

Chapter 6

THREAD SYNCHRONIZATION

Table of contents:

6.1 INTRODUCTION	6-1
6.1.1 Critical Section	6-1
6.1.2 Mutual Exclusion	6-3
6.1.3 Producer-Consumer Relation	6-3
6.1.4 Thread Synchronization	6-3
6.1.5 Bounded Buffer	6-4
6.2 MUTUAL EXCLUSION	6-5
6.2.1 Enter/Leave primitives	6-5
6.2.2 General requirements	6-6
6.2.3 Implementation	6-6
6.2.4 Busy Waiting	6-7
6.2.5 Hardware Supported Busy Waiting	6-8
6.2.6 Problems With Busy Waiting	6-10
6.2.7 Thread Blocking Approach in Uniprocessor System	6-11
6.2.8 Thread Blocking Approach in Multiprocessor System	6-17
6.2.9 Synchronization Objects	6-18
6.2.10 Deadlock	6-19
6.3 SEMAPHORES	6-20
6.3.1 Semantics of Semaphore Primitives	6-20
6.3.2 Implementation of Semaphores	6-21
6.3.3 Usage of Semaphores in Mutual Exclusion	6-22
6.3.4 Binary Semaphores	6-22
6.3.5 Simple Synchronization	6-22
6.3.6 Bounded Buffer	6-23
6.4 EVENTS	6-24
6.4.1 Semantics of Event Primitives	6-24
6.4.2 Implementation of Events	6-25
6.4.3 Generalization of Events	6-27
6.5 CONDITIONS	6-29
6.5.1 Semantics of Condition Primitives	6-29
6.5.2 Implementation of Conditions	6-29
6.6 MESSAGES	6-30
6.5.1 Implementation of Message Objects	6-31

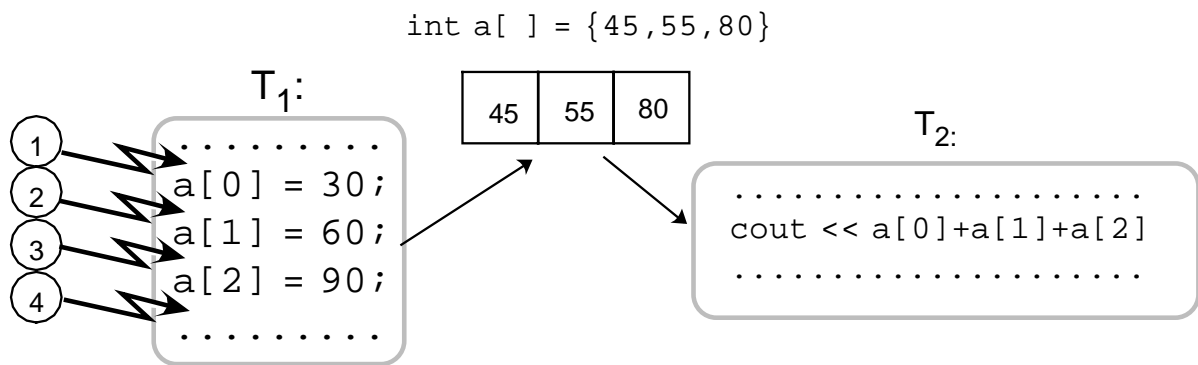
INTRODUCTION

This chapter is about synchronizatin and communication of threads which belong to the same process, i.e. they share the same address space and other resources as files and I/O devices. Next chapter will extend this to threads that belong to different processes.

Why threads need to be synchronized? Why is the thread communication an issue if they can easily access common global data?

Critical Section

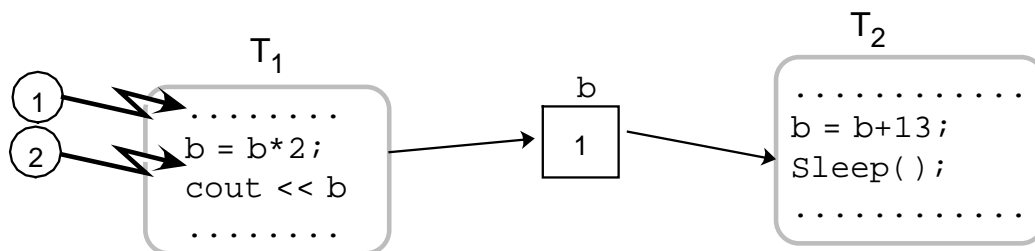
Suppose two threads are running: T_1 generates an array of three numbers (angles of a triangle), while T_2 checks if the sum of these angles is 180 degrees. Suppose T_1 is preempted by T_2 . What would be displayed if the preemption occurred in four indicated time instances?



Answer:

- (1) $45+55+80 = 180$ (correct, but old data)
- (2) $30+55+80 = 165$ (can't be a triangle)
- (3) $30+60+80 = 170$ (closer, but...)
- (4) $30+60+90 = 180$ (got it!)

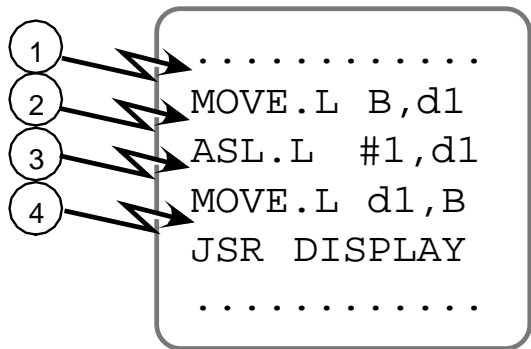
How about a scalar variable?



- (1) $2*(1+13) = 28$
- (2) $2*1+13 = 15$

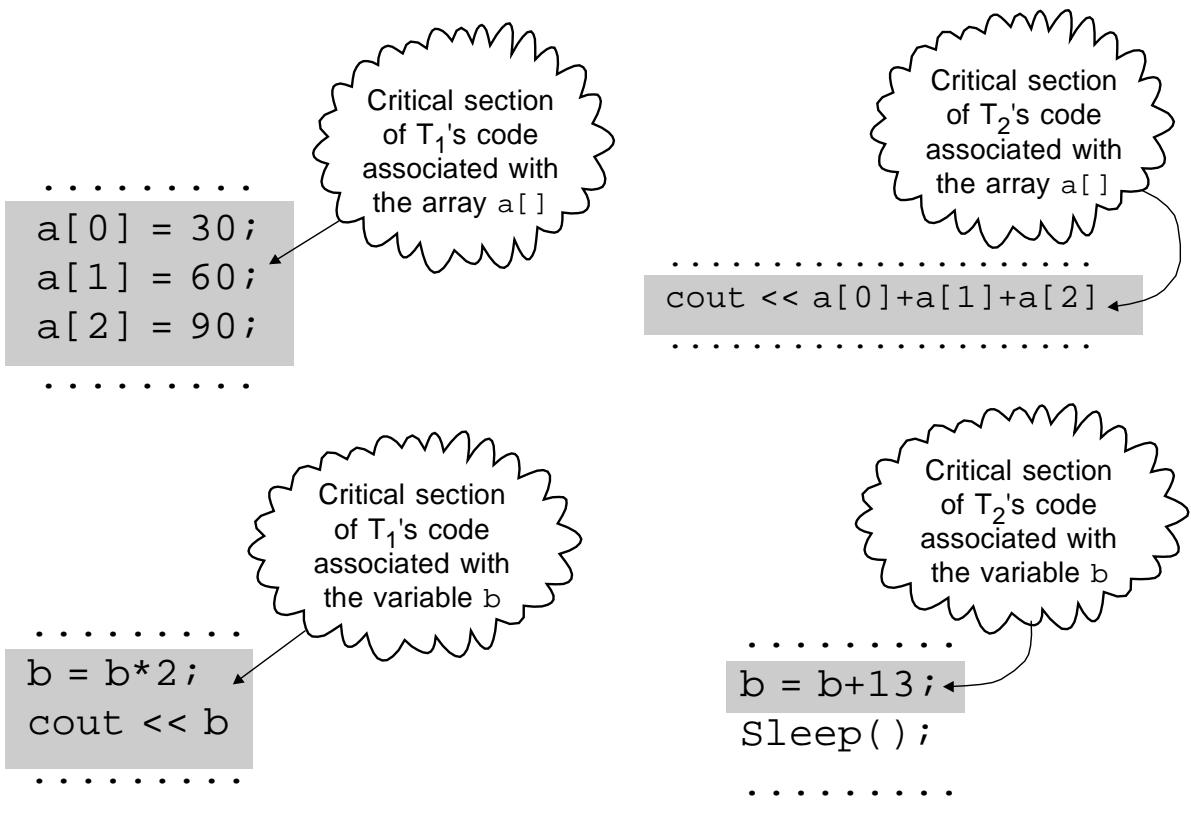
INTRODUCTION (Cont.)

