Imperial College London

Operating Systems

Synchronisation – Part II

Race Condition

- Occurs when multiple threads or processes read and write **shared data** and the final result depends on the relative timing of their execution
  - i.e. on the exact process or thread **interleaving**

- E.g., the `Extract` example → final value of account 8,000 or 9,000

Thread Interleavings

```
int a, b; // shared
void  T1()                    void T2()
{                             {
  a = 1;                        b = 2;
  b = 1;                        a = 2;
}                             }
```

a = 1	a = 1	a = 1	b = 2	b = 2	b = 2
b = 1	b = 2	b = 2	a = 2	a = 1	a = 1
b = 2	b = 1	a = 2	a = 1	a = 2	b = 1
a = 2	a = 2	b = 1	b = 1	b = 1	a = 2
(2, 2)	(2, 1)	(2, 1)	(1, 1)	(2, 1)	(2, 1)

Tutorial

Consider the following three threads:

T1	T2	T3
a = 1;	b = 1;	a = 2;
b = 2;		

How many possible interleaving are there?

a) 4
b) 8
c) 12
d) 16

Tutorial

If all thread interleavings are as likely to occur, what is the probability to have a=1 and b=1 after all threads complete execution?

All thread interleavings

a = 1	a = 1	a = 1	a = 1	a = 1	a = 1
b = 2	b = 2	b = 1	a = 2	b = 1	a = 2
b = 1	a = 2	b = 2	b = 2	a = 2	b = 1
a = 2	b = 1	a = 2	b = 1	b = 2	b = 2
(2, 1)	(2, 1)	(2, 2)	(2, 1)	(2, 2)	(2, 2)

b = 1	a = 2	b = 1	a = 2	b = 1	a = 2
a = 1	a = 1	a = 1	a = 1	a = 2	b = 1
b = 2	b = 2	a = 2	b = 1	a = 1	a = 1
a = 2	b = 1	b = 2	b = 2	b = 2	b = 2
(2, 2)	(1, 1)	(2, 2)	(1, 2)	(1, 2)	(1, 2)

Memory models

In this course, we assume *sequential consistency:*

- The operations of each thread appear in program order
- The operations of all threads are executed in some sequential order atomically

But other memory models (due to hardware behaviour and compiler optimisations) exist!

Sequential consistency vs weak memory models

```
int flag1 = 0, flag2 = 0; // shared
void  T1()                    void T2()
{                             {
  flag1 = 1;                    flag2 = 1;
  if (flag2 == 0);              if (flag1 == 0)
       critical()                    critical()
}                             }
```

- Under sequential consistency, it's impossible for both threads to read flag1 = flag2 = 0 in their if statements
- Under weak memory models, this is possible!
- Advanced reading:
  https://www.cs.princeton.edu/courses/archive/fall10/cos597C/docs/memory-models.pdf

Happens-before relationship

- Formulated by Leslie Lamport in 1976
- Partial order relation between events (e.g., instructions) in a trace
- Denoted by a $\rightarrow$ b where a, b are events in a trace

- Consider a, b with a occurring before b in the trace:
  - If a, b are in the same thread, then a $\rightarrow$ b
  - If a is unlock(L) and b is lock(L), then a $\rightarrow$ b

    (can generalise for other synchronisation mechanisms)

- Irreflexive: $\forall$a, a $\not\rightarrow$ a
- Antisymmetric: $\forall$a, b: a $\rightarrow$ b then b $\not\rightarrow$ a
- Transitive: $\forall$a, b, c: a $\rightarrow$ b $\wedge$ b $\rightarrow$ c then a $\rightarrow$ c

