Theory of concurrency

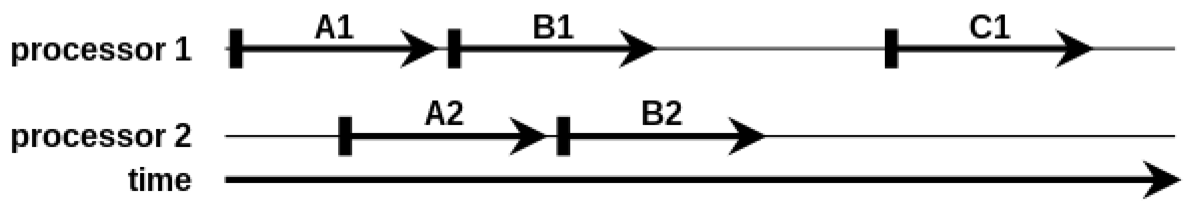
Lecture 14

Shared Resources

Shared Resources: **Shared storage**

- The behaviour of *systems* of concurrent processes can be implemented on
  - a *single* conventional stored program computer by *timesharing*:
    - a single processor executes each of the processes in *alternation*,
      - with process change on occurrence of *interrupt* from
        - an external *device* or from a regular *timer*.
- In this case, the concurrent processes may
  - *share locations* of common storage:
    - they are accessed and assigned by means of
      - the usual machine instructions within the code of the processes.

Shared Resources: Shared storage



Lamport: *sequential consistency* is met if

"the result of any execution is the same as

- if the operations of all the processors were executed in some sequential order,
- and the operations of each individual processor appear in this sequence in the order specified by its program."

- The *strong memory models*
  - processor architectures: AMD64, z/Architecture
  - programming languages: C/C ++ (default atomics), Java (volatile)
- The *weak memory models*
  - processor architectures: x86, Power, ARM, SPARC, DEC Alpha
  - programming languages: C/C ++, Java, OCaml.

Shared Resources: **Shared storage**

- A location of shared storage can be modelled a shared variable:

$$(count : VAR \mathbin{/\!/} (count.left\,!0 \rightarrow (P \mathbin{|||} Q)))$$

$$VAR = left\,?\,x \rightarrow VAR_x$$
$$VAR_x = (left\,?\,y \rightarrow VAR_y \mid right\,!\,x \rightarrow VAR_x)$$

- Shared storage vs. the local storage of sequential processes.
  - each variable is updated by at most one process
  - arbitrary interleaving of assignments from different processes.

Shared Resources: **Shared storage**

X1. (Interference) The shared variable *count* keeps a count of

- the total number of occurrences of some important event.

- On each occurrence of the event,
  - the relevant process P or Q tries to update the count by the pair of communications

$$count.right\ ?\ x;\ count.left\ !\ (x + 1)$$

- These two communications may be interleaved by
  - a similar pair of communications from the other process:

$$count.right\ ?\ x \rightarrow count.right\ ?\ y \rightarrow count.left\ !\ (y + 1) \rightarrow count.left\ !\ (x + 1) \rightarrow \ldots$$

- This kind of error is known as *interference*.
- The actual occurrence of the fault is *highly nondeterministic*
  - it is not reliably reproducible
  - it is very hard to diagnose the error by conventional testing techniques.

Shared Resources: **Shared storage**

- A *critical region* is a possible solution to this problem
  - to make sure that no change of process takes place during
    - a sequence of actions which must be protected from interleaving.

- On an implementation by a single processor,
  - the required exclusion can be achieved by
    - *inhibiting all interrupts* for the duration of the critical region.
- This solution has an undesirable effect in *delaying response to interrupts*; and
  - it fails completely as soon as *a second processor* is added to the computer.

- The binary exclusion *semaphore* by E. W. Dijkstra
  - is a better solution.

