Course 02158

Monitor Variants

Hans Henrik Løvengreen

DTU Compute

Monitors

- Hoare/Brinch Hansen 1973:

  Monitor = data abstraction + atomicity + synchronization

Data abstraction

- Given by class construct: Private data variables + operations

Atomicity

- By implicit mutual (or R/W) exclusion among operations
- By explicit use of critical sections in operations

Synchronization

- By explicit *condition queues* (*wait*, *signal*)
  - Many variations in sematics
- By use of *guards* (**when** B)

Condition Queues

- Explicit mechanism for condition synchronization within monitors
- A condition queue is associated with a monitor, e.g. by declaration

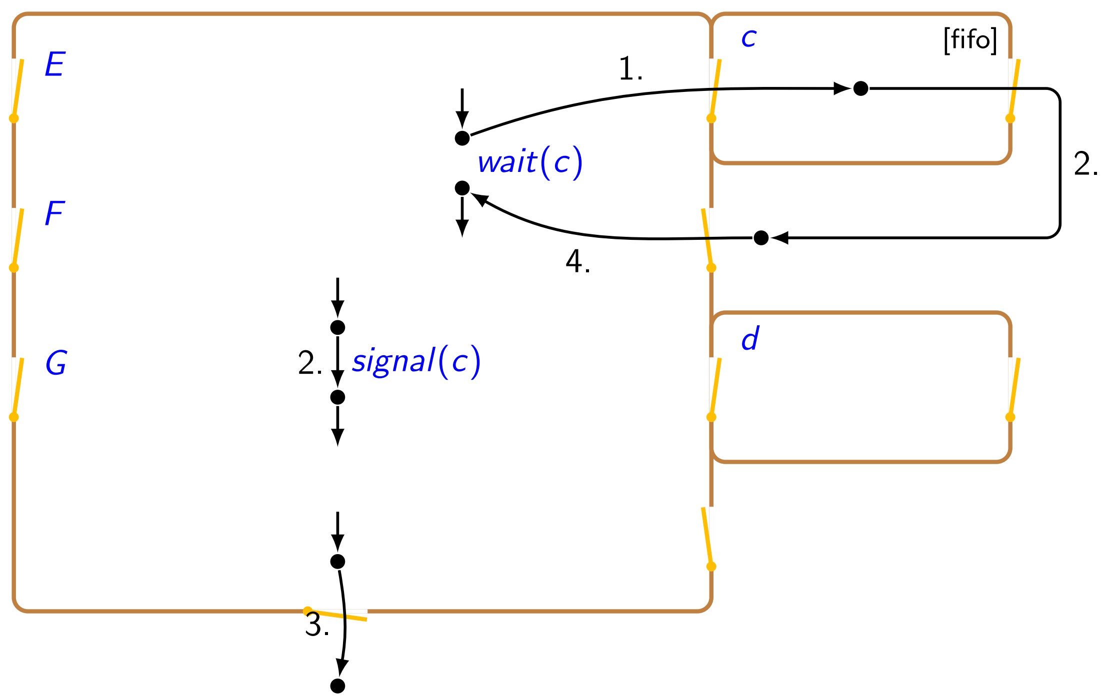
Basic queue operations

- *wait*(c) Leave monitor and enter c *atomically*
  signal(c) Wake up a single process waiting on c if any
  signal_all(c) Wake up all processes waiting on c [SC only]

