

Advanced Synchronization

- Bloom paper (online)
- Chapter 6 from Silberschatz
- slides have many illustrative examples

Support for Critical Section

- **Hardware support:** Some hardware instructions are provided to support the programmer
 - (e.g., ‘test&set’, and ‘swap’ instructions)
- **Operating System Support:** Operating system supports for the declaration of data structures and also operations on those data structures
 - (e.g., semaphores)
- **High Level Language Support:** Language provides support for data structures and operations on them to help with synchronization.
 - (e.g., critical regions, monitors, serializers, etc)

Synchronization Building Blocks

- Most synchronization on symmetric multiprocessors is based on an atomic *test and set* instruction in hardware
 - we need to do a load and store atomically
- Example:

```
try_again:
ldstub address -> register
compare register, 0
branch_equal got_it
call go_to_sleep
jump try_again
got_it:
return
```

 - *ldstub*: load and store unsigned byte (SPARC)
- Other kinds of atomic primitives at the hardware level may be even more powerful
 - e.g., *compare_and_swap(mm, reg1, reg2)*
(if mm==reg1, mm=reg2)

Limits on low-level mechanism

- no abstraction and modularity
 - i.e., a process that uses a semaphore has to know which other processes use the semaphore, and how these processes use the semaphore
 - a process cannot be written in isolation
- error-prone
 - change order or omit lock/unlock, signal/wait
- difficult to verify correctness

Higher-level constructs Language Support

- monitors, serializers, path expressions, conditional critical regions, RWlocks...
- mainly programming languages targeted for concurrent programming and object oriented languages: Concurrent Pascal (Path Pascal), Concurrent C, Java, Modula, Ada, Mesa, Eiffel, ...

- use constructs provided by language and trust compiler to translate them
 - data structures need to be created (for queues, counts, locks, etc.)
 - lock/unlock and signal/wait primitives will be invoked on the right mutex/semaphore/condition variables
- at the lowest level these will most likely translate to test&set atomic instructions or OS-supported semaphores

Requirements of Synch Mech

- modularity
 - separation of resource and its access operations, and synchronizer and its mechanisms
- expressive power
 - specifying the needed constraints (exclusion and priority constrains in terms of relevant information...)
- ease of use
 - composing a set of constraints (for complex synch. schemes)
- modifiability
 - changing the constraints if needed
- correctness
 - safety net against inadvertent user errors

Basic Mechanisms

- mutex lock
- condition variables
- semaphores
 - Proposed in 1969 by Dijkstra for process synchronization
“Cooperating Sequential Processes”
 - P(S):
while $S \leq 0$ do nothing; // busy wait
 $S = S - 1$;
 - V(S):
 $S = S + 1$;
 - Init_Sem(S, Count): $S = \text{Count}$;
 - enables mutual exclusion, process synch;

Semaphore concept

- Integer variable, with initial non-negative value
- Two atomic operations
 - `wait`
 - `signal` (not the UNIX `signal()` call...)
 - `p` & `v` (*proberen* & *verhogen*), up and down, etc
- `wait`
 - wait for semaphore to become positive, then decrement by 1
- `signal`
 - increment semaphore by 1

Semaphore without busy-wait

```
struct sem {  
    value: int;  
    L: list of processes  
} S;  
Init_Sem(S, Count)  
{  
    S.value = Count;  
}
```

– OS support

- block
- wakeup
- P and V atomic

P(S)

```
{S.value = S.value - 1;  
if (S.value < 0) {  
    add T to S.L;  
    block;  
}
```

V(S)

```
{S.value = S.value + 1;  
if (S.value <= 0) {  
    select a T from S.L;  
    wakeup(T);  
}
```

Mutual exclusion using semaphores

- Use wait and signal like lock and unlock
 - bracket critical sections in the same manner
- Semaphore value should never be greater than 1
 - This is a *binary* semaphore
- Depends on correct order of calls by programmer
- All mutex problems apply (deadlock...)
- What should the initial value be?

