4. The OS Kernel
Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and weigh only 1.5 tons.

Popular Mechanics, 1949
Overview

- 4.1 Kernel Definitions and Objects
- 4.2 Queue Structures
- 4.3 Threads
- 4.4 Implementing Processes and Threads
  - Process and Thread Descriptors
  - Implementing the Operations
Overview

• 4.5 Implementing Synchronization and Communication Mechanisms
  – Semaphores and Locks
  – Building Monitor Primitives
  – Clock and Time Management
  – Communications Kernel
• 4.6 Interrupt Handling
Kernel Definitions and Objects

- Basic set of objects, primitives, data structures, processes
- Rest of OS is built on top of kernel
- Kernel defines/provides mechanisms to implement various policies
  - process and thread management
  - interrupt and trap handling
  - resource management
  - input/output
Queues

- OS needs many different queues
- Single-level queues
  - implemented as array
    - fixed size
    - efficient for simple FIFO operations
  - implemented as linked list
    - unbounded size
    - more overhead, but more flexible operations
- Priority queues
  - implemented as array of linked lists
FIFO Queues

Figure 4-2
Priority Queues

Figure 4-3(a)
Priority Queues

Figure 4-3(b)

To processes
Processes and Threads

- Process has one or more threads
- All threads in a process share:
  - memory space
  - other resources
- Each thread has its own:
  - CPU state (registers, program counter)
  - stack
- Implemented in user space or kernel space
- Threads are efficient, but lack protection from each other

Figure 4-4
Implementing Processes/Threads

• Process Control Block (PCB)
  - State Vector = Information necessary to run process \( p \)
  - Status
    • basic types: Running, Ready, Blocked
    • additional types: Ready_active, Ready_suspended
      Blocked_active, Blocked_suspended

Figure 4-5
Process and Thread States

- **State Transition Diagram**

![State Transition Diagram]

Figure 4-6
Process Operations: Create

Create(s0, m0, pi, pid) {
    p = Get_New_PCB();          pid = Get_New_PID();
    p->ID = pid;                p->CPU_State = s0;
    p->Memory = m0;             p->Priority = pi;
    p->Status.Type = 'ready_s';
    p->Status.List = RL;
    p->Creation_Tree.Parent = self;
    p->Creation_Tree.Child = NULL;
    insert(self-> Creation_Tree.Child, p);
    insert(RL, p);
    Scheduler();
}
Process Operations: Suspend

Suspend(pid) {
    p = Get_PCB(pid);
    s = p->Status.Type;
    if ((s == 'blocked_a') || (s == 'blocked_s'))
        p->Status.Type = 'blocked_s';
    else
        p->Status.Type = 'ready_s';
    if (s == 'running') {
        cpu = p->Processor_ID;
        p->CPU_State = Interrupt(cpu);
        Scheduler();
    }
}
Process Operations: Activate

Activate(pid) {
    p = Get_PCB(pid);
    if (p->Status.Type == 'ready_s') {
        p->Status.Type = 'ready_a';
        Scheduler();
    } else
        p->Status.Type = 'blocked_a';
}
Process Operations: Destroy

Destroy(pid) {
    p = Get_PCB(pid);    Kill_Tree(p);    Scheduler();
}
Kill_Tree(p) {
    for (each q in p->Creation_Tree.Child)
        Kill_Tree(q);
    if (p->Status.Type == 'running') {
        cpu = p->Processor_ID;
        Interrupt(cpu);
    }
    Remove(p->Status.List, p);
    Release_all(p->Memory);
    Release_all(p->Other_Resources);
    Close_all(p->Open_Files);
    Delete_PCB(p);
}
Synchronization/Communication

Semaphores, locks, monitors, etc. are resources

```c
Request(res) {
    if (Free(res))
        Allocate(res, self)
    else {
        Block(self, res);
        Scheduler();
    }
}

Release(res) {
    Deallocate(res, self);
    if (Process_Blockedin(res, pr)) {
        Allocate(res, pr);
        Unblock(pr, res);
        Scheduler();
    }
}
```
Semaphores/Locks

• Need special **test_and_set** instruction: \( \text{TS}(R,X) \)

• Behavior: \( R = X; X = 0; \)
  - always set variable \( X=0 \)
  - register \( R \) indicates change:
    • \( R=1 \) if \( X \) changed (1→0)
    • \( R=0 \) if \( X \) did not change (0→0)

• TS is indivisible (atomic) operation
Spin/Spinning Locks

• Binary semaphore \( sb \) (only 0 or 1)

• Behavior of \( Pb/Vb \):
  
  \[
  Pb(sb): \quad \text{if } (sb==1) \quad sb=0; \quad \text{else wait}
  \]
  
  \[
  Vb(sb): \quad sb=1;
  \]