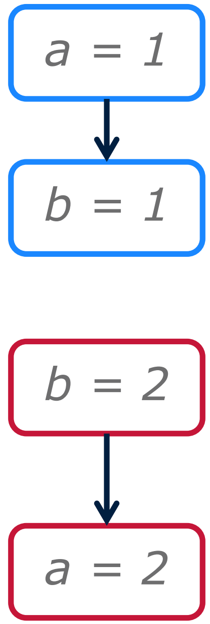
Happens-before relationship

- A data race occurs between a, b in the trace iff:
  - they access the same memory location
  - at least one of them is a write
  - they are unordered according to happens-before

Happens-before

```
int a, b; // shared
void  T1()
{
  a = 1;
  b = 1;
}

void T2()
{
  b = 2;
  a = 2;
}
```

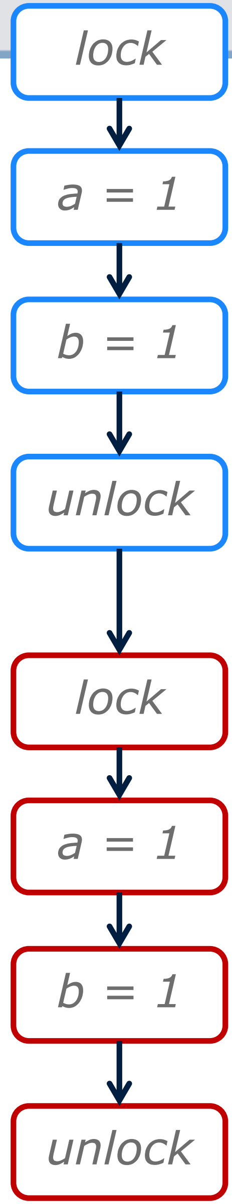


Date race between a =1, a=2
and between b = 1, b =2

Imperial College London

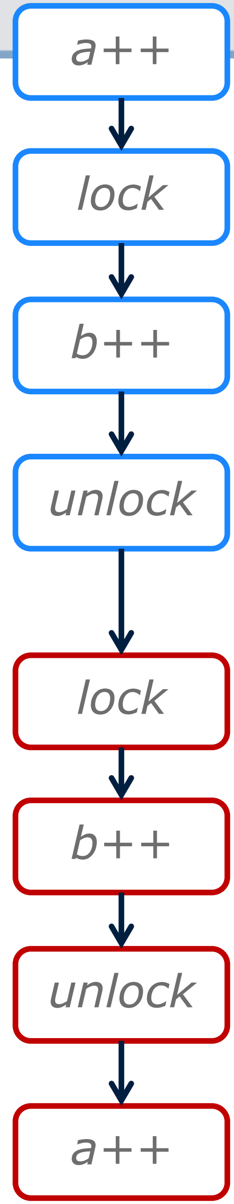
Happens-before

```
int a, b; // shared
void  T1()                 void T2()
{                          {
   lock(L);                   lock(L);
   a = 1;                     b = 2;
   b = 1;                     a = 2;
   unlock(L);                 unlock(L);
}                          }
```



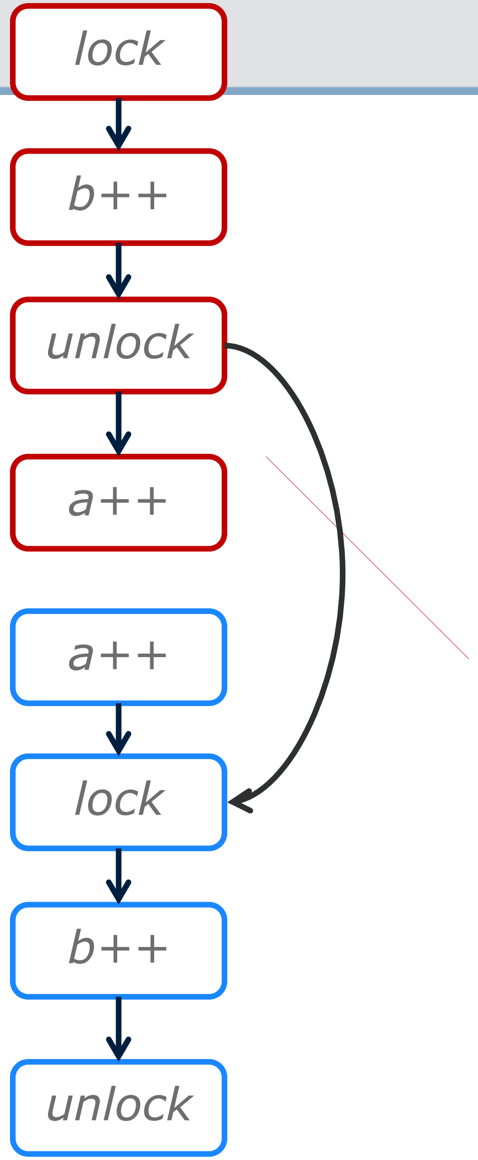
Happens-before

```
int a, b; // shared
void  T1()                void T2()
{                         {
   a++;                      lock(L)
   lock(L);                  b++;
   b++;                      unlock(L);
   unlock(L);                a++;
}                         }
```



