Processes

Rafael Ramirez
Dep Tecnologia
Universitat Pompeu Fabra
Processes

- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes
- Interprocess Communication
Process Concept

- Early systems allowed only one program to be executed at a time. This program had complete control of the system.
- Current computer systems allow multiple programs to be loaded into memory and to be executed *concurrently*.
- This resulted in the notion of *process* (which we’ll define as a program in execution).
- A system consists of a collection of processes: OS processes and user processes.
- All these processes can potentially execute concurrently with the CPU (or CPU’s) multiplexed among them.
- By switching the CPU among them the OS can make the computer more productive.
Brief Note on Concurrency

**Parallelism:** performance

**Concurrency:** reactive programming

Note
- Sequential languages may be executed in parallel
- Concurrent languages may be implemented sequentially only.

What is a concurrent program?

**Concurrent program:** a set of sequential programs (called processes) which are executed in *abstract parallelism*, i.e. not required a separate physical processor to execute each process.
An operating system executes a variety of programs:
- Batch system – jobs
- Time-shared systems – user programs or tasks

OS Textbooks use the terms *job* and *process* almost interchangeably.

Process – a program in execution; process execution must progress in sequential fashion.

A process includes:
- program counter
- stack
- data section
Note: a process is NOT a program (program is a passive entity while process is an active entity).

Two processes may be associated with the same program, e.g. user may invoke many copies of an editor program, each copy being a separate process.

A process may spawn many processes as it runs.
As a process executes, it changes *state*

- **new**: The process is being created.
- **running**: Instructions are being executed.
- **waiting**: The process is waiting for some event to occur.
- **ready**: The process is waiting to be assigned to a process.
- **terminated**: The process has finished execution.
Diagram of Process State

- new
- admitted
- interrupt
- exit
- terminated
- ready
- running
- waiting
- I/O or event completion
- scheduler dispatch
- I/O or event wait
Process Control Block (PCB)

Information associated with each process.

- **Process state:** e.g. new, ready, running, etc.
- **Program counter:** indicates the next instruction to be executed by the process
- **CPU registers:** accumulators, stack pointers
- **CPU scheduling information:** includes process priority, etc.
- **Memory-management information:** value of the base and limit registers
- **Accounting information:** includes process #, amount of CPU used, etc.
- **I/O status information:** includes the I/O allocated to the process, open files, ...
Process Control Block (PCB)

<table>
<thead>
<tr>
<th>pointer</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
<td></td>
</tr>
<tr>
<td>program counter</td>
<td></td>
</tr>
<tr>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>memory limits</td>
<td></td>
</tr>
<tr>
<td>list of open files</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
CPU Switch From Process to Process

\[
\begin{align*}
\text{process } P_0 & \quad \text{operating system} \quad \text{process } P_1 \\
\text{executing} & \quad \text{interrupt or system call} & \quad \text{idle} \\
& \quad \text{save state into PCB}_0 & \quad \text{executing} \\
& \quad \cdots & \quad \text{idle} \\
& \quad \text{reload state from PCB}_1 & \quad \text{executing} \\
& \quad \text{interrupt or system call} & \quad \text{idle} \\
& \quad \text{save state into PCB}_1 & \quad \text{idle} \\
& \quad \cdots & \\
& \quad \text{reload state from PCB}_0
\end{align*}
\]
Process Scheduling

- Basically, it is choosing the order in which processes run.
- For a uniprocessor system, only one running process. If more than one process, the rest have to wait until CPU is free.
- Multiprogramming’s goal is to have process running at all times to maximise CPU utilisation.
- Time sharing goal is to switch the CPU among processes very frequently to minimise response time.
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

- Context-switch time is overhead; the system does no useful work while switching.

- Time dependent on hardware support. (depends on memory speed, # of registers which must be copied, etc.)
Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

- **Resource sharing** (CPU time, memory, I/O devices, etc.)
  - Parent and children share all resources.
  - Children share subset of parent’s resources.
  - Parent and child share no resources.