$LP = acquire \rightarrow \mu X \bullet (left \,?\, s \rightarrow h \,!\, s \rightarrow X \mid release \rightarrow LP)$

$lp.acquire \rightarrow \ldots\ lp.left \,!\,$ "A. JONES" $\rightarrow \ldots\ lp.left \,!\, nextline \rightarrow \ldots\ lp.release \rightarrow$

Shared Resources: Shared storage

- A *semaphore* is a process which engages alternatively in actions named P and V

$$SEM = (P \rightarrow V \rightarrow SEM)$$

$$(mutex : SEM \,/\!/ \ldots)$$

Probeer (try) and ***V***erhoog (increment)

- This is a shared resource
- Each process,
  - on entry into a critical region must send the signal $mutex.P$
  - on exit from the critical region must engage in the event $mutex.V$
- The critical region with the incremented count:

$$mutex.P \rightarrow count.right \,?\, x \rightarrow count.left \,!\, (x + 1) \rightarrow mutex.V \rightarrow \ldots$$

- Two processes cannot interfere with each other's updating of the count iff
  - all processes observe this discipline.
- If any process omits a P or a V, or gets them in the wrong order,
  - the effect is chaotic
    - risk a disastrous or a subtle error.

$$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$$
$$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$$

Shared Resources: **Shared storage**

- The design of the shared storage based on
  - knowledge of its intended pattern of usage
    - another way to prevent interference.
  - If a variable is only for counting, then
    - the increment operation is a single atomic operation *count.up*
    - the shared resource is CT_0

$$count : CT_0 \mathbin{/\!/} (\ldots P \mid\mid\mid Q \ldots)$$

- Shared resource specially designed for its purpose.
  - Pure (non-specialized) storage should not be shared in the design of a system using concurrency:
    - no accidental interference.
  - The design can be implemented efficiently on
    - networks of distributed processing elements, or
    - single-processor and multiprocessor computers with physically shared store.

$LP = acquire \rightarrow \mu X \bullet (left\ ?\ s \rightarrow h\ !\ s \rightarrow X \mid release \rightarrow LP)$

$lp.acquire \rightarrow \ldots\ lp.left\ !\ \text{“A. JONES”} \rightarrow \ldots\ lp.left\ !\ nextline \rightarrow \ldots\ lp.release \rightarrow$

Shared Resources: **Multiple resources**

- *A single serially reusable* resource
  - at any given time the resource is used by at most one of the sharing processes
    - alternating input and output, or
    - alternating acquire and release signals.

- Arrays of resources with identical behaviour
  - indices in the array ensure that
    - each element communicates safely with the process that has acquired it.

$$\|_{i<12} P_i = (P_0 \parallel P_1 \parallel \ldots \parallel P_{11}) \qquad |||_{i<4} P = (P\ |||\ P\ |||\ P\ |||\ P)$$

$$\|_{i\geq 0} P_i = (P_0 \parallel P_1 \parallel \ldots) \qquad \square_{i\geq 0}(f(i) \rightarrow P_i) = (f(0) \rightarrow P_0 \mid f(1) \rightarrow P_1 \mid \ldots)$$

  - f is a one-one function
    - the choice between the alternatives is made solely by the environment.

Shared Resources: **Multiple resources**

X1. (Re-entrant subroutine)

- A serially reusable shared subroutine is used by only one calling process at a time.

- If the execution of the subroutine requires a considerable calculation,
  - there could be corresponding *delays* to the calling processes.
- Several instances of the subroutine may proceed concurrently on different processors
  - if several processors are available to perform the calculations.
- A subroutine capable of several concurrent instances is *re-entrant*,
  - it is defined as an array of concurrent processes:

$$doub : (\|_{i<27} (i : DOUBLE)) \mathbin{/\!/} \ldots$$

- A typical call of this subroutine:

$$(doub.3.left\,!\,30 \rightarrow doub.3.right\,?\,y \rightarrow SKIP)$$

Shared Resources: **Multiple resources**

- The use of the index *3* ensures that
  - the result of the call is obtained from the same instance of *doub* to
    - which the arguments were sent,
  - even though some other concurrent process may
    - at the same time call another instance of the array:

$$doub.3.left.30, \ldots\ doub.2.left.20,$$
$$\ldots\ doub.3.right.60, \ldots\ doub.2.right.40, \ldots$$

- When a process calls a *re-entrant* subroutine,
  - it really does not matter *which* element of the array responds to the call.

