

1.5 Semaphores, Resource Allocation and Scheduling

Semaphores

- Busy waiting algorithms as discussed earlier are
 - Complex to implement
 - Inefficient in multiprogramming or multithreading, since CPUs can be held up spinning instead of performing useful computation
- Synchronisation is fundamental to concurrent programming
 - It is desirable to have dedicated tools to design synchronising concurrent programs more clearly and without busy waiting
 - Process and thread scheduling systems usually include a number of synchronisation mechanisms
- Semaphores are one of the most important synchronisation tools (see OS course)
 - Can implement mutual exclusion and condition synchronisation using semaphores



Semaphores

- A semaphore s is a special kind of shared variable
- A semaphore has a non-negative integer value
 - The initial value has a bearing on the behaviour of the semaphore
- A semaphore may only be manipulated by two atomic operations
 - Signal(s) or V(s)
 $\langle s := s + 1 \rangle$
 - Wait(s) or P(s)
 $\langle \text{AWAIT } (s > 0) \ s := s - 1 \rangle$
- Fairness
 - If $s > 0$ becomes and stays true, P(s) terminates if the scheduling policy is weakly fair
 - If $s > 0$ is infinitely often true, P(s) terminates if the scheduling policy is strongly fair
 - Since V(s) is unconditional, it terminates if the scheduling policy is unconditionally fair
 - Implementations of semaphores usually reawaken waiting processes in FIFO order, hence starvation is avoided



Critical Sections using Semaphores

- One semaphore, s , with initial value 1

SEQ

```
wait(s)
... critical section
signal(s)
... non-critical section
```

- Waiting on a semaphore can
 - Busy wait
 - Cause the scheduler to block the thread, thus freeing up the host CPU to execute other threads in the meantime

depending on the semaphore implementation in use



Barriers using Semaphores

- Two thread barrier using semaphores (two way synchronisation)

```
SEM arrive1 = 0, SEM arrive2 = 0:
```

```
PROC W1
...
  signal(arrive1)
  wait(arrive2)
...
PROC W2
...
  signal(arrive2)
  signal(arrive1)
...
```

- A signalling semaphore is one that is initialised to 0
- Use the above two thread barrier to build an n thread butterfly (or dissemination) barrier
 - Only one array of semaphores is needed, as the problem with resetting flags does not occur (signal operations cannot be lost, unlike flag setting)
- Semaphores can also be used to implement a central coordinator for a barrier, or a combining tree
 - Only one semaphore required for the coordinator



Split Binary Semaphores

- Binary semaphores can only hold values 0 or 1
 - Can be used as signalling flags
- Split binary semaphores are a pair of binary semaphores where the value of one is the opposite of the value of the other
- Split binary semaphores can be used in the producer-consumer problem with one buffer
 - empty semaphore: 1 = empty buffer, 0 = non-empty buffer
 - full semaphore: 1 = full buffer, 0 = non-full buffer
 - When empty is 0, full is 1, and vice-versa



Producer Consumer Problem

- Producers and consumers using split binary semaphores, one place buffer only

```
semaphore empty := 1, full := 0:  
item buf:
```

```
PROC producer
```

```
  WHILE TRUE  
  SEQ  
    -- produce data  
    wait(empty)  
    buf := data  
    signal(full)
```

```
:
```

```
PROC consumer
```

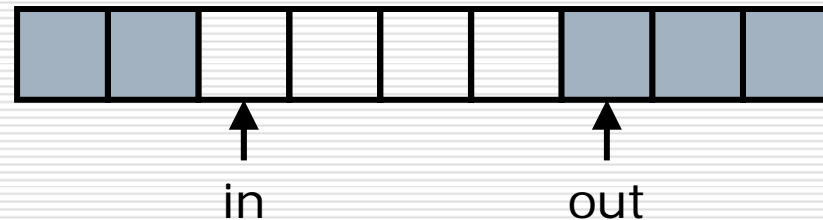
```
  WHILE TRUE  
  SEQ  
    wait(full)  
    result := buff  
    signal(empty)  
    -- consume result
```

```
:
```



Producer Consumer Problem

- To increase performance for bursty production and/or consumption of items, user a larger buffer capacity than one
 - Threads will block less frequently
 - This is the bounded buffer problem, as in OS course
 - Using semaphores as resource counters



