Chapter 2

Processes and Threads

2.1 Processes
2.2 Threads
2.3 Interprocess communication
2.4 Classical IPC problems
2.5 Scheduling

Structuring

- Functions / services - invoked by "events"
- Events occur at unpredictable times
- Events occur with varying frequency (and priority)
  - Communications and peripheral devices
  - Resource requests
  - Interactive users and contingencies (errors)
- OS must handle parallelism
- OS must deal with non-determinism
- Notion of a "process"
Sequential Process

- A sequential process ("task") is an object, resulting from the execution of a program
- A process consists of a sequence of instructions and a data area, both residing in memory
- Execution of the instructions leads to a “thread of execution” – (represented by the program counter) and to a stack with state information
- The stack is part of the “context” – state vector which comprises the allocated address space, register contents and allocated resources

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
Processes

The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Processes and Threads

- A process is a "heavyweight" object comprising the complete information vector (context, resources)
- Within that context several threads of execution (lightweight processes) may exist
- Each thread contains only a small part of the state vector - typically the register contents and the stack
  - Each thread has its own stack
  - Each thread has its own minimal context block
- Processes (heavyweight and lightweight) execute concurrently - logical and also physical to a degree
- The kernel may grant processes their own virtual address space
Processes and Threads

- Processes and threads cooperate by sharing resources, communicating and signaling synchronization.
- Processes compete for access to resources.
- Each process is dedicated to a specific function and offers it through a well defined interface.
- Processes are autonomous entities (kernel objects) executing on their private virtual CPU.
- In a real system their threads are scheduled for execution on a real processor.

Operating System

- An OS is a collection of concurrently executing processes.
- Every process has its own virtual CPU.
- These processes cooperate by sharing resources, communicating and signaling synchronization.
- They compete for access to resources.
- Each process is dedicated to a specific function and offers it through a well defined interface.
- Processes are autonomous entities (kernel objects) executing on their private virtual CPU.
- In a real system their threads are scheduled for execution on a real processor.
**Precedence Relations**

- Since processes may be created dynamically and need to cooperate and compete, they must observe certain precedence constraints (start ... stop)

- Process flow graphs are a convenient way for representing precedence relations

- Precedence flow graphs are DAGs (directed acyclic graphs)

  \[ p_i \text{ – process } "i" \]

  \[ S(p_1, \ldots, p_n) \text{ serial execution} \]

  \[ P(p_1, \ldots, p_n) \text{ parallel execution} \]

---

**Process Flow Graphs**

\[ S(p_1, S(p_2, S(p_3, p_4))) \]

\[ P(p_1, P(p_2, P(p_3, p_4))) \]
Fig 2-1 (a)

\[ S(p_1, p_2, p_3, p_4) \]

serial

Fig 2-1 (b)

\[ P(p_1, p_2, p_3, p_4) \]

parallel
serial/parallel

Fig 2-1 (c),(d)

Process Flow Graphs

well-formed = correctly nested structure
\[(a + b) \times (c + d) - \frac{e}{f}\]
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

Process Creation

Principal events that cause process creation
1. System initialization
2. Execution of a process creation system
3. User request to create a new process
4. Initiation of a batch job

Process creation can be done:
1. Implicitly – programming language constructs
2. Explicitly – operating system functions
High-level Constructs

How to express and guarantee the “well-formed”-ness of a program?

The cobegin / coend primitives

\begin{align*}
\text{begin} & \quad \text{...} \quad C_i \text{ a code segment (sequential)} \\
\text{end} & \\
\text{cobegin} & \\
\text{coend} & \\
\end{align*}

\begin{align*}
\text{cobegin} & \\
\text{executed concurrently} & \\
\text{coend} & \\
\end{align*}

Each $C_i$ may be expressed as:

\begin{align*}
P(C_1, P(C_2, \ldots P(\ldots, C_n )\ldots))
\end{align*}

\begin{align*}
S(p_{i1}, S(p_{i2}, \ldots S(\ldots, p_{im})\ldots))
\end{align*}
The “fork”, “join” and “quit” Primitives

\[ t := t - 1; \]
\[ \text{if } t == 0 \text{ then go_to } y; \]