Resource counting

- A *counting* semaphore can take on arbitrary positive values
 - max value usually system dependent but tunable
- General idea
 - initialize semaphore value to # of resources
 - wait = acquire resource
 - signal = relinquish resource
 - when all resources are taken, wait will... well, wait.
- Default semaphore type is usually counting
 - easy to make it a mutex

Monitors

- Monitors are a synchronization mechanism where the shared data are encapsulated with the locks required to access them
 - Variables/data local to a monitor cannot be directly accessed from outside.
- the data can only be accessed through the locked code

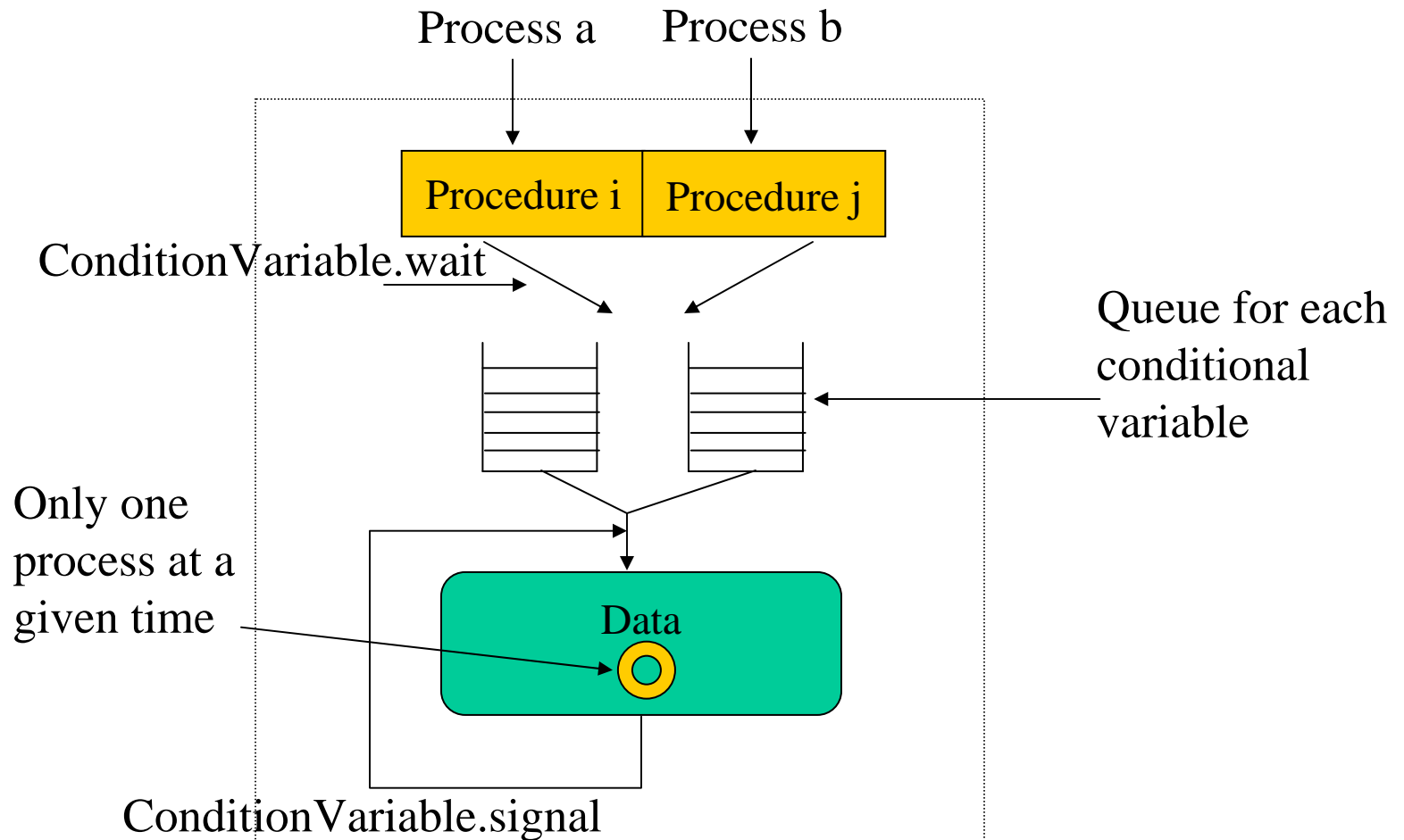
```
mon_name (ep1, ep2, ..., epn)
```

```
{  
    entry_point ep1    entry_point ep2    .....  
    {                  {                  .....  
    }                  }                  .....  
}
```

