PART 2. PROCESS MANAGEMENT

Chapter 6: Synchronization

WHAT'S AHEAD:

- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems
- Monitors
- Synchronization Examples

WE AIM:

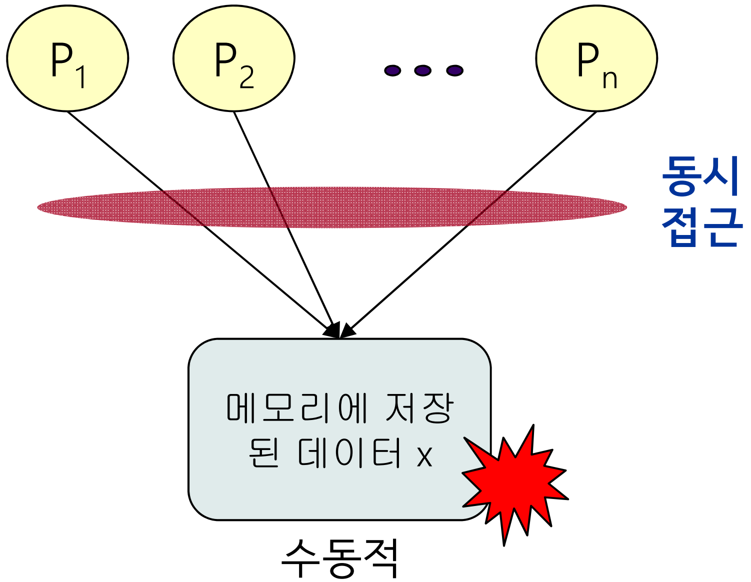
- To discuss the critical-section problem and hardware/software solutions
- To examine several classical process synchronization problems and solutions

Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심요점

왜 동기화하여야만 하는가?

복수의 프로세스가 데이터 X에 대한
읽기/쓰기(R/W) 연산을 하려고 한다



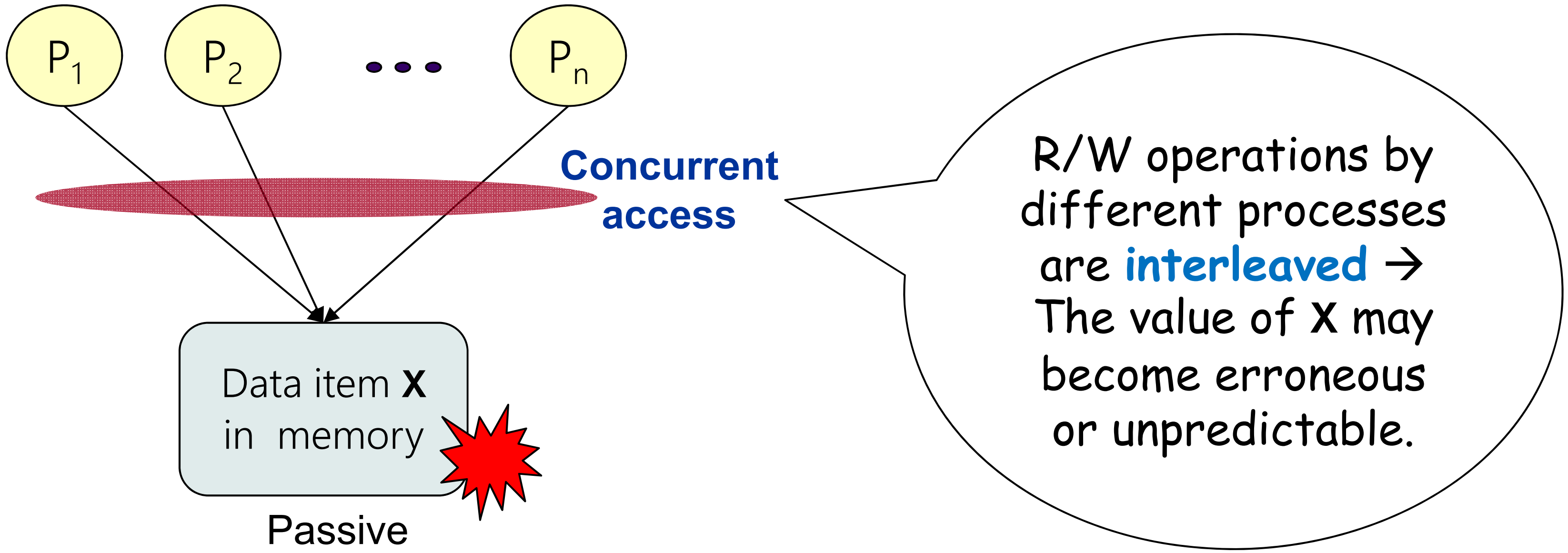
서로 다른 프로세스에 의해 수행되는 R/W 연산이 “인터리브” 된다면 → x 값이 틀려지거나 예측할 수 없게 될 수도 있음

