procedure parse_page(address : url)
    contact server, request page contents
    parse_html_header()
    while current_token in {"<p>", "<h1>", "<ul>", ...},
        "<background", "<image", "<table", "<frameset", ...} 
    case current_token of
        "<p>"  : break_paragraph()
        "<h1>" : format_heading(); match(""</h1>"")
        "<ul>" : format_list(); match(""</ul>"")
    ... 
        "<background" : 
            a : attributes := parse_attributes()
            fork render_background(a)
        "<image" : a : attributes := parse_attributes()
            fork render_image(a)
        "<table" : a : attributes := parse_attributes()
            scan forward for ""</table>"" token
            token_stream s :=...  -- table contents
            fork format_table(s, a)
        "<frameset" : 
            a : attributes := parse_attributes()
            parse_frame_list(a)
            match(""</frameset>"")
    ...
...
procedure parse_frame_list(a1 : attributes)
    while current_token in {"<frame", "<frameset", "<noframes>"} 
    case current_token of
        "<frame" : a2 : attributes := parse_attributes()
            fork format_frame(a1, a2)
type task_descriptor = record
    -- fields in lieu of thread-local variables, plus control-flow information
    ... 
    ready_tasks : queue of task_descriptor
    ...

procedure dispatch()
    loop
        -- try to do something input-driven
        if a new event E (message, keystroke, etc.) is available
            if an existing task T is waiting for E
                continue_task(T, E)
            else if E can be handled quickly, do so
            else
                allocate and initialize new task T
                continue_task(T, E)
        -- now do something compute bound
        if ready_tasks is nonempty
            continue_task(dequeue(ready_tasks), 'ok')

procedure continue_task(T : task, E : event)
    if T is rendering an image
        and E is a message containing the next block of data
            continue_image_render(T, E)
    else if T is formatting a page
        and E is a message containing the next block of data
            continue_page_parse(T, E)
    else if T is formatting a page
        and E is 'ok' -- we're compute bound
            continue_page_parse(T, E)
    else if T is reading the bookmarks file
        and E is an I/O completion event
            continue_goto_page(T, E)
    else if T is formatting a frame
        and E is a push of the "stop" button
            deallocate T and all tasks dependent upon it
    else if E is the "edit preferences" menu item
        edit_preferences(T, E)
    else if T is already editing preferences
        and E is a newly typed keystroke
            edit_preferences(T, E)
<table>
<thead>
<tr>
<th></th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Distributed computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Java, C#</td>
<td>Go</td>
<td>Erlang</td>
</tr>
<tr>
<td></td>
<td>C/C++11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extension</td>
<td>OpenMP</td>
<td></td>
<td>Remote procedure call</td>
</tr>
<tr>
<td>Library</td>
<td>pthreads, Windows threads</td>
<td>MPI</td>
<td>Internet libraries</td>
</tr>
</tbody>
</table>
Thread scheduler

User space

Thread 1a  Thread 1b  ...  Thread 1k

Thread 1b  ...  Thread 1k

Thread Ma  Thread Mb  ...  Thread Mi

OS kernel

Process 1a  ...  Process 1i

Process 1i  ...  Process 1i

Process Ma  ...  Process Mj

Core 1  Core 2  ...  Core N

Process scheduler
type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
while not test_and_set(L)
while L
    -- nothing -- spin

procedure release_lock(ref L : lock)
L := false
shared count : integer := n
shared sense : Boolean := true
per-thread private local_sense : Boolean := true

procedure central_barrier()
    local_sense := not local_sense
        -- each thread toggles its own sense
    if fetch_and_decrement(count) = 1
        -- last arriving thread
        count := n
        -- reinitialize for next iteration
        sense := local_sense
        -- allow other threads to proceed
    else
        repeat
            -- spin
        until sense = local_sense
Initially: $X = Y = 0$

Core A:

$X := 1$

Core B:

$Y := 1$

Core C:

$cx := X$
$cy := Y$

Core D:

$dy := Y$
$dx := X$
Initially:  
\[ p : \text{foo} = \text{null} \]
\[ \text{initialized} : \text{volatile Boolean} = \text{false} \]

Thread A:
\[ p := \text{new foo()} \]
\[ \text{initialized} := \text{true} \]

