TDDB68 + TDDE47

Lecture 5:
Synchronisation

Klas Arvidsson

Slides based on work by Mikael Asplund and Adrian Pop

Copyright Notice:
Thanks to Christoph Kessler and Simin Nadjm-Tehrani for some of the material behind these slides.

Reading guidelines

- Silberschatz, Galvin and Gagne, Operating System Concepts
  - 9th edition: Chapter 6.1-6.9
  - 10th edition: Chapter 6.1-6.7 + 7.1-7.3

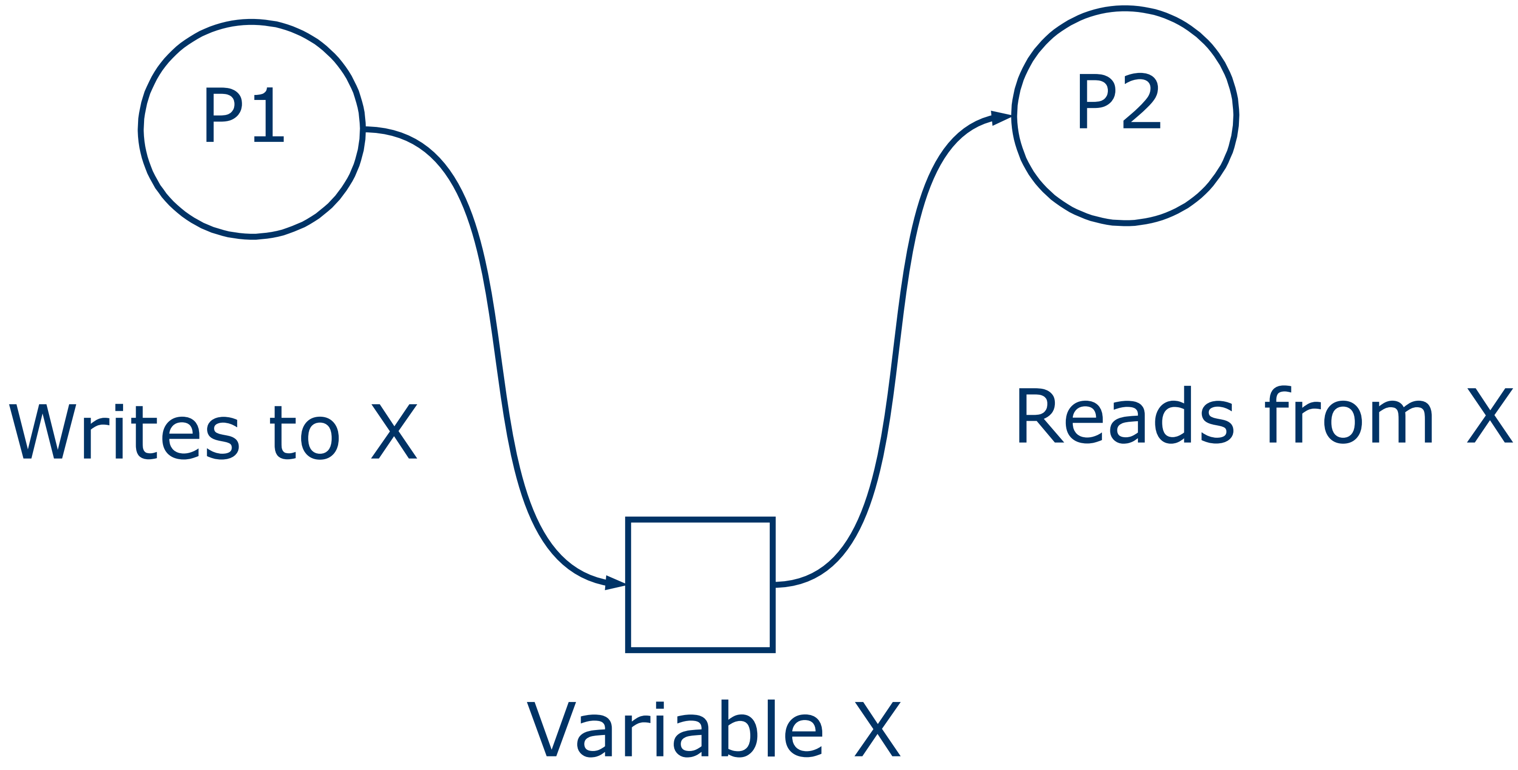
- Hint
  - Deadlock empire: https://deadlockempire.github.io/
  - Vinjett for TDDE47

Recall from lecture 3 (processes)

- Inter-process communication
  - Shared variables
  - Message passing
- Message passing is clean but gives high overhead
- What about Shared variables?