논평:

- 데이터 x 혹은 메모리가 어떤 예방조치를 취하는 것은 불가능
- 동시 접근 때문에 생기는 문제는 프로세스가 해결해야 함
- 따라서 프로세스는 R/W 연산이 적절한 순서로 수행되게 조치를 취하여야 함

Core Ideas Why Bother to Synchronize?

Two or more processes try to do read/write on X

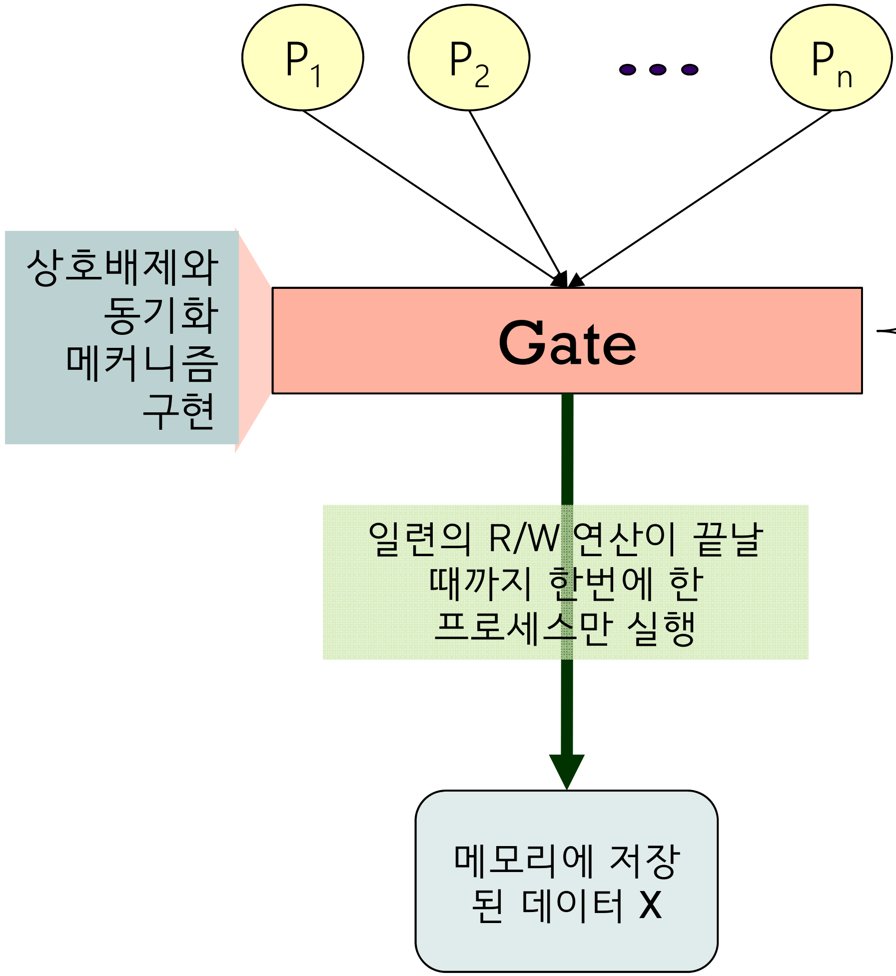


Observations:

- No ways for data item X or memory unit to take any preventive measures.
- It is up to the processes to resolve the problem caused by concurrent access.
- Hence they need to devise a mechanism to arrange R/W's in proper order.

핵심요점

어떻게 질서정연한 접근을 보장할까?



