

Hardware Solution: TSL instruction (*)

- Test and Set Lock (TSL) instruction
 - **tsl A, lock**
 - Copies contents of lock variable (stored in memory) to CPU register A ($A = lock$)
 - Sets lock variable to a non-zero value ($lock = 1$)
 - Operation is guaranteed atomic for N processors accessing a common memory
 - When process executes the TSL instruction it must run to completion or be restarted if an interrupt occurs
 - For N processors the bus is locked for the duration of the TSL instruction, thus concurrent access to the lock variable memory is impossible
 - Using the TSL instruction to solve the mutual exclusion problem

| | |
|---|--|
| <pre> TSL_enter_region: tsl A, lock cmp A, #0 jne TSL_enter_region ret </pre> | <pre> TSL_leave_region: move lock, #0 ret </pre> |
|---|--|
 - ...


```

move lock, #0          ; initialise lock to #0
call TSL_enter_region
call critical_section
call TSL_leave_region
          
```
 - Bounded waiting may not be satisfied
 - Processes compete to execute TSL and low priority processes may be starved

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.10

How does TSL solve problem?

Process executes `tsl A, lock` in order to gain access. If it finds $lock == 0$ then it proceeds (and resets $lock$ to 1), but if it finds $lock == 1$ then it loops back (`jne enter_region`) and will keep doing so until $lock == 0$. Other processes will see $lock = 1$ and similarly loop. When process leaves it sets $lock == 0$, thereby allowing another process to enter. Since TSL is atomic the processes will take turns executing TSL.

Semaphores: Producer-Consumer (*)

```

#define N 100          /* number of slots in the buffer */

semaphore mutex = 1; /* controls access to critical region */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0;  /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) { /* TRUE is the constant 1 */
        produce_item(&item); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        enter_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) { /* infinite loop */
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        remove_item(&item); /* take item from buffer */
        up(&mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}

```

• Figure 2.12, osdi2: Producer-Consumer using semaphores

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.17

mutex semaphore: mutual exclusion for access to shared buffer

full semaphore: used by consumer to wait when (full == 0) and producer to signal consumer accordingly

empty semaphore: used by producer to wait when (empty == 0) and consumer to signal producer accordingly

Initially (empty = N) and (full = 0) to indicate N-slot buffer is empty

Also (in = 0) and (out = 0) for enter_item() and remove_item():

| | |
|--------------------|----------------------|
| enter_item(item) | remove_item(&item) |
| { | { |
| buffer[in] = item; | item = buffer[out]; |
| in = (in + 1) % N; | out = (out + 1) % N; |
| } | } |

Message Passing: Mutual Exclusion (*)

- Use mailbox form of message passing
 - Main process creates and initialises mailbox
 - `create_mailbox(mutex); /* mutex is the mailbox destination */`
 - `send(mutex, msg);`
 - All processes execute:
 - ...
 - `receive(mutex, msg); /* msg can contain dummy data */`
 - `critical_region();`
 - `send(mutex, msg);`
 - ...
- Use port form of message passing
 - *Lock server* process listens on port LOCK
 - Simulates mutual exclusion operation for distributed processes
 - Relies on messages containing the source ID so server can identify client requests
 - All processes execute:
 - ...
 - `send(LOCK, "down"); /* indicate to lock server the semaphore down operation */`
 - `receive(LOCK, "go"); /* wait for lock server to allow access */`
 - `critical_region();`
 - `send(LOCK, "up"); /* indicate to lock server the semaphore up operation */`
 - ...

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.23

Mailbox

Possible with UNIX DOMAIN sockets but not INET DOMAIN sockets

→ need a lock server with INET DOMAIN sockets

Readers-Writers: Semaphore Solution (*)

- Readers have priority

```

semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the data base */
int rc = 0; /* # of processes reading or wanting to */

void reader(void)
{
  while (TRUE) { /* repeat forever */
    down(&mutex); /* get exclusive access to 'rc' */
    rc = rc + 1; /* one reader more now */
    if (rc == 1) down(&db); /* if this is the first reader */
    up(&mutex); /* release exclusive access to 'rc' */
    read_data_base(); /* access the data */
    down(&mutex); /* get exclusive access to 'rc' */
    rc = rc - 1; /* one reader fewer now */
    if (rc == 0) up(&db); /* if this is the last reader ... */
    up(&mutex); /* release exclusive access to 'rc' */
    use_data_read(); /* noncritical region */
  }
}

void writer(void)
{
  while (TRUE) { /* repeat forever */
    think_up_data(); /* noncritical region */
    down(&db); /* get exclusive access */
    write_data_base(); /* update the data */
    up(&db); /* release exclusive access */
  }
}

```

• Figure 2.19, osdi2: Readers-Writers with semaphores

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.26

How does it work?