• Indivisible implementation of \( Pb/Vb \):
  
  \[
  Pb(sb):
  \]
  
  \[
  \text{do (TS(R,sb)) while (!R);} /*wait loop*/
  \]
  
  \[
  Vb(sb): \quad sb=1;
  \]

  Note: \( sb \) is shared, but each process has its own \( R \)

• “Spinning” = “Busy Waiting”
### General Semaphores w/ Busy-Wait

**P(s)** {
  Inhibit_Interrupts;
  Pb(mutex_s);
  s = s-1;
  if (s < 0) {
    Vb(mutex_s);
    Enable_Interrupts;
    Pb(delay_s);
  }
  Vb(mutex_s);
  Enable_Interrupts;
}

**V(s)** {
  Inhibit_Interrupts;
  Pb(mutex_s);
  s = s+1;
  if (s <= 0)
    Vb(delay_s);
  else
    Vb(mutex_s);
  Enable_Interrupts;
}

- Inhibiting interrupts prevents deadlock due to context switching
- `mutex_s` needed to implement critical section with multiple CPUs
Avoiding the Busy-Wait

\( P(s) \) {
    Inhibit_Interrupts;
    \textbf{Pb}(mutex_s);
    s = s-1;
    \textbf{if} (s < 0) \{ /*Context Switch*/
        Block(self, Ls);
        \textbf{Vb}(mutex_s);
        Enable_Interrupts;
        Scheduler();
    \} \textbf{else} \{
        \textbf{Vb}(mutex_s);
        Enable_Interrupts;
    \}
}

\( V(s) \) {
    Inhibit_Interrupts;
    \textbf{Pb}(mutex_s);
    s = s+1;
    \textbf{if} (s <= 0) \{
        Unblock(q, Ls);
        \textbf{Vb}(mutex_s);
        Scheduler();
    } \textbf{else} \{
        \textbf{Vb}(mutex_s);
        Enable_Interrupts;
    \}
}

Implementing Monitors

• Need to insert code to:
  – guarantee mutual exclusion of procedures (entering/leaving)
  – implement \texttt{c.wait}
  – implement \texttt{c.signal}

• Implement 3 types of semaphores:
  1. \texttt{mutex}: for mutual exclusion
  2. \texttt{condsem\_c}: for blocking on each condition \texttt{c}
  3. \texttt{urgent}: for blocking process after \texttt{signal}
Monitor Primitives

- Each procedure guarded by \textit{mutex} Semaphore

\begin{verbatim}
P(mutex);
procedure_body;
if (urgentcnt)
  V(urgent);
else
  V(mutex);
\end{verbatim}
Monitor Primitives

- **c.wait:**
  
  ```
  condcnt_c = condcnt_c + 1;
  if (urgentcnt)
      V(urgent);
  else
      V(mutex);
  P(condsem_c);
  condcnt_c = condcnt_c - 1;
  ```

- **c.signal:**
  
  ```
  if (condcnt_c) {
      urgentcnt = urgentcnt + 1;
      V(condsem_c);
      P(urgent);
      urgentcnt = urgentcnt - 1;
  }
  ```
Clock and Time Management

- Most systems provide hardware
  - *ticker*: issues periodic interrupt
  - *countdown timer*: issues interrupt after a set number of ticks
- Build higher-level services with this hardware
- Wall clock timers
  - typical functions:
    - `Update_Clock`: increment `tnow`
    - `Get_Time`: return current time
    - `Set_Time(tnew)`: set time to `tnew`
  - must maintain *monotonicity*
Clock and Time Management

- Countdown timers
  - main use: as alarm clocks
  - typical function:
    - `Delay(tdel)` block process for `tdel` time units
  - implementation using hardware countdown:
    ```c
    Delay(tdel) {
        Set_Timer(tdel); /*set hardware timer*/
        P(delsem); /*wait for interrupt*/
    }
    Timeout() { /*called at interrupt*/
        V(delsem);
    }
    ```
Clock and Time Management

• Implement *multiple* logical countdown timers using a single hardware timer

• Functions
  - `tn = Create_LTimer()` create new timer
  - `Destroy_LTimer(tn)`
  - `Set_LTimer(tn, tdel)` block process and call `Timeout()` at interrupt
Clock and Time Management

- Implement \texttt{Set_LTimer()} using \textit{Absolute Wakeup Times}
  - priority queue \textit{TQ} records wakeup times
  - function of \texttt{Set_LTimer}(tn, tdel)
    - compute absolute time of \texttt{tdel} (using wall-clock)
    - insert new request into \textit{TQ} (according to time)
    - if new request is earlier than previous head of queue, set hardware countdown to \texttt{tdel}
Clock and Time Management  5

