Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority

- Inter-process communication (signals, messages)
  - Realization – shared memory, mailbox, rendezvous, RPC

- Scheduling algorithm parameterized
  - mechanism in the kernel

- Parameters filled in by user processes
  - policy set by user process
Kernel Operations

1. Primitives for process creation, destruction, and the basic interprocess communication
2. Primitives for allocating and releasing units of various resources such as main memory, secondary storage, files, IO devices, networking facilities.
3. Input and Output primitives; essentially read, write and control operations for initiating and supervising data transfers between main memory and peripheral devices.
4. Operations to handle interrupts triggered by various system events such as IO completion, process termination, service requests, error conditions or hardware malfunction.

Fig 3-1: A process creation hierarchy
**Software structure**

- User 1
- User 2
- 
- User n
- (Graphical) User Interface
- Applic.-progr. 1
- utility-program
- SW-tool
- Applic.-progr. n
- Operating System Functions
- Operating System Kernel
- Hardware

**Microkernel Architecture (Mach)**

- User 1
- User 2
- 
- User n
- Interface to system calls and functions
- Memory Management
- Terminal I/O
- File System
- Micro-kernel: Scheduler, Interprocess Communic. Basic I/O
- Hardware
Descriptors

- **Process descriptors**
  - ""control block" (* task/process/thread)
  - Created used and modified by the basic process and resource operations
  - descriptor categories:
    - Identification
    - State vector
    - Resources (memory, SRs & CRs)
    - Status information (running/ready/blocked)
    - Other information
Descriptors

- **Resource descriptors**
  - Inventory of available units
  - Waiting list of blocked processes
  - Allocator for assignment decisions (scheduler)
- **Descriptor categories**
  * Identification, Type, Origin
  * Inventory list
  * Waiting process list
  * Allocator routine
  * Additional information
### Waiting Lists - Queues

- **Logical structure** - defined by the “policy”
  - Description by an ADT (abstract data type)
  - May be seen as an abstract object (class)

- **Representation - implementation strategy**
  - Number of different implementation techniques
  - Implementation should be supported by hardware
  - Pointers, arrays, linked lists, hash tables etc. are potential implementation choices (mechanism)
  - Can be implemented in an object-oriented style
  - Crucial for the efficiency
Bounded buffer FIFO

Linked list FIFO
Doubly linked list FIFO

Fig 3-4: organization of a priority queue
Static Priority Queues

Dynamic Priority Queues

(b)
**Dynamic Priority Queues**

**Heap Implementation**

1. \[ i: \rightarrow \begin{cases} 2 \cdot i \\ 2 \cdot i + 1 \end{cases} \]

**Process Control Block - PCB**

- **Processors**
  - ID
  - CPU_State
  - Processor_ID

- **Resources**
  - Memory
  - Open_Files
  - Other_Resources

- **Status**
  - Type | List
  - Parent | Child

- **Creation_Tree**
  - Priority

- **Other information**
Process States (1)

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Basic Operations on Processes

- Create
- Destroy
- Activate
- Suspend
Basic Process States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>running</td>
<td>$p$ is currently running on a processor (the one designated by $p$-&gt;Processor_ID)</td>
</tr>
<tr>
<td>ready</td>
<td>$p$ is ready to run, waiting for a processor</td>
</tr>
<tr>
<td>blocked</td>
<td>$p$ cannot logically proceed until it receives a particular resource for example, a lock, file, table, message, I/O device, or semaphore</td>
</tr>
</tbody>
</table>

Basic Operations on Processes

1. **Create:** establish a new process
2. **Destroy:** remove one or more processes
3. **Suspend:** change process status to "suspended"
4. **Activate:** change process status to "active"
5. **Change_priority:** set a new priority for the process
Task, Process, Thread

- Task – an executing “job” – in the more narrow sense either a process or a thread:
  
- “process” – a program in execution, “owns” resources such as main memory, disk storage, devices, and temporarily also the CPU

- A process switch is an expensive operation (heavyweight processes)

- Allowing parallel processes which share the expensive resources such as memory, files, devices etc. leads to “lightweight” processes = “threads”

- Threads run concurrently within the context of one process and share the state vector

Process and Threads of Execution
Basic Operations on Processes

```c
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(); }
```
Basic Operations on Processes

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()
    }
} //Suspend

Basic Operations on Processes

Activate(pid) {
    p = Get PCB(pid);
    if (p-> Status.Type == 'ready s') {
        p-> Status.Type = 'ready a';
        Scheduler();
    }
    else p -> Status.Type = 'blocked a';
}
Basic Operations on Processes

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);
    }
}

continuation - next slide

Basic Operations on Processes

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);
}
Linux Process States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>running</td>
<td>$p$ is either using or waiting for the CPU; Thus this state jointly represents the two states we defined as running and ready; the distinction is implied by the assignment of the CPU</td>
</tr>
<tr>
<td>interruptible</td>
<td>$p$ is blocked on a resource; this corresponds to our blocked state; when the resource becomes available, $p$ is moved to the running state</td>
</tr>
<tr>
<td>stopped</td>
<td>$p$ has been explicitly suspended; this corresponds jointly to our two states ready and blocked</td>
</tr>
<tr>
<td>zombie</td>
<td>$p$ has terminated its execution but its PCB is kept active so that its parent can obtain information about $p$'s status</td>
</tr>
</tbody>
</table>

Linux process creation hierarchy
Processes and Threads

process p

th1

th2

PC1

PC2

stack 1

stack 2

shared data

process q

th1

th2

thn

PC1

PC2

PCn

stack 1

stack 2

stack n

shared data

running

ready_a

ready_s

blocked_a

blocked_s

Request

Scheduler

Activate

Release

Suspension

process state changes
Basic Operations on Resources

1. Create_RC: create the descriptor for a new resource class
2. Destroy_RC: destroy the descriptor for a resource class
3. Request: request some units of a resource class
4. Release: release some units of a resource class