Basic operation

- Communication using shared variables



Sharing variables

- Often requires atomicity
- Consider the two processes using a shared variable x initialised at 0:
- What is the outcome of running them both to completion?

```
P0 {                P1 {
    x = x + 1;          x = x + 1;
}                   }
```

Machine instructions

x = x+1 is really:

```
LD R, x  // load register R from x
INC R    // increment register R
ST R, x  // store register R to x
```

- The program will be compiled, and the compiler may optimize for a specific architecture
- What can you assume about the compiler, the runtime environment and the architecture? (Nothing, or read the specs!)

Non-atomic operations

```
P0 {                      P1 {
    x = x + 1;                x = x + 1;
}                         }
```

Can become:

```
P0: LD R, x
P0: INC R
                P1: LD R, x
                P1: INC R
P0: ST R, x
                P1: ST R, x
```

How?

// Example of events that may couse this interleaving

P0: LD R, x

P0: INC R

// Timer interrupt, P0 time slice run out, P1 scheduled

P1: LD R, x

P1: INC R

// Device interrupt, long handling, P1 time slice run out, P0 scheduled

P0: ST R, x

// P0 completed, P1 scheduled

P1: ST R, x

Menti.com 6214 8470

```
P0 {
    x = x - 1;
}
```

```
P1 {
    x = x + 1;
}
```

What are the possible results after both thread run once? X start at 255. Select all possible results.

- 254
- 255
- 256
- 510

Shared data

- Primitive data types
  - Atomic access often supported by hardware
  - May not require special protection (read the specs!), but the compiler must be made aware of atomic intentions!
- Composite data types
  - E.g., update date, time and stock value
  - Atomic access needs to be implemented in software

Shared data example

```
M = [
   ('A', 4),
   ('F', 0),
   ('K', 7),
   ('X', 1),
];
```

Shared data example

- **Task to run in thread A and thread B:**
  - Check if C in M, memorize position
  - If so, increment value of memorized pos
  - If not, add C to M with value 1

  Menti.com 6214 8470

Shared data example

- **Alternate task to run in thread B:**
  - Check if C in M, memorize position
  - If so, decrement value at memorized pos
  - If decr. value == 0, remove pos from M

Live performance!

- Thread A:
    - Check if 'G' in M, memorize position
- Thread B:
    - Check if 'G' in M, memorize position
    - If so, increment value of memorized pos
    - If not, add 'G' to M with value 1
- Thread A:
    - If so, increment value of memorized pos
    - If not, add 'G' to M with value 1

Live performance!

- Thread B:
  - Check if 'F' in M, memorize position
  - If so, decrement value at memorized pos
- Thread A:
  - Check if 'F' in M, memorize position
  - If so, increment value of memorized pos
  - If not, add 'G' to M with value 1
- Thread B:
  - If decr. value == 0, remove pos from M

Race condition

If the order of operations performed by multiple processes can affect the outcome of the computation, and if this is **unintended**, then the system suffers from a **race condition**

Critical section

- Consider n processes that need to exclude concurrent execution of some parts of their code

```
Process Pi {
    entry-protocol
    critical-section
    exit-protocol
    non-critical-section
}
```

- Fundamental problem to design entry and exit protocols for critical sections

Critical-Section Problem

- Mutual Exclusion
- Progress
- Bounded waiting

Mutual Exclusion

If process P is executing in critical section C, then no other processes can be executing in C (accessing the same shared data/resource).

Note: Several code sections may be labelled C if they touch the same shared data/resource. Ony one process should be allowed in any section C.

