Imperial College
London

Operating Systems
Synchronisation II

Course 211
Spring Term 2018-2019

http://www.imperial.ac.uk/computing/current-students/courses/211/calendar/

(Slides courtesy of Cristian Cadar)

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

Semaphores

Blocking synchronisation mechanism invented by Dijkstra

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()

`up()` also called V()

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 initialised 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
```

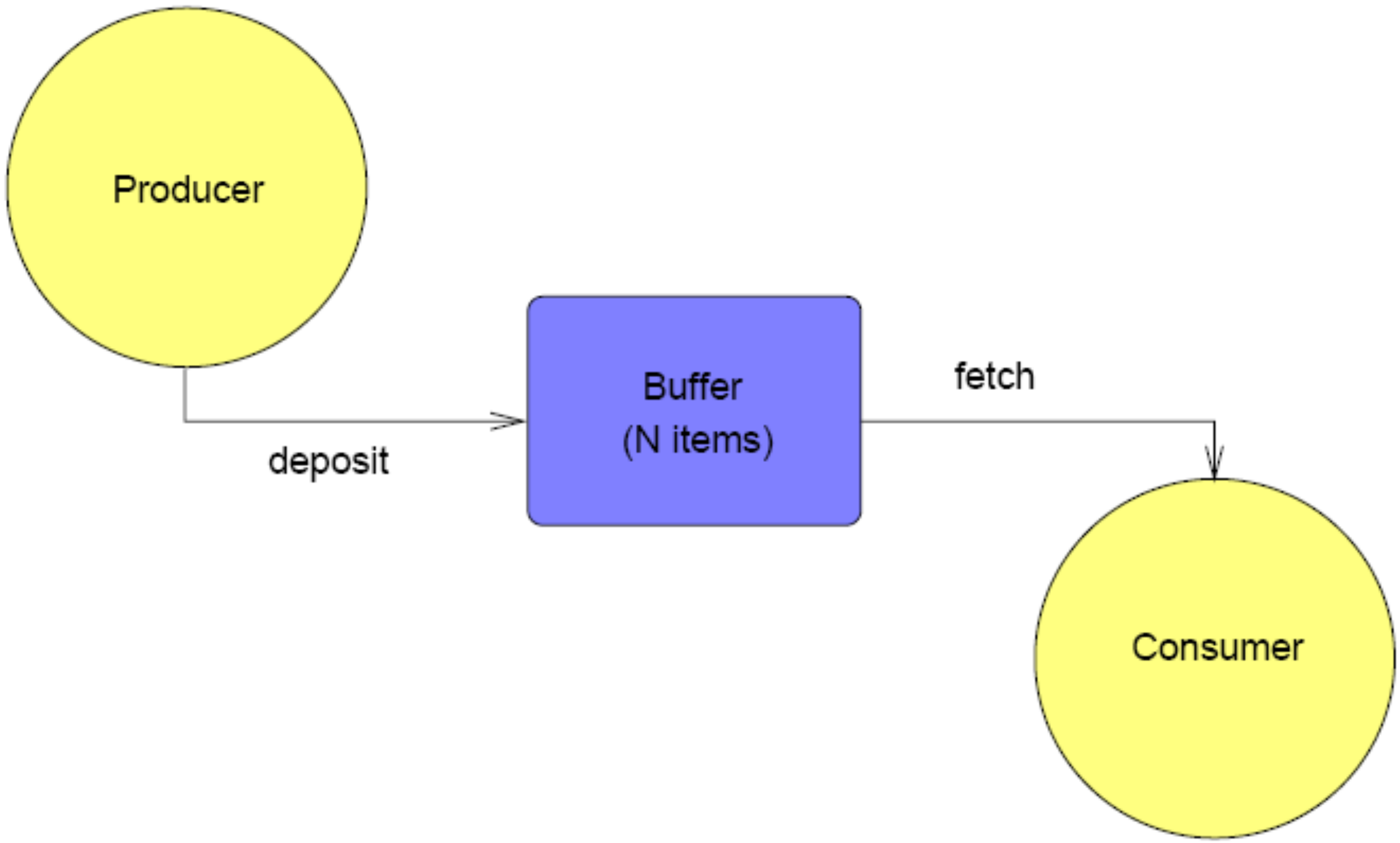
```
var s:Semaphore
...
init(s, 0) /* initialise semaphore */
...
   start processes A and B in random order
...
```

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
```

Monitors

Higher-level synchronisation 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

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?

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()

Summary

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

Tutorial Question

Two threads in the same process can synchronise 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