

CHAPTER 3: CONCURRENT PROCESSES AND PROGRAMMING

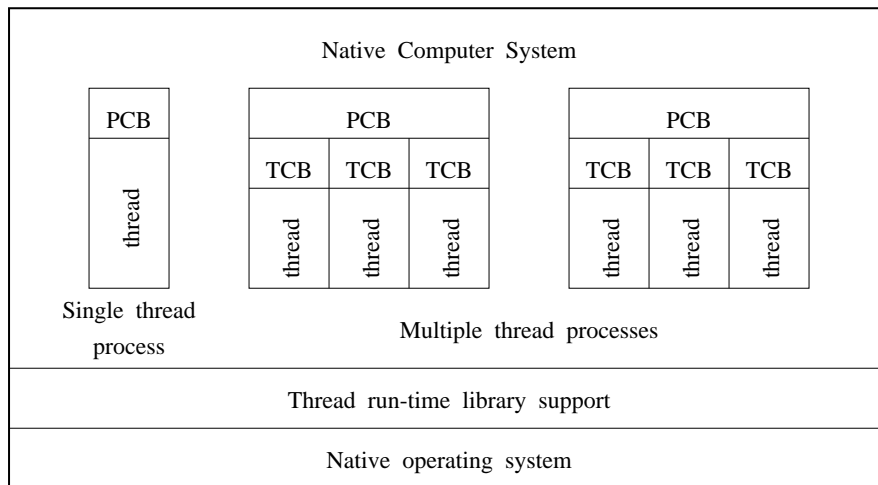
Chapter outline

- Thread implementations
- Process models
- The client/server model
- Time services
- Language constructs for synchronization
- Concurrent programming systems

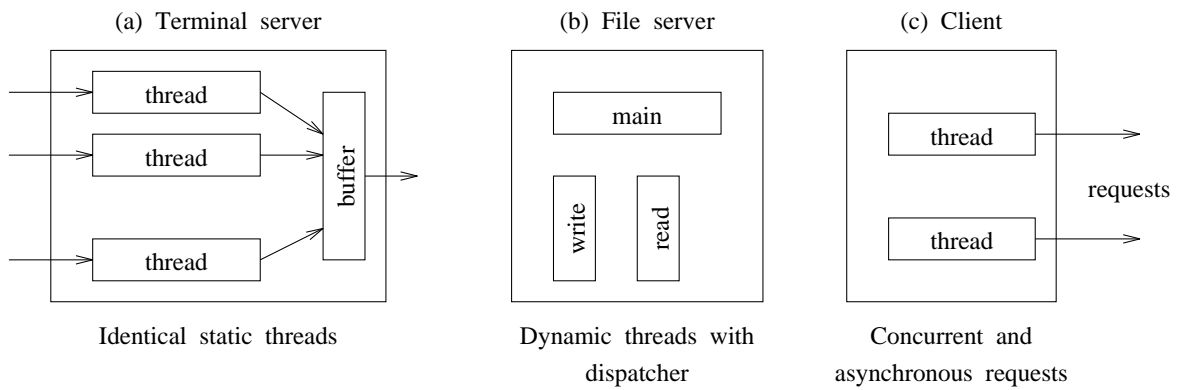
Processes and threads

- *Processes*: separate logical address space
- *Threads*: common logical address space
- Light weight context switching
- Blocking and scheduling