- all ep's are mutually exclusive
- multiple threads inside the monitor?
 - yes, but only one active
 - others?
 - waiting on an event

Monitors ..

Silberschatz textbook 6.7. example of Monitor implementation with semaphores (structure for each condition x, actions for x.wait and x.signal...)



Designing a Monitor

- When a process is active inside a monitor, other processes get queued up.
- Queuing up: done by operation *wait*. Dequeuing by *signal*.
- Queuing/dequeuing up: more than 1 reason possible. E.g., waiting for a reader to depart, waiting for a writer to finish, ..
- Condition variable may be associated with wait and signal.
 - E.g., OKtoread.wait, OKtoread.signal, ...
- Queues: generally FIFO, priorities may be implemented with a parameter.
- signaling threads immediately relinquish control of the monitor (in original definition)
 - this means they cannot signal multiple condition variables at the same time!

Monitor-style programming

- With mutexes and condition variables you can implement any critical section

CS_enter(); [controlled code] CS_exit();

```
void CS_enter() {
```

```
    Lock(m) {
```

```
        while (![condition])
```

```
            Wait(c, m)
```

```
        [change shared data
```

```
         to reflect in_CS]
```

```
        [broadcast/signal as needed]
```

```
    }
```

```
}
```

```
void CS_exit() {
```

```
    Lock(m) {
```

```
        [change shared data
```

```
         to reflect out_of_CS]
```

```
        [broadcast/signal as needed]
```

```
    }
```

```
}
```

- Readers/Writer example structure:

```
monitor
```

```
{ Condition OKtoread, OKtowrite;
```

```
  int readercount;
```

```
  // data decls
```

```
void StartRead() { ... }
```

```
void StartWrite() { ... }
```

```
void FinishRead() { ... }
```

```
void FinishWrite() { ... } }
```

Reader's Priority: Monitors

```
readers-writers: monitor;  
begin // the monitor  
    readercount: integer;  
    busy: boolean;  
    OKtoread, OKtowrite: condition;  
  
    procedure StartRead;  
        begin  
            if busy then OKtoread.wait;  
            readercount := readercount + 1;  
            OKtoread.signal; // all readers can start  
        end StartRead;  
  
    procedure EndRead;  
        begin  
            readercount := readercount - 1;  
            if readercount = 0 then OKtowrite.signal;  
        end EndRead;
```

Reader's Priority: Monitors ...

```
procedure StartWrite;
begin
    if busy OR readcount != 0 then
        OKtowrite.wait;
    busy := true;
end StartWrite;

procedure EndWrite;
begin
    busy := false;
    if OKtoread.queue then
        OKtoread.signal;
    else OKtowrite.signal;
end EndWrite;

begin // initialization
    readercount := 0; busy := false;
end;
end readers-writers.
```

Readers-Writers: Monitors

Reader:

```
    StartRead();  
    ReadFile();  
    EndRead();
```

Writer:

```
    StartWrite();  
    WriteFile();  
    EndWrite();
```

Monitors: Drawbacks

- Only one active process inside a monitor: no concurrency.
- Previous example: File NOT inside monitor to allow concurrency. -> Responsibility of readers and writers to ensure proper synchronization.
- Nested monitor calls can lead to deadlocks:
 - Consider monitors X and Y with procedures A and B. Let X.A call Y.B and vice-versa
 - A process P calls X.A, process Q calls Y.B.
 - P is blocked on Y.B and Q is blocked on X.A
- Responsibility of valid programs shifts to programmers, difficult to validate correctness.
- “low-level” – explicit signalling needed, no connection between abstract condition and signalling, signaller has to choose which queue to signal – explicit priorities...