- **Execution**
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
Process Creation (Cont.)

- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.

- UNIX examples
  - `fork` system call creates new process (with same address space)
  - `execve` system call used after a `fork` to replace the process’ memory space with a new program.
Process Termination

- Process executes last statement and asks the operating system to terminate it (exit).
  - Output data from child to parent (via wait).
  - Process’ resources are deallocated by operating system.
- Parent may terminate execution of children processes (abort).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.
Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process.

Advantages of process cooperation

- Information sharing (we must provide for concurrent access to share resources)
- Computation speed-up (break task into subtasks; exec. concurrently. >1 CPU)
- Modularity (divide functions into separate processes)
- Convenience (even a user may have >1 tasks to work on at one time)
Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
Bounded-Buffer – Shared-Memory Solution

- Shared data
  
  \[
  \begin{align*}
  & \text{var } n; \\
  & \text{type } \text{item} = \ldots; \\
  & \text{var } \text{buffer. array}[0..n-1] \text{ of item}; \\
  & \quad \text{in, out: } 0..n-1; \text{ initially } = 0
  \end{align*}
  \]

- Producer process

  \[
  \text{repeat} \\
  \quad \ldots \\
  \quad \text{produce an item in } nextp \\
  \quad \ldots \\
  \quad \text{while } \text{in+1 mod } n = \text{out do no-op; } \\
  \quad buffer[\text{in}] := nextp; \\
  \quad \text{in :=in+1 mod } n; \\
  \quad \text{until false;}
  \]
Bounded-Buffer (Cont.)

- Consumer process

```plaintext
repeat
  while in = out do no-op;
  nextc := buffer [out];
  out := out+1 mod n;
  ...
  consume the item in nextc
  ...
until false;
```

- Solution is correct, but can only fill up n–1 buffer.
Threads

- A thread (or lightweight process) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space

- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources collectively know as a task.

- A traditional or heavyweight process is equal to a task with one thread
Multiple Threads within a Task
Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

- Java threads are managed by the JVM.
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`
- If $P$ and $Q$ wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)
Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
Direct Communication

- Processes must name each other explicitly:
  - **send** \((P, \text{message})\) – send a message to process P
  - **receive** \((Q, \text{message})\) – receive a message from process Q

- Properties of communication link
  - Links are established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be unidirectional, but is usually bi-directional.
Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports).
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.

- Properties of communication link
  - Link established only if processes share a common mailbox.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
Indirect Communication (Continued)

- Mailbox sharing
  - $P_1$, $P_2$, and $P_3$ share mailbox A.
  - $P_1$, sends; $P_2$ and $P_3$ receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Buffering

Queue of messages attached to the link; implemented in one of three ways.

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous).

2. Bounded capacity – finite length of $n$ messages
   Sender must wait if link full.

3. Unbounded capacity – infinite length
   Sender never waits.
Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Concurrent programming in Java
Concurrent access to shared data may result in data inconsistency.

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Shared-memory solution to bounded-buffer problem allows at most $n - 1$ items in buffer at the same time. A solution, where all $N$ buffers are used is not simple.

Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer
Bounded-Buffer

- Shared data
  
  ```
  type item = ... ;
  var buffer array [0..n-1] of item;
  in, out: 0..n-1;
  counter: 0..n;
  in, out, counter := 0;
  ```

- Producer process
  
  ```
  repeat
  ...
  produce an item in nextp
  ...
  while counter = n do no-op;
  buffer [in] := nextp;
  in := in + 1 mod n;
  counter := counter +1;
  until false;
  ```
Bounded-Buffer (Cont.)

- Consumer process
  
  repeat
  
  \[ \text{while } \text{counter} = 0 \text{ do no-op;} \]
  \[ \text{nextc} := \text{buffer}[\text{out}]; \]
  \[ \text{out} := \text{out} + 1 \mod n; \]
  \[ \text{counter} := \text{counter} - 1; \]
  
  ... 
  
  consume the item in \text{nextc}
  
  ...
  
  until false;

- The statements:
  
  \[ \text{counter} := \text{counter} + 1; \]
  \[ \text{counter} := \text{counter} - 1; \]

  must be executed atomically.