Progress

If no process is executing in critical section C and there exist some processes that wish to enter C, then the selection of the process that will enter C next cannot be postponed indefinitely.

Note: It's about making sure someone is entering the section if no-one is working there, making progress on the work in the section.

Bounded waiting

A bound must exist on the number of times that other processes are allowed to enter critical section C after a process has made a request to enter C and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

Note: Progress is not enough, we need to avoid starvation!

Solutions for critical section problem

- Software-only solutions

- Solutions with hardware support

- Synchronization primitives

Software-only solutions

Dijkstras mutual exclusion (1965)

```
Process P1
while (true) {
  flag1 = up
  while (flag2 == up) {
    // do nothing
  }
  critical section
  flag1 = down
  non-critical section
}
```

```
Process P2
while (true) {
  flag2 = up
  while (flag1 == up) {
    // do nothing
  }
  critical section
  flag2 = down
  non-critical section
}
```

Second attempt

```
Process P1
while (true) {
    while (flag2 == up) {
      //do nothing
  }
  flag1 = up
  critical section
  flag1 = down
  non-critical-section
}
```

```
Process P2
while (true) {
    while (flag1 == up) {
      //do nothing
  }
  flag2 = up
  critical section
  flag2 = down
  non-critical-section
}
```

Third attempt

```
Process P1
while (true) {
  while (turn == 2) {
    //do nothing (busy waiting)
  }
  critical section
  turn = 2
  non-critical-section
}
```

```
Process P2
while (true) {
  while (turn == 1) {
    //do nothing (busy waiting)
  }
  critical section
  turn = 1
  non-critical-section
}
```

Peterson’s algorithm

```
Process P1
while (true) {
   flag1 = up // P1 want to enter
   Turn = 2    // let P2 go first
   while (flag2 == up) and
          (turn == 2) {
      //do nothing, wait for P2
   }
   critical section
   flag1 = down // P1 leaves
   non-critical-section
}
```

```
Process P2
while (true) {
   flag2 = up // P2 want to enter
   Turn = 1    // let P1 go first
   while (flag1 == up) and
          (turn == 1) {
      //do nothing, wait for P1
   }
   critical section
   flag2 = down // P2 leaves
   non-critical-section
}
```

- What assumptions about compiler and hardware must hold true?

Hardware support

Hardware Atomic Support for Synchronization

- **TestAndSet**: test memory word and set value atomically
- **Swap**: swap contents of two memory words atomically
- **CompareAndSwap**: compare memory and set atomically
- https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

If multiple atomic instructions are executed simultaneously (each on a different CPU in a multiprocessor), then they take effect sequentially in some arbitrary order.

TestAndSet Instruction

Atomic: state only observable after all instructions are done (or none of them).

Definition in pseudocode:

```
boolean TestAndSet (boolean *target) {
    boolean old_value = *target
    *target = true
    return old_value
}
```

CS Solution using TestAndSet

```
lock = false //shared variable

while (true) {
  while (TestAndSet (&lock)) {
      // do nothing (busy waiting)
  }
  critical section
  lock = false
  non-critical section
}
```

Swap Instruction

- Definition in pseudocode:

```
void Swap (boolean *a, boolean *b) {
      boolean save_a = *a
      *a = *b
      *b = save_a
}
```

CS Solution using Swap

```
lock = false     //shared variable

while (true) {
  tmp = true;    //local variable (not shared)
  while ( tmp == true) {
    swap (&lock, &tmp );     // busy waiting…
  }
  critical section
  lock = false;
  non-critical section
}
```

CompareAndSwap Instruction

- Definition in pseudocode:

```
int CompareAndSwap (int *ptr, int cmp, int new) {
        int old = *ptr
        if ( *ptr == cmp )
            *ptr = new
        return old
}
```

Synchronization primitives

Programming language support

- Would be useful to have support from an operating system or a programming language

- Modern programming languages have explicit support:
  - java.util.concurrency provides good support.
  - Ada: built-in run-time support with explicit task synchronisation entry points (Rendezvous)
  - Python: threading import *
  - C: pthreads, C++: std::thread