But here is how the program really looks like:



- (1) $2*(1+13) = 28$
- (2) $2*1 = 2$
- (3) $2*1 = 2$
- (4) $2*1+13 = 15$

Apparently there are some sections of thread's code associated with shared data, which must not be overlapped in execution. These section of code are called critical sections, or critical regions.



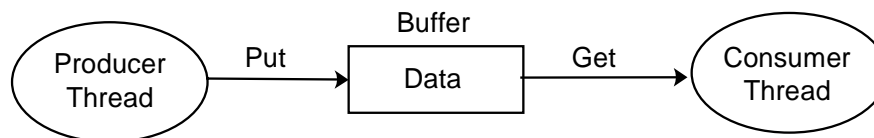
INTRODUCTION (Cont.)

Mutual Exclusion

Critical sections (CS) can be associated with any shared resource: a data structure in memory, a file, an I/O device. When two or several threads are accessing the same resource, their respective CS (which are associated with the resource) must be mutually exclusive in execution. This is called mutual exclusion (ME) problem. This problem is specially important in preemptive multithreaded environments where it is difficult to secure mutual exclusion unless...

Producer-Consumer Relation

In example above, thread T_1 generates (produces) angles of a triangle, while thread T_2 uses (consumes) them. This is very common relationship between threads, which is called producer-consumer relationship: thread T_1 is the producer thread and thread T_2 is the consumer thread.



Thread Synchronization

The mutual exclusion is not the only problem in thread communication.

Suppose that the threads in triangle example work iteratively: T_1 produces angles (in groups of three), while T_2 sums the angles, group by group. Suppose that we have solved somehow the mutual exclusion problem and suppose that both threads are working with unpredictable relative speeds. What happens if T_2 works faster, checks one group of angles, then wants to check the next group, before T_1 has produced it? Also, what happens if T_1 works faster than T_2 , produces new group of angles before T_2 has checked the previous group?

Generalization: producer thread produces data and puts them into a buffer, while the consumer thread gets the data from the buffer and consumes them. Both threads work iteratively with unpredictable relative speeds. When consumer takes data from the buffer, the buffer becomes empty; when producer puts data into the buffer, the buffer becomes full. The two threads have to be synchronized, so that the producer can put data into the buffer only if it is not full otherwise it has to wait until the consumer empties the buffer. Similarly, the consumer can take data only if the buffer is not empty, otherwise it has to wait until the producer puts new data into the buffer. The mechanism in which the threads are forced to wait and are prompted to continue has to be automatic and transparent to the application programmer.

INTRODUCTION (Cont.)

Bounded Buffer

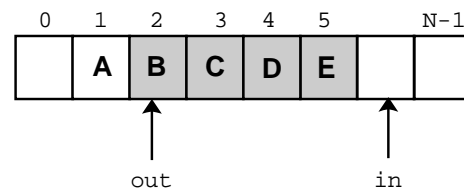
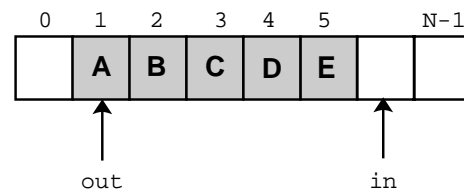
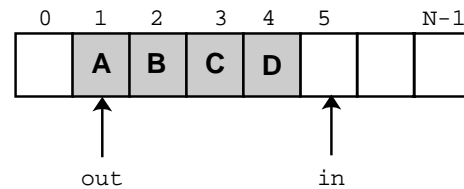
A general form of a buffer is the multi-slot buffer or bounded buffer. This is a fixed-size array of structures (slots, items) which is accessed in a cyclic manner (therefore it is sometimes called cyclic buffer.)

```
typedef struct {.....} Slot;
```

```
Slot B[N];
int in=0, out=0;
```

```
void put(Slot x)
{
    B[in] = x;
    in = (1+in)%N;
}
```

```
void get(Slot *x)
{
    *x = B[out];
    out = (1+out)%N;
}
```



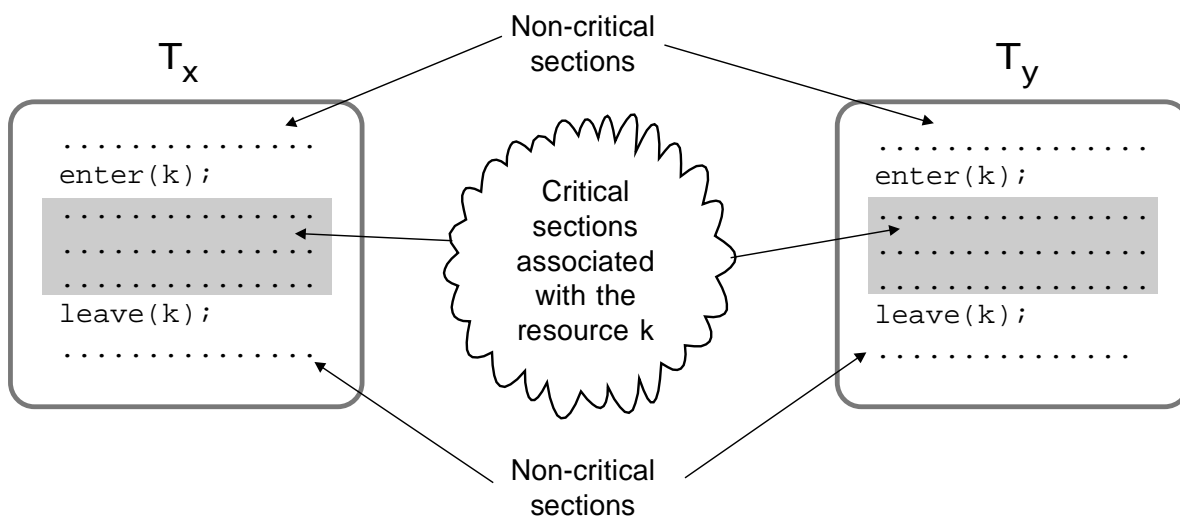
NOTICE:

The program which calls `put()` and `get()` is responsible not to overflow or underflow the buffer.

MUTUAL EXCLUSION

Enter/Leave Primitives

From the user point of view the ME of a CS associated with a resource k will be enforced through a pair of two system primitives `enter(k)` and `leave(k)` which surround the respective CS of each thread.



The semantics of the primitives:

`enter(k)` - The calling thread will continue execution passed this call if none of other threads is in a CS associated with the resource k . If another thread is executing in a respective CS, the calling thread will be forced to wait until that thread leaves its CS.

`leave(k)` - The calling thread is leaving its CS associated with the resource k . If there is another thread waiting to enter its respective CS, it will be permitted to enter immediately. If there are several threads waiting to enter their respective CS, only one thread will be permitted to enter,

MUTUAL EXCLUSION (Cont.)

General Requirements

In order to make the mechanism which supports ME safe and useful, the following requirements have to be satisfied:

- (1) ME must be enforced: only one thread at a time is allowed into its CS, among all threads that have CS for the same resource or shared object.
- (2) A thread that halts in its noncritical section must do so without interfering with other threads.
- (3) It must not be possible for a thread requiring access to a CS to be delayed indefinitely: no deadlocks or starvation.
- (4) When no thread is in a CS, any thread that requests entry to its CS must be permitted to enter without delay.
- (5) No assumptions are made about relative speeds or number of threads.
- (6) A thread remains inside its CS for a finite time only.