- use
 - embed resource in mon (e.g. access to b-buffer)
 - problem?
all ops mutually exclusive
 - resource outside mon
 - permission to access inside mon
 - does not prevent resource being called directly
- monitors vs. semaphores?
 - exercise: implement one using the other
- scorecard for monitor
 - modularity and correctness (low)
 - ease of use, modifiability, expr. power (OK)

Monitors in Java

- keyword “synchronized” can be used to identify monitor regions (statements, methods, etc).
- compiler generates monitorenter and monitorexit bytecodes when monitor region is statements within a method
- JVM acquires and releases lock to corresponding object (or class)
- locks are recursive
- single condition variable associated with object that has monitor
- notify() semantic is Signal-and-Continue, i.e. have to verify that indeed the condition is true!

Example - pseudocode

```
import Utilities.*;
import Synchronization.*;

class BoundedBuffer extends MyObject {

    private int size = 0;
    private double[] buf = null;
    private int front = 0, rear = 0, count = 0;

    public BoundedBuffer(int size) {
        this.size = size;
        buf = new double[size];
    }
}
```

```
public synchronized void deposit(double data) {
    if (count == size) wait();
    buf[rear] = data;
    rear = (rear+1) % size;
    count++;
    if (count == 1) notify();
}

public synchronized double fetch() {
    double result;
    if (count == 0) wait();
    result = buf[front];
    front = (front+1) % size;
    count--;
    if (count == size-1) notify();
    return result;
}
}
```

Producer/Consumer

Bounded Buffer with Monitor

Monitor

```
PBBuffer b : BBuffer; // This is an unprotected
bounded buffer
count : Integer;
empty, full : condition;
```

```
procedure Init;
```

```
begin
```

```
    init(empty); init(full); init(b); count := 0;
```

```
end;
```

```
procedure Enqueue(I : Item)
begin
    if count == BufferSize(b) then wait(full);
        //BufferSize returns the maximum size of b
    count++;
    Enqueue(b, I);
    signal(empty);
end;

procedure Dequeue(I : out Item)
begin
    if count == 0 then wait(empty);
    count--;
    Dequeue(b, I);
    signal(full);
end;
```

Dining Philosophers Problem

[this solution is not *fair*]:

```
Monitor DiningPhilosophers
```

```
HMNY : constant integer = {the number of  
                             philosophers};
```

```
HMNY1 : constant integer = HMNY - 1;
```

```
state : array[0 .. HMNY1] of (thinking, hungry,  
                               eating) = (thinking, .., thinking);
```

```
self : array[0 .. HMNY] of condition;
```

```
{We assume self's conditions  
are initialized}
```

```
function Left(K : 0 .. HMNY1) return 0 .. HMNY1 is  
    {This is an operation only  
    used within the monitor}
```

```
begin
```

```
    return (K+1) mod HMNY;
```

```
end;
```

```
function Right(K : 0 .. HMNY1) return 0 .. HMNY1 is
    {This is an operation only
     used within the monitor}

begin
    return (K-1) mod HMNY;
end;

procedure Test(K : 0 .. HMNY1)
    {This is an operation only
     used within the monitor}

begin
    if state[Left(K)] /= eating and
       state[K] == hungry and
       state[Right(K)] /= eating
    then
        { state[K] = eating; signal(self[K]); }
    end;
end;
```

```
procedure Pickup(I : 0 .. HMNY1)
begin
    state[I] = hungry;
    Test(I);
    if state[I] /= eating
    then wait(self[I]);
end;
procedure PutDown(I : 0 .. HMNY1)
begin
    state[I] = thinking;
    Test(Left(I));
    Test(Right(I));
end;
```