Two-level concurrency of processes and threads



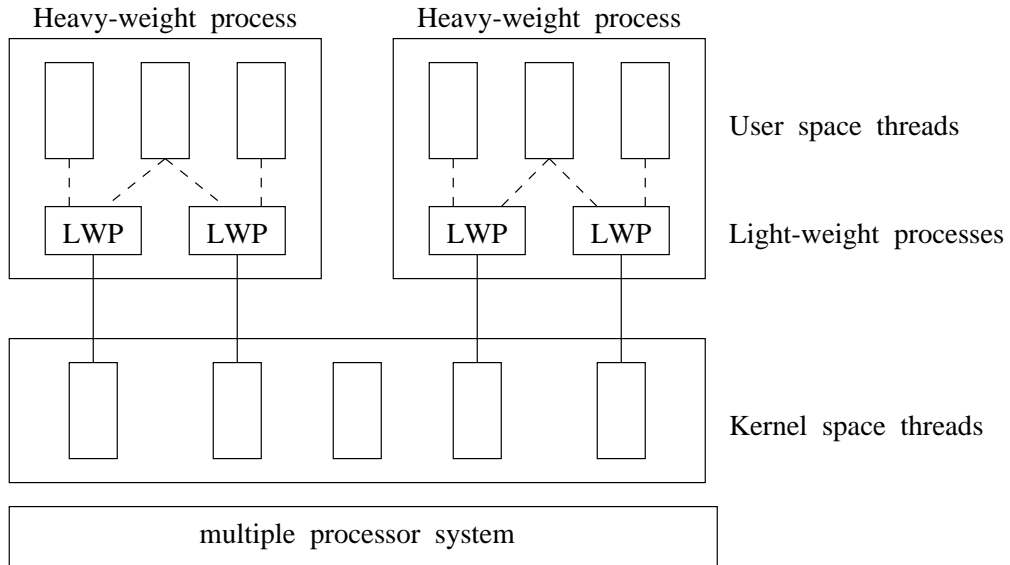
Thread applications



Thread implementations

- *User space*: simple but non-preemptable
- *Kernel space*: efficient but not portable