Thread B:
\[ \text{repeat} \]
\[ -- \text{nothing} -- \text{spin} \]
\[ \text{until initialized} \]
\[ x := p.a \]
shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule()
    -- assume that scheduler_lock is already held and that timer signals are disabled
    t : thread
    loop
        t := dequeue(ready_list)
        if t \neq null
            exit
        -- else wait for a thread to become runnable
        release_lock(scheduler_lock)
        -- window allows another thread to access ready_list (no point in reenabling
        -- signals; we're already trying to switch to a different thread)
        acquire_lock(scheduler_lock)
        transfer(t)
    -- caller must release scheduler_lock and reenable timer signals after we return

procedure yield()
    disable_signals()
    acquire_lock(scheduler_lock)
    enqueue(ready_list, current_thread)
    reschedule()
    release_lock(scheduler_lock)
    reenable_signals()

procedure sleep_on(ref Q : queue of thread)
    -- assume that caller has already disabled timer signals and acquired
    -- scheduler_lock, and will reverse these actions when we return
    enqueue(Q, current_thread)
    reschedule()
type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
  while not test_and_set(L)
    count := TIMEOUT
  while L
    count :=:= 1
    if count = 0
      OS_yield() -- relinquish core and drop priority
      count :=:= TIMEOUT

procedure release_lock(ref L : lock)
  L := false
type semaphore = record
    N : integer    -- always non-negative
    Q : queue of threads
end

procedure P(ref S : semaphore)
    disable_signals()
    acquire_lock(scheduler_lock)
    if S.N > 0
        S.N := 1
    else
        sleep_on(S.Q)
    end
    release_lock(scheduler_lock)
    reenable_signals()
end

procedure V(ref S : semaphore)
    disable_signals()
    acquire_lock(scheduler_lock)
    if S.Q is nonempty
        enqueue(ready_list, dequeue(S.Q))
    else
        S.N := 1
    end
    release_lock(scheduler_lock)
    reenable_signals()
shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : bdata)
    P(empty_slots)
    P(mutex)
    buf[next_empty] := d
    next_empty := next_empty mod SIZE + 1
    V(mutex)
    V(full_slots)

function remove() : bdata
    P(full_slots)
    P(mutex)
    d : bdata := buf[next_full]
    next_full := next_full mod SIZE + 1
    V(mutex)
    V(empty_slots)
    return d
monitor bounded_buf
imports bdata, SIZE
exports insert, remove

buf : array [1..SIZE] of bdata
next_full, next_empty : integer := 1, 1
full_slots : integer := 0
full_slot, empty_slot : condition

entry insert(d : bdata)
  if full_slots = SIZE
    wait(empty_slot)
    buf[next_empty] := d
    next_empty := next_empty mod SIZE + 1
    full_slots +:= 1
    signal(full_slot)

entry remove() : bdata
  if full_slots = 0
    wait(full_slot)
  d : bdata := buf[next_full]
  next_full := next_full mod SIZE + 1
  full_slots -:= 1
  signal(empty_slot)
  return d
buffer : record
    buf : array [1..SIZE] of bdata
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0

procedure insert(d : bdata)
    region buffer when full_slots < SIZE
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots := full_slots + 1

function remove() : bdata
    region buffer when full_slots > 0
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots := full_slots + 1
    return d
procedure write(x : address, v : value)
    write_map[x] := v

procedure commit()
    try
        lock_map : map address → orece := ∅
done : Boolean := false
for x : address ∈ write_map.domain
    o : orece := oreces[hash(x)]
    if o ≠ ⟨true, me⟩
        if o.owned then throw abort
        if not CAS(&oreces[hash(x)], o, ⟨true, me⟩)
            throw abort
    lock_map[x] := o
n : time := 1 + fetch_and_increment(&clock)
validate()
done := true
for ⟨x, v⟩ : ⟨address, value⟩ ∈ write_map
    *x = v     -- write back
finally
    -- do this however control leaves the try block
    for ⟨x, o⟩ : ⟨address, orece⟩ ∈ lock_map
        oreces[hash(x)] := if done
            then ⟨false, n⟩     -- update
        else o           -- restore
procedure read(x : address) : value
    if x ∈ write_map.domain then return write_map[x]
loop
    repeat
        o : orece := oreces[hash(x)]
    until not o.owned
    t : time := o.val      -- when last modified
    if t > valid_time
        -- may be inconsistent with previous reads
        invalidate()          -- attempt to extend valid_time
    v : value := *x
    if o = oreces[hash(x)]
        read_set += {x}
    return v

procedure invalidate()
    t : time := clock
    for x : address ∈ read_set
        o : orece := oreces[hash(x)]
        if (not o.owned and o.val > valid_time)
            throw abort
        or (o.owned and o.val ≠ me)
        throw abort
    valid_time := t

struct orece
    owned : Boolean
    val : union (time, transaction_id)
Initially: inspected = false; X = 0

Core A:
inspected := true
xa := X

Core B:
X := 1
ib := inspected