Producer Consumer Problem

```
semaphore empty := n, full := 0:  
int in := 0, out := 0:  
[n]item buf:
```

```
PROC producer
```

```
  WHILE TRUE
```

```
    SEQ
```

```
      -- produce item and place in buf  
      wait(empty)  
      buf[in] := data  
      in := (in + 1) % n  
      signal(full)
```

```
:
```

```
PROC consumer
```

```
  WHILE TRUE
```

```
    SEQ
```

```
      wait(full)  
      result := buf[out]  
      out := (out + 1) % n  
      signal(empty)  
      -- consume item in result
```



Producer Consumer Problem

- The algorithm provided assumes one producer and one consumer
- For multiple producers and consumers we need critical sections (semaphores initialised to 1)

SEQ

```
wait(mutexP)
buf[in] := data
in := (in + 1) % n
signal(mutexP)
```

SEQ

```
wait(mutexC)
result := buf[out]
out := (out + 1) % n
signal(mutexC)
```



Readers/Writers Problem

- ❑ Reader threads and writer threads share some data
- ❑ Individual transactions in isolation maintain the shared data in a consistent state
- ❑ Uncontrolled sharing of the data may lead to inconsistencies in the shared data
- ❑ Selective mutual exclusion
 - Multiple reader threads may access the shared data concurrently
 - ❑ May have to wait for a writer thread
 - A writer thread requires exclusive access to the shared data
 - ❑ May have to wait for reader and writer threads
- ❑ Two solutions: mutual exclusion and condition synchronisation



Readers/Writers Problem

- Overconstrained solution

```
PROC reader
```

```
  WHILE TRUE
```

```
    SEQ
```

```
      ...
```

```
      wait(rw)
```

```
      -- read shared data
```

```
      signal(rw)
```

```
:
```

```
PROC writer
```

```
  WHILE TRUE
```

```
    SEQ
```

```
      ...
```

```
      wait(rw)
```

```
      -- write shared data
```

```
      signal(rw)
```

```
:
```



Readers/Writers Problem (mutex)

- Mutual exclusion approach: reader's preference solution gives priority to readers
 - Unfair: writers may be starved

```
SEMAPHORE rw := 1:
```

```
PROC writer
  WHILE TRUE
    SEQ
      ...
      wait(rw)
      -- write shared data
      signal(rw)
```



Readers/Writers Problem (mutex)

```
PROC reader
  WHILE TRUE
    SEQ
    < nr := nr + 1
    IF
      nr = 1
        wait(rw)
    TRUE
    SKIP
  >
  -- read shared data
  < nr := nr - 1
  IF
    nr = 0
      signal(rw)
  TRUE
  SKIP
>
:
```



Readers/Writers Problem (baton)

- Using condition synchronisation instead of mutual exclusion
- New technique: passing the baton
 - Uses split binary semaphores
 - Implements condition synchronisation
 - Controls waking order of delayed processes
- Count the number of readers (nr) and the number of writers (nw) trying to access the shared data
 - Constrain the counter values in order to avoid bad states
- The bad states are
 $(nr > 0 \wedge nw > 0) \vee nw > 1$
- It follows that the good states are given by global invariant
 $(nr = 0 \vee nw = 0) \wedge nw \leq 1$



Readers/Writers Problem (baton)

- Outline of reader and writer processes

SEQ

```
< nr := nr + 1 >  
-- read shared data  
< nr := nr - 1 >
```

SEQ

```
< nw := nw + 1 >  
-- write shared data  
< nw := nw - 1 >
```

- Need to guard the above assignments to ensure that the global invariant is preserved
 - Reader: ensure $nw = 0$ before incrementing nr
 - Writer: ensure $(nr = 0 \wedge nw = 0)$ before incrementing nw
 - No guards required on decrementing nr or nw



Readers/Writers Problem (baton)

```
INT nr := 0, nw := 0:
```

```
PROC reader
```

```
  WHILE TRUE
```

```
    SEQ
```

```
    ..
```

```
    < AWAIT (nw = 0) nr := nr + 1 >
```

```
    -- read shared data
```

```
    < nr := nr - 1 >
```

```
:
```

```
PROC writer
```

```
  WHILE TRUE
```

```
    SEQ
```

```
    ..
```

```
    < AWAIT (nr = 0 AND nw = 0) nw := nw + 1 >
```

```
    -- write shared data
```

```
    < nw := nw - 1 >
```

```
:
```