“atomic” = indivisible instruction

Evaluation of arithmetic expression of Fig. 2-2

\[ n:= 2; \]
\[ \text{fork } p3; \]
\[ m := 2; \]
\[ \text{fork } p2; \]
\[ t1 := a + b; \]
\[ \text{join } m, p4; \]
\[ \text{quit}; \]
\[ p2 : \]
\[ t2 := c + d; \]
\[ \text{join } m, p4; \]
\[ \text{quit}; \]
\[ p4 : \]
\[ t4 := t1 \times t2; \]
\[ \text{join } n, p5; \]
\[ \text{quit}; \]
\[ p3 : \]
\[ t3 := e / f; \]
\[ \text{join } n, p5; \]
\[ \text{quit}; \]
\[ p5 : \]
\[ t5 := t4 - t3; \]

Future disadvantage: these primitives may be used indiscriminately which can lead to poorly structured code

ork, join, quit

Evaluation of the precedence graph of Fig. 2-1(d);
(Si denotes the statement sequence for process pi)

\[ t6 := 2; \]
\[ t8 := 3; \]
\[ S1; \]
\[ \text{fork } p2; \]
\[ \text{fork } p5; \]
\[ \text{fork } p7; \]
\[ \text{quit}; \]
\[ p2: \]
\[ S2; \]
\[ \text{fork } p3; \]
\[ \text{fork } p4; \]
\[ \text{quit}; \]
\[ p5: \]
\[ S5; \]
\[ \text{join } t6, p6; \]
\[ \text{quit}; \]
\[ p7: \]
\[ S7; \]
\[ \text{join } t8, p8; \]
\[ \text{quit}; \]
\[ p3: \]
\[ S3; \]
\[ \text{join } t8, p8; \]
\[ \text{quit}; \]
\[ p4: \]
\[ S4; \]
\[ \text{join } t6, p6; \]
\[ \text{quit}; \]
\[ p6: \]
\[ S6; \]
\[ \text{join } t8, p8; \]
\[ \text{quit}; \]
\[ p8: \]
\[ S8; \]
\[ \text{quit}; \]

Main disadvantage: these primitives may be used indiscriminately which can lead to poorly structured code
Process Creation

process p {
    declarations_for_p;
    executable_code_for_p;
}

Creates a separate unit of execution;
Is statically created when the enclosing scope is created;
Contains local variables, functions etc.

Dynamic Process Creation

process p {
    process p1 {
        declarations_for_p1;
        executable_code_for_p1;
    }
    process type p2 {
        declarations_for_p2;
        executable_code_for_p2;
    }
    . . .
    q = new p2;
    . . .
}

p1 is statically created, p2 is dynamically created.
**Static and dynamic process creation - Ada**

```
process p
process p_1
begin
    . . .
end
process type p_2
begin
    . . .
end
Other declarations for p
begin
    . . .
q := new p_2
    . . .
end
```

**Static process creation**

**Dynamic process creation**

**process “template”**

---

**Static and dynamic process creation - Ada**

```
process “p”

process “p_1”

process “p_2”

interaction, communication

Need other tools (primitives)
```
Critical Sections

Process 1

```
cobegin
  p_1:  ... 
  x := x + 1;
  ... 
  p_2:  ... 
  x := x - 1; // or x + 1, or x := n 
  ... 
coend
```

The classical "race condition"
Critical Sections

\[
\text{cobegin}
\]
\[
p1 \{ \ldots x++ \ldots \}
\]
\[
p2 \{ \ldots x++ \ldots \}
\]
\[
\text{coend}
\]

Critical Sections

\[
\ldw R1, x
\]
\[
x++ \quad \text{addi} \ R1,1
\]
\[
\ldw R1, x
\]
\[
\text{addi} \ R1,1
\]
\[
\stw R1, x
\]
\[
\stw R1, x
\]
Critical Sections

- Any segment of code involved in
  - Reading
  - Writing
- A shared data area is called:
- A Critical Section
- Additional assumptions:
  - Reading and Writing are indivisible operations when 2 processes attempt simultaneous access one will “win”
  - Critical sections do not have priorities associated
  - The relative speeds of the processes are unknown
  - A program may halt only outside of a critical section.

```plaintext
p1 { . . .
    LOOP
    access critical section;
    program (non critical)
    ENDLOOP
}
```
Critical Sections - 2

- A enters critical region
- A leaves critical region
- B attempts to enter critical region
- B enters critical region
- B leaves critical region

Mutual exclusion using critical sections

Critical Sections - 3

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region
General Critical Section Problem

cobegin
p_1: loop CS_1; program_1; end {loop}
| p_2: loop CS_2; program_2; end {loop}
| :  |
| p_n: loop CS_n; program_n; end {loop}
coend