[Stallings, OS, 1998]

Implementation

There are two basic approaches in implementing the ME:

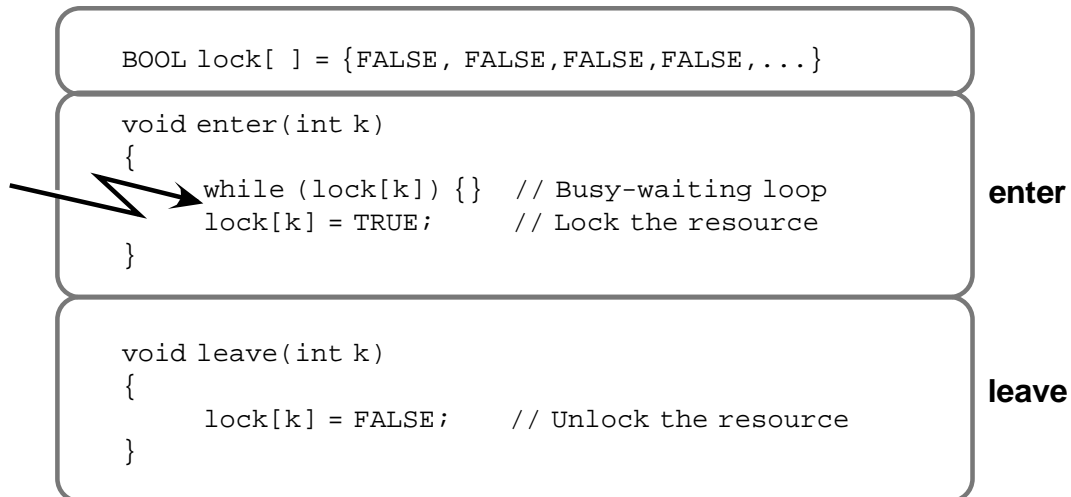
- Busy-Waiting Approach
- Thread Blocking Approach

NOTICE:

There is a possibility of disabling interrupts (i.e. disabling the thread preemption) at the entrance of the CS and enabling them back on the leave of CS. However, this possibility is rejected because the time spent in CS can be unpredictably long and interrupts must not be disabled for long time. In addition, the user must not have a power of disabling/enabling interrupts. This would be disastrous for multiprogrammed systems. Finally, disabling interrupts wouldn't work in multiprocessor systems. Disabling/enabling interrupts works only for the processor that is running the program which does disabling/enabling of interrupts, but not for other processors in the system.

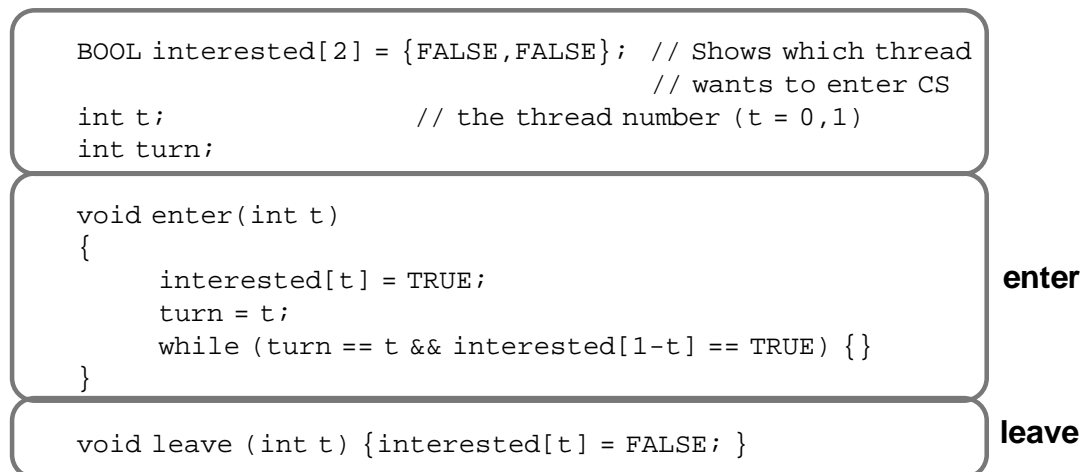
MUTUAL EXCLUSION (Cont.)**Busy Waiting**

A flag `lock[k]` is associated with the resource `k`:



Unfortunately this would not work: the timer interrupt can happen after the calling thread gets off the loop but before the lock is set. Another thread can then call `enter(k)`, find that the resource is free and get through. Result: two threads are inside their CS.

There are pure software solutions which work for two threads. First solution was proposed by Dutch mathematician T. Dekker (before 1965), later a simpler solution was proposed by G. L. Peterson (in 1981). Here is Peterson's solution (suppose two threads and a single resource, `k` omitted):

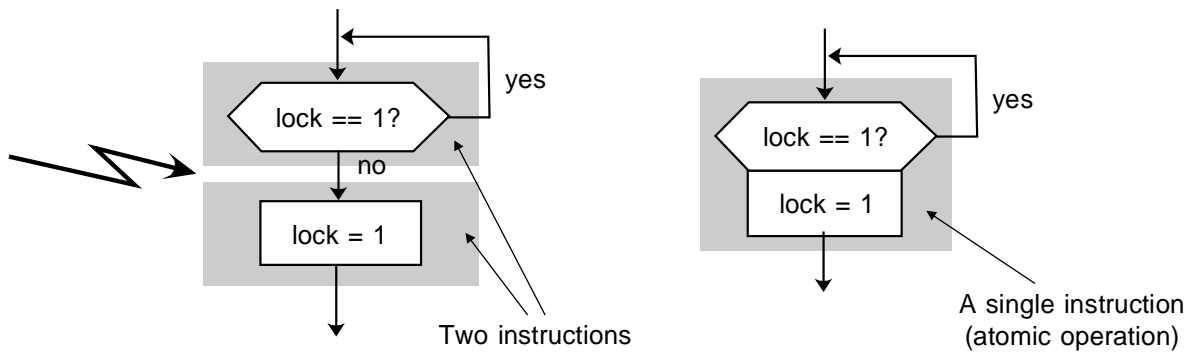


MUTUAL EXCLUSION (Cont.)

Hardware Supported Busy Waiting

Software solutions get more complicated and can cause longer waits if more than two threads are involved. Therefore the solutions which are used today are based on direct hardware support. There are two machine instructions: test-and-set, and test-and-set with memory lock which test and set the local variable in one atomic operation, that is in one indivisible, or uninterruptable operation.

Test-And-Set



Example:

```
BSET #n, LOCK      (Test bit and set, an MC680xx instruction)
```

Semantics of the instruction:

```
Atomic operation {
    if (LOCK[n] == 1) { // If bit LOCK[n] is set, then
        SR.Z = 0;      // Clear Z-bit in status register
    }
    else { // Otherwise
        SR.Z = 1;      // Set the Z-bit
        LOCK[n] = 1;  // and set LOCK[n]
    }
}
```

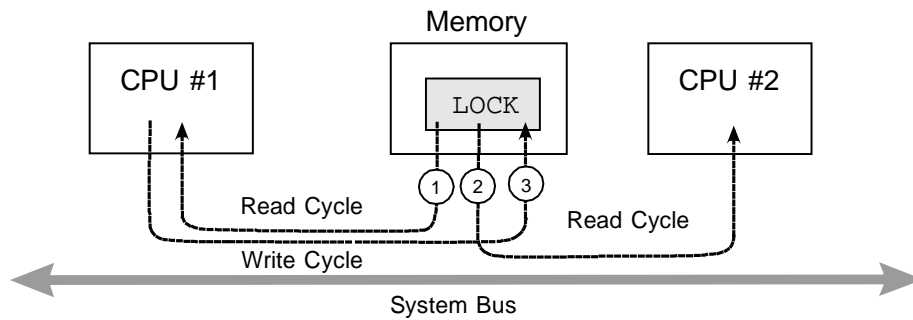
Usage of the instruction:

```
ENTER: BSET #7,LOCK // Test and set bit 7 in LOCK
      BNE ENTER    // Loop if Z-bit clear
      .....
      ..... CS ..... // Critical section
      .....
LEAVE: BCLR #7,LOCK // Clear bit 7 in LOCK
```

MUTUAL EXCLUSION (Cont.)

Test-And-Set With Memory Bus Lock

Test-and-set instruction works well in a uniprocessor machine, but wouldn't work in a multiprocessor machine. Reason: the instruction requires two bus cycles: read LOCK from the memory, test the bit, then write new LOCK into memory. If there is another CPU which can arbitrate the bus, it can read LOCK before the first CPU has written back the variable. Consequently, two threads running on different CPUs will be inside their respective CS.