  (Atomic operation means an operation that completes in its entirety without interruption).
The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process $P_i$

```plaintext
repeat
  entry section
  critical section
  exit section
  reminder section
until false;
```
1. **Mutual Exclusion.** If process $Pi$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting.** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the $n$ processes.
Initial Attempts to Solve Problem

- Only 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (other process $P_j$)
  
  ```
  repeat
  entry section
  critical section
  exit section
  reminder section
  until false;
  ```

- Processes may share some common variables to synchronize their actions. (no update to these vars in critical section)
Algorithm 1

Shared variables:
  - var turn: (0..1);
    initially turn = 0
  - turn = i ⇒ P_i can enter its critical section
  - Process P_i

repeat
  while turn ≠ i do no-op;
  turn := j;
until false;

Satisfies mutual exclusion, but not progress
Algorithm 2

Shared variables
- **var** *flag*: array [0..1] of boolean;
  - initially *flag*[0] = *flag*[1] = false.
- *flag*[i] = true ⇒ *P*<sub>i</sub> ready to enter its critical section
- Process *P*<sub>i</sub>

```
repeat
    *flag*[i] := true;
    while *flag*[j] do no-op;
    critical section
    *flag*[i] := false;
    remainder section
until false;
```

- Satisfies mutual exclusion, but not progress requirement.
Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process $P_i$

```
repeat
  flag $[i] := \text{true}$;
  turn $:= j$;
  while (flag $[j]$ and turn $= j$) do no-op;
  critical section
  flag $[i] := \text{false}$;
  remainder section
until false;
```

- Meets all three requirements; solves the critical-section problem for two processes.
Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

- If processes \( P_i \) and \( P_j \) receive the same number, if \( i < j \), then \( P_i \) is served first; else \( P_j \) is served first.

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
Bakery Algorithm

Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (j = 1; j <= NUM_THREADS; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { /* nothing */ }
    }
}

unlock(integer i) {
    Number[i] = 0;
}

Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}

(a, b) < (c, d)
defined as
(a < c) or ((a == c) and (b < d))
Semaphore

- Synchronization tool that does not require busy waiting.
- Scales better to $n$ processes
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

\[
\text{wait} \ (S): \quad \text{while } S \leq 0 \ \text{do no-op;}
\]
\[
S := S - 1;
\]

\[
\text{signal} \ (S): \quad S := S + 1;
\]
Example: Critical Section of $n$ Processes

- Shared variables
  - `var mutex : semaphore`
  - Initially $mutex = 1$
- Process $P_i$

```
repeat
  wait(mutex);
  critical section
  signal(mutex);
  remainder section
until false;
```
Define a semaphore as a record

\[\text{type } \text{semaphore} = \text{record} \]
\[\text{value: integer} \]
\[L: \text{list of } \text{process}; \]
\[\text{end}; \]

Assume two simple operations:
- block suspends the process that invokes it.
- \text{wakeup}(P) resumes the execution of a blocked process P.
Semaphore operations now defined as

\[ \text{wait}(S): \quad S.value := S.value - 1; \]
\[ \text{if } S.value < 0 \]
\[ \quad \text{then begin} \]
\[ \quad \quad \text{add this process to } S.L; \]
\[ \quad \quad \text{block}; \]
\[ \quad \text{end;} \]

\[ \text{signal}(S): \quad S.value := S.value + 1; \]
\[ \text{if } S.value \leq 0 \]
\[ \quad \text{then begin} \]
\[ \quad \quad \text{remove a process } P \text{ from } S.L; \]
\[ \quad \quad \text{wakeup}(P); \]
\[ \quad \text{end;} \]
Semaphore as General Synchronization Tool

- Execute \( B \) in \( P_j \) only after \( A \) executed in \( P_i \)
- Use semaphore \( \text{flag} \) initialized to 0
- Code:

\[
\begin{align*}
\text{\( P_i \)} & \quad \text{\( P_j \)} \\
\ldots & \quad \ldots \\
A & \quad \text{\( \text{wait(flag)} \)} \\
\text{\( \text{signal(flag)} \)} & \quad B
\end{align*}
\]
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let $S$ and $Q$ be two semaphores initialized to 1
  
  ```
  P_0
  wait(S);
  wait(Q);
  ..
  signal(S);
  signal(Q);
  
  P_1
  wait(Q);
  wait(S);
  ..
  signal(Q);
  signal(S);
  ```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.

- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.

- Can implement a counting semaphore $S$ as a binary semaphore.
Implementing $S$ as a Binary Semaphore

- **Data structures:**
  
  ```
  var
  S1: binary-semaphore;  (for mutex)
  S2: binary-semaphore;  (for waiting)
  C: integer;            (val of count $S$)
  ```

- **Initialization:**
  
  1. $S1 = 1$
  2. $S2 = 0$
  3. $C = \text{initial value of semaphore } S$
Implementing $S$ (Cont.)

- **wait operation**

  ```
  wait(S1);
  C := C - 1;
  if C < 0
   then begin
      signal(S1);
      wait(S2);
   end
  else signal(S1);
  ```

- **signal operation**

  ```
  wait(S1);
  C := C + 1;
  if C ≤ 0 then signal(S2);
  signal(S1);
  ```
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

Shared data

**type** item = ...

**var** buffer = ...

full, empty, mutex: semaphore;
nextp, nextc: item;
full :=0; empty := n; mutex :=1;
Producer process

\begin{verbatim}
repeat
  ... 
  produce an item in nextp 
  ... 
  wait(empty); 
  wait(mutex); 
  ... 
  signal(mutex); 
  signal(full); 
until false;
\end{verbatim}
Consumer process

\[\text{repeat}\]
\[\text{wait}(\text{full})\]
\[\text{wait}(\text{mutex});\]
\[\ldots\]
\[\text{remove an item from } \text{buffer to } \text{nextc}\]
\[\ldots\]
\[\text{signal}(\text{mutex});\]
\[\text{signal}(\text{empty});\]
\[\ldots\]
\[\text{consume the item in } \text{nextc}\]
\[\ldots\]
\[\text{until } \text{false};\]
Readers-Writers Problem

- Shared data
  
  ```
  var mutex, wrt: semaphore (=1);
  readcount: integer (=0);
  ```

- Writer process
  
  ```
  wait(wrt);
  ...
  writing is performed
  ...
  signal(wrt);
  ```
Readers-Writers Problem (Cont.)

- Reader process

```
wait(mutex);
readcount := readcount + 1;
if readcount = 1 then wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount := readcount - 1;
if readcount = 0 then signal(wrt);
signal(mutex):
```
Dining-Philosophers Problem

- Shared data

```
var chopstick: array [0..4] of semaphore;
(=1 initially)
```
Dining-Philosophers Problem (Cont.)

- Philosopher $i$:

```
repeat
  wait(chopstick[i])
  wait(chopstick[i+1 mod 5])
  ...
  eat
  ...
  signal(chopstick[i]);
  signal(chopstick[i+1 mod 5]);
  ...
  think
  ...
  until false;
```
Synchronization Mechanisms

- Shared variables
  
  "while turn = 1 do no-op;"

- Synch. Hardware
  
  "while test-and-set(lock) do no-op;"

- Semaphores
  
  "wait(S); CS; signal(S);"

- Monitors (like Java)

```
Airline database

User 1

User 2

User N
```
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```pascal
type monitor-name = monitor
variable declarations
procedure entry P1 : (...);
  begin ... end;
procedure entry P2 (...);
  begin ... end;
..;
procedure entry Pn (...);
  begin ... end;
begin
  initialization code
end
```
To allow a process to wait within the monitor, a *condition* variable must be declared, as