CS_i is the critical section for process p_i

Software Solutions

• First attempt

int turn = 1;
cobegin
pl: loop
    while turn == 2 do
        {waitloop};
    CS_1; turn = 2;
    program_1
endloop;
p2: loop
    while turn == 1 do
        {waitloop};
    CS_2; turn = 1;
    program_2
endloop;
coend
Software Solutions

- Second attempt

```c
int c1 = 0, cs = 0;
cobegin
p1: loop
    c1 = 1;
    while c2 do
        {waitloop} ;
        CS_1; c1 = 0;
        program_1
    endloop;

p2: loop
    c2 = 1;
    while c1 do
        {waitloop} ;
        CS_2; c2 = 0;
        program_2
    endloop;
coend
```

- Third attempt

```c
int c1 = 0, cs = 0;
cobegin
p1: loop
    c1 = 1;
    if (c2) c1 = 0;
    else {
        CS_1; c1 = 0;
        program_1;
    }
endloop;

p2: loop
    c2 = 1;
    if (c1) c2 = 0;
    else {
        CS_2; c2 = 0;
        program_2;
    }
endloop;
coend
```
Software Solutions

- Dekker’s Solution to mutual exclusion:

```plaintext
var c1, c2: boolean; turn: integer
  c1 := c2 := true;
  cobegin
    p1: loop
      c1 := false; turn := 1;
      while ¬c2 & turn = 1 do
        {waitloop} ;
        CS_1; c1 := true;
        program_1
      endloop
  p2: loop
      c2 := false; turn := 2;
      while ¬c1 & turn = 2 do
        {waitloop} ;
        CS_2; c2 := true;
        program_2
    endloop
  coend
```

- Peterson’s Solution to mutual exclusion:

```plaintext
int c1 = 0, c2 = 0, willWait;
  cobegin
    p1: loop
      c1 = 1; willWait = 1;
      while (c2 && (willWait == 1)) do
        {waitloop} ;
        CS_1; c1 = 0;
        program_1
      endloop
    p2: loop
      c2 := false; turn := 2;
      while (c1 && (willWait == 2)) do
        {waitloop} ;
        CS_2; c2 := true;
        program_2
      endloop
  coend
```
Classical Problem

Producer $p_1$ deposit

Shared buffer

Consumer $p_2$ remove

Cooperation - Coordination - Synchronization

Circular Buffer

Shared variables

in = in+1; add one entry
out = out+1; remove one entry

empty: in = out;
full: in+1 MOD n = out;

in = in+1;
counter = counter+1;
add one entry
out = out+1;
counter = counter-1;
remove one entry

counter = 0; buffer empty

Unsafe!
### Classical Problem

- **Message Passing**
  - Client: $p_1$ sends message to server $p_2$.
  - Server: Receiver receives message, sends it back to $p_1$.

### Cooperation - Coordination - Synchronization

#### Semaphores

- **V(s):** Increment $s$ by 1 in a single indivisible action; the fetch, increment, and store cannot be interrupted, and $s$ cannot be accessed by another process during the Operation.

- **P(s):** Decrement $s$ by 1, if possible. If $s = 0$, then it is not possible to decrement $s$ and still remain in the domain of nonnegative integers; the process invoking the P( ) Operation then waits until it is possible.

The successful testing and decrementing of $s$ are also an indivisible (atomic) operation. This requires support by special hardware functions, e.g. Test_and_Set or Compare_and_Swap instructions.
Mutexes

```pascal
var mutex: semaphore
mutex := 1;

cobegin
    p_1: loop . . . end {loop}
    |
    |
    p_i: loop
        P(mutex); CS_I; V(mutex);
        program_i
        end; {loop}
        |
        |
    p_j: loop
        P(mutex); CS_I; V(mutex);
        program_j
        end; {loop}
        |
        |
    p_n: loop . . . end {loop}
coend
```

This solution handles n parallel processes!