Happens-before

```
int a, b; // shared
void  T1()              void T2()
{                       {
  a++;                    lock(L)
  lock(L);                b++;
  b++;                    unlock(L);
  unlock(L);              a++;
}                       }
```



Imperial College London

Semaphores

- Blocking synchronization mechanism invented by Dijkstra in 1965

- Idea: Processes will cooperate by means of *signals*
  - A process will stop, waiting for a specific signal
  - A process will continue if it has received a specific signal

- **Semaphores** are *special variables*, accessible via the following *atomic* operations:
  - **`down(s)`**: receive a signal via semaphore **s**
  - **`up(s)`**: transmit a signal via semaphore **s**
  - **`init(s, i)`**: initialise semaphore **`s`** with value **`i`**

- **down()** also called **P()** (*probeer te verlagen)*
- **up()** also called **V()** (*verhogen*)

Imperial College London

Semaphores

- Semaphores have two private components:
  - A counter (non-negative integer)
  - A queue of processes currently waiting for that semaphore

Semaphore Operations

```
init(s, i) ::= counter(s) = i
               queue(s) = {}
```

```
down(s) ::= if counter(s) > 0
               counter(s) = counter(s) - 1
            else
               add P to queue(s)
               suspend current process P
```

```
up(s) ::= if queue(s) not empty
             resume one process in queue(s)
          else
             counter(s) = counter(s) + 1
```

Semaphores: Mutual Exclusion

- **Binary semaphore:** counter is initialized to 1
- Similar to a lock/mutex

```
process A                          process B
  ...                                ...
  down(s)                            down(s)
    critical section                   critical section
  up(s)                              up(s)
end                                end


main() {
  var s:Semaphore
  ...
  init(s, 1) /* initialise semaphore */
  ...
    start processes A and B in random order
  ...
}
```

Semaphores: Ordering Events

Process A must execute its critical section before process B can execute its critical section

```
process A
   ...
      critical section
   up(s)
end
```

```
process B
   ...
   down(s)
      critical section
end
```

