



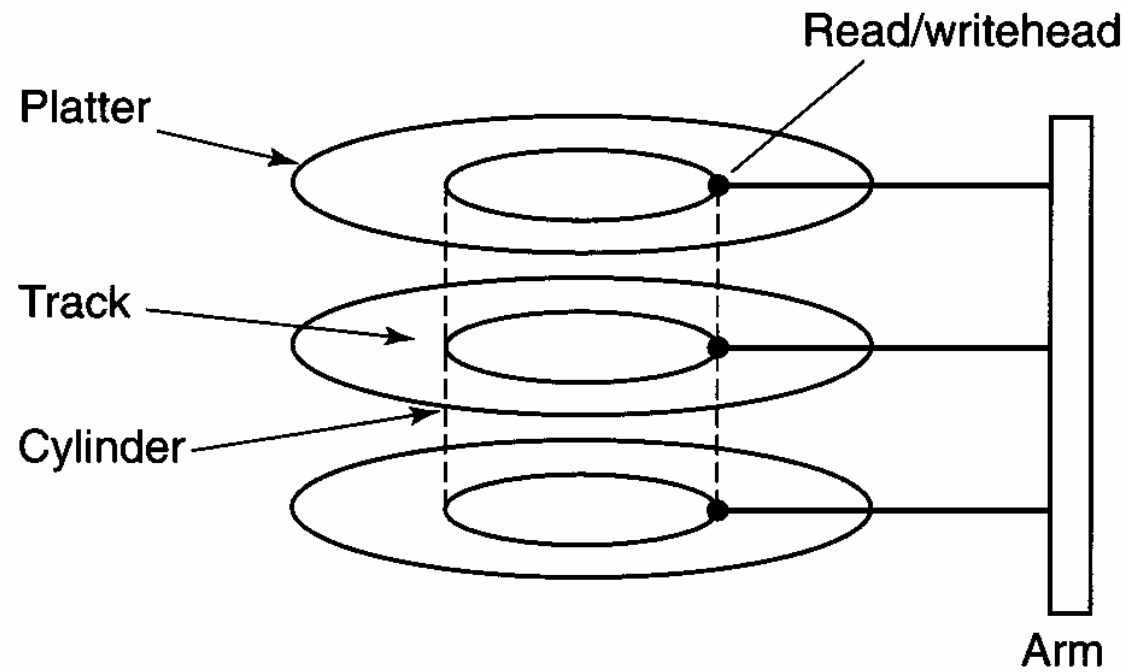
T-106.420

Concurrent Programming

More Monitor Examples



Example: Disk scheduling





Example: disk scheduling

- Physical disk structure
- physical disk address: cylinder, track number, offset (distance from a fixed clock point)
- disk I/O parameters: physical address, number of bytes to transfer, direction, address of data buffer



disk scheduling (1)

- disk access time depends on
 - seek time
 - rotational delay
 - data transmission time
- seek time \gg rotational delay
- we assume several clients that use the disk
not much sense to deny access when there's only one client who wants access



scheduling strategies

- SST – shortest seek time
 - select the pending request that is **closest** to the present position
 - unfair, can starve requests for positions far away
 - no bound on how long a request might be delayed
 - used in UNIX (buy more disks ! 😊)



scheduling strategies (1)

- SCAN/LOOK/ELEVATOR
 - move disk heads in one direction until all requests are served
 - reverse direction, when no more pending requests
 - fair scheduling
 - worst case for a pending request just behind the present track
 - large variance in expected waiting time



scheduling strategies (2)

- **CSCAN/CLOOK/**
 - **C** from circular
 - variance in *expected waiting time* greatly reduced
 - request serviced in *only one direction*, e.g. outermost to innermost cylinder
 - when no more requests ahead of current position then searching starts over again, e.g. from outermost



disk-scheduling solutions

1. with [separate monitor](#), see [readers/writers problem](#)
 2. with monitor as intermediary between *disk user processes* and the *disk driver process* actually doing the disk access
 3. with nested monitors
 - 1st performs scheduling and
 - 2nd performing disk access
- all three use the CSCAN strategy