Each philosopher P_i will execute a simple loop:

```
loop
  think;
  DiningPhilosophers.PickUp(i);
  eat;
  DiningPhilosophers.PutDown(i);
forever;
```

Semaphore with Monitor

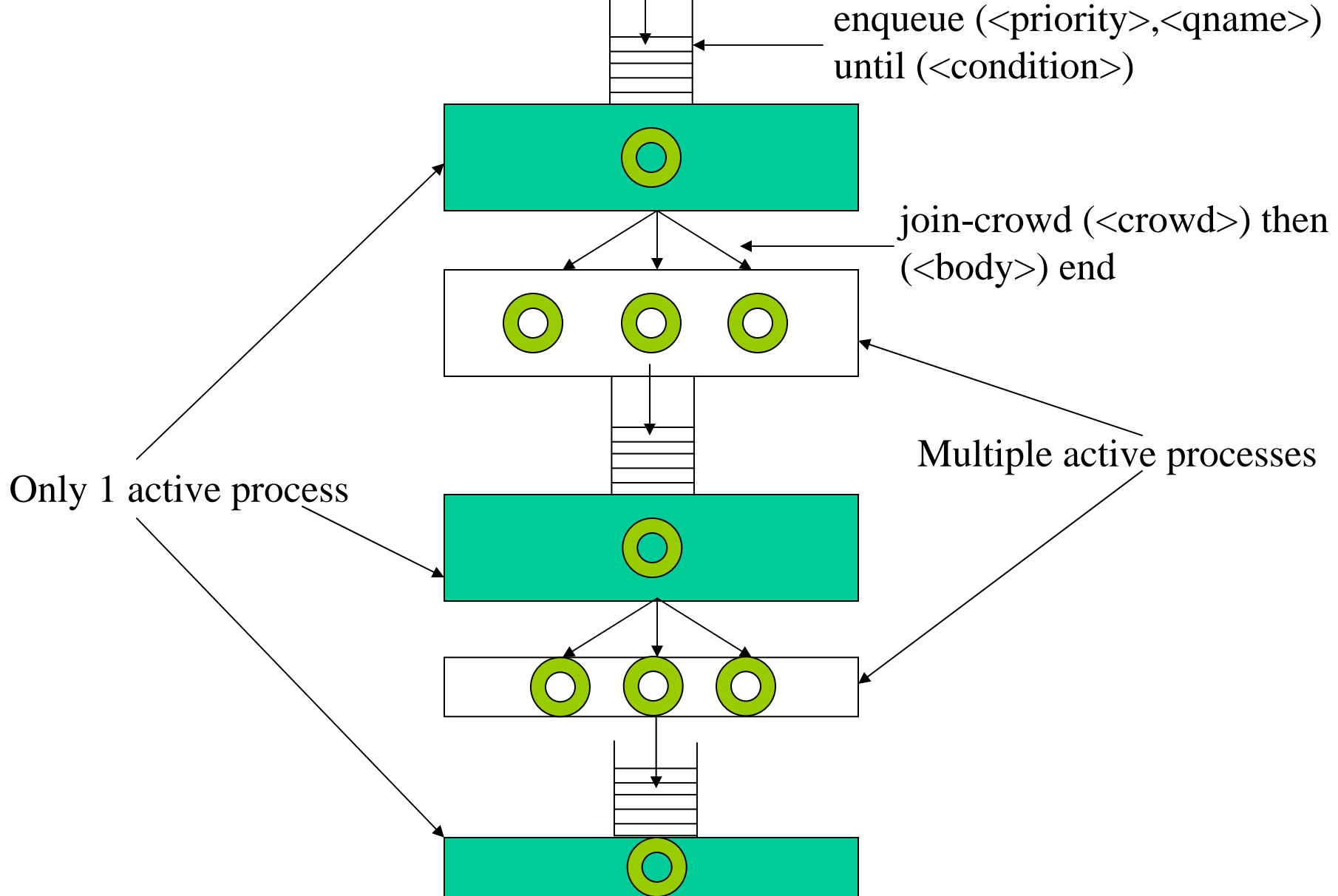
```
MONITOR monSemaphore;  
VAR  
    semvalue : INTEGER;  
    notbusy : CONDITION;  
PROCEDURE monP;  
BEGIN  
    IF (semvalue = 0) THEN  
        WAITC(notbusy)  
    ELSE  
        semvalue := semvalue - 1;  
END;
```

```
PROCEDURE monV;  
BEGIN  
    IF (EMPTY(notbusy)) THEN  
        semvalue := semvalue + 1  
    ELSE  
        SIGNALC(notbusy);  
END;  
BEGIN { initialization code }  
    semvalue := 1;  
END; // of monSemaphore  
monitor
```