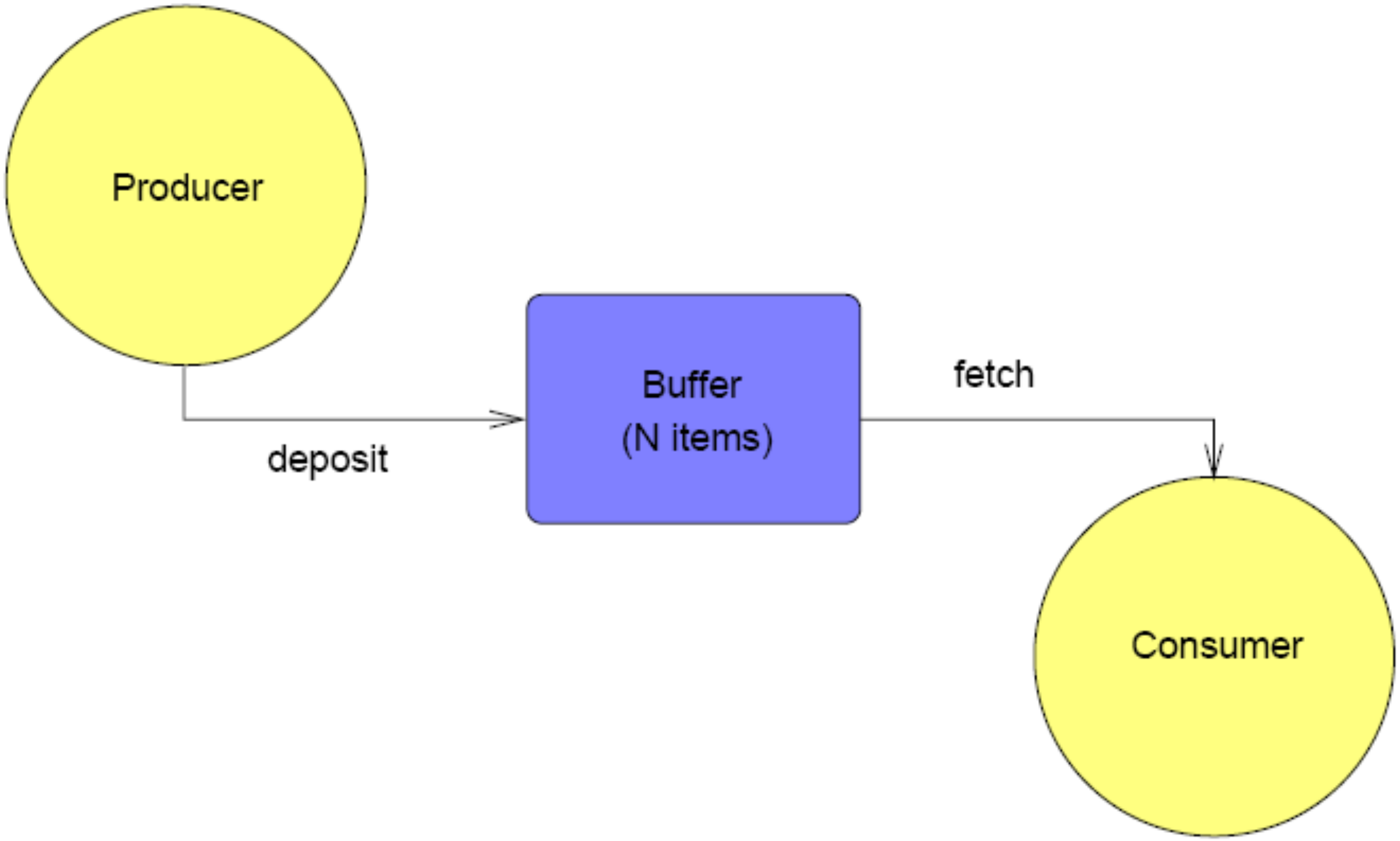
```
var s:Semaphore
...
init(s, 0) /* initialise semaphore */
...
   start processes A and B in random order
...
```

Imperial College London

General Semaphores

- The initial value of a semaphore counter indicates how many processes can access shared data at the same time
- `counter(s) >= 0:` how many processes can execute down without being blocked

Producer / Consumer



There can be multiple producers and consumers

Producer / Consumer

- Producer constraints:
  - Items can only be deposited in buffer if there is space
  - Items can only be deposited in buffer if mutual exclusion is ensured
- Consumer constraints:
  - Items can only be fetched from buffer if it is not empty
  - Items can only be fetched from buffer if mutual exclusion is ensured
- Buffer constraints:
  - Buffer can hold between **0** and **N** items