- A calling process should leave the selection arbitrary
  - rather than specifying a particular index *2* or *3*

$$\square_{i \geq 0}(doub.i.left \,!\, 30 \rightarrow doub.i.right \,?\, y \rightarrow SKIP)$$

  - The same index is used for sending the arguments and for receiving the result.

Shared Resources: **Multiple resources**

- An arbitrary limit of 27 simultaneous activations of the subroutine.
- For single processor which divides its attention among a larger number of processes,
  - we introduce an infinite array of concurrent processes:

$$doub : (\|_{i \geq 0}\ i : D)$$

$$doub : (\|_{i<27}\ (i : DOUBLE)) \mathbin{/\!/} \ldots$$

  - D serves only a single call and then stop.

$$D = left\ ?\ x \rightarrow right\ !\ (x + x) \rightarrow STOP$$

- A *procedure* is a subroutine with no bound on its re-entrancy.
- In a procedure, the effect of each call

$$\square_{i \geq 0}(doub.i.left\ !\ x \rightarrow doub.i.right\ ?\ y \rightarrow SKIP)$$

  - is identical to the call of a subordinate process D
    - declared immediately adjacent to the call

$$(doub : D \mathbin{/\!/} (doub.left\ !\ x \rightarrow doub.right\ ?\ y \rightarrow SKIP))$$

Shared Resources: **Multiple resources**

- A *local* procedure call suggests execution of the procedure on
  - the same processor as the calling process;
- A *remote* call is the call of a shared procedure
  - it suggests execution on a separate possibly distant processor.
- The effect of remote and local calls is intended to be the same
  - the reasons for using the remote call can be political or economic
    - to keep the code of the procedure secret,
    - to run it on a machine with special expensive facilities.
      - a high-volume backing store: a disk or bubble memory.

LOC: $(doub : D \,/\!/\, (doub.left\ !\ x \rightarrow doub.right\ ?\ y \rightarrow SKIP))$

REM: $\square_{i \geq 0}(doub.i.left\ !\ x \rightarrow doub.i.right\ ?\ y \rightarrow SKIP)$

$COPY = \mu X \bullet (left \,?\, x \rightarrow right \,!\, x \rightarrow X)$

$VAR = left \,?\, x \rightarrow VAR_x$
$VAR_x = (left \,?\, y \rightarrow VAR_y \mid right \,!\, x \rightarrow VAR_x)$

Shared Resources: **Multiple resources**

X2. (Shared backing storage)

A storage medium is split into B sectors which can be read and written independently.

- Each sector stores one block of information,
  - which it inputs on the left and outputs on the right.
- The storage medium is implemented in a technology with *destructive read-out*,
  - each block written can be *read only once*.
  - Each sector behaves like *COPY* rather than *VAR*.
- The whole backing store is an array of such sectors: $BSTORE = \|_{i<B}\ i : COPY$
- This store is intended for use as a subordinate process $(back : BSTORE \,/\!/ \ldots)$
- Within its main process, the store may be used by the communications

$$back.i.left \,!\, bl \rightarrow \ldots \; back.i.right \,?\, y \rightarrow \ldots$$

Shared Resources: **Multiple resources**

X2. (Shared backing storage)

- The backing store may also be shared by concurrent processes.
- The action simultaneously *acquires* an arbitrary free sector with number i, and
  - *writes* the value of bl into it: $\square_{i<B}(back.i.left\ !\ bl \rightarrow \ldots)$
- The single action both *reads* the content of sector i into x and
  - *release* this sector for use on another occasion: $back.i.right\ ?\ x$

- Successful sharing of this backing store requires the utmost discipline:
  - A process may input from a sector only if
    - the same process has most recently output to that very sector.
  - Each output must eventually be followed by such an input.
- Failure to observe such disciplines will lead to deadlock, or even worse confusion.