First reader will block if there are writers, subsequent readers will not bother checking since a reader with access implies all readers can have access

Writer will only enter when the last reader leaves but readers enter immediately → as long as readers arrive there is no last reader and writer starves

Example Scenario:

R1 enter (**rc** = 1 → sets **db** = 0)
 R2 enter
 W1 block (*blocks* since **db** = 0)
 R3 enter
 R4 enter
 R2 exit
 R1 exit
 R3 exit
 R4 exit (**rc** = 0 → signals **db**)
 W1 enter (wakes up from *waiting* on **db**)

Readers-Writers: Monitor Solution (*)

```

monitor ReadersWriters
condition ReadingAllowed, WritingAllowed;
boolean SomeoneIsWriting;
integer readers;

procedure BeginReading
begin
  if SomeoneIsWriting or queue(WritingAllowed)
  then wait(ReadingAllowed);
  readers := readers + 1;
  signal(ReadingAllowed)
end;

procedure FinishedReading
begin
  readers := readers - 1;
  if readers = 0 then signal(WritingAllowed)
end;

procedure BeginWriting
begin
  if readers > 0 or SomeoneIsWriting
  then wait(WritingAllowed);
  SomeoneIsWriting := TRUE
end

procedure FinishedWriting
begin
  SomeoneIsWriting := FALSE;
  if queue(ReadingAllowed)
  then signal(ReadingAllowed)
  else signal(WritingAllowed)
end;

readers := 0;
SomeoneIsWriting := FALSE
end monitor;

(* procedures using monitor *)
procedure writer;
begin
  while true do
  begin
    ReadersWriters.BeginWriting;
    Write_DataBase();
    ReadersWriters.FinishedWriting
  end
end;

procedure reader;
begin
  while true do
    ReadersWriters.BeginReading;
    Read_DataBase();
    ReadersWriters.FinishedReading
  end
end;

```

Writers have priority

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.27

queue(cond) returns TRUE if there is a process in the *cond* blocked queue otherwise it returns FALSE

Writers have priority over readers as follows:

As long as there are no writers as many readers as want to are allowed to access database

When a writer arrives all subsequent readers are blocked since *queue(WritingAllowed)* now returns TRUE and subsequent readers block on a *wait(ReadingAllowed)*. The writer blocks on *wait(WritingAllowed)*

Remaining readers leave and the last reader does a *signal(WritingAllowed)* to allow writer to proceed

When writer finishes it signals other writers or readers waiting, with priority to readers which are queued

The first reader on the ReadingAllowed queue that is allowed to enter does a *signal(ReadingAllowed)* to allow the next reader in → cascade effect and all readers on the ReadingAllowed queue are allowed in.

Example Scenario:

R1,R2 enter

W1 block (**readers** > 0 → **wait(WritingAllowed)**)

R3 block (**queue(WritingAllowed)** = TRUE)

R4 block

R2 exit

R1 exit (**readers** = 0 → **signal(WritingAllowed)**)

W1 enter (wakes up from *waiting* on **WritingAllowed**)

W1 exits (**queue(ReadingAllowed)** = TRUE → **signal(ReadingAllowed)**)

R3 enter (wakes up from *waiting* on **ReadingAllowed** → **signal(ReadingAllowed)**)

R4 enter (wakes up from *waiting* on **ReadingAllowed**)

Dining Philosophers: Semaphore Solution (*)

```

void take_forks(int i)      /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);          /* enter critical region */
    state[i] = HUNGRY;     /* record fact that philosopher is hungry */
    test(i);               /* try to acquire 2 forks */
    up(&mutex);             /* exit critical region */
    down(&s[i]);            /* block if forks were not acquired */
}

void put_forks(int i)      /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);          /* enter critical region */
    state[i] = THINKING;   /* philosopher has finished eating */
    test(LEFT);            /* see if left neighbor can now eat */
    test(RIGHT);           /* see if right neighbor can now eat */
    up(&mutex);            /* exit critical region */
}

void test(int i)           /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING; /* Yes! the forks are free for philosopher i */
        up(&s[i]);          /* signal philosopher i that he can now eat */
    }
}

```

- Figure 2.18(a), osdi2: Dining Philosophers using semaphores

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.31

How does it work?

When philosopher wants to eat it tests whether the 2 forks are available, if so it signals itself to continue and eat, else it blocks

When philosopher finishes eating he/she checks whether the LEFT and RIGHT neighbour can eat if they were waiting for the forks to become free

Deadlock Avoidance (*)

- Banker's Algorithm (Dijkstra in 1965)
 - Define the state of a system
 - List of processes
 - number and type of resources used
 - maximum number of resources of each type that can be held
 - Number of resources of each type that are still available
 - Safe, Unsafe and Deadlock states
 - *Safe*: OS can allocate remaining available resources in such a way that all possible requests will be satisfied and no process will block
 - *Unsafe*: A particular sequence of process requests exist which can lead to a potential deadlock no matter what the OS does
 - *Deadlock*: No more resources are available and all processes have been blocked
 - Allocate resources so that the state of the system is never unsafe
 - Algorithm has to examine all subsequent requests in order to see if the system can ever get into a deadlock state → if so request will make system unsafe
 - If process attempts to make an unsafe request it is blocked (even though the resource it wants is free)

3/02/00

COS214, Roberto Togneri, E&E Eng, Univ. of Western Australia

6.38

Why Banker's Algorithm?

Processes → customers making loans which are paid back when finished

Resources held → current outstanding customer loan

Maximum resources expected → maximum loan that can be requested

Available resources → money in the bank

Banker relies on there being enough money to satisfy the maximum loan from any one customer. If the loan request from a customer cannot guarantee this then customer is told to “come back later” since an unsafe state may arise.