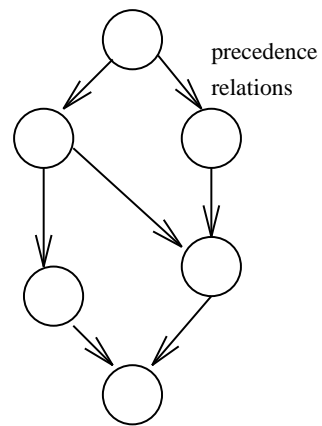
Solaris thread implementation



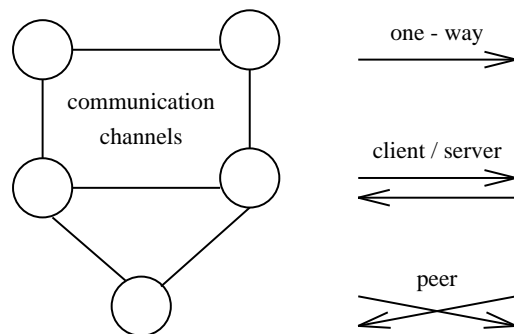
Process models

Synchronous Process, Asynchronous Communication, Time-Space

Graph representations

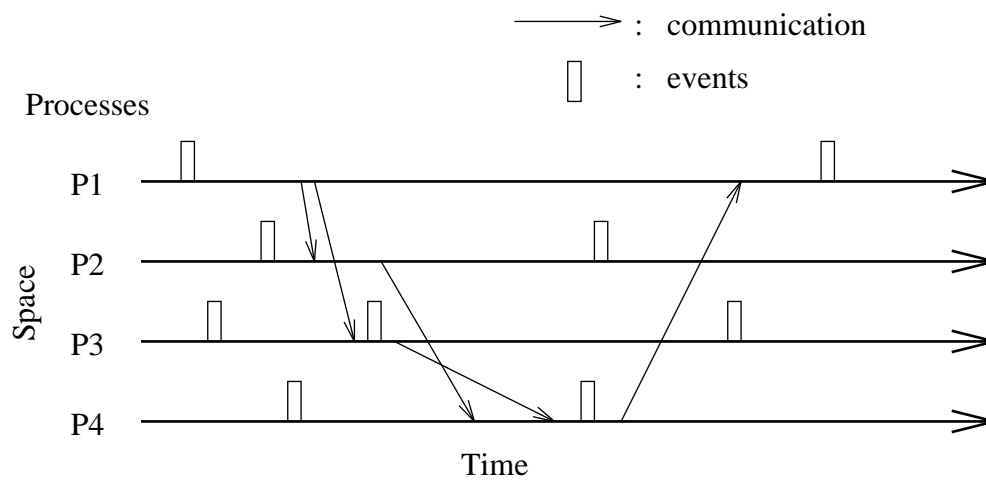


Synchronous process graph

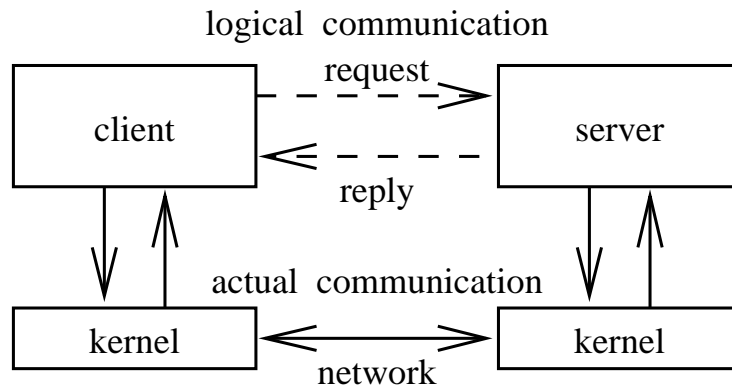


Asynchronous process graph and communication scenarios

Time-space model



Client/server model

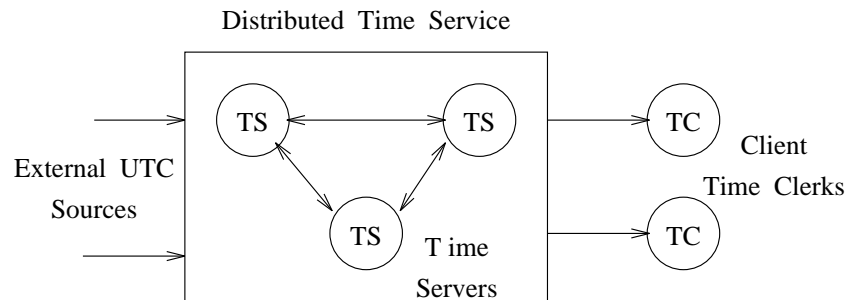


Time services

- time and timer
- physical and logical clocks

Physical clock

A distributed time service architecture



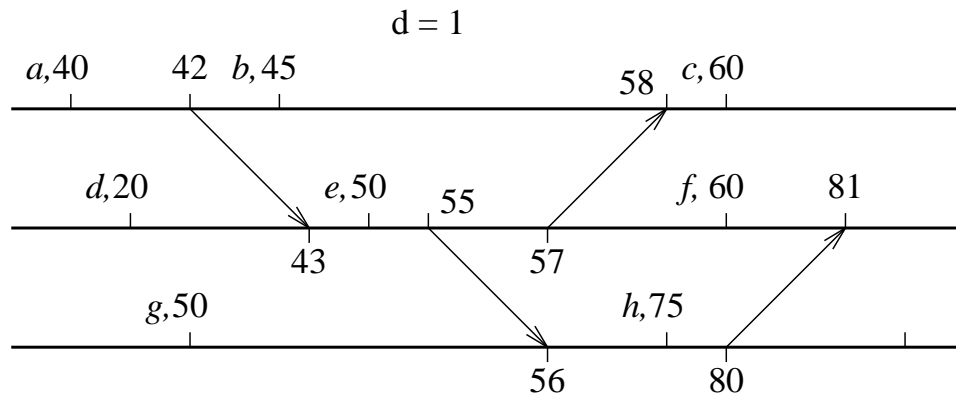
Logical clock

The *happens-before* relationship:

1. If $a \rightarrow b$ within a same process then $C(a) < C(b)$.
2. If a is the sending event of P_i and b is the corresponding receiving event of P_j , then $C_i(a) < C_j(b)$.

Implementation:

$C(b) = C(a) + d$ and $C_j(b) = \max(TS_a + d, C_j(b))$, where TS_a is the timestamp of the sending event and d is a positive number.

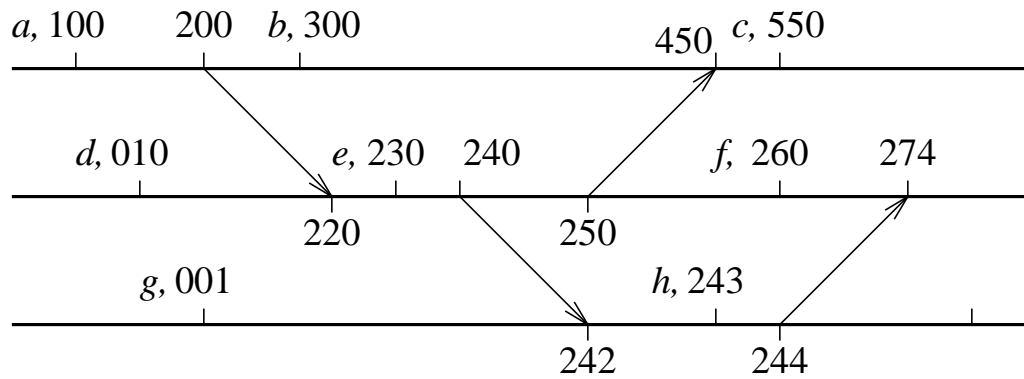


Vector logical clock

To tell if $C_i(a) < C_j(b)$ then $a \rightarrow b$.

