keep critical sections \textit{as short as possible}. 
Condition Synchronization

» Semaphores can be used for more than just *mutual exclusion*

» **Condition synchronization**: permit processes to wait for events to occur without wasting resources (busy-waiting).

» Also called *counting semaphores*: opposite of binary semaphores (i.e, regular semaphores that can take on any value).

» Typically, one counting semaphore per event type
Producer & Consumer Example

Setting:

» one process, the **producer**, creates data items

» another process, the **consumer**, consumes data times

» shared, *limited-size* pool of **buffers** to hold produced, but not yet consumed data items

*What are the requirements?*
Producer & Consumer Example

Requirements:

» consumer must wait for data to be available
   → wait for “data produced” event

» producer must wait for buffer space to be available
   → wait for “buffer emptied” event

» at most one process must manipulate buffer at the same time
   → mutual exclusion
Producer & Consumer Example

How many counting and binary semaphores do we need?

What are their initial values?

Assume: we have space for numBuffers data items.
Producer & Consumer Example

Two counting semaphores:
- buffer_empty, initialized to numBuffers
- buffer_filled, initialized to zero

One binary semaphore:
- buffer_pool_mutex, initialized to one
Producer Process

Idea: wait for space, get empty buffer, produce, make full buffer available
Producer Process

P(buffer_emptyed);
P(buffer_pool_mutex);
get buffer from pool of empty buffers;
V(buffer_pool_mutex);
produce data in buffer;
P(buffer_pool_mutex);
add buffer to pool of full buffers;
V(buffer_pool_mutex);
V(buffer_filled);
Consumer Process

Idea: *wait for data, get full buffer, consume, make empty buffer available*
Consumer Process

P(buffer_filled);
P(buffer_pool_mutex);
get buffer from pool of full buffers;
V(buffer_pool_mutex);
process data in buffer;
P(buffer_pool_mutex);
add buffer to pool of empty buffers;
V(buffer_pool_mutex);
V(buffer_emptyied);
Discussion

» Why does the producer $P(\text{buffer}_\text{emptied})$, but $V(\text{buffer}_\text{filled})$?

» What changes are required to add a second consumer?

» Could we have separate binary semaphores $\text{empty_buffer_mutex}$ and $\text{full_buffer_mutex}$?

» Can we change the order of the $V()$ operations? (i.e., $V(\text{buffer_pool_mutex})$ after $V(\text{buffer}_\text{emptied})$?)

» Can we change the order of the $P()$ operations? (i.e, $P(\text{buffer_pool_mutex})$ before $P(\text{buffer}_\text{filled})$?)
Deadlock

Or “deadly embrace” [Dijkstra].

» Cycle in the wait–for graph.

» $A$ is waiting for $B$, $B$ is waiting for $C$, …, $Y$ is waiting for $Z$, and $Z$ is waiting for $A$

To avoid deadlock, always acquire nested locks in the same order. Examples:

» $P(X); P(Y); V(Y); V(X)$ || $P(Y); P(X); V(X); V(Y)$ will deadlock.

» $P(X); P(Y); V(Y); V(X)$ || $P(X); P(Y); V(Y); V(X)$ is fine.
Another Synchronization Example

Setting:

- a shared database
- multiple *readers* may access database simultaneously
- each *writer* requires exclusive access

*Which constraints do we need to enforce?*
Shared Database — Specification

- **writers** can only proceed if there are no active readers or writers

- **readers** can only proceed if there are no active or waiting writers
Shared Database — Variables

Four (non-atomic) state variables:

- \(\text{AR} \& \text{WR}\): number of active \& waiting readers
- \(\text{AW} \& \text{WW}\): number of active \& waiting writers

Semaphores:

- protect state variables with semaphore Mutex
- **writers** use semaphore OKToWrite to wait
- **readers** use semaphore OKToRead to wait
Initial Values

AR = AW = WR = WW = 0

Mutex = 1

OKToWrite = 0

OKToRead = 0
Reader Process

>> readers can only proceed if there are no active or waiting writers

Idea:

1. first, check for any writers
2. start reading if none are present; otherwise wait
3. don’t forget to let (later-arriving) writers know a read is in progress
4. the last reader to leave must notify a waiting writer (if any)
Reader Process

Reader entry:

P(Mutex);
if (AW + WW == 0) {
    V(OKToRead);
    AR = AR + 1;
} else {
    WR = WR + 1;
}
V(Mutex);
P(OKToRead);
[...start reading DB...]