Shared Resources: **Multiple resources**

X3. (Two line printers)

- Two identical line printers are available to serve the demands of a collection of processes.
- They both need the kind of protection from interleaving that was provided by *LP*.
- An array of two instances of *LP*:

$$LP2 = (0 : LP \parallel 1 : LP)$$

- This array may itself be given a name for use as a shared resource

$$(lp : LP2 \mathbin{/\!/} \ldots)$$

- Each instance of *LP* is now prefixed twice
  - once by a name and once by an index
  - communications with the using process have three or four components:

lp.0.acquire, lp.1.left.“A.JONES”,…

Shared Resources: **Multiple resources**

X3. (Two line printers)

- When a process needs to acquire one of an array of identical resources
  - it cannot matter which element of the array is selected on a given occasion.
    - as in the case of a re-entrant procedure.
  - any element which is ready to respond to the acquire signal will be acceptable:

$$\square_{i \geq 0}(lp.i.acquire \rightarrow \ldots\ lp.i.left\ !\ x \rightarrow \ldots\ lp.i.release \rightarrow SKIP)$$

- The initial *lp.i.acquire*
  - acquires whichever of the two *LP* processes is ready for this event.
    - If neither is ready, the acquiring process will wait.
    - If both are ready, the choice between them is nondeterministic.
- After the initial acquisition,
  - the bound variable *i* takes as its value the index of the selected resource, and
  - all subsequent communications will be correctly directed to that same resource.

$(doub : D \mathbin{/\!/} (doub.left\ !\ x \rightarrow doub.right\ ?\ y \rightarrow SKIP))$

Shared Resources: **Multiple resources**

X3. (Two line printers)

- The shared resource is intended to behave exactly like
  - a locally declared subordinate process, communicating only with its using process:

$$(myfile :: lp \mathbin{/\!/} \ldots\ myfile.left\ !\ x \ldots)$$

- instead of $\square_{i\geq 0}(lp.i.acquire \rightarrow \ldots\ lp.i.left\ !\ x \ldots\ ;\ lp.i.release \rightarrow SKIP)$
- The local name *myfile* is stand for the indexed name *lp.i*, and
  - the technicalities of acquisition and release have been conveniently suppressed.
- The *remote subordination* "::" is distinguished from the ":" in that
  - it takes on its right, not a complete process, but
    - the name of a remotely positioned array of processes.

$(lp : LP2 \mathbin{/\!/} \ldots)$

$LP2 = (0 : LP \parallel 1 : LP)$

$LP = acquire \rightarrow \mu X \bullet (left\ ?\ x \rightarrow h\ !\ s \rightarrow X \mid release \rightarrow LP)$

$\square_{i \geq 0}(lp.i.acquire \rightarrow \ldots\ lp.i.left\ !\ x \ldots\ ;\ lp.i.release \rightarrow SKIP)$

Shared Resources: **Multiple resources**

X4. (Two output files)

- A using process requires simultaneous use of two line printers
  - to output two files, *f1* and *f2*

$$(f1 :: lp \,/\!/\, (f2 :: lp \,/\!/ \ldots\ f1.left\ !\ s1 \rightarrow f2.left\ !\ s2 \rightarrow \ldots))$$

- The using process interleaves output of lines to the two different files;
  - each line is printed on the appropriate printer.
- Deadlock will be the result if
  - declare *three* printers simultaneously
  - declaring two printers simultaneously in each of two concurrent processes
    - Ann and Mary $(myfile :: lp \,/\!/\, (A \parallel M))$

$(f1 :: lp \,/\!/\, (f2 :: lp \,/\!/\, (f3 :: lp \,/\!/ \ldots$

$(lp : LP2 \,/\!/ \ldots)$ $\quad LP2 = (0 : LP \parallel 1 : LP)$

$LP = acquire \rightarrow \mu X \bullet (left\ ?\ x \rightarrow h\ !\ s \rightarrow X \mid release \rightarrow LP)$

Shared Resources: **Multiple resources**

X5. (Scratch file) A scratch file is used for output of a sequence of blocks.