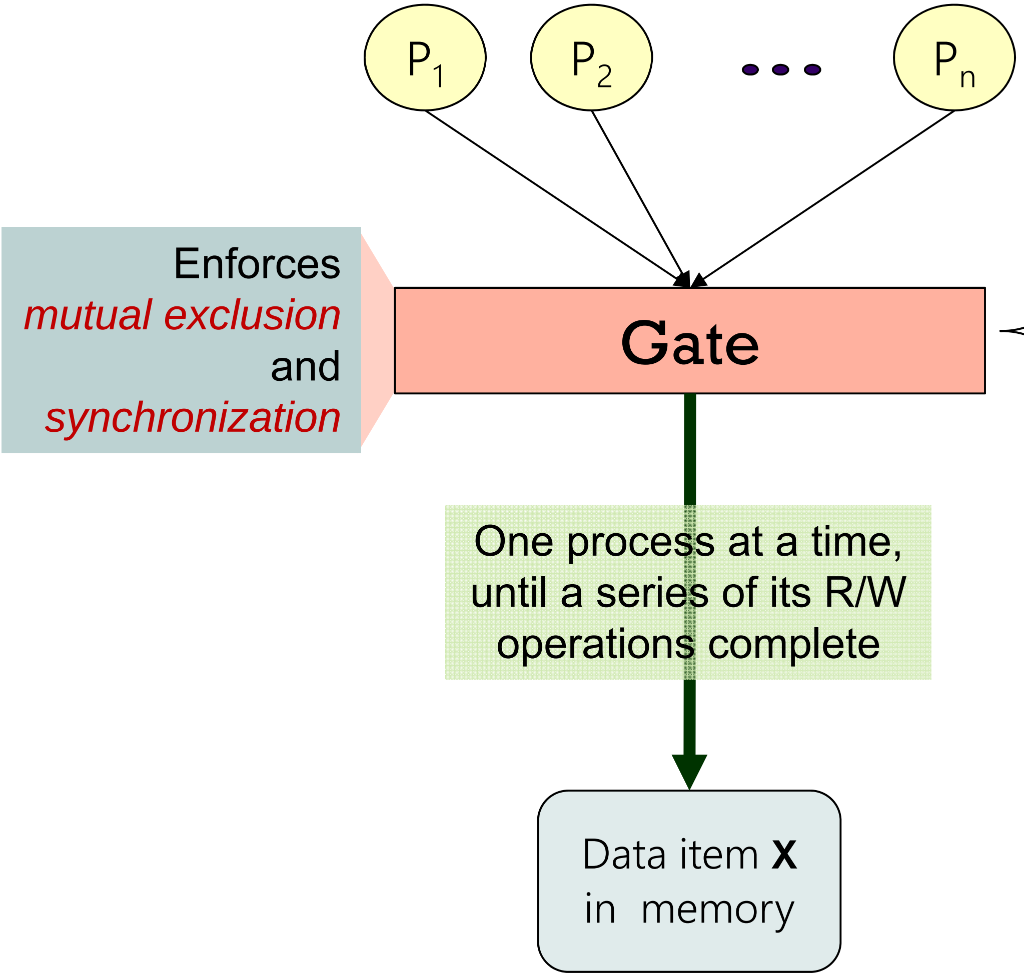
임계영역(critical section)이라 불리는 특별한 코드 세그먼트 사용

- 락(lock)이나 세마포어(semaphore)에 의해 보호되는 일련의 R/W 연산

모니터(monitor)라 불리는 특별한 객체 사용

- 프로시저와 내부 변수의 집합체 (추상적 데이터 유형 (abstract data type))