Serializers

- **Serializers** was a mechanism proposed in 1979 to overcome some of the monitors' shortcomings
 - more automatic, high-level mechanism
 - basic structure is similar to monitors
- Data types
 - queue – takes place of CVs, but there is no signal op, the wait is replaced by enqueue
 - crowd
- operations
 - enqueue (queue_name) until <predicate>
 - join (crowd_name) then { <stmts> }
 - implicitly relinquishes control of serializer
 - empty(crowd_name or queue_name)
 - resource access

Serializers



serializer vs. monitor

- difference from monitor
 - resource ops can be inside the serializer (***)
 - no explicit signaling
 - explicit enqueueing on queues
 - automatic thread resumption (dequeue)
 - Serializers are similar to monitors with two main differences:
 - they allow concurrency
 - they have an automatic signalling mechanism
 - A serializer allows access to a resource without mutual exclusion, although the resource is inside the serializer
 - built in priorities:
 - threads on queue when condition true
 - threads leaving crowd
 - threads entering crowd

- Operation semantics:
 - enqueue is like Wait, only the Signal happens automatically when condition is true, the thread is at the head of the queue, and some other thread leaves the serializer
 - join_crowd leaves the serializer, executes a block without mutual exclusion, but returns to the serializer when the block finishes
- Use of Serializers
 - Usual sequence of events for a thread:
 - enter serializer
 - enqueue waiting for event (if needed)
 - dequeue (automatic)
 - join crowd to start using resource
 - leave crowd (automatic)
 - exit serializer

Reader's Priority: Serializers

```
Readerwriter: serializer
```

```
var
```

```
    readq: queue; writeq: queue;
```

```
    rcrowd: crowd; wcrowd: crowd;
```

```
    db: database;
```

```
procedure read(k:key; var data: datatype);
```

```
begin
```

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

```
    joincrowd (rcrowd) then
```

```
        data:= read-db(db[key]);
```

```
end
```

```
return (data);
```

```
end read;
```

```
procedure write(k:key, data:datatype);
```

```
begin
```

```
    enqueue(writeq) until
```

```
        (empty(rcrowd) AND empty(wcrowd) AND
```

```
        empty(readq));
```

```
    joincrowd (wcrowd) then write-db(db[key], data);
```

```
end
```

```
end write;
```

Readers-Writers in Serializers ...

- Weak reader's priority
 - enqueue(writeq) until
(empty(wcrowd) AND empty(rcrowd));
 - A writer does not wait until readq becomes empty
- Writer's priority
 - enqueue(writeq) until
(empty(wcrowd) AND empty(rcrowd));
 - enqueue(readq) until
(empty(wcrowd) AND empty(writeq));

Serializers: Drawbacks

- More complex, may be less efficient
- More work to be done by serializers
- “crowd” : complex data structure; stores identity of processes,...
- “queue”: count, semaphore, predicate...
- Assumes automatic signaling feature: test conditions of every process at the head of every queue every time a process comes out of a serializer.
- Though it (automatic signalling) helps in avoiding deadlocks and race conditions.