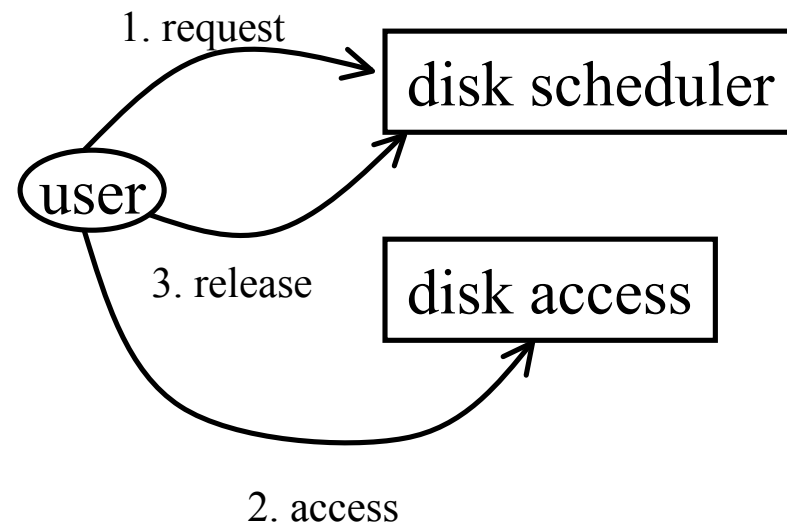


1. separate monitor

- 3 elements
 - user
 - disk scheduler (monitor)
 - disk access

- user interface is a protocol of three steps:

```
disk_scheduler.request(cyl);  
access the disk;  
disk_scheduler.release();
```





readers and writers: (reminder from the previous slideset)

```
monitor RW_Controller
  int nr=0, nw=0;          ## (nr == 0 v nw == 0) ^ nw <= 1
  cond oktoread,          # signaled when nw==0
    oktowrite;           # signaled when nr==0 and nw==0
  procedure request_read() {
    while (nw>0) wait(oktoread);
    nr=nr+1;
  }
  procedure release_read() {
    nr=nr-1;
    if (nr==0) signal(oktowrite); # awaken one writer
  }
  procedure request_write() {
    while (nr>0 OR nw>0) wait(oktowrite);
    nw=nw+1;
  }
  procedure release_write() {
    nw=nw-1;
    signal(oktowrite); # awaken one writer and
    signal_all(oktoread) # all readers (BOADCAST)
  }
}
```

11/13/05



separate disk scheduler monitor

- C: requests greater than current position
- N: requests less or equal than current position
- position: the current head position; -1 if no access
- invariant DISK:

C and N are ordered sets \wedge

all elements of set C are $>$ position \wedge

all elements of set N are \leq position \wedge

$(\text{position} == -1) \Rightarrow (C == \emptyset \wedge N == \emptyset)$



separate disk scheduler monitor (1)

```
monitor Disk_Scheduler {                                     ## Invariant DISK
  int position=-1, c=0, n=1;
  cond scan[2];                                           # signaled when disk released

  procedure request (int cyl) {
    if (position== -1) position=cyl;                       # disk is free, so return
    elseif (position != -1 && cyl > position) wait(scan[c],cyl);
    else wait(scan[n],cyl); }

  procedure release() {
    int temp;
    if (!empty(scan[c]))
      position=minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp=c; c=n; n=temp;          # swap c and n
      position = minrank(scan[c]); }
    else
      position=-1;
    signal(scan[c]); } } }
```



separate disk scheduler monitor (2)

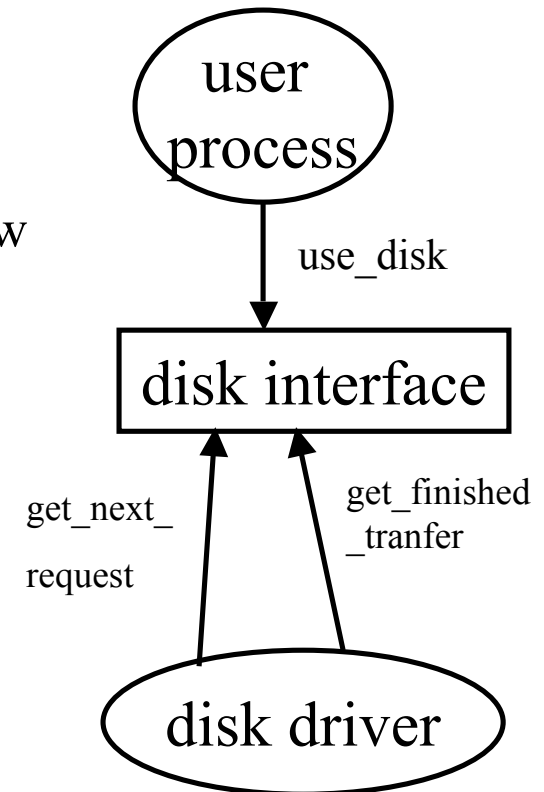
- problems
 - scheduler is visible to processes:

```
disk_scheduler.request(cyl);  
access the disk;  
disk_scheduler.release();
```
 - if scheduler is deleted, processes must be changed
 - all users **must** follow the required protocol, if any doesn't, the scheduling is *defeated*
 - after access is granted communication with the disk via
 - bounded buffers: adds 2 instances of monitors, i.e. 4 monitor calls/disk operation
 - sleeping barber: still separate monitor for interaction
- workaround
 - access to disk and disk scheduler embedded into a procedure (extra layer of procedures)