```
var x, y: condition
```

Condition variable can only be used with the operations *wait* and *signal*.

- The operation
  
  ```
  x.wait;
  ```
  
  means that the process invoking this operation is suspended until another process invokes
  
  ```
  x.signal;
  ```
  
- The *x.signal* operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.
Schematic View of a monitor
Monitor with condition variables
Dining Philosophers Example

type dining-philosophers = monitor

var state : array [0..4] of : (thinking, hungry, eating);
var self : array [0..4] of condition;

procedure entry pickup (i: 0..4);
begin
  state[i] := hungry,
  test (i);
  if state[i] ≠ eating then self[i], wait,
end;

procedure entry putdown (i: 0..4);
begin
  state[i] := thinking;
  test (i + 4 mod 5);
  test (i + 1 mod 5);
end;

Phil(l):
  dp.pickup(l);
  EAT
  dp.putdown(l);
procedure test(k: 0..4);
begin
  if state[k+4 mod 5] ≠ eating
      and state[k] = hungry
      and state[k+1 mod 5] ≠ eating
      then begin
          state[k] := eating;
          self[k].signal;
      end;
end;

begin
  for i := 0 to 4
    do state[i] := thinking;
end.
Monitor Implementation Using Semaphors

- **Variables**
  
  ```
  var mutex: semaphore (init = 1)       (mutex in monitor)
  next: semaphore (init = 0)           (for susp. inside monitor)
  next-count: integer (init = 0)      (# of susp. proc. in next)
  ```

- Each external procedure $F$ will be replaced by
  ```
  wait(mutex);
  ...
  body of $F$;
  ...
  if next-count > 0
  then signal(next)
  else signal(mutex);
  ```

- Mutual exclusion within a monitor is ensured.
Monitor Implementation (Cont.)

- For each condition variable $x$, we have:
  
  ```
  var $x$-sem: semaphore (init = 0)
  $x$-count: integer (init = 0)
  ```

- The operation $x$.wait can be implemented as:

  ```
  $x$-count := $x$-count + 1;
  if next-count > 0
    then signal(next)
    else signal(mutex);
  wait($x$-sem);
  $x$-count := $x$-count − 1;
  ```
Monitor Implementation (Cont.)

- The operation \( x\.\text{signal} \) can be implemented as:

  \[
  \text{if } x\text{-count} > 0 \\
  \text{then begin} \\
  \quad \text{next-count} := \text{next-count} + 1; \\
  \quad \text{signal(x-sem);} \\
  \quad \text{wait(next);} \\
  \quad \text{next-count} := \text{next-count} - 1; \\
  \text{end;}
  \]
Monitor Implementation (Cont.)

- **Conditional-wait construct:** $x.wait(c)$;
  - $c$ – integer expression evaluated when the wait operation is executed.
  - Value of $c$ *(priority number)* stored with the name of the process that is suspended.
  - When $x.signal$ is executed, process with smallest associated priority number is resumed next.

- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.
Class **Thread** has several constructors. The constructor:

- **Public Thread( String threadname )**
  Constructs a **Thread** object with name **threadname**.

- **Public Thread()**
  Constructs a **Thread** whose name is **Thread**- concatenated with a number (e.g. **Thread-1**)

- The code of a thread is placed in its **run** method.
- The run method can be overridden in a subclass of **Thread**.
- Program launches a thread execution by calling the thread **start** method.
- Which in turn calls the **run** method.
- Caller executes concurrently with the launched thread.
Methods or parts of methods can be declared synchronized.

Each object has a lock associated with it.

Threads can hold the lock of one or more objects.

Only one thread can hold the lock of an object at any time.

To execute synchronized code of an object, a thread needs to hold the lock of this object.
Wait and Notify

- In synchronized code, an object can invoke `wait()`. In that case the current thread suspends until another thread notifies the object. Invoking `wait()` releases the lock of the object.

- In synchronized code, an object can invoke `notify()`. In that case one thread, who is waiting for the lock of this object, is resumed. Notification is asynchronous; the waiting thread needs to re-obtain the object’s lock.

- Variants: `sleep(_), notifyAll()`