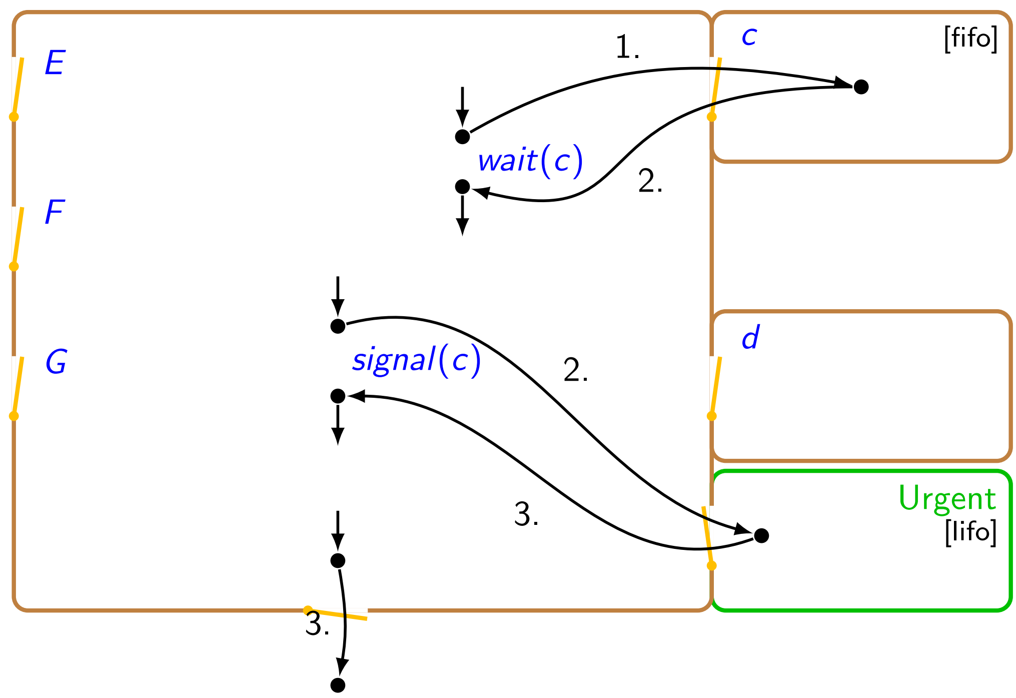
- Different semantics for signalling process:

  SC Signal-and-continue. Signaller continues in monitor
  SW Signal-and-wait. Woken process takes over monitor
  Hoare Signal-and-urgent-wait
  SW + signalling processes have priority to reenter

Monitor Access — signal-and-continue



Monitor Access — signal-and-waiturgent-wait (Hoare)



Barber Shop

- Using Hoare's semantics (*signal-and-urgent-wait*)
- **monitor** *Barber_Shop*

```
var barber_available : condition;
    condchair_occupied : condition;
    conddoor_open : condition;

procedure get_haircut()
  if ¬empty(chair_occupied) then wait(barber_available);
  signal(chair_occupied);
  wait(door_open);

procedure get_next_customer()
  if empty(barber_available) then wait(chair_occupied);
                              else signal(barber_available);

procedure finished_cut()
  signal(door_open);

end
```

Auxiliary Condition Queue Operations

- Given: **var** *c* : *condition*

Queue size

- *empty*(*c*) No processes are currently waiting on *c*
- Queue length can be maintained in explicit monitor variables.

Priority Queuing

- *wait*(*c*, *rank*) Wait in *c* according to *rank* (ascending)
  minrank(*c*) Return rank of first process waiting in *c*

Shortest-Job-Next Monitor

- **monitor** *SJN*

```
    var free : boolean := true;
        turn : condition;

    procedure request(time : integer)
      if free then
        free := false
      else
        wait(turn, time)

    procedure release()
      if empty(turn) then
        free := true
      else
        signal(turn)

end
```

Timer Monitor — covering condition

- **monitor** *Timer*

  var $tod : integer := 0;$
  $\quad check : condition;$

  procedure $delay(interval : integer)$
  $\quad$**var** $waketime : integer;$
  $\quad waketime := tod + interval;$
  $\quad$**while** $waketime > tod$ **do**
  $\quad\quad wait(check);$