- Absolute Wakeup Times Example:

  \texttt{Set\_LTimer(tn,35)}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4-8.png}
\caption{Figure 4-8}
\end{figure}
Clock and Time Management

- Implement `Set_LTTimer()` using *Time Differences*
  - priority queue $TQ$ records only time increments, no wall-clock is needed
  - function of `Set_LTTimer(tn, tdel)`
    - find the two elements $L$ and $R$ between which $tdel$ is to be inserted (add differences until $tdel$ is reached)
    - split the current difference between $L$ and $R$ into difference between $L$ and new element and difference between new element and $R$
Clock and Time Management

- Time Differences Example: \texttt{Set_LTimer}(tn, 35)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4-9.png}
\caption{Figure 4-9}
\end{figure}
Communication Primitives

• **Message Passing** as basic form of communication between processes/threads
  – most natural with processes executing in separate address spaces or machines
  – heavily used in distributed, cluster computing environment (MPI library)

• Two generic basic operations:
  – `send(p, m)` send message `m` to process `p`
  – `receive(q, m)` receive a message `m` from process `q`
  – either can be **blocking** or **non-blocking**
  – or (non-)selective (specify process explicitly)
In a shared-memory system:

- **send** and **receive** each use a buffer to hold message
  - in the user-space of sender/receiver
SM-Message Passing

• System copies message from **sbuf** to **rbuf** upon a send call

• Important issues:
  – 1. How does sender know that **sbuf** may be reused (overwritten)?
  – 2. How does system know that **rbuf** may be reused (overwritten)?

• Possible solutions include blocking the sender/receiver
  – completely synchronous if both
SM-Message Passing

• Reusing *sbuf*:
  – use blocking send:
    • reuse as soon as *send* returns
    • sender blocked even if *sbuf* not reused!
  – provide a flag (or interrupt) to indicate release of *sbuf*
    • sender needs to poll flag!

• Reusing *rbuf*:
  – provide a flag to system to indicate release of *rbuf*
    • receiver must explicitly set flag and system must test repeatedly!

• Solutions are awkward
SB-Message Passing

- Better solution: use pool of system buffers
  - send and receive each use intermediate system buffer(s) to hold message(s)
    - system simply copies messages from/to user buffers sbuf/rbuf to/from system buffers
SB-Message Passing

- **Send** copies `sbuf` to a system buffer
  - sender is free immediately after copy is made
    - non-blocking with large pool of buffers
  - sender may continue generating messages
    - until no more system buffers are available (block!)
    - selective sends include send/recv process IDs

- System copies (or reallocates) full buffers to receiver
  - asynchronous
SB-Message Passing

- *Receive* copies a system buffer to `rbuf`
  - frees *consumed* system buffer
  - receiver decides when to reuse `rbuf` for next message

- Blocking if no more system buffers with messages are available
  - non-blocking: returns corresponding error condition
Networked Message Passing

- Use of system buffers important in distributed systems
  - no shared memory between CPUs
  - large delays in message transmissions
  - extra processing:
    - packeting, routing, error handling etc.

Figure 4-10c
Interrupts

- **Unpredictable** event that forces transfer of control from running process
  - abstraction and handling of asynchronous events
  - *external*: generated by hardware (asynchronous)
    - I/O completion or GUI handling
  - *internal*: consequence of current computation (synchronous)
    - exceptional (error) conditions (also called *exceptions* or *traps*)
Interrupt Handling

Example: external hardware device accessed by process $p$ via procedure $Fn$

- $Fn$ initiates device and returns results to caller
  - blocks itself to await completion of device
- OS schedules other processes while device busy
- upon termination of device an interrupt is generated
  - $Fn$ has to be unblocked and completed
Interrupt Handling

- Standard interrupt-handling sequence:
  1. save state of interrupted process/thread (q)
  2. identify interrupt type and invoke IH
  3. IH services interrupt
  4. restore state of interrupted process (q or of another one p waiting for that interrupt)

Figure 4-11a
Interrupt Handling

Main challenges:

- $Fn$ must be able to block itself on a given event
  - lowest-level device functions $Fn$ generally provided by OS
- $IH$ must be able to unblock $p$ and to return from interrupt
  - unblock and make process ready to run which was waiting for the device
  - $IH$ is generally provided by OS
- Abstract $Fn$ and $IH$ more from application
Interrupt Handling

- View hardware device as a process:
  - implement $Fn$ and $IH$ as monitor functions
  - $Fn$ waits on $c$
    - after initializing device
  - $IH$ invoked by hardware process
    - $IH$ signals $c$
    - condition $c$ is associated with device

Figure 4-11b