- When the output is complete,
  - the entire sequence of blocks is read back from the beginning.
- When all the blocks have been read,
  - the scratch file will then give only *empty* signals; no further reading/writing.
- A scratch file behaves like
  - a file output to compact disc, which must be set to the start before being read.
- The *empty* signal serves as an end-of-file marker.

$$SCRATCH = WRITE_{<>}$$
$$WRITE_s = (left\ ?\ x \rightarrow WRITE_{s\frown<x>} \mid rewind \rightarrow READ_s)$$
$$READ_{<x>\frown s} = (right\ !\ x \rightarrow READ_s) \quad READ_{<>} = (empty \rightarrow READ_{<>})$$

- This may conveniently be used as a simple unshared subordinate process

$$(myfile : SCRATCH \mathbin{/\!/} \ldots\ mfile.left\ !\ v\ \ldots\ myfile.rewind\ \ldots$$
$$(myfile.right\ ?\ x \rightarrow \ldots \mid myfile.empty \rightarrow \ldots)\ldots)$$

Shared Resources: **Multiple resources**

X6. (Scratch files on backing store)

- The scratch file X5 can be implemented by
  - holding the stored sequence of blocks in the *main* store of a computer.
- If the blocks are large and the sequence is long,
  - it would be better to store the blocks on a *backing* store.
- A backing store X2 with destructive read-out will suffice
  - each block in a scratch file is read and written only once.
- An ordinary scratch file (held in main store) is used to
  - hold the sequence of indices of the sectors of backing store on which
    - the corresponding actual blocks of information are held;
    - this ensures that the correct blocks are read back, and in the correct sequence.

X2. $BSTORE = \|_{i<B}\ i : COPY$

X5. $SCRATCH = WRITE_{<>}$

$WRITE_s = (left\ ?\ x \rightarrow WRITE_{s\hat{}<x>}\ \mid\ rewind \rightarrow READ_s)$

$READ_{<x>\hat{}s} = (right\ !\ x \rightarrow READ_s)$ $\quad READ_{<>} = (empty \rightarrow READ_{<>})$

Shared Resources: **Multiple resources**

$$\begin{aligned}
&BSCRATCH = (pagetable : SCRATCH \mathbin{/\!/} \\
&\quad \mu\, X \bullet (left\ ?\ x \rightarrow (\ \Box_{i<B}\ back.i.left\ !\ x \rightarrow pagetable.left\ !\ i \rightarrow X) \\
&\qquad \mid rewind \rightarrow pagetable.rewind \rightarrow \\
&\qquad \mu\, Y \bullet (pagetable.right\ ?\ i \rightarrow back.i.right\ ?\ x \rightarrow right\ !\ x \rightarrow Y \\
&\qquad\quad \mid pagetable.empty \rightarrow empty \rightarrow Y)))
\end{aligned}$$

- *BSCRATCH* uses the name *back* to address a backing store X2 as a subordinate process:

$$SCRATCHB = (back : BSTORE \mathbin{/\!/} BSCRATCH)$$

- *SCRATCHB* can be used as a simple unshared subordinate process as in X5:

$$(myfile : SCRATCHB \mathbin{/\!/} \ldots\ myfile.left\ !\ v \ldots)$$

- The effect is exactly the same as use of *SCRATCH*, except that
  - the maximum length of the scratch file is limited to *B* blocks.

X2. $BSTORE = \|_{i<B}\ i : COPY$ $\quad COPY = \mu\, X \bullet (left\ ?\ x \rightarrow right\ !\ x \rightarrow X)$

Shared Resources: **Multiple resources**

X7 (Serially reused scratch files)

- Share the scratch file on backing store among a number of interleaved users:
  - they acquire, use, and release it one at a time (a shared line printer X3).
- Adapt *BSCRATCH* to accept *acquire* and *release* signals.
- If a user releases his scratch file before reading to the end,
  - the unread blocks on backing store may never be reclaimed.
- A loop reads back these blocks and discards them

  $$SCAN = \mu X \bullet (pagetable.right\,?\,i \rightarrow back.i.right\,?\,x \rightarrow X \mid pagetable.empty \rightarrow SKIP)$$

- A shared scratch file acquires its user, and then behaves as *BSCRATCH*:

  $$SHBSCR = acquire \rightarrow (BSCRATCH \triangle (release \rightarrow SCAN))$$

  - The release signal causes an interrupt to the *SCAN* process
- The serially reusable scratchfile is provided by $back : BSTORE \,/\!/\, {}^*SHBSCR$
  - the simple loop ${}^*SHBSCR$ which uses *BSTORE* as a subordinate process

Shared Resources: **Multiple resources**

X8. (Multiplexed scratch files)