2. using monitor as an intermediary

- All disk accesses are delegated to a special disk driver process.
- User process communicate with it through a monitor with one call: `use_disk(params)`
- Benefits:
 - 1 call/disk operation: no multi-step protocol to follow
 - scheduling is transparent
 - easy to extend to several disks and drivers
- Solution is a version of the sleeping barber:
 - rename procedures
 - parameterize the monitor procedures, i.e. barber's chair, exit door → communication buffers
 - add scheduling, i.e. driver services the preferred request





disk interface monitor outline

```
monitor disk_interface {
    permanent variables for status, scheduling and data transfer
    procedure use_disk(int cyl, transfer and result parameters {
        wait for turn to use driver
        store transfer parameters in permanent variables
        wait for transfer to be completed
        retrieve results from permanent variables
    }
    procedure get_next_request(someType &results) {
        select next request
        wait for transfer parameters to be stored
        set results to transfer parameters
    }
    procedure get_finished_transfer(someType &results) {
        store results in permanent variables
        wait for results to be retrieved by client
    }
}
```



disk interface monitor / `procedure use_disk` wait for your turn to use driver

```
monitor Disk_Interface
.
.
procedure use_disk(int cyl; argType transfer_params; resultType
&result_params) {
  if (position==-1) position=cyl;
  elseif (position != -1 && cyl > position) wait(scan[c],cyl);
  else wait(scan[n],cyl);
  arg_area=transfer_params; args=args+1; signal(args_stored);
  while (results==0) wait(results_stored);
  result_params=result_area; results=results-1; signal(results_retrieved);
}
procedure get_next_request(argType &transfer_params;) {
.
.
}
procedure finished_transfer(resultType result_vals) {
.
.
  }}
}}
```



disk interface monitor / `procedure use_disk` store transfer parameters in permanent variables

```
monitor Disk_Interface
.
  argType arg_area; resultType result_area;
.
.
procedure use_disk(int cyl; argType transfer_params; resultType
&result_params) {
  if (position==-1) position=cyl;
  elseif (position != -1 && cyl > position) wait(scan[c],cyl);
  else wait(scan[n],cyl);
  arg_area=transfer_params; args=args+1; signal(args_stored);
  while (results==0) wait(results_stored);
  result_params=result_area; results=results-1; signal(results_retrieved);
}
procedure get_next_request(argType &transfer_params;) {
.
.
}
procedure finished_transfer(resultType result_vals) {
.
.
  }}
}
```



disk interface monitor / `procedure use_disk` wait for transfer to be completed

```
monitor Disk_Interface
.
.
procedure use_disk(int cyl; argType transfer_params; resultType
&result_params) {
    if (position==-1) position=cyl;
    elseif (position != -1 && cyl > position) wait(scan[c],cyl);
    else wait(scan[n],cyl);
    arg_area=transfer_params; args=args+1; signal(args_stored);
    while (results==0) wait(results_stored);
    result_params=result_area; results=results-1; signal(results_retrieved);
}
procedure get_next_request(argType &transfer_params;) {
.
.
}
procedure finished_transfer(resultType result_vals) {
.
.
    }}
}}
```



disk interface monitor / `procedure use_disk` retrieve results from permanent variables

```
monitor Disk_Interface
.
  argType arg_area; resultType result_area;
.
procedure use_disk(int cyl; argType transfer_params; resultType
&result_params) {
  if (position== -1) position=cyl;
  elseif (position != -1 && cyl > position) wait(scan[c],cyl);
  else wait(scan[n],cyl);
  arg_area=transfer_params; args=args+1; signal(args_stored);
  while (results==0) wait(results_stored);
  result_params=result_area; results=results-1; signal(results_retrieved);
}
procedure get_next_request(argType &transfer_params;) {
.
.
}
procedure finished_transfer(resultType result_vals) {
.
.
  }}
11/13/05
```