Core Ideas How to Ensure Orderly Accesses



Use a special code segment called *critical section*

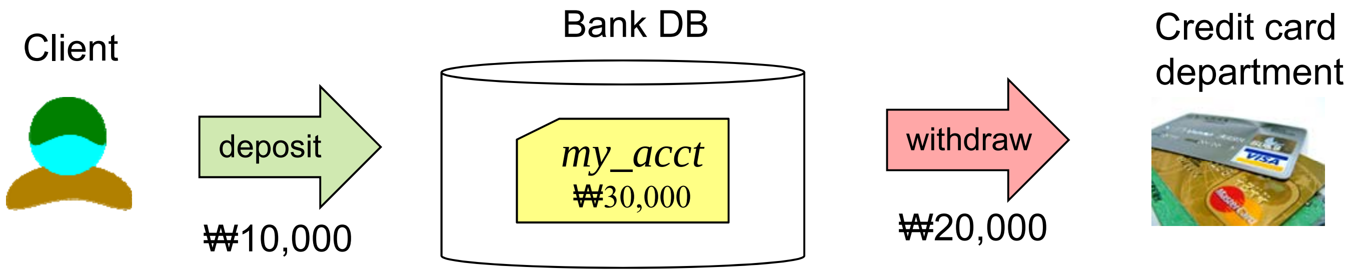
A group of R/W operations guarded by shared variable(s), *locks* or *semaphores*

Use a special construct called *monitor*

A group of procedures plus internal variables (*abstract data type*)

Macroscopic View of Concurrency

Scenario: What would happen if you deposit ₩10,000 in your account whose current balance is ₩30,000. and the credit card (CC) department tries to withdraw ₩20,000 from your account, simultaneously.



- Deposit by client
  (D1) read *my_acct* → *balance*
  (D2) add ₩10,000 to *balance*
  (D3) write balance to *my_acct*

- Withdrawal by CC department
  (W1) read *my_acct* → *cc_balance*
  (W2) subtract ₩20,000 from *cc_balance*
  (W3) write *cc_balance* to *my_acct*

What if these operations are performed in the following sequences?

(1) D1→W1→D2→W2→D3→W3

(2) D1→D2→D3→W1→W2→W3

Which one do you think is correct? Why do they result in different answers?

Consequently, what counts is the order of execution, or *interleaving* sequence.

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem: consumer-producer problem

  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer and Consumer

```
while (true) {
   /* produce an item in next_produced */

   while (counter == BUFFER_SIZE) ;
       /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Producer

What would happen to the shared variable "**counter**" if the two processes run *concurrently*?