  procedure $tick()$
  $\quad tod := tod + 1;$
  $\quad signal\_all(check)$

  end

Timer Monitor — utilizing priority wait

- **monitor** *Timer*

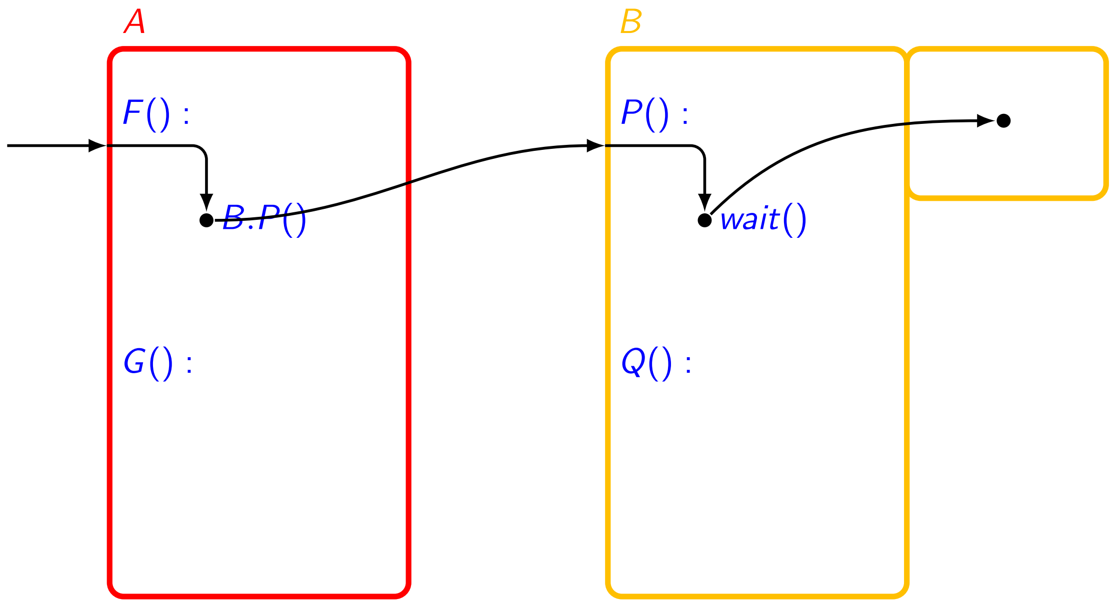
  var $tod : integer := 0;$
  $\quad check : condition;$

  procedure $delay(interval : integer)$
  $\quad$**var** $waketime : integer;$
  $\quad waketime := tod + interval;$
  $\quad$**if** $waketime > tod$ **then**
  $\quad\quad wait(check, waketime);$

  procedure $tick()$
  $\quad tod := tod + 1;$
  $\quad$**while** $\neg empty(check) \wedge rank(check) \leq tod$ **do** $signal(check)$