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

Release(res) {
    Deallocate(res, self);
    if (Process Blocked on(res, pr)) {
        Allocate(res, pr);
        Unblock(pr, res);
        Scheduler();
    }
}

Examples

• Some common applications:

• Semaphore operations
  – P(s) – Request(s, W, W);
  – V(s) – Release(s, W);

• Message Passing
  – Transmit: - Release(M, (p, m));
  – Receive – Request(M, p, m);

• Main Storage Allocation:
  – Allocate: Request(Main_Storage, No_words, Base_address)
  – Free: Release(Main_Storage, (Base_address, No_words))
Mechanisms and Implementations

- P() and V() operations on semaphores
- Operations embedded in monitor procedures
- Clock / timer functions
- Send / Receive primitives

Semaphores and Locks

Required are un-interruptible, indivisible (atomic) instructions
(I.e. hardware support)
e.g. TestAndSet, CompareAndSwap ...
Disabling and Enabling interrupts.

TestAndSet: \( TS(R, X) \)

\[ \begin{align*}
R & := X; \\
x & := 0;
\end{align*} \]
i.e. register \( R \) stores the previous value of variable \( X \) and \( X \) is set to 0.
Semaphores

- **Binary semaphores** - only two values
  - 1 - available
  - 0 - allocated
  - (spin locks – busy wait)

- **Counting semaphores** - integer values
  - n >= 0 - available
  - n < 0 - pending requests (waiting queue)
  - (blocking – context switch)

Simple Semaphore Operations

```c
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;
}
```
**Simple Semaphore Operations**

\[
V(s) \{
    \text{Inhibit Interrupts;}
    Pb(\text{mutex } s);
    s = s + 1;
    \text{if } (s \leq 0) \text{ Vb(delay } s)\text{; }
    \text{else Vb(\text{mutex } s)};
    \text{Enable Interrupts;}
\}
\]

**Blocking Semaphore Operations**

\[
P(s) \{ \text{Inhibit Interrupts;}
    Pb(\text{mutex}_s);
    s = s - 1;
    \text{if } (s < 0) \{ /*Context Switch*/
        \text{Block(\text{self, } Ls)};
        \text{Vb(\text{mutex}_s)};
        \text{Enable Interrupts;}
        \text{Scheduler();}
    \}
    \text{else }
        \text{Vb(\text{mutex}_s)};
        \text{Enable Interrupts;}
\}
\]
Blocking Semaphore Operations

V(s) {  
  Inhibit Interrupts;
  Pb(mutex_s);
  s = s+1;
  if (s <= 0) {
    Unblock(q, Ls);
    Vb(mutex_s);
    Enable_Interrupts;
    Scheduler();
  }
  else {
    Vb(mutex_s);
    Enable Interrupts;
  }
}

Monitors

Required are four primitives for:

- Entering
- Leaving
- Waiting
- Signaling

this can be implemented using low-level semaphores; the general Hoare monitor can be implemented using 3 types of semaphores – restrictions can lead to more efficient implementations.
Monitors

Every monitor procedure must be "wrapped" in a protective envelop:

```c
P(mutex);
// protective entry ceremony
procedure body;
// protective exit ceremony
if (urgentcnt) V(urgent);
else V(mutex);
```

Monitors

Each `c.wait` within a procedure body has to be coded as:

```c
condcnt_c = condcnt_c + 1;
if (urgentcnt) V(urgent);
else V(mutex);
P(condsem_c); /* The process waits here. */
condcnt_c = condcnt_c - 1;
```
Monitors

Each *c.signal* in a monitor has to be replaced by the code:

```c
if (condcnt_c) {
    urgentcnt = urgentcnt + 1;
    V(condsem c); P(urgent);
    urgentcnt = urgentcnt - 1;
}
```

Clocks and Timers

<table>
<thead>
<tr>
<th>hardware-timers</th>
<th>Wall-clock</th>
<th>Count-down</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

organization of a timer queue
**Clocks and Timers**

Count-down timer queue with differences (delta timer)

**IPC – Message Passing**

Message transfer – copying user buffers directly
message transfer – copying via system buffers

message transfer – copying across network
Exceptions and Interrupts

- **Exception types**
  - External interrupts – asynchronous
  - Internal traps and contingencies – synchronous and asynchronous

- **Low-level exchange operation**
  - $P \rightarrow IH \rightarrow q$ context switch
  - Save / restore process state
  - Return immediate or through scheduler
Basic interrupt service:

1. Hardware device calls `F()`.
2. `F()` calls `Init_block`.
3. `Init_block` calls `IH()`.
4. `IH()` calls `Unblock p`.
5. `Unblock p` calls `Return_From_Interrupt`.
6. `Return_From_Interrupt` calls `Run some other process in the meantime`.
7. `Run some other process in the meantime` calls `OS`.
8. `OS` calls `IH()`.
9. `IH()` calls `OS`.
10. `OS` calls `F(n)`.
11. `F(n)` returns.

Monitor device interface:

1. Hardware device calls `F()`.
2. `F()` calls `Init c.wait`.
3. `Init c.wait` calls `IH()`.
4. `IH()` calls `Unblock p`.
5. `Unblock p` calls `c.signal`.
6. `c.signal` calls `c.wait`.
7. `c.wait` calls `OS`.
8. `OS` calls `IH()`.
9. `IH()` calls `OS`.
10. `OS` calls `F(n)`.
11. `F(n)` returns.