Producer/Consumer

```
var item, space, mutex:  Semaphore
init(item, 0)  /* Semaphore to ensure buffer is not empty */
init(space, N)  /* Semaphore to ensure buffer is not full */
init(mutex, 1)  /* Semaphore to ensure mutual exclusion */

procedure producer()
  loop
    produce item
    down(space)
    down(mutex)
    deposit item
    up(mutex)
    up(item)
  end loop
end producer

procedure consumer()
  loop
    down(item)
    down(mutex)
    fetch item
    up(mutex)
    up(space)
    consume item
  end loop
end producer
```

Imperial College London

Monitors

- Higher-level synchronization primitive
- Introduced by Hansen (1973) and Hoare (1974)
- Refined by Lampson (1980)

Monitors

- Shared data
- Entry procedures
  - Can be called from outside the monitor
- Internal procedures
  - Can be called only from monitor procedures
- An (implicit) monitor lock
- One or more condition variables

- Processes can only call entry procedures
  - cannot directly access internal data
- Only one process can be in the monitor at one time

Condition Variables

- Associated with high-level conditions
  - “some space has become available in the buffer”
  - “some data has arrived in the buffer”
- Operations:
  - **wait(c)**: releases monitor lock and waits for **c** to be signalled
  - **signal(c)**: wakes up one process waiting for **c**
  - **broadcast(c)**: wakes up all processes waiting for **c**
- Signals do not accumulate
  - If a condition variable is signalled with no one waiting for it, the signal is lost

What happens on signal?

[Hoare] A process waiting for signal is immediately scheduled

+ Easy to reason about
– Inefficient: the process that signals is switched out, even if it has not finished yet with the monitor
– Places extra constraints on the scheduler

[Lampson] Sending signal and waking up from a wait not atomic

– More difficult to understand, need to take extra care when waking up from a wait()
+ More efficient, no constraints on the scheduler
+ More tolerant of errors: if the condition being notified is wrong, it is simply discarded when rechecked (see next slides)

*Usually **[Lampson]** is used (including Pintos)*

Producer/Consumer with Monitors

```
monitor ProducerConsumer
    condition not_full, not_empty;
    integer count = 0;

    entry procedure insert(item)
        if (count == N) wait(not_full);
        insert_item(item); count++;
        signal(not_empty);

    entry procedure remove(item)
        if (count == 0) wait(not_empty);
        remove_item(item); count--;
        signal(not_full);
end monitor
```

Does this work?

Imperial College London

Producer/Consumer with Monitors

```
monitor ProducerConsumer
    condition not_full, not_empty;
    integer count = 0;

    entry procedure insert(item)
       while (count == N) wait(not_full);
       insert_item(item); count++;
       signal(not_empty);

    entry procedure remove(item)
       while (count == 0) wait(not_empty);
       remove_item(item); count--;
       signal(not_full);
end monitor
```

Monitors

- Monitors are a language construct
- Not supported by C

- Pintos
  - explicit monitor lock

- Java
  - synchronized methods
  - no condition variables
    - wait() and notify()

Recap

- Lock
  - Reader/writer locks
  - Often exposed with Monitor language construct
  - Within a process
  - 1 process/thread in critical section
- Mutex
  - Like lock, but can work across processes too
- Semaphore
  - Like mutex, but can let in N processes/threads

Bohr and Heisenbugs

- Bohrbugs:
  - Deterministic, reproducible bugs
  - Behave similar to Bohr’s atom model where electrons deterministically orbit the nucleus
- Heisenbugs
  - Non-deterministic, hard to reproduce bugs
    - Often caused by race conditions
  - Suffer from the observer effect (Heisenberg Uncertainty Principle): attempts to observe them (i.e., printfs) make them disappear!
- Which bug would you rather have?
  - During development/testing: _______
  - During deployment: _______

Tutorial

- Two threads in the same process can synchronize using a kernel semaphore:
  (1) Only if they are implemented by the kernel
  (2) Only is they are implemented in user space
  (3) Both if implemented by the kernel or in user-space