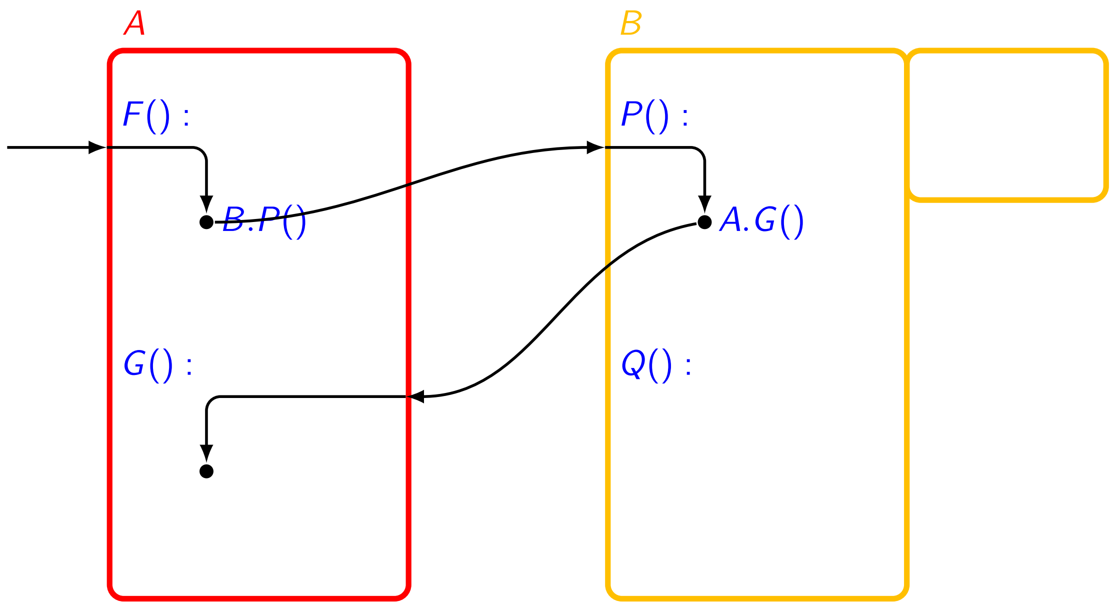
  end

Nested Monitor Calls — Wait



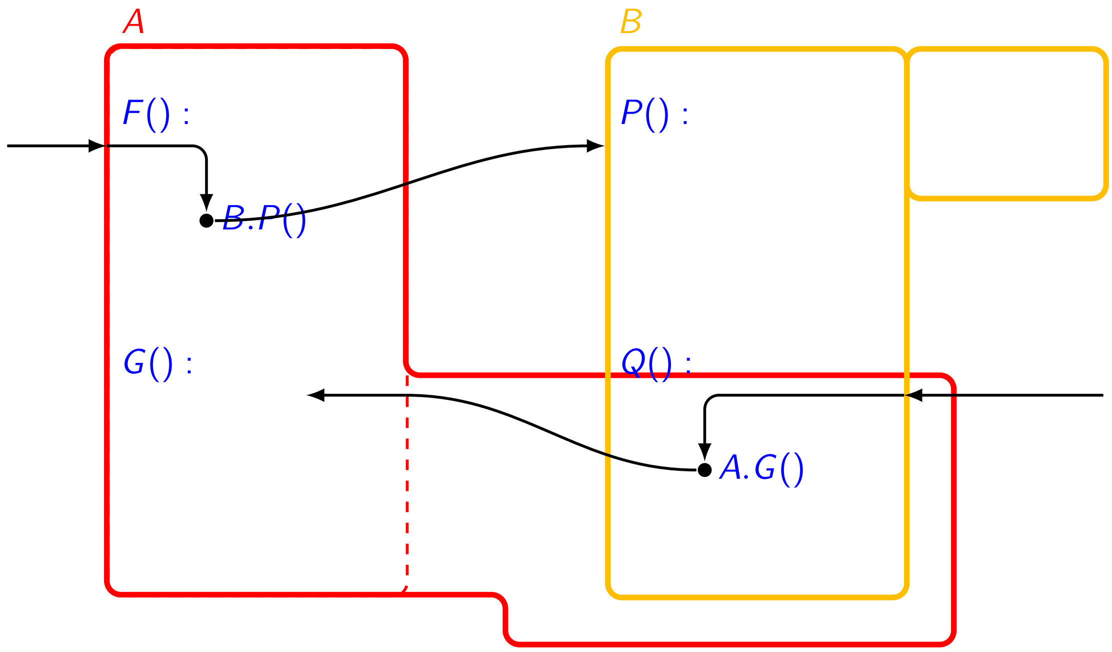
- Consensus: Only innermost monitor is released

Nested Monitor Calls — Callback



- Java has *reentrant*/*recursive locks*: Allows for callback

Nested Monitor Calls — Deadlock prevention



Monitors in Current Programming Langauges

Common

- No syntactic monitor construct (exception Ada)
- Critical sections or mutex locs
- Signal-and-continue semantics
- Conditions queues not ordered
- No length/empty operation on condition queues
- Many allow for *spurious wakeups*

Examples

- C$^{\#}$ $\sim$ Java
- Pthreads Mutex (non-recursive) + condition (dynamic)
- C++ Mutex (non-recursive) + condition (static)
- Python Condition (incl. lock)
- Rust Mutex + condition
- Go Condition (incl. lock)