Define $VC_i(a) = [TS_1, TS_2, \dots, C_i(a), \dots, TS_n]$, where n is the number of cooperating processes.

Use *pair-wise maximum*.



Matrix logical clock

$$MC_i[i, i] = MC_i[i, i] + d$$

$$MC_j[j, l] = \max(MC_j[j, l], TS_i[j, l]) \quad : \quad l = 1 \dots n$$

$$MC_j[k, l] = \max(MC_j[k, l], TS_i[k, l]) \quad : \quad k = 1 \dots n, \quad l = 1 \dots n$$

Concurrent languages

- Specification of concurrent activities
- Synchronization of processes
- Interprocess communication
- Nondeterministic execution of processes

Language constructs

- Program structure
- Data structure
- Control structure
- Procedure and system call
- Input and output
- Assignment

Synchronization mechanisms and language facilities

<i>Synchronization Methods</i>	<i>Language Facilities</i>
<i>Shared-Variable Synchronization</i>	
semaphore	shared variable and system call
monitor	data type abstraction
conditional critical region	control structure
serializer	data type and control structure
path expression	data type and program structure
<i>Message Passing Synchronization</i>	
communicating sequential processes	input and output
remote procedure call	procedure call
rendezvous	procedure call and communication

Shared-variable synchronization

- *Semaphore and conditional critical region*
- *Monitor and serializer*
- *Path expression*

Example: the reader/write problems, synchronization + concurrency

- *reader preference*
- *strong reader preference*
- *weak reader preference*
- *weaker reader preference*
- *writer preference*

Semaphore solution to the weak reader preference problem

```
var mutex, db: semaphore; rc: integer
```

reader processes

```
P(mutex)
```

```
rc := rc + 1
```

```
if rc = 1 then P(db)
```

```
V(mutex)
```

```
read database
```

```
P(mutex)
```

```
rc := rc - 1
```

```
if rc = 0 then V(db)
```

```
V(mutex)
```

writer processes

```
P(db)
```

```
write database
```

```
V(db)
```

Monitor solution

```
rw : monitor
var rc : integer; busy : boolean; toread, towrite : condition;

procedure startread
begin
if busy then toread.wait;
rc := rc + 1;
toread.signal;
end

procedure startwrite
begin
if busy or rc  $\neq$  0
then towrite.wait;
busy := true;
end

procedure endread
begin
rc := rc - 1;
if rc = 0 then towrite.signal;
end

procedure endwrite
begin
busy := false;
toread.signal or towrite.signal;
end

begin rc := 0; busy := false end
```

reader processes

```
rw.startread
read database
rw.endread
```

writer processes

```
rw.startwrite
write database
rw.endwrite
```

CCR solution

```
var db: shared; rc: integer;
```

reader processes

```
region db begin rc := rc + 1 end;  
read database  
region db begin rc := rc - 1 end;
```

writer processes

```
region db when rc = 0  
begin write database end
```

Serializer solution

```
rw : serializer
```

```
var readq, writeq: queue; rcrowd, wcrowd: crowd;
```

```
procedure read
```

```
begin
```

```
enqueue(readq) until empty(wcrowd);
```

```
joincrowd(rcrowd) then begin read database end;
```

```
end
```

```
procedure write
```

```
begin
```

```
enqueue(writeq) until (empty(wcrowd) and empty(rcrowd));
```

```
joincrowd(wcrowd) then begin write database end;
```

```
end
```

Path Expression solution

```
path 1:([read],write) end
```

Message Passing Synchronization

- *Asynchronous*: non-blocking send, blocking receive
- *Synchronous*: blocking send, blocking receive

Mutual exclusion using asyn. msg. passing

process P_i	channel server	process P_j
begin	begin	begin
receive(channel)	create channel	receive(channel)
critical section	send(channel)	critical section
send(channel)	manage channel	send(channel)
end	end	end

Mutual exclusion using syn. msg. passing

process P_i	semaphore server	process P_j
begin	loop	begin
send(sem,msg)	receive(pid,msg)	send(sem,msg)
critical section	send(pid,msg)	critical section
receive(sem,msg)	end	receive(sem,msg)
end		end

Communicating Sequential Processes (CSP)

