Processes and Threads

Resource Ownership
- Process includes a virtual address space to hold the process image
- The OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution
- Follows an execution path that may be interleaved with other processes
- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

The unit of dispatching is referred to as a thread or lightweight process
- The concept of a thread is not recognized
- MS-DOS is an example

Multithreading - The ability of an OS to support multiple, concurrent paths of execution within a single process

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.
Each process can contain multiple threads

A Java run-time environment is an example of a system of one process with multiple threads

Process
- Defined in a multithreaded environment as “the unit of resource allocation and a unit of protection”
- Associated with processes:
  - A virtual address space that holds the process image
  - Protected access to:
    - Processors
    - Other processes (for interprocess communication)
    - Files
    - I/O resources (devices and channels)

Thread
- Associated with threads:
  - An execution state (Running, Ready, etc.)
  - A saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its processes, shared with all other threads in that process

Key Benefits of Threads
- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs
**Thread Synchronization**

- It is necessary to synchronize the activities of the various threads
- All threads of a process share the same address space and other resources
- Any alteration of a resource by one thread affects the other threads in the same process

**User-Level Threads (ULTs)**

- All thread management is done by the application
- The kernel is not aware of the existence of threads

**Kernel-Level Threads (KLTs)**

- Thread management is done by the kernel
- There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
- Windows is an example of this approach
Concurrency Arises in Three Different Contexts:

- **Multiple Applications**
  - Invented to allow processing time to be shared among active applications

- **Structured Applications**
  - Extension of modular design and structured programming

- **Operating System Structure**
  - OS themselves implemented as a set of processes or threads

---

**Mutual Exclusion: Software Approaches**

- Software approaches can be implemented for concurrent processes that execute on a single-processor or a multiprocessor machine with shared main memory

- These approaches usually assume elementary mutual exclusion at the memory access level

- Dijkstra reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker

- Following Dijkstra, we develop the solution in stages

---

**Table 5.1** Some Key Terms Related to Concurrency

- **atomic operation**: A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interpret the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.

- **critical section**: A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

- **deadlock**: A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

- ** livelock**: A situation in which two or more processes continuously change their states (in response to changes in the other processes) without doing any useful work.

- **mutual exclusion**: The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

- **race condition**: A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

- **starvation**: A situation in which a runnable process is overlooked indefinitely by the scheduler, whether it is able to proceed, it is never chosen.

---

**Figure 5.1 Mutual Exclusion Attempts (page 1)**

(a) First attempt

```
/* PROCESS 0 */
   while (turn = 0)
      /* do nothing */;
   /* critical section*/;
   turn = 1;
```

(b) Second attempt

```
/* PROCESS 0 */
   while (flag[0])
      /* do nothing */;
   flag[0] = true;
   /* critical section*/;
   flag[0] = false;
```

```
/* PROCESS 1 */
   while (turn = 0)
      /* do nothing */;
   /* critical section*/;
   turn = 1;
```

```
/* PROCESS 1 */
   while (flag[0])
      /* do nothing */;
   flag[0] = true;
   /* critical section*/;
   flag[0] = false;
```
Figure 5.1 Mutual Exclusion Attempts (page 2)

(c) Third attempt

```c
/** PROCESS 0 */
.
flag[0] = true;
while (flag[0]) { /* do nothing */
  /* critical section */
  flag[0] = false;
}

/** PROCESS 1 */
.
flag[1] = true;
while (flag[1]) { /* do nothing */
  /* critical section */
  flag[1] = false;
}

(d) Fourth attempt

```