Test-and-Set

```pascal
PROCEDURE TestAndSet(VAR target: BOOLEAN):
    BOOLEAN;
    VAR tmp .. BOOLEAN;
    BEGIN
        tmp := target ;
        target := TRUE;
        RETURN tmp ;
    END P;
```

![Diagram for Test-and-Set]
Resource Management - Events

```plaintext
var s: semaphore
s := 0;
cobegin
p_1: begin
begin
P( s ); {wait for signal}
end
| begin
| V( s ); {send wakeup signal}
| end
| coend
```

Resource Management

```plaintext
Var e, f, b: semaphore;
e := n; f := 0; b := 1;
cobegin
producer: loop
produce next record
P( e ); P( b );
add to buffer;
V( b ); V( f );
end_loop;
| consumer: loop
| P( f ); P( b );
take from buffer;
V( b ); V( e );
| process record {consume}
| end_loop
| coend
```
Events

- Event - change in state
- Types of events:
  - Synchronous
  - Asynchronous
- Support of events:
  - Define events \( E \)
  - \( E.post \)
  - \( E.wait \)
  - "memoryless"?
  - unicast / broadcast
- Event handler

Events

- UNIX - signals
  - \( \text{kill(pid, sig)} \)
  - \( \text{sigaction()} \)
  - \( \text{pause()} \)

- Windows2K - dispatcher objects:
  - signaled
  - nonsignaled
  - \( \text{waitForSingleObject} \)
  - \( \text{waitForMultipleObjects} \)
  - unicast / broadcast
  - Objects: process, thread, semaphore, file, timer, que, . . .
Higher-Level Synchronization and Communication

- Shared memory
- Distributed S&C
- Classic synch problems

Semaphores and Events

- Low-level constructs
- No support for structuring concurrent programs
- No direct support for critical sections
- CS must be protected by correctly placed semaphore or event operations
- Correct sequence is crucial - danger or deadlocks

- Provide higher-level alternative constructs
- Concentrate and encapsulate all accesses to a shared resource
  - For shared memory systems
  - For isolated and distributed systems
Share Memory Methods

- Monitors and Protected Types
- Based on concept of Abstract Data Types (ADTs)

- Monitor:
  - a collection of data representing the state of the resource (object)
  - A set of functions (procedures) which manipulate the resource data
  - The implementation must guarantee:
    * Access to the resource is only possible through a procedure
    * Procedures are mutually exclusive - only one thread is "inside", others must wait "outside"

Monitors

- concentrate all accesses to the shared object on a one-at-a-time basis
- idea of a monitor is based on the principles of abstract data types, which suggest that for any distinct data type there should be a well-defined set of operations through which any instance of that data type must be manipulated
- a monitor is defined as a collection of data, representing the resource to be controlled by the monitor [attributes] and a set of procedures [methods] to manipulate that resource.
- this is a special type of class/object in today's view of object-oriented software technologies
The implementation of the monitor construct must guarantee the following:

1. Access to the resource is possible only via one of the monitor procedures.
2. Procedures are mutually exclusive; that is, at any given time only one process may be executing inside the monitor. During that time, other processes calling a monitor procedure are delayed until the process leaves the monitor.

Monitors provide a special type of variable called condition and two operations, `wait` and `signal`, which operate on conditions and can only be used inside monitor procedures. The operation `c.wait` causes the executing process to be suspended (blocked) and placed on a queue associated with the condition `c`. Performing the signal operation `c.signal` causes one of the waiting processes to reenter the monitor.

### Monitor solution for bounded buffer

```pascal
buffer: monitor;

type buf_storage = array[0 .. n - 1] of char;

var Buf: buf_storage; nextin, nextout, full_cnt: integer;
    notempty, notfull: condition;

procedure deposit( data: char );
begin
    if full_cnt = n then notfull.wait
    Buf[nextin] := data;
    nextin := nextin + n 1;
    full_cnt := full_cnt + 1;
    notempty.signal
end;

procedure remove(data: char);
begin
    if full_cnt = 0 then notempty.wait
    data := Buf[nextout];
    nextout := nextout + n 1;
    full_cnt := full_cnt - 1;
    notful.signal
end;

begin
    full_cnt := nextin := nextout := 0 {initialization }
end (buffer)
```
Monitor Implementation

1. Execution of procedures must be mutually exclusive.
2. A wait must block the current process on the corresponding condition.
3. When a process exits or is blocked on a condition and there are processes waiting to enter or reenter the monitor, one must be selected. If there is a process suspended as the result of executing a signal operation, then it is selected; otherwise, one of the processes from the initial queue of entering processes is selected (processes blocked on conditions remain suspended).
4. A signal must determine if any process is waiting on the corresponding condition. If this is the case, the current process is suspended and one of these waiting processes is reactivated (say, using a first-in/first-out discipline); otherwise, the current process continues.