There is another machine instruction specifically designed for multiprocessor systems:

TAS LOCK (Test bit and set, use read-write-modify memory cycle, an MC680xx instruction)

Semantics of the instruction:

```

Atomic operation {
    if (LOCK[7] == 1) { // If the MSB of LOCK is set, then
        SR.Z = 1; // Set Z-bit in status register
        SR.N = 1; // Set N-bit in status register
    }
    else { // Otherwise
        SR.Z = 0; // Clear the Z-bit
        SR.N = 0; // Clear the N-bit
        LOCK[7] = 1; // and set the MSB of LOCK
    }
}
A single Read-Modify-Write bus cycle
    
```

Usage of the instruction:

```

ENTER: TAS LOCK // Test and set bit 7 in LOCK
      BMI ENTER // Loop on minus (N-bit set)
      .....
      ..... CS .....
      .....
LEAVE: BCLR #7, LOCK // Clear bit 7 in LOCK
    
```

MUTUAL EXCLUSION (Cont.)**COMMENTS:**

Alternative semantics for TAS instruction:

```
TAS: tmp = LOCK[7];
      SR.N = tmp;
      SR.Z = tmp;
      LOCK[7] = 1;
```

Similar instruction is available on any modern architecture which supports multiprocessing. In the discussions that follow the test-and-set instruction with memory lock will be generally referred to as follows:

```
BOOL tsl(BOOL lock); // Test and set with memory lock:
                      // set lock to TRUE and return its original
                      // value
```

Usage of `tsl()` in a busy waiting loop:

```
while (tsl(lock)) {} // Loop until lock becomes FALSE
                      // then set lock TRUE
```

This form of waiting is also called spinlock.

Problems With Busy Waiting

Bad utilization of CPU. Waiting in a busy loop is wasteful for the CPU. The CPU could do some useful work, for example run another thread. This is specially acute in case of long CS.

Priority Inversion Problem. Suppose there are two threads: \mathcal{T}_1 waiting in a busy loop to enter its CS, and \mathcal{T}_2 which has gained access to the CS associated with the same resource. Suppose that \mathcal{T}_1 has high priority, while \mathcal{T}_2 has low priority and \mathcal{T}_1 is currently running. When the time quantum of \mathcal{T}_1 expires it will be preempted, however because it has priority higher than \mathcal{T}_2 , it will be dispatched immediately with a full time quota. Consequently, \mathcal{T}_2 will never have a chance to run, finish its work in CS and leave it, and \mathcal{T}_1 will busy wait forever. This is an example when higher priority doesn't help a thread to progress.

Long busy waits are not recommended for multiprogrammed systems. Therefore another approach has to be taken, in which the OS organizes the thread waiting.

MUTUAL EXCLUSION (Cont.)

Thread Blocking Approach in Uniprocessor System

Instead of waiting in a busy loop, the waiting thread is blocked and another thread is dispatched. For that purpose a new queue of blocked threads waiting to enter the critical section has to be added to the scheduler. Creation and initialization of the new queue and variable `lock` for a given resource will be done by an auxiliary primitive `init_CS()`.

Since we are dealing now with the scheduler and since the variable `lock` must be hidden from the user, part of the primitives `enter()`, `leave()` and `init_CS()` must execute in system mode. Therefore they will wrap system calls (using `TRAP`, see section 3.3).

Conceptually these primitives are implemented as follows:

API Library Function
(linked statically or dynamically
with the user code)

```
typedef void *HANDLE;      // Generic handle

HANDLE init_CS(void)      // API library function
{
    TRAP(SYSCALL, CS, h); // Trap calling sequence,
    return h;             // Returns handle
}
```

```
typedef struct {          // Internal type used in trap handler
    BOOL lock;           // FALSE - CS is free, TRUE - CS is occupied
    QUEUE q;            // Queue of threads waiting to enter CS
} CS;

INIT_CS:                 // Trap handler for create_CS()
    Disable interrupts;
    CS *p = new(CS);     // Allocate space for a new CS
    p->lock = FALSE;    // Initialize the lock variable
    Initialize queue p->q to empty;
    Return h = (HANDLE)p via register or stack;
    RTE;                // Return from trap
```

TRAP handler for `init_CS()`
(Executes in system mode)

The generic handle `h` is passed to the user in order to hide the implementation details like the structure `CS` and its instances.

MUTUAL EXCLUSION (Cont.)

API Library Functions
(linked statically or dynamically
with the user code)

```
void enter(HANDLE h){           // API library function
    TRAP(SYSCALL, ENTER, h);   // Trap calling sequence
}
```

```
void leave(HANDLE h){          // API library function
    TRAP(SYSCALL, LEAVE, h);   // Trap calling sequence
}
```

```
ENTER:
    Disable interrupts;
    Get p = (CS *)h via register or stack;
    if (p->lock){               // If CS occupied
        running2blocked(p->q);  // Block the running thread
        ready2running();       // Dispatch the highest pty thread
    }
    else{                       // If CS free
        p->lock = TRUE;         // Set lock and
    }
    RTE;                        // Return to API
```

```
LEAVE:
    Disable interrupts;
    Get p = (CS *)h via register or stack;
    if (p->q not empty){        // If there are waiting threads
        running2ready();       // Preempt running thread
        blocked2ready(p->q);   // Unblock the highest pty thread
        ready2running();       // Dispatch the highest pty thread
    }
    else {
        p->lock = FALSE;       // Reset lock
    }
    RTE;                       // Return to API
```

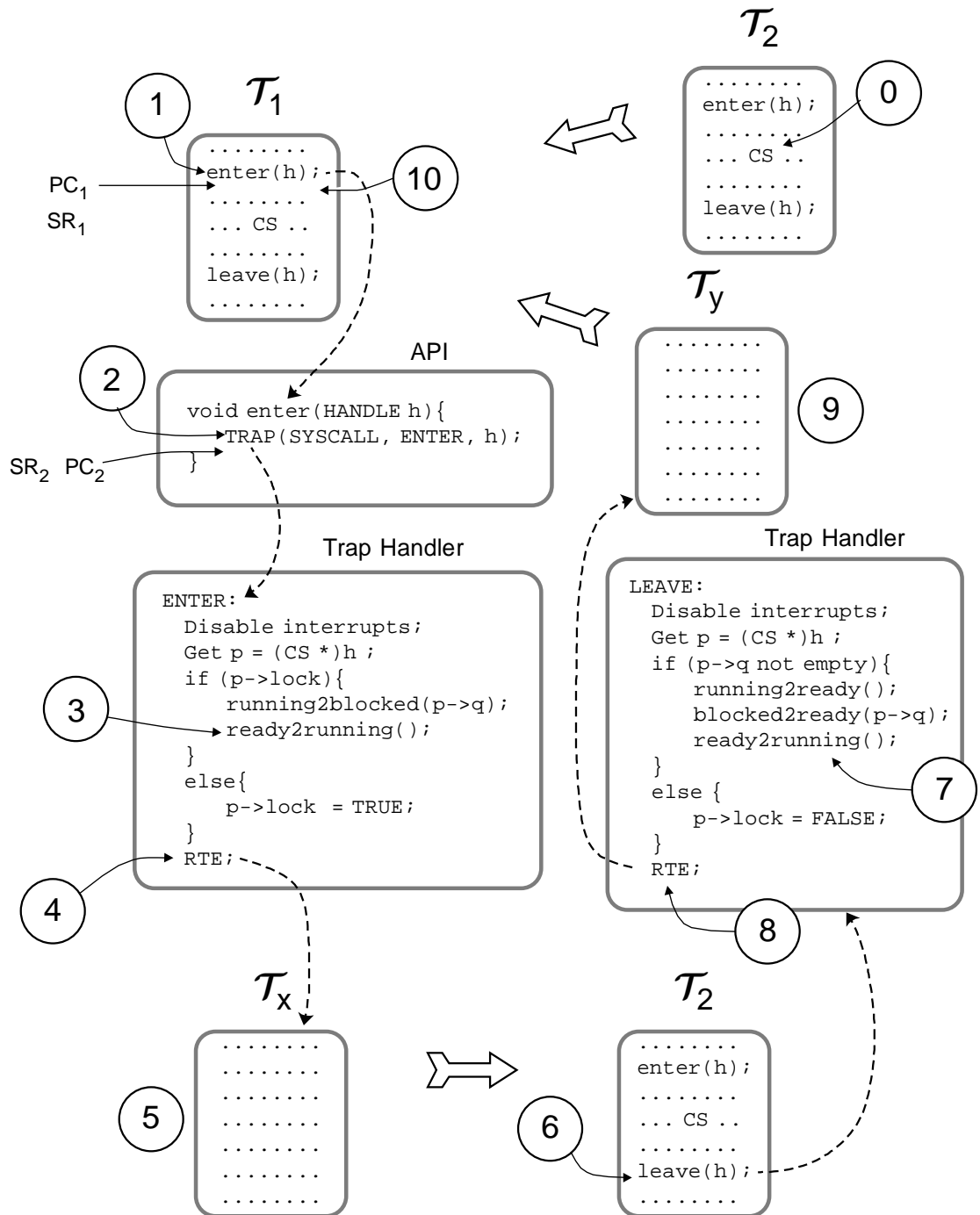
TRAP handlers for enter() and leave()
(Execute in system mode)