Passing the Baton

- Removing the await statements from the last algorithm is problematic
 - A single semaphore is not sufficiently expressive
- Need a general technique for implementing await statements (condition synchronisation)
 - 'passing the baton' is one solution
- To solve readers/writers problem we need
 - Binary semaphore e , initialised to 1, used for mutex
 - One semaphore and one counter for each await guard, all initialised to 0
 - The semaphores r and w will be used to block waiting readers and writers, respectively
 - The counter variables dr and dw will be used to count the number of waiting readers and writers, respectively



Passing the Baton: Reader

The reader code becomes

```
SEQ
  wait(e)
  IF
    nw > 0
      SEQ
        dr := dr + 1
        signal(e)
        wait(r)
      TRUE
    SKIP
  nr := nr + 1
  SIGNAL

  -- read shared data

  wait(e)
  nr := nr - 1
  SIGNAL
```



Passing the Baton: Writer

The writer code becomes

```
SEQ
  wait(e)
  IF
    (nr > 0) OR (nw > 0)
    SEQ
      dw := dw + 1
      signal(e)
      wait(w)
    TRUE
    SKIP
  nw := nw + 1
  SIGNAL

  -- write to shared data

  wait(e)
  nw := nw - 1
  SIGNAL
```



Passing the Baton: SIGNAL

The code for SIGNAL is

```
SEQ
  IF
    (nw = 0) AND (dr > 0)
      SEQ
        dr := dr - 1
        signal(r)
    (nr = 0) AND (nw = 0) and (dw > 0)
      SEQ
        dw := dw - 1
        signal(w)
  TRUE
    signal(e)
```

- SIGNAL will signal exactly one of the three semaphores



Passing the Baton

- Semaphores e , r and w form a split binary semaphore
 - At most one of them is 1 at any point in time
 - Every execution path starts with signal and ends with wait
 - Statements between wait and signal execute with mutual exclusion
- The invariant $(nr = 0 \vee nw = 0) \wedge nw \leq 1$ is true initially and just before each wait, so it is true whenever one of the semaphores is 1
- Each guard is guaranteed to be true when the statement it guards is executed
- No deadlock, since r or w are signalled only if some process is waiting on (or about to wait on) the semaphore



Passing the Baton

- Holding the baton indicates permission to execute a critical section
- Executing the SIGNAL code passes the baton to another process
 - If a process is waiting for a condition that is now true, the baton is passed to it
 - If no process is waiting on a condition, the baton is passed to the next process that executes wait(e)
- The SIGNAL code may be further simplified for specific cases
- This solution, like previous ones, gives preference to readers, but this prioritisation may be changed by modifying the SIGNAL code



Passing the Baton: Reader

```
SEQ
  wait(e)
  IF
    nw > 0
      SEQ
        dr := dr + 1
        signal(e)
        wait(r)
      TRUE
      SKIP
  nr := nr + 1
  IF
    dr > 0
      SEQ
        dr := dr - 1
        signal(r)
      TRUE
      signal(e)

-- read shared data

wait(e)
nr := nr - 1
IF
  (nr = 0) AND (dw > 0)
    SEQ
      dw := dw - 1
      signal(w)
    TRUE
    signal(e)
```



Passing the Baton: Writer

```
SEQ
  wait(e)
  IF
    (nr > 0) OR (nw > 0)
    SEQ
      dw := dw + 1
      signal(e)
      wait(w)
    TRUE
    SKIP
  nw := nw + 1
  signal(e)

-- write to shared data

wait(e)
nw := nw - 1
IF
  dr > 0
  SEQ
    dr := dr - 1
    signal(r)
  dw > 0
  SEQ
    dw := dw - 1
    signal(w)
  TRUE
  signal(e)
```



Readers/Writers: Writer Preference

- New readers are delayed if a writer is waiting: change first IF in reader to

```
IF
  (nw > 0) OR (dw > 0)
  SEQ
    dr := dr + 1
    signal(e)
    wait(r)
TRUE
SKIP
```

- A delayed reader is woken only if no writers are waiting: change final IF in writer to

```
IF
  dw > 0
  SEQ
    dw := dw - 1
    signal(w)
  dr > 0
  SEQ
    dr := dr - 1
    signal(r)
TRUE
  signal(e)
```