Reader exit:

[...finish reading DB...]
P(Mutex);
AR = AR - 1;
if (AR == 0 && WW > 0) {
    V(OKToWrite);
    AW = AW + 1;
    WW = WW - 1;
}
V(Mutex);
Some Examples

1. Single reader enters and leaves system
2. Two readers enter and leave system
Writer Process

writers can only proceed if there are no active readers or writers

Idea:

1. first, check for any writers or readers
2. start writing if nobody else is present; otherwise wait
3. when leaving, unblock next writer...
4. ...or all readers if no writer is waiting
Writer Process

Writer entry:

P(Mutex);
if (AW + AR + WW == 0) {
    V(OKToWrite);
    AW = AW + 1;
} else {
    WW = WW + 1;
}
V(Mutex);
P(OKToWrite);

[...start writing DB...]

Writer exit:

[...finish writing DB...]
P(Mutex);
AW = AW - 1;
if (WW > 0) {
    V(OKToWrite);
    AW = AW + 1;
    WW = WW - 1;
} else while (WR > 0) {
    V(OKToRead);
    AR = AR + 1;
    WR = WR - 1;
}
V(Mutex);
More Examples

1. Single writer $W_1$ enters and leaves system
2. Two readers $R_1, R_2$ enter system
3. Writer $W_2$ enters system and waits
4. Reader $R_3$ enters system and waits
5. Readers $R_1, R_2$ leave system, writer $W_2$ continues
6. Writer $W_2$ leaves system, reader $R_3$ continues and leaves
Discussion

» Is the “+ wW” necessary in the writer entry check?

» If there are both readers and writers, who gets priority? Always?

» Which values do AW, OKToRead, and OKToWrite assume?

» Is the first writer to execute P(Mutex) guaranteed to be the first writer to access the DB?
Semaphore Implementation

- Semaphores are a powerful, higher-level abstraction...
- ... but are not provided by hardware.
- The OS must provide a semaphore implementation based on the available atomic primitive operation provided by hardware.
How to Implement Semaphores

» Could use atomic reads and writes, like in *too-much-milk* example…

» …but that leads to busy-waiting and inelegant solution.
Semaphore System Calls

» Instead, realize \texttt{P()} and \texttt{V()} as \textit{system calls} in the kernel.

» \textit{Block} (or \textit{suspend}) threads that must wait in \texttt{P()} by
  » setting their state to \texttt{WAITING} and
  » removing them from the ready queue.

» \textit{Unblock} (or \textit{resume}) waiting threads in \texttt{V()} by
  » setting their state to \texttt{READY} and
  » adding them to the ready queue.
Semaphore Sketch

typedef struct {
    int count;
    queue q;
} Semaphore;

» P(): *atomically* check count and add process to q if count ≤ 0; otherwise decrement count

» V(): *atomically* resume process in q (if any); otherwise increment count

» But access to the struct is not atomic…

» …how to make sure that operations are *effectively* atomic?
Uniprocessor Solution