MUTUAL EXCLUSION (Cont.)**COMMENTS:**

- (1) System calls are never done directly from an application program. Instead they are done from within a function which hides details of the system call, like parameter checking and TRAP calling sequence, including parameter handling and stack clean-up. All such functions are organized into a library which is statically, or dynamically linked with the application program. Such library is usually called Application programming Interface (API).
- (2) Primitives `enter()` and `leave()` have different names in different OS. The names used here are the most common names used in literature. In Window NT these primitives are called `EnterCriticalSection()` and `LeaveCriticalSection()` when used within a process boundary, and `WaitForSingleObject()` and `ReleaseMutex()` when used across the process boundary. In most practical systems the synchronization primitives return a status variable (which was omitted here for simplicity).
- (3) The internal variables associated with the implementation of the synchronization primitives `enter()` and `leave()` (`lock` and thread queue `q`) must be strictly hidden from the user. Therefore they are defined in system space and are accessible only in the system mode (by the OS kernel).
- (4) The handle `h` returned from the initialization function `init_CS()` is used by the primitives `enter()` and `leave()` to distinguish between different CS, i.e. different resources. This variable is passed to the kernel which can use it. Attempt to use `h` as a pointer from the user program would result into an access violation error.
- (5) Application programmer is responsible to create a CS for each resource and to surround each CS with the corresponding pair of `enter()` and `leave()` primitives. In addition, application programmer must make sure that a thread stays in a CS a limited amount of time.
- (6) The pseudo-codes on previous page explicitly show disabling of interrupts, however enabling of interrupts is not shown explicitly. This is done automatically by the `RET` instruction. This instruction pops the user's status register of the kernel stack, which always has unmasked interrupts. In other words, return to the user's program via `RET` instruction always enables interrupts.
- (7) The dispatcher macro `blocked2ready()` assumes deblocking of exactly one thread from the waiting queue. In prioritized scheduling systems the thread with the highest priority is selected. Other threads waiting in the queue, for the same resource to be freed, have to wait for the next occasion, when some other thread leaves the respective CS and calls `leave()`.

MUTUAL EXCLUSION (Cont.)

How a thread \mathcal{T}_1 gets into a CS which is previously occupied by thread \mathcal{T}_2 .



MUTUAL EXCLUSION (Cont.)

- (0) \mathcal{T}_2 is executing in CS, then gets preempted
- (1) \mathcal{T}_1 is dispatched. It wants to enter CS and calls `enter()`
- (2) API `enter()` calls trap handler `ENTER`
- (3) Trap handler finds CS occupied (`lock = 1`) and calls dispatcher which blocks \mathcal{T}_1 and puts it into waiting queue. The program counter PC_2 and status register SR_2 will be saved into \mathcal{T}_1 's PCB. Dispatcher selects a new thread (\mathcal{T}_x) from the ready queue. Handler immediately proceeds to instruction `RTE`.
- (4) Handler executes instruction `RTE`. Consequently thread \mathcal{T}_x starts to run.
- (5) Thread \mathcal{T}_x runs for a while, then gets either blocked for some reason, or preempted. Suppose the next thread is \mathcal{T}_2 .
- (6) \mathcal{T}_2 is continuing to execute code within its CS and eventually calls `leave()`.
- (7) API `leave()` calls trap handler `LEAVE`. Trap handler finds that thread \mathcal{T}_1 is waiting in the queue and puts the thread into ready queue. The scheduler then selects a ready thread and makes it running. This could be \mathcal{T}_1 , but suppose some other thread (\mathcal{T}_y) with higher priority gets running.
- (8) Trap handler executes `RTE`, which starts execution of \mathcal{T}_y .
- (9) Thread \mathcal{T}_y runs for a while until it gets either blocked for some reason, or preempted. Thread \mathcal{T}_1 , which is waiting in ready queue gets eventually dispatched.
- (10) Execution of the last `RTE` causes that the program counter PC_2 and status register SR_2 gets popped from the kernel stack, which means that \mathcal{T}_1 continues execution of the API `enter()`, immediately behind the trap call, and returns from there at PC_1 , which is inside the CS!

MUTUAL EXCLUSION (Cont.)

The sequence of dispatcher macros in traps ENTER and LEAVE will be used in other synchronization primitives. Therefore we introduce new macros:

```

block(h):    running2blocked(h);
             ready2running();

unlock(h):  running2ready();
            blocked2ready(h);
            ready2running();

```

Which simplifies the handlers on page 8-12:

```

ENTER:
  Disable interrupts;
  Get p = (CS *)h via register or stack;
  if (p->lock)           // If CS occupied
    block(p->q);
  else                   // If CS free
    p->lock = TRUE;      // Set lock and
RTE;                    // Return to API

LEAVE:
  Disable interrupts;
  Get p = (CS *)h via register or stack;
  if (p->q not empty)    // If there are waiting threads
    unlock(p->q);
  else
    p->lock = FALSE;     // Reset lock
RTE;                    // Return to API

```

MUTUAL EXCLUSION (Cont.)

Thread Blocking Approach in Multiprocessor System

As shown earlier, the scope of the interrupt inhibition is the processor on which the inhibiting thread is running. Therefore, the method described on previous pages will not work on a multiprocessor machine.

For that reason the variable `lock` and the waiting queue `q` must be considered as mutually exclusive resources. In other words, the bodies of trap handlers `ENTER` and `LEAVE` can also be considered as CSs associated with `lock` and `q`. Since the CSs must work on multiprocessor, we use spinlocks for their protection. In this case the usage of spinlocks is not critical since the CS which they have to protect are short (see below).

If we use separate spinlocks for each CS, then the CS's data structure has to be extended by a new locking variable `guard`:

```
typedef struct {           // Internal type used in trap handler
    BOOL lock;             // FALSE - CS is free, TRUE - CS is occupied
    QUEUE q;               // Queue of threads waiting to enter CS
    BOOL guard;         // Protects lock and q from simultaneous
} CS;                      // access by other processors
```

```
ENTER:Disable interrupts;
Get p = (CS *)h via register or stack;
while (tsl(p->guard)) {}; // Spinlock to protect lock, q
if (p->lock)                // If CS occupied
    block(p->q);
else                        // If CS free
    p->lock = TRUE;         // Set lock and
p->guard = FALSE;       // Release spinlock
RTE;                        // Return to API
```

Short
Critical
Sections

```
LEAVE:Disable interrupts;
Get p = (CS *)h via register or stack;
while (tsl(p->guard)) {}; // Spinlock to protect lock, q
if (p->q not empty){        // If there is a waiting
    unblock(p->q);          // thread - unblock it
else
    p->lock = FALSE;       // Reset lock
p->guard = FALSE;       // Release spinlock
RTE;                        // Return to API
```

MUTUAL EXCLUSION (Cont.)

Synchronization Objects

API functions

```
HANDLE init_CS(void);  
void enter(HANDLE h);  
void leave(HANDLE h);
```

are linked (statically or dynamically) with the user application. They are wrappers for the calling sequences to the appropriate trap handlers and they execute in user mode. Once the trap instruction is executed, the execution continues in system mode.

The structure

```
typedef struct {  
    BOOL lock;  
    QUEUE q;  
    BOOL guard;  
} CS;
```

and its instances are completely unreachable and transparent to the user, they are strictly part of the system space. The only connection between a particular instance of the structure and the user is the handle which is returned to the user by `init_CS()`. The handle is necessary if there are several different sharable resources which need to be garded separately and which have to be identified. The handle can be used only as a parameter of `enter()` and `leave()`, and can not be used in a user program as a pointer.

Encapsulation of an instance of the structure CS and the operations `init_CS()`, `enter()` and `leave()` will be called synchronization object of type mutual exclusion.

MUTUAL EXCLUSION (Cont.)

Deadlock

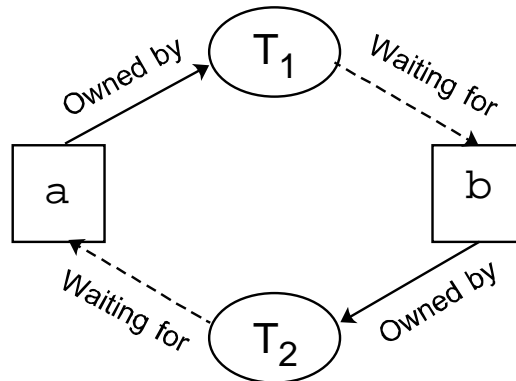
Suppose two resources and two synchronization objects with handles a and b, and two threads:

```

T1: .....
    enter(a);
    enter(b);
    .....

T2: .....
    enter(b);
    enter(a);
    .....
  