Readers/Writers: Fair

- The conditions in the code need to be changed to reflect the following fair behaviour
 - Delay a new reader when a writer is waiting
 - Delay a new writer when a reader is waiting
 - Awakening one waiting writer (if any) when readers finish
 - Awaken all waiting readers (if any) when a writer finishes; otherwise awaken one waiting writer (if any)



Resource Allocation and Scheduling

- Resource allocation is the problem of deciding when a process is granted access to a resource
 - Processes compete for use of units of a shared resource
 - A process requests one or more units, with parameters indicating the number of units and/or any other characteristics of the request
 - Each unit of the shared resource is either free or in use at any one point in time
 - The request may delay the process until it is satisfied, i.e. when all requested units are free
 - After using allocated resources, the process releases them to the free pool, possibly returning them in a different order/amounts
- We may arbitrarily order processes' access to resources using the 'passing the baton' technique



Resource Allocation and Scheduling

- Using the passing the baton technique
- Requesting a resource

```
SEQ
  wait(e)
  IF
    request not satisfied
    WAIT
  TRUE
    SKIP
  -- take units
  SIGNAL
```

- Releasing a resource

```
SEQ
  wait(e)
  -- return units
  SIGNAL
```

- There must be one semaphore for each delay condition
- The implementation of WAIT must somehow queue up the request and block the process on a condition semaphore
- The implementation of SIGNAL depends on the delay conditions for the particular problem



Shortest-Job-Next Allocation

- ❑ Several processes compete for the use of a single shared resource
- ❑ A request takes the form $\text{request}(\text{time}, \text{id})$, where time is the length of time for which the resource will be used and id is the requesting process id
- ❑ When a request is made, if the resource is free it is granted to the process; if not, the process waits
- ❑ After using a resource, a process releases it by calling release
- ❑ When the resource is released, it is allocated to the waiting process (if any) with the smallest time value in its request
- ❑ If two or more processes have the same time value, the resource is allocated to the process that has waited longest
- ❑ SJN minimises average completion time but is unfair under heavy resource contention, unless aging is used (see OS course)



Shortest-Job-Next Allocation

- Let **free** be a boolean variable that is true when the resource is available and false when it is in use
- Let **pairs** be a set of (time, id) pairs ordered by time; if two records have the same time value, use FIFO to order them
- The following is a global invariant
(**pairs** is an ordered set \wedge **free**) \Rightarrow (**pairs** = \emptyset)
- When a request cannot be satisfied, WAIT is called
 - Insert request parameters in **pairs**
 - Release critical section: signal(e)
 - Wait on a condition semaphore until request can be satisfied
- When the resource is freed, if **pairs** is not empty, choose a request from **pairs** in accordance with SJN and signal a semaphore to free it up



Shortest-Job-Next Allocation

- In this case, we need one semaphore per process waiting in **pairs**, since if we were to queue up multiple processes on a single semaphore they would be woken up according to semaphore's queuing policy not SJN
 - A semaphore is a **private semaphore** if exactly one process waits on it
 - In this case we use one private semaphore per waiting process; in other cases, where the queuing order within a single condition is unimportant, we use one semaphore per condition instead
- The variables required by the SJN algorithm follow

BOOLEAN `free` := **TRUE**:

SEMAPHORE `e` := **1**:

[**n**] **SEMAPHORE** `b` := [0, 0, ..., 0]:

ORDEREDPAIRSET `pairs`:



Shortest-Job-Next Allocation

```
PROC request(INT time, id)
  SEQ
  wait(e)
  IF
    NOT free
      SEQ
        insert (time, id) in pairs
        signal(e)
        wait(b[id])
      TRUE
      SKIP
  free := FALSE
  signal(e)
:
```

```
PROC release
  SEQ
  wait(e)
  IF
    pairs is not empty
      SEQ
        remove first pair
        signal(b[id])
      TRUE
      signal(e)
:
```



Shortest-Job-Next Allocation

- The algorithm can be generalised to handle resources with multiple allocation units
 - Request and release have an additional parameter, amount
 - Free is replaced with a variable that keeps track of available units
 - Request tests whether $\text{amount} \leq \text{available}$ before allocating
 - Release increases available by amount and check if the process at the head of **pairs** can now be satisfied