Synchronization primitives

- Abstraction layer
  - Easier to use, but must be implemented
- Do not solve all synchronization problems
- Examples:
  - Semaphores
  - Locks
  - Condition variables
  - Monitors

Semaphores

- A semaphore S is a *non-negative* integer variable on which only two atomic operations wait and signal can be performed

```
wait(S):
    wait until S > 0
    S = S-1

signal(S):
    S = S+1
```

CS solution with semaphore

```
semaphore S = 1

while (true) {
  wait(S)
  critical section
  signal(S)
  non-critical section
}
```

Atomicity of semaphore implementation must be provided by the supporting environment

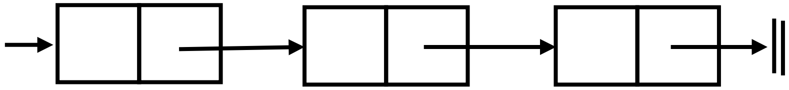
Implementation considerations

Spin locks

- All entry protocols so far (including semaphore wait) uses a busy wait loop – called a spin lock
- Sometimes necessary (kernel-level programming)
- Wasteful for synchronization of user processes
- Ok for short waits when thread on other core is expected to complete fast (real hw concurrency)

Eliminate Busy Waiting

- With each semaphore there is an associated **waiting queue**. Each entry in a waiting queue contains:
  - Process table index, e.g. pid
  - Pointer to next entry



- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

Samaphore datastructure with a queue

```
typedef struct {
    int value;
    struct process *wqueue;
} semaphore;
```

Wait implementation w/o busy waiting

```
void wait ( semaphore *S ) {
  S->value--;
   if (S->value < 0) {
      add this process to S->wqueue;
      block();  // I release the lock on the critical
   }            // section for S and release the CPU
 }
```

- This code is in itself a critical section, what if two threads read value == 1 “simultaneous”?

Signal Implementation w/o busy waiting

```
void signal ( semaphore *S ) {
   S->value++;
   if (S->value <= 0) {
     remove a process P from S->wqueue;
    wakeup (P);  // append P to ready queue
   }
}
```

Counting semaphores

- When more than one instance of a resource is available, e.g. print servers
- Processes can use up to max available but no more
- The semaphore can be initialised to provide access for n processes

- Keeps track of ***available*** resources
- Good for time sync – make sure X happend in thread A before performing Y in thread B

Semaphore initialization

- Crutial to determine the semantics of the semaphore

- Must be stated in your exam answers!

A semaphore with maximum value 1 is called a **binary semaphore**, useful to implement lock.

Locks

- Binary semaphore

- Operations often called
  - **Acquire** (instead of wait)
  - **Release** (instead of signal)

- Only the thread that acquired the lock can release it – built in error checks!

Complex data structures

Complex data structures

- Data is often structured
  - Lists
  - Objects
  - Structs

- Consistency requirements
  - cannot change one part of the data structure but not the other part

Two options

- One big lock
  - Safe (no synchronization problems)
  - Slow (reduces concurrency of solution)
- Multiple synchronization primitives
  - Fast (allows higher degree of synchronization)
  - Potentiall dangerous (introduces new concurrency problems)

Multiple synchronization primitives

- **Conditional action**
  - Purpose is to avoid busy waiting
  - Examples:
    - Compute the interest when all transactions have been processed
    - Book a flight seat only if seats are available

- **Mutual exclusion**
  - Purpose is to avoid errors
  - Example:
    - Two customers shall not be booked on the same seat

Deadlock and starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused only by some of the waiting processes. But they can not cause the event when waiting...
- **Starvation** – indefinite blocking. Other threads keep getting priority to a resource resulting in one thread waiting indefinitly.
  - A process may never be removed from the semaphore queue in which it is suspended.

Focus on the resource (data)!