Threading: Java vs. C$^\#$

Facility	Java	C$^\#$
Thread embedding	`Runnable` object	`void` $\rightarrow$ `void` procedure
Critical section	`synchronized` P`( ... )`	
	`synchronized (`o`) { ... }`	`lock (`o`) { ... }`
		`Monitor.Enter(`o`)`
		`Monitor.Exit(`o`)`
		`Monitor.TryEnter(`o`,`t`)`
Condition queue	o`.wait()`	`Monitor.Wait(`o`)`
	o`.notify()`	`Monitor.Pulse(`o`)`
	o`.notifyAll()`	`Monitor.PulseAll(`o`)`

C$^\#$ Semaphore Monitor

- ```
  class Sem {

    int S = 0;

    public void P() {
      lock(this) {
        while (S == 0) Monitor.Wait(this);
        S--;
      }
    }

    public void V() {
      lock(this) {
        S++;
        Monitor.Pulse(this);
      }
    }
  }
  ```
  ```

Pthreads

Background

- Around 1985: Many versions of commercial Unix system (but no Linux!)
  ATT&T System V, Berkeley Unix, HP-Unix, ...
- Many common notions, but also many variations
- POSIX: IEEE-lead standardization of a systems programming API
- Many systems "almost comply" with POSIX, e.g. Linux

POSIX Threads

- Standardized 1995 [IEEE 1003.1c]
- Defines *threads*, *mutexes* and *conditions*
- Also covers cancellation, real-time scheduling and thread-local data
- Adapts to prior *semaphore* standard
- Later standard [IEEEE 1003.1j] adds *barriers* and *r/w-locks*
- Standard for writing concurrent C programs
- C++ (11) has developed its own concurrency libraries

Pthread Mutex'es

- A synchronization primitive to establish a critical region

Operations

- `pthread_init`(*mutexp*, *attrp*)
- `pthread_lock`(*mutexp*)
- `pthread_unlock`(*mutexp*)
- *mutexp* is a pointer to a *handle* of type `pthread_mutex_t`
- *attrp* is usually `NULL`
- Pthread mutexes are **not** recursive

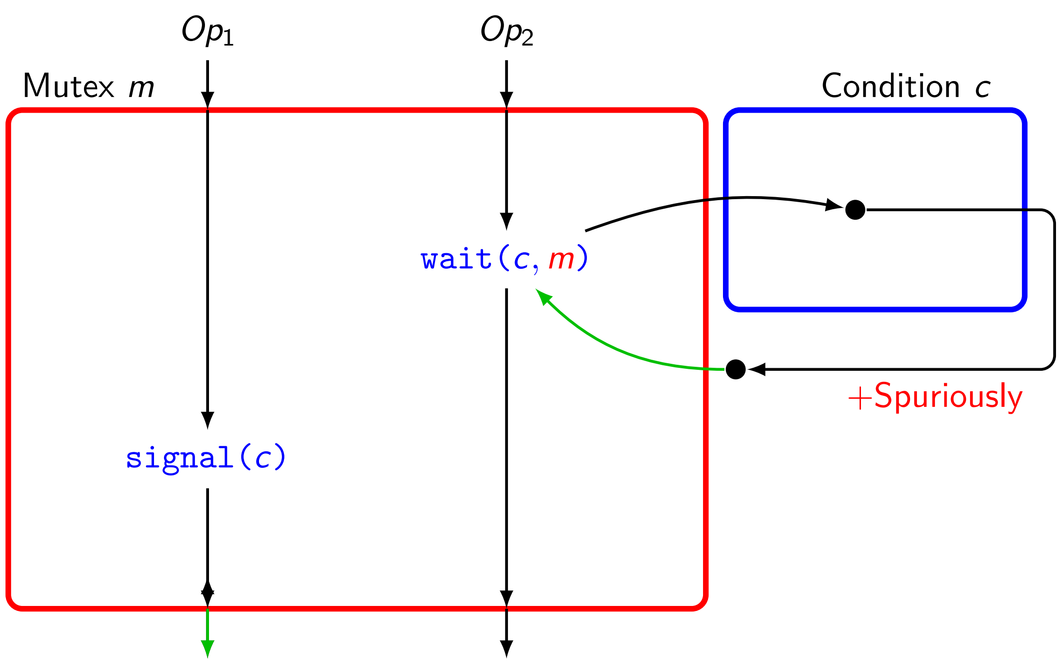
Pthread Condition Variables

- Waiting places associated with a (mutex-based) critical region

Operations

- `pthread_cond_wait`(*condp*, *mutexp*)
- `pthread_cond_signal`(*condp*)
- `pthread_cond_broadcast`(*condp*)
- Wait atomically leaves region and enters condition
- Signal prepares a thread for reentry to region
- Broadcast prepares all waiting threads for reentry to region
- No ordering
- *Spurious wakeups* may occur

Pthread Condition Variables



Pthread Monitor — Semaphore

