Programs, Processes, and Threads

- Process Management
  - What is a process?
  - How to control processes.
  - How to allocate the available resources to the execution of the processes (scheduling)
  - How to coordinate processes among themselves (synchronization)

Processes and Process Control

- Q: What is a process?
- Process as execution of a Program
- We can trace the execution of a process
- Process as minimal entity for resource allocation (for example memory).
The Execution Trace of Processes

- Two processes and a dispatcher

<table>
<thead>
<tr>
<th>δ</th>
<th>dispatcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>program A</td>
</tr>
<tr>
<td>β</td>
<td>program B</td>
</tr>
</tbody>
</table>

Traces of processes A and B

<table>
<thead>
<tr>
<th>α</th>
<th>β</th>
<th>δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>β+1</td>
<td>δ+1</td>
</tr>
<tr>
<td>α+1</td>
<td>β+2</td>
<td>δ+2</td>
</tr>
<tr>
<td>α+2</td>
<td>β+3</td>
<td>δ+3</td>
</tr>
<tr>
<td>α+3</td>
<td>β+4</td>
<td>δ+4</td>
</tr>
<tr>
<td>α+4</td>
<td>β+5</td>
<td>δ+5</td>
</tr>
<tr>
<td>α+5</td>
<td>β+6</td>
<td>δ+6</td>
</tr>
<tr>
<td>α+6</td>
<td>β+7</td>
<td>δ+7</td>
</tr>
<tr>
<td>α+7</td>
<td>β+8</td>
<td>δ+8</td>
</tr>
<tr>
<td>α+8</td>
<td>β+9</td>
<td>δ+9</td>
</tr>
<tr>
<td>α+9</td>
<td>β+10</td>
<td>δ+10</td>
</tr>
<tr>
<td>α+10</td>
<td>β+11</td>
<td>δ+11</td>
</tr>
</tbody>
</table>

Trace of dispatcher

<table>
<thead>
<tr>
<th>δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>δ+1</td>
</tr>
<tr>
<td>δ+2</td>
</tr>
<tr>
<td>δ+3</td>
</tr>
<tr>
<td>δ+4</td>
</tr>
</tbody>
</table>

States of a Process

- **User view**: A process is executing continuously
- **In reality**: Several processes compete for the CPU and other resources
- A process may be
  - **running**: it holds the CPU and is executing instructions
  - **blocked**: it is waiting for some I/O event to occur
  - **ready**: it is waiting to get back on the CPU
Process Creation

• When?
  - Submission of a batch job
  - User logs on
  - Create process to provide service such as printing
  - Spawned by existing processes

• How?
  - In UNIX:
    all processes created by fork() system call

Example: Vanilla Command Interpreter

```c
char command[MAX_COMMAND_LENGTH];

do {
  command = read_command(stdin);
  if (fork() != 0) {
    /* parent */
    if (last_char(command) != '&') {
      /* run in foreground, i.e. wait */
      waitpid(-1, &status, ...);
    }
  }
  else {
    /* child */
    execve(command, ...);
  }
} while (strcmp(command, "exit") != 0); /* ?!? */
```
The Process Control Block (PCB)

- Mechanism of a process switch:
  1. Preempt Process A and store all relevant information.
  2. Load information about Process B and continue execution.
  3. Preempt Process B and store all relevant information.
  4. Load information about Process A and continue execution.
  5. (idle)

- The PCB contains all information specific to a process.

Example for the Use of PCBs: Process Queues

- ready
- running
- waiting
- executing process
- ready queue
- disk 1
- disk 2
- serial I/O
- 1/O device queues
### Elements of a PCB

<table>
<thead>
<tr>
<th>Process Identification</th>
<th>Processor State Information</th>
<th>Processor Control Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>process id</td>
<td>register set</td>
<td>process state</td>
</tr>
<tr>
<td>parent process id</td>
<td>condition codes</td>
<td>scheduling information</td>
</tr>
<tr>
<td>user id</td>
<td>processor status</td>
<td>event (wait-for)</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
<td>memory-mgmt information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>owned resources</td>
</tr>
</tbody>
</table>

### Processes in UNIX

```
fork()  

created  
| not enough memory | enou...  

preempted  
| preempt | interrupt | system call |

kernel running  
| reschedule process | return | user running |

user running  
| return to user |

ready swapped  
| swap in | swap out |

ready  
| swap out |

sleep swapped  
| swap out |

sleep in memory  
| sleep |

zombie  
| exit |
```

- `fork()`: Creates a new process.
- `created`: The process is newly created.
- `preempted`: The process is preempted.
- `ready swapped`: The process is ready to run and swapped into memory.
- `ready`: The process is ready to run.
- `sleep swapped`: The process is swapped out and sleeping.
- `sleep in memory`: The process is sleeping in memory.
- `zombie`: The process is a zombie.
- `system call`: The process makes a system call.
- `interrupt`: The process is interrupted.
- `return`: The process returns from a system call.
- `return to user`: The process returns to the user.
- `enough memory`: The process has enough memory.
- `not enough memory`: The process does not have enough memory.
- `swap in`: The process is swapped in.
- `swap out`: The process is swapped out.
- `reschedule process`: The process is rescheduled for execution.
- `exit`: The process exits.
- `wakeup`: The process is woken up.
- `swap in`: The process is swapped in.
- `swap out`: The process is swapped out.
**Threads**

- Traditionally, processes interact very little:

```
  processes as jobs
  in batch queue
```

- This is not true in modern systems: Some applications may want to have multiple, tightly-coupled processes.

---

**Problems with traditional (heavy-weight) processes**

- Heavy-weight processes have separate address spaces:
  - Process creation is expensive
  - Process switch is expensive
  - Sharing memory areas among processes non-trivial
Threads

- Threads share address space:
  - Thread creation much simpler than process creation (no need to create and initialize address space, etc.)
  - Thread switch simple
  - Threads fully share the address space

- Convenience
  - Communication between threads

- Efficiency
  - Multiprogramming within a process (Netscape vs. Mosaic)
  - Multiprocessors

User-Level vs. Kernel-Level Threads

- User-level: kernel not aware of threads
- Kernel-level: all thread-management done in kernel
Potential Problems with Threads

- General: Several threads run in the same address space:
  - Protection must be explicitly programmed (by appropriate thread synchronization)
  - Effects of misbehaving threads limited to task
- User-level threads: Some problems at the interface to the kernel: With a single-threaded kernel, as system call blocks the entire task.

Singlethreaded vs. Multithreaded Kernel

- Protection of kernel data structures is trivial, since only one process is allowed to be in the kernel at any time.
- Special protection mechanism is needed for shared data structures in kernel.
Threads in Solaris 2.x

- Processes
- User-level threads
- Light-weight processes
- Kernel threads
- Kernel
- CPUs