COT 4600 Operating Systems Fall 2009

Dan C. Marinescu
Office: HEC 439 B
Office hours: Tu-Th 3:00-4:00 PM
Lecture 19

- Last time:
  - Enforcing Modularity in Memory

- Today:
  - Sharing a processor among multiple threads
  - Implementation of the YIELD
  - Creating and terminating threads
  - Preemptive scheduling

- Next Time:
  - Thread primitives for sequence coordination
Virtualization of threads

- Implemented by the operating system for the three abstractions:
  1. Threads → a thread is a virtual processor; a module in execution
     1. Multiplexes a physical processor
     2. The state of a thread: (1) the reference to the next computational step (the Pc register) + (2) the environment (registers, stack, heap, current objects).
  3. Sequence of operations:
     1. Load the module’s text
     2. Create a thread and lunch the execution of the module in that thread.
  4. A module may have several threads.
  5. The thread manager implements the thread abstraction.
     1. Interrupts → processed by the interrupt handler which interacts with the thread manager
     2. Exception → interrupts caused by the running thread and processed by exception handlers
     3. Interrupt handlers run in the context of the OS while exception handlers run in the context of interrupted thread.
### Basic primitives for processor virtualization

<table>
<thead>
<tr>
<th>Memory</th>
<th>CREATE/DELETE_ADDRESS SPACE</th>
<th>ALLOCATE/FREE_BLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP/UNMAP</td>
<td>UNMAP</td>
</tr>
<tr>
<td>Interpreter</td>
<td>ALLOCATE_THREAD</td>
<td>DESTROY_THREAD</td>
</tr>
<tr>
<td></td>
<td>EXIT_THREAD</td>
<td>YIELD</td>
</tr>
<tr>
<td></td>
<td>AWAIT</td>
<td>ADVANCE</td>
</tr>
<tr>
<td></td>
<td>TICKET</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACQUIRE</td>
<td>RELEASE</td>
</tr>
<tr>
<td>Communication channel</td>
<td>ALLOCATE/DEALLOCATE_BOUNDED_BUFFER</td>
<td>SEND/RECEIVE</td>
</tr>
</tbody>
</table>
The state of a thread and its associated virtual address space
Processor sharing

- Possible because threads spend a significant percentage of their lifetime waiting for external events.

- Called:
  - Time-sharing
  - Processor multiplexing
  - Multiprogramming
  - Multitasking

- The kernel must support a number of functions:
  - Creation and destruction of threads
  - Allocation of the processor to a ready to run thread
  - Handling of interrupts
  - Scheduling – deciding which one of the ready to run threads should be allocated the processor
Thread states and state transitions

- Thread states: NOT ALLOCATED, RUNNABLE, RUNNING, WAITING
- State transitions: ALLOCATE THREAD, EXIT THREAD, SCHEDULER, YIELD, ADVANCE, AWAIT
Switching the processor from one thread to another

- **Thread creation:**
  
  \[
  \text{thread\_id} \leftarrow \text{ALLOCATE\_THREAD}(\text{starting\_address\_of\_procedure, address\_space\_id});
  \]

- **YIELD** → function implemented by the kernel to allow a thread to wait for an event.
  - Save the state of the current thread
  - Schedule another thread
  - Start running the new thread – dispatch the processor to the new thread

- **YIELD**
  - cannot be implemented in a high level language, must be implemented in the machine language.
  - can be called from the environment of the thread, e.g., C, C++, Java
  - allows several threads running on the same processor to wait for a lock. It replaces the busy wait we have used before.
shared structure buffer
   message instance message[N]
   integer in initially 0
   integer out initially 0
   lock instance buffer_lock initially UNLOCKED

procedure SEND (buffer reference p, message instance msg)
   ACQUIRE (p_buffer lock)
   while p.in – p.out = N do /* if buffer full wait
      RELEASE (p_buffer lock)
      ACQUIRE (p_buffer lock)
      p.message [p.in modulo N] <-msg /* insert message into buffer cell
      p.in <- p.in + 1 /* increment pointer to next free cell
      RELEASE (p_buffer lock)
   return

procedure RECEIVE (buffer reference p)
   ACQUIRE (p_buffer lock)
   while p.in = p.out do /* if buffer empty wait for message
      RELEASE (p_buffer lock)
      ACQUIRE (p_buffer lock)
      msg<- p.message [p.in modulo N] /* copy message from buffer cell
      p.out <- p.out + 1 /* increment pointer to next message
   return msg
shared structure buffer

message instance message[N]
integer in initially 0
integer out initially 0
lock instance buffer_lock initially UNLOCKED

procedure SEND (buffer reference p, message instance msg)
    ACQUIRE (p_buffer_lock)
    while p.in – p.out = N do /* if buffer full wait */
        RELEASE (p_buffer_lock)
        YIELD()
        ACQUIRE (p_buffer_lock)
    p.message [p.in modulo N] \rightarrow msg /* insert message into buffer cell */
    p.in \leftarrow p.in + 1 /* increment pointer to next free cell */
    RELEASE (p_buffer_lock)

procedure RECEIVE (buffer reference p)
    ACQUIRE (p_buffer_lock)
    while p.in = p.out do /* if buffer empty wait for message */
        RELEASE (p_buffer_lock)
        YIELD()
        ACQUIRE (p_buffer_lock)
    msg \leftarrow p.message [p.in modulo N] /* copy message from buffer cell */
    p.out \leftarrow p.out + 1 /* increment pointer to next message */
    return msg
Implementation of YIELD