$P: Q!exp$, $Q: P?var$, and *guarded* commands

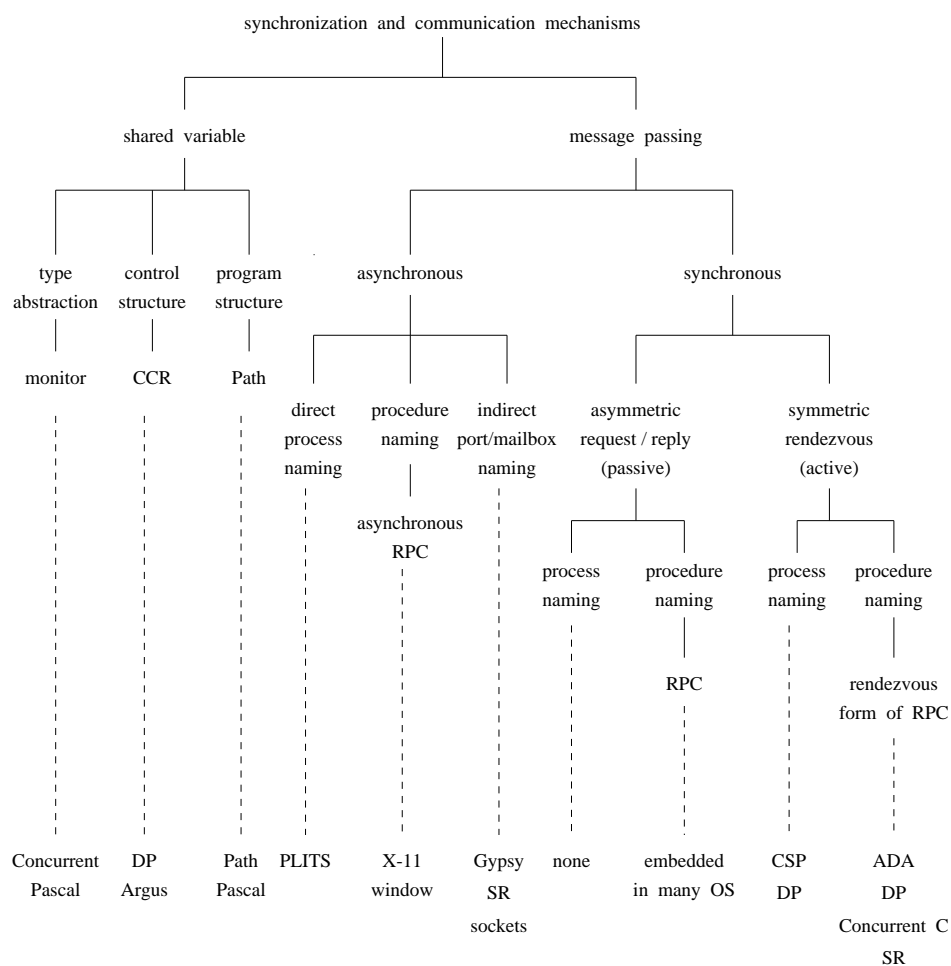
ADA rendezvous

```
task rw is
    entry startread;
    entry endread;
    entry startwrite;
    entry endwrite;
end

task body rw is
    rc: integer := 0;
    busy: boolean := false;
begin
loop
    select
        when busy = false →
            accept startread do rc := rc + 1 end;
        or
            →
            accept endread do rc := rc - 1 end;
        or
            when rc = 0 and busy = false →
                accept startwrite do busy = true end;
        or
            →
            accept endwrite do busy = false end;
    end loop
end;
```

Concurrent Programming Languages

A taxonomy



Coordination languages

- *OCCAM*: based on CSP process model, use PAR, ALT, and SEQ constructors, use explicit global links for communication.
- *SR*: based on resource (object) model, use synchronous CALL and asynchronous SEND and rendezvous IN, use *capability* for channel naming.
- *LINDA*: based on distributed data structure model, use tuples to represent both process and object, use blocking IN and RD and non-blocking OUT for communication.

	System	Object model	Channel naming
OCCAM	concurrent programming language	processes	static global channels
SR	concurrent programming language	resources	dynamic capabilities
LINDA	concurrent programming paradigm	distributed data structures	associative tags

Distributed and Network Programming

Programming languages for loosely coupled systems:

ORCA

fork process-name(parameters) [**on** (processor-number)];

operation op(parameters)
guard condition **do** statements;
guard condition **do** statements;

invoke(object, operation, parameters)

$t[1] = 6, A, 8$

$t[6] = 0, B, 0$

$t[8] = 0, C, 0$

JAVA

- Well-defined standard interfaces for integrating software modules
- Capability of running software modules on any machine
- Infrastructure for coordinating and transporting software modules

Applet and system security