- A backing store is usually sufficiently large for simultaneously existing many scratch files
  - each occupying a disjoint subset of the available sectors.
- The backing store can be shared among an *unbounded* array of scratch files.
- Each scratch file acquires a sector on outputting to it, and
  - releases it automatically on inputting that block again.
- The backing store is shared by the technique of multiple labelling,
  - using as labels the same indices which are used for the array of sharing processes

$FILESYS = \mathrm{N} : (back : BSTORE) \mathbin{/\!/} (\|_{i \geq 0}\ i : SHBSCR)$, where $\mathrm{N} = \{i \mid i \geq 0\}$

- This filing system is a subordinate process, shared by among any number of users:

$$filesys : FILESYS \mathbin{/\!/} \ldots (USER1 \mathbin{|||} USER2 \mathbin{|||} \ldots)$$

- In each user, a fresh scratch file is acquired, used, & released by remote subordination

$$myfile : filesys \mathbin{/\!/} (\ldots\ myfile.left\ !\ v \ldots\ myfle.rewid \ldots\ myfile.right\ ?\ x \ldots)$$

- which has the same effect as the simple subordination of a private scratch file X5:

$$(myfile : SCRATCH \mathbin{/\!/} \ldots\ myfile.left\ !\ v \ldots\ myfile.rewind \ldots\ myfile.right\ ?\ x \ldots)$$

Shared Resources: **Multiple resources**

- Inside a user processor a scratch file is created by remote subordination

$$myfile :: filesys \;/\!/\; (\ldots\ myfile.left\ !\ v \ldots\ myfile.rewind \ldots\ myfile.right\ ?\ x \ldots)$$

- By definition of remote subordination this is equivalent to

$$(\Box_{i \geq 0}\ filesys.i.acquire \rightarrow filesys.i.left\ !\ v \ldots\ filesys.i.rewind \ldots$$
$$filesys.i.right\ ?\ x \ldots\ filesys.i.release \rightarrow SKIP)$$

- All communications between *filesys* and its users begin with *filesys.i*... where
  - *i* is the index of the *particular instance* of *SHBSCR* which is
    - acquired by a particular user on a particular occasion.
    - Each occasion of its use is surrounded by a matching pair of signals
      - *filesys.i.acquire* and *filesys.i.release*.

Shared Resources: **Multiple resources**

- Each virtual scratchfile begins by acquiring its user, and
  - then continues according to the pattern of X5 and X6

$$(acquire \rightarrow \ldots\ left\ ?\ x \ldots\ rewind \ldots\ right\ !\ v \ldots\ release \ldots)$$

- All other communications of the virtual scratch file are
  - with the subordinate *BSTORE* process, and
  - concealed from the user.
- Each instance of the virtual scratch file is indexed by a different index *i*,
  - and then named by the name *filesys*.
- The externally visible behaviour of each instance is

$$(filesys.i.acquire \rightarrow filesys.i.left\ ?\ x \ldots\ filesys.i.rewind \ldots\ filesys.i.right\ !\ v \ldots\ filesys.i.release)$$

- The matching pairs of *acquire* and *release* signals ensure that
  - no user can interfere with a scratch file that has been acquired by another user.

Shared Resources: **Multiple resources**

- Communications within *FILESYS* between the virtual scratch files and the backing store.
  - are concealed from the user,
  - do not have the name *filesys* attached to them.
- The relevant events are:
  - $i.back.j.left.v$ — communication of block v from
    - the i^{th} element of the array of scratch files to the j^{th} sector of backing store
  - $i.back.j.right.v$ — a communication in the reverse direction.
- Each sector of the backing store behaves like *COPY*.
- After indexing with a sector number j and naming by *back*, the j^{th} sector behaves like

$$\mu X \bullet (back.j.left\ ?\ x \rightarrow back.j.right\ !\ x \rightarrow X)$$

- After multiple labelling by natural numbers it behaves like

$$\mu X \bullet (\square_{i \geq 0}\ i.back.j.left\ ?\ x \rightarrow (\square_{k \geq 0}\ k.back.j.right\ !\ x \rightarrow X))$$

  - communicate on any occasion with any element of the array of virtual scratch files.
- Each scratch file reads only from sectors the most recently written by it.