```

If T_1 calls `enter(a)` and T_2 calls `enter(b)` before T_1 calls `enter(b)` - both threads become blocked indefinitely. This situation is called deadlock and it is very likely to occur whenever there are circular calls of blocking primitives.



Another deadlock situation:

```

    enter(a);
    .....
    enter(a); // probably wanted to use leave(a)
  
```

The thread executing the code above will be blocked for ever.

SEMAPHORES

Primitives `enter()` and `leave()` are convenient to ensure the mutual exclusiveness of CS, however they are not so convenient for thread synchronization. Therefore a more powerful primitives called semaphores were proposed by E. Dijkstra in 1965.

["Cooperating Sequential processes" Technological University, Eindhoven, The Netherlands, 1965; reprinted in: "Great Papers in Computer Science, P. Laplante, ed., IEEE Press, New York, NY 1996]

The primitives are called $P()$ and $V()$, originally named by Dijkstra:

P stands for "try to decrease" (in Dutch: *prolagen* which is an acronym for *proberen* = try, and *verlagen* = decrease, decrement)
 V stands for "Increase" (in Dutch: *verhogen* = increase, increment)

Alternative names for $P()$ and $V()$, commonly used today, are `wait()` and `signal()` respectively.

Semantics of Semaphore Primitives

Semaphores are based on a nonnegative integer variable s , which is usually initialized to some positive value N .

$P()$ - If $s > 0$ then decrement s and exit $P()$, i.e. continue to execute the calling thread. If $s = 0$ then block the calling thread. The thread will remain blocked until s becomes positive.

$V()$ - Increment s by 1 and check if there is any thread blocked on this semaphore. If yes, unblock the waiting thread.

Operations $P()$ and $V()$ are atomic.

COMMENT:

There are various flavors of these definitions. Some allow negative values for semaphore variable.

SEMAPHORES (Cont.)**Implementation of Semaphores**

API library functions:

```

HANDLE init_semaphore(int n); // Create and initialize a new
                                // semaphore
void P(HANDLE h);           // P-operation on semaphore h
void V(HANDLE h);           // V-operation on semaphore h

```

Trap handlers:

```

typedef struct { // Semaphore type
    int s; // Semaphore variable
    QUEUE q; // Queue of threads waiting on semaphore
    BOOL guard; // Mutual exclusion variable for s and q
} Semaphore;

```

```

P:  Disable interrupts;
    Get p = (Semaphore *)h via register or stack;
    while (tsl(p->guard)) {}; // Spinlock to protect s and q
    if (p->s == 0) // If not s>0
        block(p->q);
    else // If s>0
        p->s -= 1; // Decrement s
    p->guard = FALSE; // Release spinlock
    RTE; // Return to API

```

```

V:  Disable interrupts;
    Get p = (Semaphore *)h via register or stack;
    while (tsl(p->guard)) {}; // Spinlock to protect s and q
    if (p->q not empty){ // If there is a waiting
        unblock(p->q); // thread - unblock it
    }
    else
        p->s += 1; // Increment s
    p->guard = FALSE; // Release the spinlock
    RTE; // Return to API

```

SEMAPHORES (Cont.)

Usage of Semaphores in Mutual Exclusion

Each CS must be assigned a semaphore initialized to 1:

```
HANDLE h1, ...;
h1 = init_semaphore(1); // Create a semaphore for critical
                        // section CS1, initialize semaphore
                        // variable to 1
```

```
P(h1); // Enter CS1
.....
CS1
.....
V(h1); // Leave CS1
```

} Each thread must surround
its CS with the semaphore
primitives

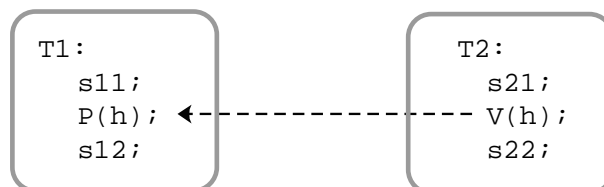
Binary Semaphores

Semaphore whose internal variable s takes only values 0 and 1 is called binary semaphore. The V-operation of a binary semaphore doesn't increment semaphore variable, it sets it to 1 instead. In other words, a repeated call of $V()$ would not permit equivalent repeated P-calls without blocking.

Simple Synchronization

Suppose thread T_1 executes instruction s_{11} , then must wait until thread T_2 executes instruction s_{21} . After that T_1 can proceed and execute instruction s_{12} . For this a binary semaphore can be used, which must be initialized to 0:

```
h = init_semaphore(0);
```



SEMAPHORES (Cont.)

Bounded Buffer

Bounded buffer can be used to exchange data between threads. However there are two problems with that:

- (1) Mutual exclusion of buffer operations: calling `put()` and `get()` must be mutually exclusive, i.e. `put()` and `get()` are critical sections.
- (2) Synchronization: `put()` must not be called if buffer is full, and `get()` must not be called if buffer is empty.

Both problems can be solved with three semaphores:

- mutex* - For mutual exclusion of buffer access (a binary semaphore initialized to 1)
- space* - To count the free space in buffer (an integer semaphore, initialized to N)
- slots* - To count the number of slots in buffer (an integer semaphore, initialized to 0)

```

Producer: while(1)
{
    produce slot x;
    P(space);    // Will wait if buffer becomes full
    P(mutex);   // Enter CS
    put(x);     // Put slot into buffer
    V(mutex);   // Leave CS
    V(slots);   // Tell Consumer there are slots in buffer
}

```

```

Consumer: while(1)
{
    P(slots);   // Wait if buffer is empty
    P(mutex);   // Enter CS
    get(&x);    // Get slot from buffer
    V(mutex);   // Leave CS
    V(space);   // Tell Producer buffer is not full
    use slot x;
}

```

NOTICE:

For mutual exclusion could be used `enter(mutex)` and `leave(mutex)` instead of `P(mutex)` and `V(mutex)`.

EVENTS

In simple synchronization example one thread is synchronized with another thread, i.e. thread \mathcal{T}_2 notifies thread \mathcal{T}_1 to continue execution. This can be done with binary semaphores. If however one thread wants to notify several threads to continue execution, the semaphores wouldn't be convenient. Reason: a V-operation unblocks only one thread. If we want to unblock all threads waiting for a single condition (event), we can use more convenient event objects, which are supported by most of the modern OS.

Events were first used in operating system RTX-11 written for PDP-11 family, and in operating system VMS for VAX systems. Very reach variations of events are available in Windows NT.

Semantics of Event Primitives

Events are based on an event flag e which is initially reset ($e = \text{FALSE}$). The names of event primitives are generic.

`wait_event()` - If $e = \text{FALSE}$, the calling thread is blocked, and remains blocked until some other thread calls `set_event()`.
If $e = \text{TRUE}$, the primitive has no effect.

`set_event()` - Set event flag $e = \text{TRUE}$ and unblock all threads which are waiting for that event.

`reset_event()` - Set event flag $e = \text{FALSE}$.

EVENTS (Cont.)

Implementation of Events

API library functions:

```
HANDLE init_event();           // Create and initialize an event
void wait_event(HANDLE h);     // Wait for event h to occur
void set_event(HANDLE h);      // Set event h (the event is occurring)
void reset_event(HANDLE h);    // Turn off the event (event is no
                                // longer in effect
```

Trap handlers:

```
typedef struct {               // Event type
    BOOL e;                    // Event flag
    QUEUE q;                   // Queue of threads waiting on event
    BOOL guard;                // Mutual ex. variable for e and q
} Event;
```

```
WAITEVENT:
Disable interrupts;
Get p = (Event *)h via register or stack;
while (tsl(p->guard)) {}; // Spinlock to protect e and q
if (p->e == FALSE)        // If event not set
    block(p->q);           // block the calling thread
p->guard = FALSE;        // Release spinlock
RTE;                     // Return to API
```

```
SETEVENT:
Disable interrupts;
Get p = (Event *)h via register or stack;
while (tsl(p->guard)) {}; // Spinlock to protect e and q
p->e = TRUE;              // Set the event flag
if (p->q not empty) {     // If there is a thread waiting
    running2ready();      // .. preempt the calling thread
    while(p->q not empty) // Unblock all waiting threads
        blocked2ready(p->q);
    ready2running();      // Dispatch a ready thread to run
}
p->guard = FALSE;        // Release the spinlock
RTE;                     // Return to API
```

EVENTS (Cont.)