```
int s;
pthread_mutex_t mutex;
pthread_cond_t pos;

void sem_init() { pthread_mutex_init(&mutex, NULL);
                  pthread_cond_init(&pos, NULL); s = 0; }

void sem_P() {
  pthread_mutex_lock(&mutex);
  while (s == 0) pthread_cond_wait(&pos, &mutex);
  s--;
  pthread_mutex_unlock(&mutex);
}

void sem_V() {
  pthread_mutex_lock(&mutex);
  s++;
  pthread_cond_signal(&pos);
  pthread_mutex_unlock(&mutex);
}
```

C++ Monitor — Semaphore

```
class sem {
private:
    int s = 0;
    std::mutex m;
    std::condition_variable pos;
public:
    P() {
        std::unique_lock<std::mutex> lock(m);
        pos.wait(lock, [] {return s > 0; } );
        s--;
    }

    V() {
        std::lock_guard<std::mutex> lock(m);
        s++;
        pos.notify_one();
    }
}
```

Concurrency Extensions in Java 1.5

`java.util.concurrent`

- Adoptation of Dough Lea's utility classes [Lea 00]
- Synchronization: *semaphores*, *barriers*, *latches*, *buffers*
- Execution control: *thread pools*, *futures*

`java.util.concurrent.atomic`

- Simple atomic operations on scalar values
- *Read-compare-update* operations for non-blocking synchr.
- May be implemented by monitors, OS, or hardware

`java.util.concurrent.locks`

- Basic locking mechanisms for critical regions (as in C#)
- Provides *condition queues* (as in Pthreads)
- Uses a primitive event-mechanism for implementation

Locks

- `Lock`: Common interface for mutex-like synchronization primitives

Operations

- For a *Lock* object `l` protecting a critical region:
- `l.lock()` Enter critical region
  `l.unlock()` Leave critical region
  `l.tryLock()` Try to enter region — abandon if occupied.
  `l.newCondition()` Get a condition queue attached to the region.
- Also timed and interruptible versions of the `lock` operation.
- Class *ReentrantLock* is an implementation of `Lock`

Reader/Writer Locks

- A *ReadWriteLock* is a reading lock coupled with a writing lock.

Condition

- Interface for condition queues associated with locks.

Operations

- For a *Condition* object `c` associated with region of lock `l`.
- `c.await()` Atomically leave region and enter queue `c`.
  `c.signal()` Wake a thread in `c` (if any) and continue.
  `c.signalAll()` Wake all threads in `c` and continue.
- Also timed and interruptible versions of `await`
- No means of inspecting queue (size etc.)
- Allows for *spurious wakeups* (due to idiosyncrasy of Pthreads)

Full Java Monitors

-
```
class BoundedBuffer {
  final Lock mutex = new ReentrantLock();
  final Condition notFull = mutex.newCondition();
  final Condition notEmpty = mutex.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    mutex.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      mutex.unlock();
    }
  }
}
```