Serializers: pros and cons

- Pros:
 - clean and powerful model addresses monitors' drawbacks
 - allows concurrency of encapsulated resources
 - automatic signaling simplifies programming
- Cons
 - more complex so less efficient
 - automatic signaling requires testing conditions every time possession of serializer is relinquished
- scorecard for serializer
 - modularity and correctness (high); ease of use (high)
 - modifiability (OK); expr. power (OK)
 - efficiency (low)

Path Expressions

- Path expressions are declarative specifications of allowed behaviors of a concurrent program
 - synchronization is a mere side-effect of ordering the executions of operations on a resource
- To implement path expressions, a run-time system (path controller for each instance of shared resource) is needed to check the validity of orderings
 - it keeps track of operation start and end
 - it blocks threads if their execution of an operation will violate the path expression
 - *important: automatically unblocks when execution can go on*
- Path expressions do *not* cause the operations to be executed and do *not* determine who executes the operations

Path Expressions

- sync for data abstraction part of definition
- path expression specifies allowed orderings
- syntax: **path S end;**
 - S is an expression whose variables are the operations on the resource, and the operators are
 - ; (sequencing)
 - + (selection)
 - { } (concurrency)
 - path-end (repetition)
 - unsynch. access to ops not in path

Operator semantic

- ; (sequencing) defines a obliged sequencing order between operations.
 - no concurrency between operations
- + (selection) means only one of the operations can be executed at a time
 - the order of executions does not matter
- { } (concurrency) means any number of instances of the embraced operations can be executing at a time

- example
 - path {read} + write end
 - multiple readers or single writer
 - priority?
 - none
 - path expression does not cause op invocation
 - several path expressions in a module; the ordering has to be consistent with all paths
 - after a legal execution, another legal execution may follow
 - path open; read; close; end
 - sequentially one after the other
 - no mention of WHO executes the op in path expr.

- path A end
 - a sequence of As (the path between path and end can be repeated)
- path { A } end
 - concurrent As
- path { A;B }+C end
 - the nth B can start only after n As have finished
 - any C will be preceded by zero or n concurrent A;B, where all As and Bs have finished
- path { A + B };C end
 - the nth B can start independent of how many As have started or finished before it
 - all the As and Bs that have started have to complete before a C is allowed to go

- Readers/Writer (basic):
path { read } + write end
- Writes and reads interleaved (at least 1 read):
path write ; { read } end
- path a + b; c end
- path a + (b; c) end;
- path {a} + {b} end;
- path {a + b} end;
- path {a; b} end;
- path {(a;b) + c} end
- path {a; b+c} end

Usage

- Each path expression is associated with a single resource
- Path expressions are used encapsulated with the operations for a resource:

```
class file {  
  path write ; { read } end  
  void write(...) { ... };  
  int read() { ... } }
```

Path Expressions in Path Pascal

- Path Pascal – extension of Pascal that includes directives for specifying Path Expressions, and uses Object Encapsulations and Processes
- to specify priorities may need to introduce “artificial” operations
 - in Bloom’s paper, not just “read/write” but “start read/write”, “read/write”...

- Readers priority (weak):

path {read} + write end

- either several reads or a write

- Writer's priority:

path write end

path start_read + {start_write; write} end

path {start_read; read} + write end

- 3rd expression: no reader can execute start_read when a writer is writing
- 2nd expression: a writer can start a write when a writer is writing or when a reader is reading (start_read cannot be concurrent with start_write, however read can)
- 1st expression: only one reader at a time

```
/* A CS for up to 2 concurrent threads */
path request_enter + exit + do_enter end
// means these are mutually exclusive
path enter end
// means that instances are exclusive
path {wait ; release ; do_enter} + do_enter
end
count = 0;
private:
wait() { }
release() { }
request_enter() {if (count >= 2) wait();}
do_enter() { count++ }
public:
enter() { request_enter(); do_enter(); }
exit() { count--; release(); }
```