Idea: disable interrupts to avoid interleaving “in the middle” of a $P()$ or $V()$ operation.
void P(Semaphore &s) {
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
    } else {
        set_state(current_thread, WAITING);
        remove_from_ready_queue(current_thread);
        add_to_queue(&s->q, current_thread);
        schedule(); /* context-switch away */
    }
    Enable interrupts;
}
Uniprocessor Solution: \texttt{V()}

```c
void \texttt{V(Semaphore \&s)} \{
    \texttt{Disable interrupts;}
    \texttt{if (isEmpty(s->q)) \{}
        \texttt{s->count += 1;}
    \texttt{\}} \texttt{else \{}
        \texttt{thread = RemoveFirst(s->q);}
        \texttt{set_state(thread, READY);}
        \texttt{add_to_ready_queue(thread);}
    \texttt{\}}
    \texttt{Enable interrupts;}
\}
```
The Multiprocessor Case

» Why does the previous solution not work on a multiprocessor?
The Multiprocessor Case

» Why does the previous solution not work on a multiprocessor?
   → Concurrent modification of Semaphore struct

» Must exclude both:
   » local interleaving (as on a uniprocessor)
   » accesses on remote processors

» Can we just turn off interrupts on all processors?
Multiprocessor Approach

1. Turn off interrupts to protect against local interleaving.

2. Use a flag and busy-waiting to synchronize with other cores (→ a spin lock).

   » spin_lock(int*) / spin_unlock(int*)

   » Wait, isn’t busy-waiting “bad”?

   » Why is it ok here?
Multiprocessor Solution

» Add a **spin lock**: an int variable to serve as a “operation is currently in progress” flag.

```c
typedef struct {
    int slock; /* initially 0 */
    int count;
    queue q;
} Semaphore;
```
void P(Semaphore &s) {
    Disable interrupts;
    spin_lock(&s->slock);
    if (s->count > 0) {
        s->count -= 1;
        spin_unlock(&s->slock);
    } else {
        set_state(current_thread, WAITING);
        remove_from_ready_queue(current_thread);
        add_to_queue(&s->q, current_thread);
        spin_unlock(&s->slock);
        schedule(); /* context-switch away */
    }
    Enable interrupts;
}
void V(Semaphore &s) {
    Disable interrupts;
    spin_lock(&s->slock);
    if (isEmpty(&s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(&s->q);
        set_state(thread, READY);
        add_to_ready_queue(thread);
    }
    spin_unlock(&s->slock);
    Enable interrupts;
}
How to Implement Spin Locks?

» Most CISC machines provide some sort of **atomic read-modify-write** instruction.

» Commonly available: **test-and-set (TAS)** operation
  
  » always sets variable to one
  
  » returns old value prior to write

» RISC alternative: **load-linked (LDL)** and **store-conditional (STC)** instructions
  
  » LDL establishes link between memory location and processor
  
  » any write to a linked memory location destroys its links
  
  » STC fails if written-to memory location is not linked
Test-and-Test-and-Set (TTAS) Spin Lock

Idea: busy-wait until old value was zero (= unlocked)
Test-and-Test-and-Set (TTAS) Spin Lock

Idea: busy-wait until old value was zero (= unlocked)

```c
void spin_lock(int *lock) {
    do {
        while (*lock)
            /*do nothing*/;
    } while (TAS(lock) == 1);
}

void spin_unlock(int *lock) {
    *lock = 0;
}
```
LDL–STC Spin Lock

Idea: \textit{emulate TAS with LDL–STC}
LDL–STC Spin Lock

Idea: emulate TAS with LDL–STC

```c
int TAS(int *x) {
    do {
        old_value = LDL(x);
    } while (STC(x, 1) == STORE FAILED);
    return old_value;
}
```
Spin Lock Discussion

>> A real implementation must worry about compiler barriers and memory fences (⇒ weak memory consistency).

>> A simple TTAS lock ensures no order.
   >> Starvation possible under heavy contention, especially on large multicore.

>> Polling of shared variable is not at all friendly to cache-consistency protocol.

>> Much better spin locks exist…
Semaphore Key Points

Two **fundamental uses** for semaphores (*review both!*):

» mutual exclusion

» condition synchronization

Semaphores are an example of **layering**:

» provide powerful abstraction (simple, portable, as many as needed)

» deal with atomic operations offered by hardware just once in the OS kernel to implement semaphores