disk interface monitor / `procedure get_next_request` select next request

```
monitor Disk_Interface
..
procedure use_disk(int cyl; argType transfer_params; resultType
  &result_params) {
  .. }
procedure get_next_request(argType &transfer_params;) {
  int temp;
  if (!empty(scan[c])) position=minrank(scan[c]);
  elseif (empty(scan[c]) && !empty(scan[n])) {
    temp=c; c=n; n=temp; # swap c and n
    position = minrank(scan[c]);}
  else position=-1;
  signal(scan[c]);
  while (args==0) wait(args_stored);
  transfer_params=arg_area; args=args-1;  }

  procedure finished_transfer(resultType result_vals) {
    ..  }}
}
```



disk interface monitor / `procedure get_next_request` wait for transfer parameters to be stored

```
monitor Disk_Interface
..
procedure use_disk(int cyl; argType transfer_params; resultType
  &result_params) {
  .. }
procedure get_next_request(argType &transfer_params;) {
  int temp;
  if (!empty(scan[c])) position=minrank(scan[c]);
  elseif (empty(scan[c]) && !empty(scan[n])) {
    temp=c; c=n; n=temp; # swap c and n
    position = minrank(scan[c]);}
  else position=-1;
  signal(scan[c]);
  while (args==0) wait(args_stored);
  transfer_params=arg_area; args=args-1;  }
procedure finished_transfer(resultType result_vals) {
  ..  }}
```



disk interface monitor / `procedure get_next_request` set results to transfer parameters

```
monitor Disk_Interface
..
procedure use_disk(int cyl; argType transfer_params; resultType
  &result_params) {
  .. }
procedure get_next_request(argType &transfer_params;) {
  int temp;
  if (!empty(scan[c])) position=minrank(scan[c]);
  elseif (empty(scan[c]) && !empty(scan[n])) {
    temp=c; c=n; n=temp; # swap c and n
    position = minrank(scan[c]);}
  else position=-1;
  signal(scan[c]);
  while (args==0) wait(args_stored);
  transfer_params=arg_area; args=args-1;    }
procedure finished_transfer(resultType result_vals) {
  ..  }}
```



disk interface monitor / **procedure finished_transfer** store results in permanent variables

```
monitor Disk_Interface
..
procedure use_disk(int cyl; argType transfer_params;
  resultType &result_params) {
  .. }
procedure get_next_request(argType &transfer_params;) {
  .. }
procedure finished_transfer(resultType result_vals) {
  result_area = result_vals; results=results+1;
  signal(results_stored);
  while (results>0) wait(results_retrieved);  }}
```



disk interface monitor / **procedure finished_transfer** wait for results to be retrieved by client

```
monitor Disk_Interface
..
procedure use_disk(int cyl; argType transfer_params;
  resultType &result_params) {
  .. }
procedure get_next_request(argType &transfer_params;) {
  .. }
procedure finished_transfer(resultType result_vals) {
  result_area = result_vals; results=results+1;
  signal(results_stored);
  while (results>0) wait(results_retrieved);
}
}
```