Producer/Consumer Problem

```
CONST nbuf = 5;
TYPE bufrange = 1..5;
ring = OBJECT
    PATH nbuf:(1:(put); 1:(get)) END;
    VAR buffer : ARRAY [bufrange] OF CHAR;
        inp, outp : bufrange;
    ENTRY PROCEDURE put(x : CHAR);
BEGIN inp := (inp MOD nbuf) + 1;
        buffer[inp] := x
END
    ENTRY FUNCTION get: CHAR;
BEGIN outp := (outp MOD nbuf) + 1;
        get := buffer[outp]
END
INIT;
BEGIN inp := nbuf;
        outp := nbuf
END
END (* end of OBJECT *)
```

```
VAR buf : ring;  
    c : CHAR;  
BEGIN buf.put('a');  
        c := buf.get  
END;
```

Dining philosopher Problem

```
CONST nphilopopers = 5;
      maxindex = 4; (* nphilopopers - 1 *)
TYPE diner = 0..maxindex;
VAR i: integer;
    table : OBJECT
      PATH maxindex:(starteating; stopeating) END;
      VAR fork: ARRAY [diner] OF OBJECT
          PATH 1:(pickup; putdown) END;
          ENTRY PROCEDURE pickup; BEGIN END;
          ENTRY PROCEDURE putdown; BEGIN END;
      END;
      ENTRY PROCEDURE starteating(no: diner);
      BEGIN fork[no].pickup;
          fork[(no+1) MOD nphilosophers].pickup;
      END;
      ENTRY PROCEDURE stopeating(no: diner);
      BEGIN fork[no].putdown;
          fork[(no+1) MOD nphilosophers].putdown;
      END;
END; (* table *)
```

```
PROCESS philosopher (mynum: diner);  
BEGIN REPEAT  
    delay(rand(seed));  
    table.starteating(mynum);  
    delay(rand(seed));  
    table.stopeating(mynum);  
UNTIL FALSE; END;  
  
(* main *)  
BEGIN  
    FOR i:= TO maxindex DO philosopher(i)  
END.
```

Comments on Path Expressions

- Path expressions can be complex
 - they may require the addition of “artificial” operations
 - it is not clear that a specification is correct when it spans several different resources (each with its own path expressions and operations depending on each other)
- But the specifications are declarative and centralized (i.e., there is no need to look through the code to find the synchronization primitives)
- For synchronization on a single resource, path expressions may be fine

– scorecard

- expressive power, modularity (high)
- priority constraints (low)
- correctness (depends)

ReaderWriter Locks

- Abstraction in Java, MS .NET, other places
- Manages read and write locks so you don't have to
- Also handles scheduling of blocked threads
 - Java implements `WriterPreference`, `ReaderPreference` classes which implement `ReadWriteLock` interface

RWLocks

- Bracket critical sections as with normal mutexes
- You say whether you're locking for read or write
 - it does the “right” thing
- Downgrade/upgrade
 - reduce overhead when switching from read to write
 - priority over those with no locks when upgrading
- Very useful for caches
 - or anywhere readers \gg writers – Java Swing GUI threads, etc
- For small critical sections, it may be overkill
 - need larger CS to get benefit of concurrent readers

Conditional Critical Regions

- Silverschatz textbook 6.6 (and implementation with semaphores)

```
var v: shared t
region v when {Condition} do Statements
```

- the *developer* does not have to program the semaphore or alternate synchronization explicitly
- the *compiler* ``automatically'' plugs in the synchronization code using predefined libraries
- once done carefully, *reduces* likelihood of mistakes in designing the delicate synchronization code

Synchronization support for concurrency problems using programming languages

- monitors (shared memory support)
 - Concurrent Pascal (Hansen), Modula (Wirth), Mesa (Xerox), uC++ (Buhr), Emerald (Raj)
- path expressions (shared memory support)
 - Path Pascal (Habermann and Campbell)
- message passing (non-shared memory support)
 - CSP: Communicating Sequential Processes (Hoare)
- serializers
 - Eifel
- RPC/rendezvous (non-shared memory support)
 - Ada for rendezvous

Lock free synchronization

- basic idea
 - if mostly just read operations, and no updates, then don't lock
 - instead:
 - catch update/write operations
 - allow to perform a change in a separate copy
 - wait until all current reads complete (or something similar) then apply the update