Thread table

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>Saved SP</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'2000'</td>
<td>RUNNABLE</td>
</tr>
<tr>
<td>1</td>
<td>X'10000'</td>
<td>RUNNING</td>
</tr>
<tr>
<td>4</td>
<td>X'10000'</td>
<td>RUNNING</td>
</tr>
<tr>
<td>6</td>
<td>X'4000'</td>
<td>RUNNABLE</td>
</tr>
</tbody>
</table>

Processor Table

<table>
<thead>
<tr>
<th>Processor Id</th>
<th>Thread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor A</td>
<td>1</td>
</tr>
<tr>
<td>Processor B</td>
<td>4</td>
</tr>
</tbody>
</table>
shared structure processor_table(7)
   integer thread_id
shared structure thread_table(7)
   integer topstack
   integer state
shared lock instance thread_table_lock

procedure GET_THREAD_ID() return processor_table(CPUID).thread_id

procedure YIELD()
   ACQUIRE (thread_table_lock)
   ENTER_PROCESSOR_LAYER(GET_THREAD_ID())
   RELEASE(thread_table_lock)
   return

procedure ENTER_PROCESSOR_LAYER(this_thread)
   thread_table(this_thread).state ← RUNNABLE
   thread_table(this_thread).topstack ← SP
   SCHEDULER()
   return

procedure SCHEDULER()
   j ← _GET_THREAD_ID()
   do
     j ← j+1 (mod 7)
     RELEASE(thread_table_lock)
     while thread_table(j).state ≠ RUNNABLE
     thread_table(j).state ← RUNNING
     processor_table(CPUID).thread_id ← j
     EXIT_PROCESSOR_LAYER(j)
     return

procedure EXIT_PROCESSOR_LAYER(new)
   SP ← thread_table(new).topstack
   return
More about thread creation and termination

- What if want to create/terminate threads dynamically → we have to:
  - Allow a thread to self-destroy and clean-up → EXIT_THREAD
  - Allow a thread to terminate another thread of the same application → DESTROY_THREAD

- What if no thread is able to run →
  - create a dummy thread for each processor called a processor_thread which is scheduled to run when no other thread is available
  - the processor_thread runs in the thread layer
  - the SCHEDULER runs in the processor layer
  - The procedure followed when a kernel starts

---

**Procedure** RUN_PROCESSORS()

```plaintext
for each processor do
  allocate stack and setup processor thread /*allocation of the stack done at processor layer
  shutdown ← FALSE
  SCHEDULER()
  deallocate processor_thread stack /*deallocation of the stack done at processor layer
  halt processor
```
Switching threads with dynamic thread creation

- Switching from one user thread to another requires two steps
  - Switch from the thread releasing the processor to the processor thread
  - Switch from the processor thread to the new thread which is going to have the control of the processor
  - The last step requires the SCHEDULER to circle through the thread_table until a thread ready to run is found

- The boundary between user layer threads and processor layer thread is crossed twice

- Example: switch from thread 1 to thread 6 using
  - YIELD
  - ENTER_PROCESSOR_LAYER
  - EXIT_PROCESSOR_LAYER
shared structure processor_table(7)
    integer topstack
    byte reference stack
    integer thread_id

shared structure thread_table(7)
    integer topstack
    integer state
    boolean kill_pr_continue
    byte reference stack

shared lock instance thread_table_lock

procedure GET_THREAD_ID() return processor_table(CPUID).thread_id
procedure YIELD()
    ACQUIRE(thread_table_lock)
    ENTER_PROCESSOR_LAYER(GET_THREAD_ID())
    RELEASE(thread_table_lock)
    return

procedure SCHEDULER()
    while shutdown = FALSE do
        ACQUIRE(thread_table_lock)
        for i from 0 until 7 do
            if thread_table(i).state = RUNNABLE then
                thread_table(i).state ← RUNNING
                processor_table(CPUID).thread_id ← i
                EXIT_PROCESSOR_LAYER(CPUID,i)
            if (thread_table(i).kill_or_continue = KILL) then
                thread_table(j).state = RUNNABLE
                thread_table(j).state ← FREE
                DEALLOCATE(thread_table(i).stack)
                thread_table(i).kill_or_continue ← CONTINUE
                RELEASE(thread_table_lock)
        return

procedure ENTER_PROCESSOR_LAYER(thread_id, processor)
    thread_table(thread_id).state ← RUNNABLE
    thread_table(thread_id).topstack ← SP
    SCHEDULER()
    return

procedure EXIT_PROCESSOR_LAYER(processor,thread_id)
    processor_table(processor).topstack ← SP
    SP ← thread_table(thread_id).topstack
    return