- Non-shared data does not need protecting
  - Automatic (stack, local) variables
- Same resource must be protected with the same synchronization primitive
- Consistency requirements and access patterns determine the granularity of synchronization

Hints

- Identify all shared variables/data/resources
- Understand the purpose of the code, what is the intentions, and what data states should be possible/impossible?
- Identify where knowledge of data is built up in the local thread (result of calculation or check base on the shared data)
- Strive to place synch primitives with the data to protect for most paralellism (global synch primitives lead poor parallelism)

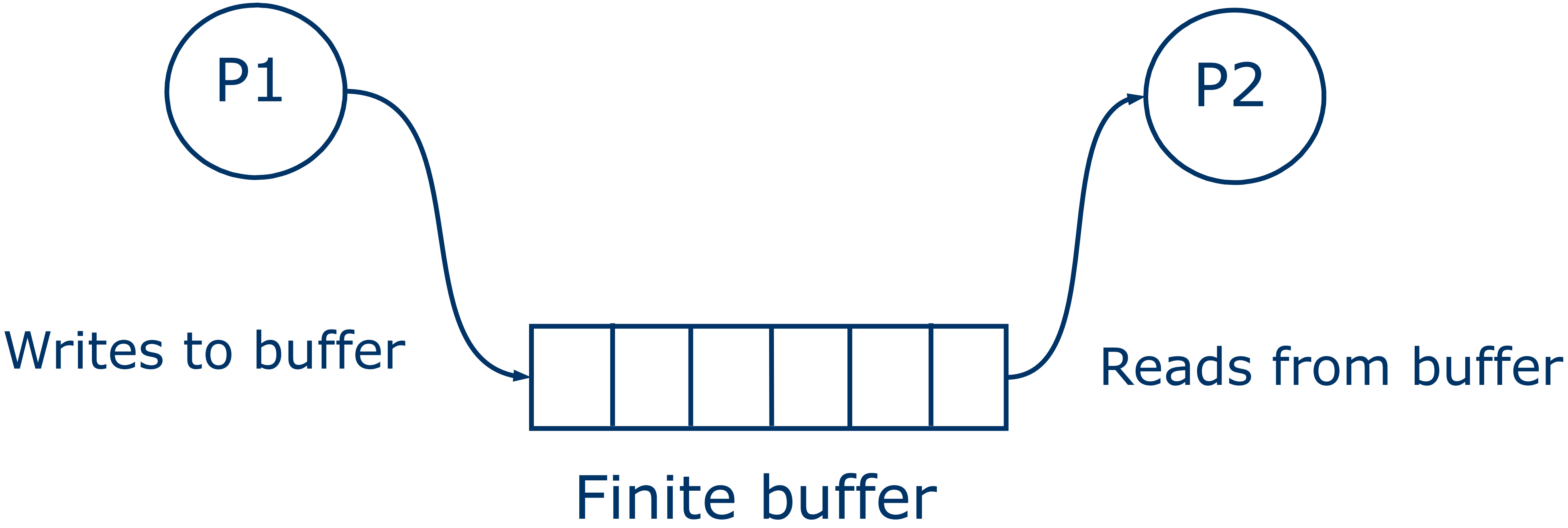
Common mistakes

- Omitting **wait** (mutex) or **signal** (mutex) (or both)
- **wait** (mutex) …. **wait** (mutex)
- **wait** (mutex1) …. **signal** (mutex2)
- Multiple semaphores with different orders of **wait**() calls
  - Example: Each philosopher first grabs the chopstick to its left → risk for deadlock!
- Not counting available resources

Two more useful synchronization primitives

- Condition variables
- Monitors (in lecture 9)

Example: bounded buffer



P1

P2

Writes to buffer

Reads from buffer

Finite buffer

Issues

- Writing to full buffer (conditional action)
- Reading from empty buffer (conditional action)
- Two write operations to the same element (mutual exclusion)

Condition variables

- Declared as special synchronisation variables:

  condition X;

- With two designated operations:

  wait: suspend the calling process (releasing lock!)

  signal/notify: if there are suspended processes on this variable, wake one up

- Wait will always wait, signal may have nothing to wake up – Very different from semaphore semantics!

Examples in Progviz!

On lab computer:
/courses/TDDE47/progviz.sh

At home:
https://storm-lang.org/index.php?q=01-Introduction%

(Progvis created by Filip Strömbäck)

Final remarks

- Concurrency is hard!
  And thus worthwhile to be an expert in!

- Get some practice
  - Pintos labs
  - Deadlock empire
  - Exam synchronization question