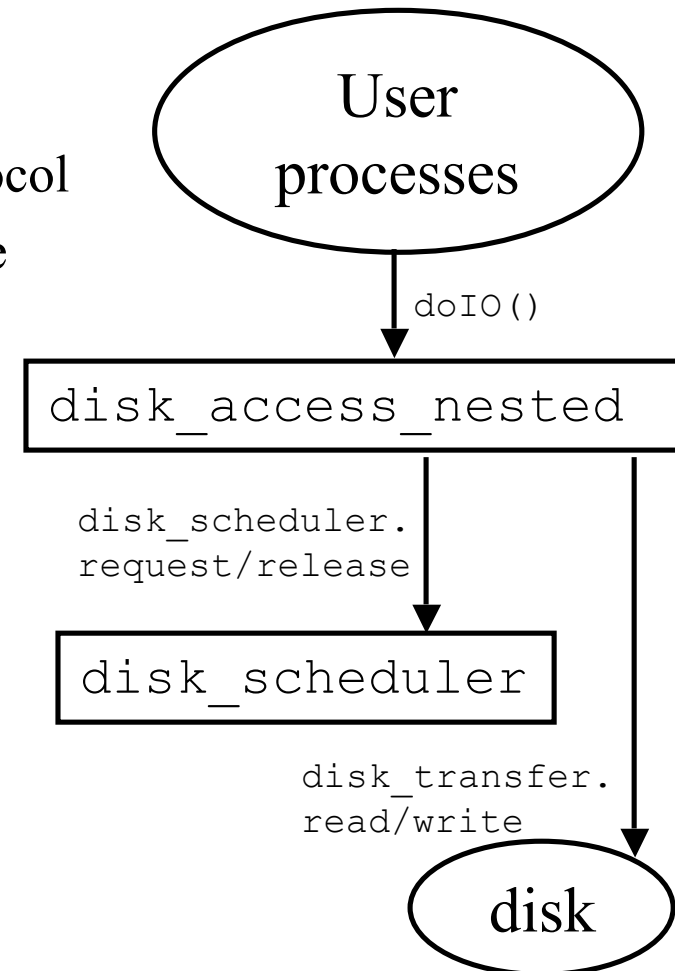
disk interface monitor: the whole story

```
monitor Disk_Interface {
  int position=-2,c=0,n=1,args=0,results=0;
  cond scan[2];
  cond args_stored, results_stored, results_retrieved;
  argType arg_area; resultType result_area;
  procedure use_disk(int cyl; argType transfer_params; resultType &result_params) {
    if (position==-1) position=cyl;
    elseif (position != -1 && cyl > position) wait(scan[c],cyl);
    else wait(scan[n],cyl);
    arg_area=transfer_params; args=args+1; signal(args_stored);
    while (results==0) wait(results_stored);
    result_params=result_area; results=results-1; signal(results_retrieved); }
  procedure get_next_request(argType &transfer_params;) {
    int temp;
    if (!empty(scan[c])) position=minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp=c; c=n; n=temp;    # swap c and n
      position = minrank(scan[c]);}
    else position=-1;
    signal(scan[c]);
    while (args==0) wait(args_stored);
    transfer_params=arg_area; args=args-1;  }
  procedure finished_transfer(resultType result_vals) {
    result_area = result_vals; results=results+1; signal(results_stored);
    while (results>0) wait(results_retrieved);  }
}
```



3. Using Nested Monitors

- Overview
 - separate monitor: users have to follow protocol
 - intermediary: single request, but much more sophisticated monitor
- Now: 2 monitors
 - disk access nested (caller)
 - scheduler (called)
 - calls from the monitor to another
 - **release exclusion** in the caller: *open call* (we use this)
 - **keep exclusion** in the caller: *closed call*





Problem: when making a nested monitor call is mutual exclusion of the calling monitor is released ?

- first alternative: YES - a open call
 - permanent variables are protected from concurrent access on an open call, as long as such variables are not passed by reference as arguments
 - the invariant of the calling monitor must be true when making nested call
- second alternative: NO - a closed call
 - if the process is delayed at a wait statement in the course of the nested call, it cannot be awakened by another process that has to make the same set of nested calls (deadlock danger!)
 - monitor invariant do not have to hold by the call



nested monitor disk interface outline

```
monitor disk_access_nested {  
  permanent variables as in disk_scheduler;  
  procedure doIO(int cyl, transfer and result parameters) {  
    actions of disk_scheduler.request;  
    call disk_transfer.read or disk_transfer.write;  
    actions of disk_scheduler.release;  
  }  
}
```

- **Why to use nested open calls ?**
 - otherwise only one process in doIO at a time (making request and release superfluous)
 - allowed because
 - only local variables (parameters to doIO)
 - disk scheduler invariant (DISK) is valid before read/write



mutual exclusion within monitors

- because one procedure executes at a time, permanent variables cannot be accessed concurrently
- Mutual exclusion of the monitor is not always necessary to avoid interference if:
 - the monitor procedure is “at-most-once” w.r.t monitor shared state: it can be executed concurrently with any monitor procedure.
 - e.g. add a "read_clock" procedure to the interval timer example



interval timer (priority wait) with read_clock

```
monitor Timer
  int tod=0;                ## Invariant CLOCK
  cond check;              # signaled when minrank(check) <= tod

  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
    if (wake_time > tod) wait(check, wake_time)
  }
  procedure tick () {
    tod=tod+1;
    while (!empty(check) ^ minrank(check) <= tod) signal(check);
  }
  procedure read_clock() {
    return tod      # the program calling it can know that the returned time is
                   # no greater than the current value of tod
  }
}
```