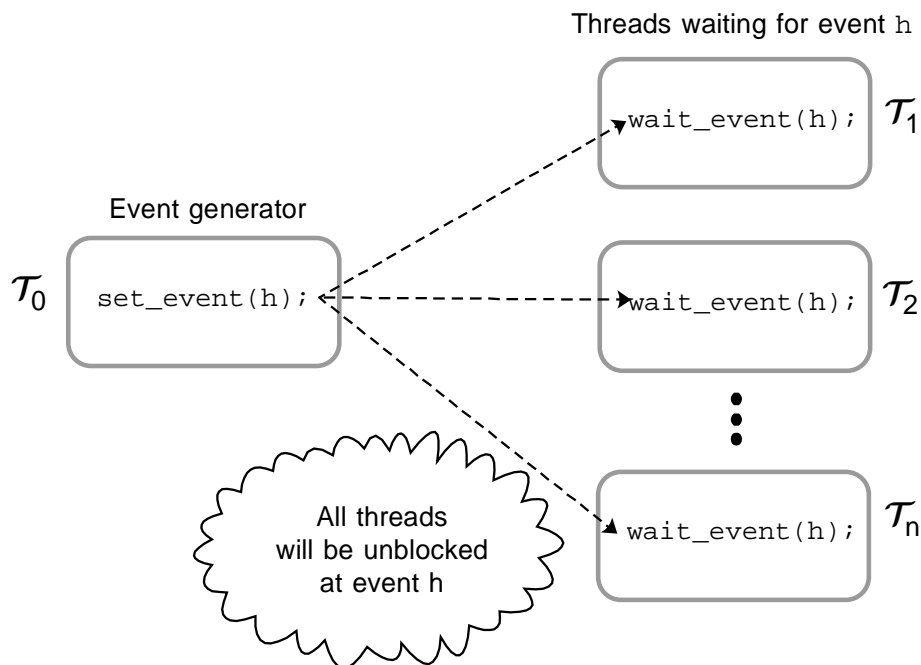
```

RESETEVENT:
    Disable interrupts;
    Get p = (Event *)h via register or stack;
    while (tsl(p->guard)) {}; // Spinlock to protect e
    p->e = FALSE;           // Reset the event flag
    p->guard = FALSE;       // Release the spinlock
    RTE;                   // Return to API
  
```

COMMENTS:

The implementation of the primitive `set_event()` is essentially different than the implementation of its counterparts `leave()` and `V()`: the latter primitives unblock only one waiting thread, no matter how many threads are waiting in the blocked queue. Other threads have to be unblocked by subsequent calls of `leave()` or `V()`. The primitive `set_event()` however unblocks all threads which are waiting for the event.

This behavior can be modified in some systems (Windows NT): it can be specified that the event resets automatically after the first waiting thread gets unblocked. Other threads waiting in the blocked queue for the same event have to be unblocked one by one by calling `set_event()` repeatedly.



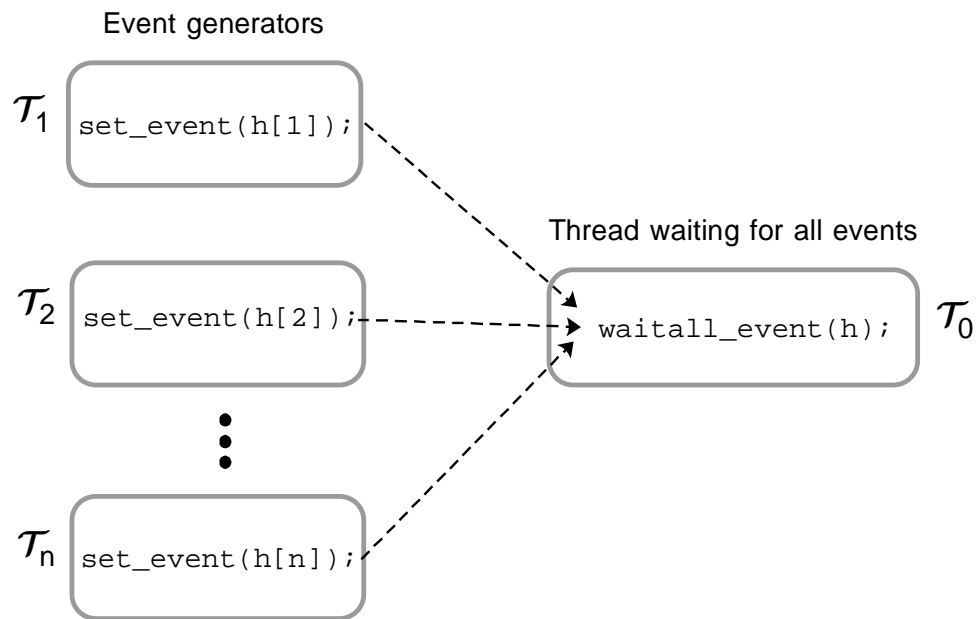
EVENTS (Cont.)

Generalization of Events

So far a thread (or threads) can wait for a single event. Sometimes it is useful to wait for conjunction (AND) or disjunction (OR) of several events h_1, h_2, \dots, h_n :

`waitall_event(h)` - wait for all events $h[1], h[2], \dots, h[n]$ to occur

`waitany_event(h)` - wait for any of event $h[1], h[2], \dots, h[n]$ to occur



NOTICE:

Events h_1, h_2, \dots, h_n don't have to be signalled from different threads, they can be signalled from the same thread or from a combination of threads.

Several threads can be waiting for several events.

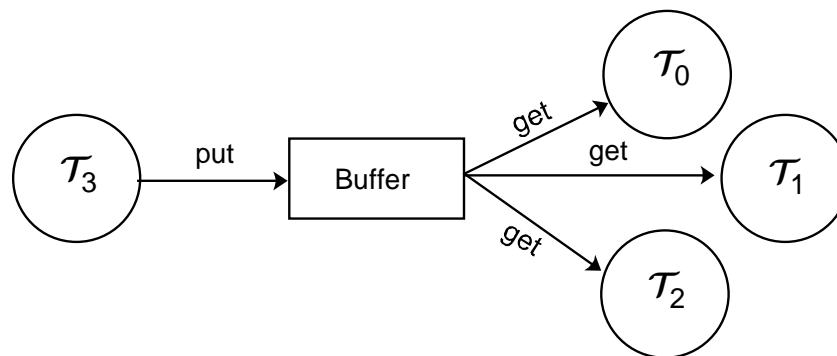
EVENTS (Cont.)

Example:

A producer thread \mathcal{T}_3 places data in a single-slot buffer and waits until all threads \mathcal{T}_0 , \mathcal{T}_1 and \mathcal{T}_2 take data from the buffer. This problem requires six events:

taken[i] - thread \mathcal{T}_i has taken data from the buffer ($i=0,1,2$)

full[i] - thread \mathcal{T}_3 has put data into the buffer and has notified \mathcal{T}_i ($i=0,1,2$)



```

T[3]: while(1){
    produce x;
    waitall_event(taken);
    put(x);
    for(i=0;i<3;i++){
        reset_event(taken[i]);
        set_event(full[i]);
    }
}
  
```

```

T[i]: while(1){
    wait_event(full[i]);
    get(&x);
    reset_event(full[i]);
    set_event(taken[i]);
    use x;
}
  
```

Before threads \mathcal{T}_i ($i=0,1,2,3$) are started the events have to be created and initialized:

```

HANDLE taken[3], full[3];

for(i=0;i<3;i++){
    taken[i] = init_event();
    full[i] = init_event();
}
  
```

CONDITIONS

Semaphores and events are higher-level synchronization tools, which are difficult to use in some situations without degrading them. Therefore a lower-level primitives would be desired, so the users can build their own customized higher-level synchronization primitives. These are `wait()` and `signal()` and the corresponding synchronization object is called condition.

Conditions were proposed by P. Brinch-Hansen and C. A. R. Hoare in early 70's.

Semantics of Condition Primitives

Synchronization objects have only a queue of threads blocked on condition, no other variables except the usual spinlock variable.