Shared Resources: **Multiple resources**

- The natural numbers i and j
  - permit any scratch file to communicate
    - with any sector on disc,
    - safely with the user that has acquired it.
  - a mathematical description of a kind of crossbar used in
    - a telephone exchange to allow any subscriber to communicate with another one.
- If the number of sectors in the backing store is infinite,
  - *FILESYS* behaves like an array of simple scratch files

$$\|_{i \geq 0}\ i : (acquire \rightarrow (SCRATCH \mathbin{\triangle} (release \rightarrow STOP)))$$

- With a backing store of finite size,
  - there is a danger of deadlock if
    - the backing store gets full at a time when all users are still writing to their files.
- In practice, this risk is usually reduced by
  - delaying acquisition of new files when the backing store is nearly full.

Shared Resources: **Multiple resources**

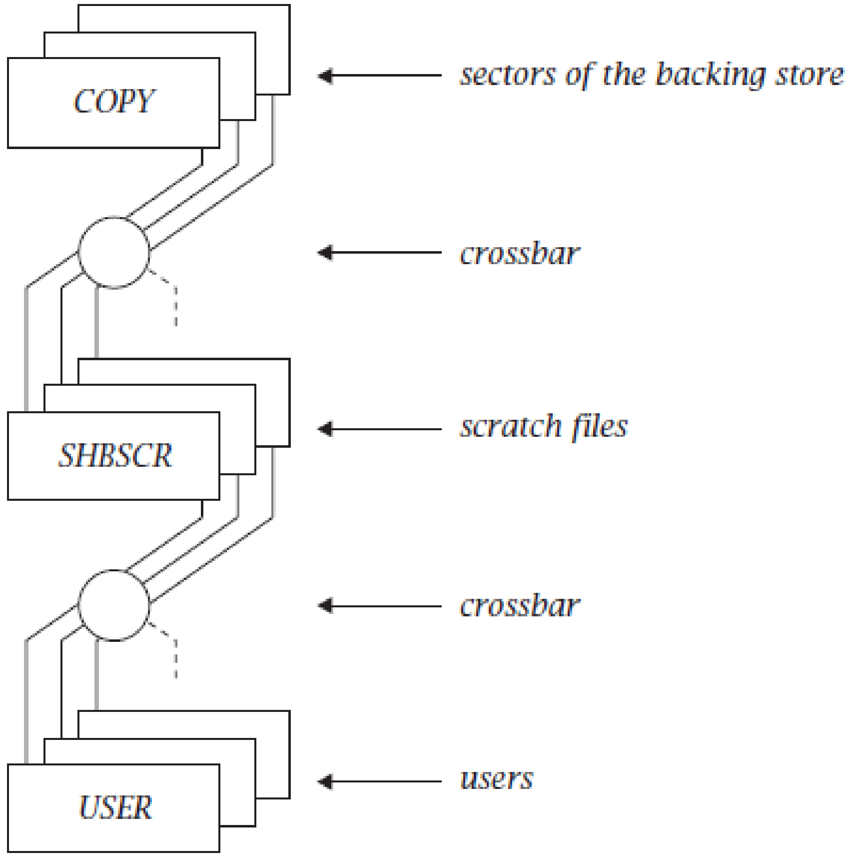


Figure 6.2

Shared Resources: **Multiple resources**

- The problem of sharing a *limited number* of resources among
  - an *unknown number* of users.
    - the structure of the filing system X8 and its mode of use.
- The users do not communicate directly with the resources;
  - there is an intermediary *virtual resource* (the *SHBSCR*) which
    - they declare and use as though it were a private subordinate process.
- The function of the virtual resource is twofold.

1. It provides a nice clean interface to the user;
   - *SHBSCR* glues together into a single contiguous scratch file
     - a set of sectors scattered on backing store.
2. It guarantees a proper, disciplined access to the actual resources;
   - the process *SHBSCR* ensures that each user reads only from sectors allocated to it,
     - and cannot forget to release sectors on finishing with a scratch file.

- Point (1) ensures that the discipline of Point (2) is painless.