- | | |
|-----------------------|--|
| <code>wait()</code> | - Block the calling thread unconditionally until some thread executes <code>signal()</code> operation. |
| <code>signal()</code> | - If the blocked queue for that condition is not empty, unblock one waiting thread, otherwise the primitive has no effect. |

Implementation of Conditions

```
HANDLE init_condition();    // API functions
void wait(HANDLE h);
void signal(HANDLE h);
```

```
typedef struct {           // Condition type
    QUEUE q;               // Queue of thr. waiting on condition
    BOOL guard;           // Mutual exclusion variable for q
} Condition;

// Trap handlers
```

```
WAIT:  Disable interrupts;
Get p = (Condition *) h via register or stack;
while(tsl(p->guard){} // Spinlock
block(p->q);           // Block the caller unconditionally
p->guard = FALSE;     // release spinlock
RTE;
```

```
SIGNAL: Disable interrupts;
Get p = (Condition *) h via register or stack;
while(tsl(p->guard){} // Spinlock
if(p->q not empty)    // If there is a waiting thread
    unblock(p->q);    // unblock one
p->guard = FALSE;    // release spinlock
RTE;
```

MESSAGES

A common method for communication of threads which are in producer-consumer relationship is bounded buffer with mutual exclusion and synchronization. A semaphore solution shown earlier can be interpreted as follows:

```

// Initialize all necessary semaphores:

HANDLE mutex = init_semaphore(1);
HANDLE space = init_semaphore(N);
HANDLE slots = init_semaphore(0);

// Initialize buffer:

Slot *B;           // Create space for buffer
B = new Slot[N];
int in = 0, out = 0; // Initialize buffer pointers

// Sending data into buffer

P(space);
P(mutex);
B[in] = x;
in = (1+in)%N;
V(slots);
V(mutex);
} send

// Receiving data from the buffer

P(slots);
P(mutex);
*x = B[out];
out = (1+out)%N;
V(space);
V(mutex);
} receive

```

What is wrong with this solution?

1. Too many system calls (makes the program more complicated and less efficient)
2. Must be very careful with the usage of semaphores
 - correct initialization,
 - correct order
 - consistent usage in all participating threads
 - must not use P() and V() anywhere else in program on the same object
3. Buffer variables B[], int and out are visible and could be accessed accidentally outside the context of send and receive operations.

MESSAGES (Cont.)

A safer, more efficient and more convenient method for thread communication via bounded buffer is to implement a new higher level system object message buffer:

Implementation of Message Objects

API library functions:

```
HANDLE init_message_buffer(int N); // Create a new message object

void send(HANDLE h, Slot *x); // Send message

void receive(HANDLE h, Slot *x); // receive message
```

Trap handlers:

```
// Message type (hidden in kernel)
typedef struct {
    int N; // Maximal number of slots
    int in, out; // Buffer pointers
    int count; // Number of slots placed into buffer
    Slot *B; // Pointer to the buffer space
    BOOL lock; // CS lock to protect access to buffer
    QUEUE mutex; // Queue of thr. waiting to access the buff.
    QUEUE nonfull; // Queue of thr. waiting for non-full cond.
    QUEUE nonempty; // Queue of thr. waiting for non-empty cond.
    BOOL guard; // To protect lock, mutex, nonfull, nonempty
} Message_buffer;
```

```
SEND: // prefix "p->" omitted for simplicity
Disable interrupts;
Get p = (Message_buffer *)h via register or stack;
enter(mutex);
if (count == N) wait(nonfull, mutex);
B[in] = *x; in = (1+in)%N;
count++;
signal(nonempty, mutex);
RTE;
```

```
RECEIVE:
Disable interrupts;
Get p = (Message_buffer *)h via register or stack;
enter(mutex);
if (count == 0) wait(nonempty, mutex);
*x = B[out]; out = (1+out)%N;
count--;
signal(nonfull, mutex);
RTE;
```

enter(), wait() and signal() are internal kernel functions, they are not called via TRAP, however they return via RTE (see next page)

MESSAGES (Cont.)

```

enter(m):
    push(SR);                // Save status register
    while (tsl(guard)) {};   // To protect lock
    if (lock){
        block(m);
    }
    else{
        lock = TRUE;
    }
    guard = FALSE;          // Release spinlock
    RTE;

wait(c,m):
    push(SR);                // Save status register
    while (tsl(guard)) {};   // To protect lock
    running2blocked(c);
    if (queue m not empty)
        blocked2ready(m);
    else
        lock = FALSE;
    ready2running;
    guard = FALSE;          // Release spinlock
    RTE;

signal(c,m):
    push(SR);                // Save status register
    while (tsl(guard)) {};   // To protect lock
    if( queue c not empty)
        unblock(c);
    else {
        if(queue m not empty)
            unblock(m);
        else
            lock = FALSE;
    }
    guard = FALSE;          // Release spinlock
    RTE;

```

MESSAGES (Cont.)

FREQUENTLY ASKED QUESTION

- Q1: Why we need to ensure mutual exclusion of buffer variables N , out , in , $B[]$, $count$, $lock$...by using $enter()$? If we disable interrupts in traps SEND and RECEIVE, then no other thread can be scheduled while some thread is executing SEND or RECEIVE, therefore the mutual exclusion is already achieved.
- A1: That is true only for uniprocessor machines. In multiprocessor machines we must have $enter()$.
- Q2: Isn't it too dangerous to disable interrupts in SEND and RECEIVE? There is for example $wait()$ which blocks the thread if the buffer is full/empty. The thread can be blocked until some other thread removes/puts slots from/into the buffer.
- A2: If a thread is blocked it doesn't mean that the execution is stuck in kernel. Blocking means updating and relinking the thread/process control blocks plus modifying the kernel stack (see pages 3-11 to 3-15). The RTE will make sure that the execution continues somewhere else without waiting for the blocking condition to be changed, i.e. that someone removes/puts a slot from/to the buffer.
- Q3: Yes, but what if execution continues in kernel. That means, the interrupt is disabled during thread switching, which can be relatively long if the thread is outswapped?
- A3: True, this is a real danger if OS has virtual memory. Therefore Microsoft has introduced deferred procedure calls (DPC), (see chapter 12). Shortly, an interrupt service routine (ISR) has interrupts disabled only for a critical period of time, which is short enough and which doesn't cover the thread switching.
- Q4: Why we need to push processor status register (SR) onto kernel stack in kernel functions $enter()$, $wait()$ and $signal()$?
- A4: Because these functions exit with RTE, there must be a correct balance on the kernel stack.
- Q5: Why do we need RTE in $enter()$, $wait()$ and $signal()$ anyway? These are functions, not traps and they are called by JSR (jump to subroutine), consequently they can exit with RTS (return from subroutine). Both, JSR and RTS push/pop only PC.
- A5: RTE doesn't necessarily mean that the execution will continue at the first instruction after the function call. The execution can continue in some other thread. Essentially $enter()$, $wait()$ and $signal()$ do not follow FIFO principle, they behave more like coroutines. If the execution continues somewhere else, specially in user space, the user environment has to be restored, i.e. SR has to be restored.

MESSAGES (Cont.)

FREQUENTLY ASKED QUESTION (Cont.)

Q6: Why the variable *lock* is not reset in than part of the outer if statement in *signal()*?
If we are unblocking a thread we also need to make the critical section free for the unblocked thread.

A6: The lock variable must be left TRUE. Consider *signal(nonfull,mutex)* in RECEIVE trap.

(a) Suppose that the unblocked thread (which was waiting due to a call of *wait(nonfull,mutex)*), gets running (*unblock (nf)* macro). This thread will continue execution imediately after the *wait(nonfull,mutex)* function call in SEND trap. Apparently, this thread won't have chance to set the *lock* variable, so it needs to be left TRUE. Same holds for *signal(nonempty,mutex)* in SEND trap.

(b) Suppose now that the thread that called *signal(nonfull,mutex)* gets running (while the unblocked thread remains in ready queue). This thread will immediately leave the trap and return to the user space. Consequently noone is virtually in critical section and the lock variable is TRUE. That is however OK because the blocked thread will eventually get running, and we have the case (a).

Q7: Why the variable *lock* is not reset in else part of the outer if statement in *signal()*?
Like in Q6, we are leaving trap and critical section, therefore we need to reset the *lock* variable.

A7: Answer is similar to A6. The only difference is that the other thread is blocked on *enter(mutex)* instead of *wait(nonfull,mutex)*, and will continue execution after this call. The *lock* variable must therefore remain TRUE.

Q8: Why do we need RTE at the end of SEND and RECEIVE traps? WE already have RTE at the end of *signal()*.

A8: RTE in *signal()* and *wait()* exist for reason considered above. Eventually each call to SEND and RECEIVE trap will end up with PC loaded with address immediately after *signal()*. From there we need to return from the trap to the user space.