```
while (true) {
   while (counter == 0)
       ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
   /* consume the item in next_consumed */
}
```

Consumer

Race Condition

Who gets there first?

- `counter++` could be implemented as

  (i1) `register1 ← counter`
  (i2) `register1 ← register1 + 1`
  (i3) `counter ← register1`

- `counter--` could be implemented as

  (d1) `register2 ← counter`
  (d2) `register2 ← register2 - 1`
  (d3) `counter ← register2`

- Consider this execution interleaving with "counter = 5" initially:
  • producer: `counter++` • consumer: `counter--` ► counter = 5

S0: producer execute (i1) `register1 ← counter` {register1 = 5}
S1: producer execute (i2) `register1 ← register1 + 1` {register1 = 6}
S2: consumer execute (d1) `register2 ← counter` {register2 = 5}
S3: consumer execute (d2) `register2 ← register2 – 1` {register2 = 4}
S4: producer execute (i3) `counter ← register1` {counter = 6 }
S5: consumer execute (d3) `counter ← register2` {counter = 4}

Wrong!

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$

Although the final target of R/W operations is the shared data themselves, there are no ways to secure or protect them directly. Thus, we achieve this end by controlling the access to the data, which is implemented as a code block called *critical section*..

- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process p_i is

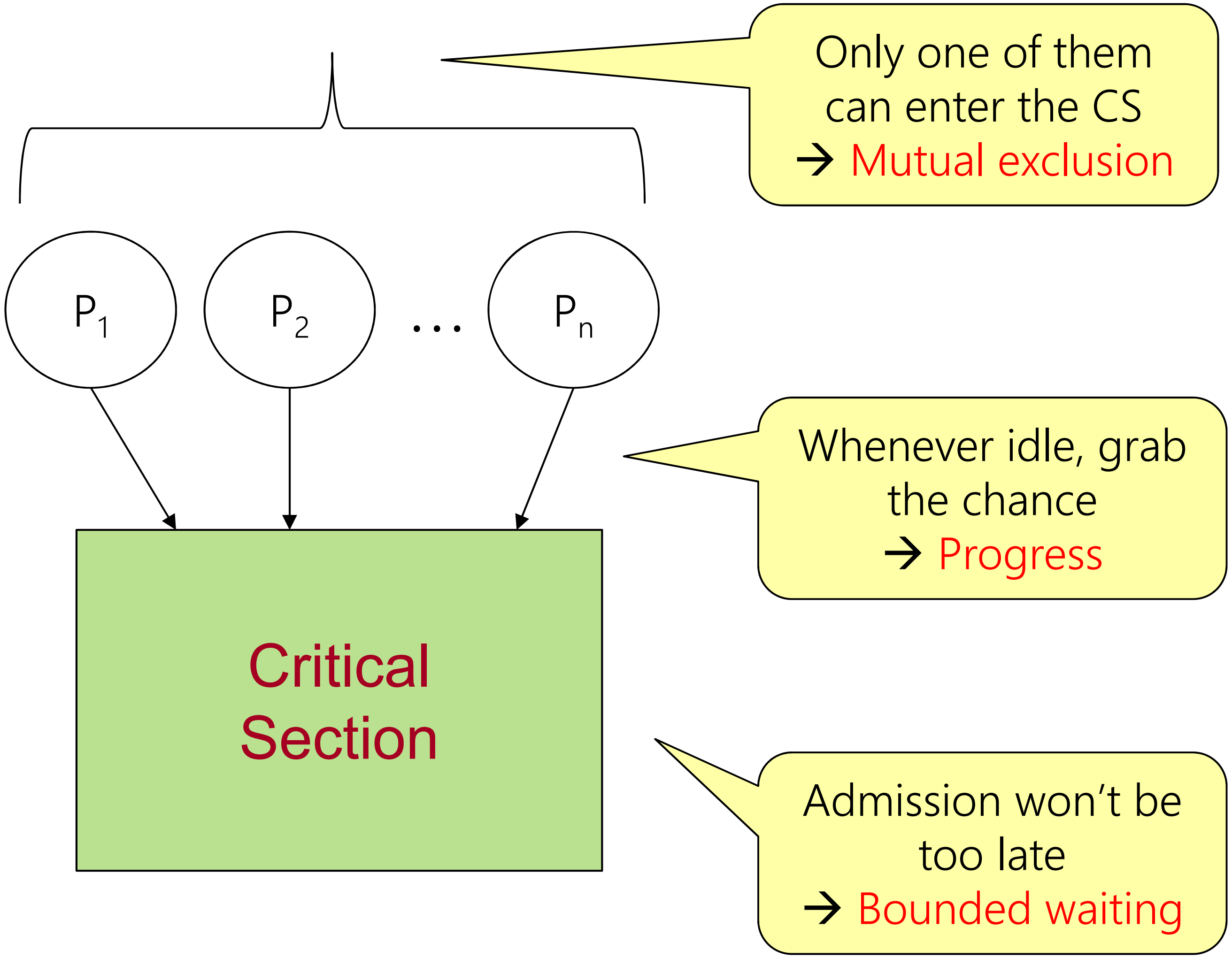
```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

- Two approaches depending on if kernel is preemptive or non-preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Solution to Critical-Section Problem

- Three requirements must be satisfied
- 1. Mutual Exclusion
  - If process P_i is executing in its critical section(CS), then no other processes can be executing in their CSs.
- 2. Progress
  - If no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the processes that will enter the CS next cannot be postponed indefinitely
- 3. Bounded Waiting
  - A bound must exist on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the n processes

Solution to C-S Problem (Cont.)



Peterson's Solution

- Good algorithmic description of solving the problem
- Two-process solution
- Assume that the **load** and **store** instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process P_i is ready!

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
            critical section
    flag[i] = false;
            remainder section
} while (true);
```

Giving way to the other process

- Provable that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - All other processors need be notified of this (disable)
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - ***Atomic*** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Atomic operation: provided by hardware as machine instruction

Solution to Critical-Section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

test_and_set Instruction

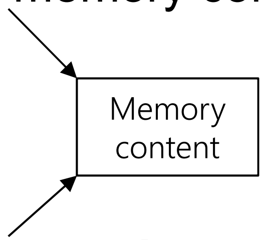
- Definition:

```
boolean test_and_set (boolean *target)
   {
         boolean rv = *target;
         *target = TRUE;
         return rv:
   }
```

- Solution using `test_and_set()`
  - Shared boolean variable lock, initialized to FALSE
  - Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

test → return the memory content



set → change the memory content to 1

compare_and_swap Instruction

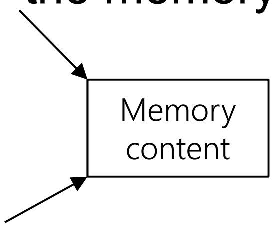
- Definition:

```
int compare_and_swap(int *value, int expected, int
   new_value) {
    int temp = *value;
    if (*value == expected)
       *value = new_value;
    return temp;
}
```

- Solution using **compare_and_swap()**
  - Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
  - Solution:

```
do {
    while (compare_and_swap
             (&lock, 0, 1) != 0)
       ; /* do nothing */
       /* critical section */
    lock = 0;
       /* remainder section */
} while (true);
```

compare → return the memory content



swap → (if content == *expected*) change the memory content to *new_value*

Bounded-waiting Mutual Exclusion with `test_and_set`

shared data structures
```
boolean waiting[n];
boolean lock;
```
all initialized to *false*

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key) /* line c1 */
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

If this line is reached (no other processes are `waiting`!), *`lock`* and then line `c1` becomes false because *`key`* becomes false. → Any process that executes `test_and_set()` first in the future enters c. s.

If this line is reached (*`j`* is `waiting`), *`lock`* remains true and line `c1` for *`j`* becomes false. → *`j`* now enters c. s. → **bounded waiting**

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
     while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}

do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

busy waiting → spin lock

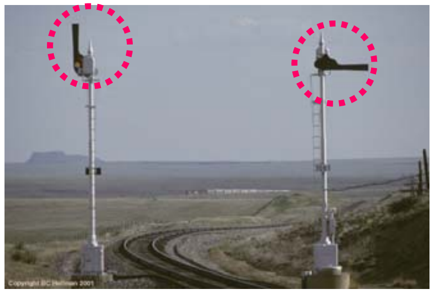
A mutex lock has a boolean variable *available* whose value indicates if the lock is available or not. If the lock is available, a call to `acquire()` succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

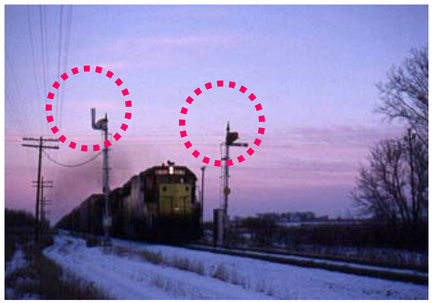
Semaphore

- Synchronization tool that does not require busy waiting, proposed by Edsger Dijkstra in 1965
- Semaphore ***S*** : integer variable
- Two standard operations modify ***S***
  - **`wait()`** and **`signal()`**, originally called **`P()`** and **`V()`**
  - Less complicated
  - Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
     while (S <= 0)
         ; // busy wait
     S--;
}

signal (S) {
     S++;
}
```





Semaphore Usage

- Counting semaphore
  - integer value can range over an unrestricted domain
- Binary semaphore
  - integer value can range only between 0 and 1
  - Then a mutex lock
- Can implement a counting semaphore *S* as a binary semaphore
- Can solve various synchronization problems
- Example: [event ordering] Consider P_1 and P_2 that require S_1 to happen before S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Mutex lock ≈ binary semaphore
But the process that locks the mutex must be the one to unlock it.

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting (Cont.)

```
typedef struct{
   int value;
   struct process *list;
} semaphore;
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
        add this process to S->list;
      block();
   }
}
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
        remove a process P from S->list;
      wakeup(P);
   }
}
```

This (<=) means there is [are] a process [processes] waiting to be awakened.

Now the calling process and P run concurrently. But there is no way to know which process will continue on a uniprocessor system

Deadlock and Starvation

- **Deadlock**
  - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`.`	`.`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Yellow arrows indicate an execution order

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended, where the removal is, for example, based on LIFO
- **Priority Inversion**
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process – blocking possible
  - Solved via **priority-inheritance protocol** – Lower-priority blocking process is given the priority of [higher-priority] blocked process

More about Priority Inversion

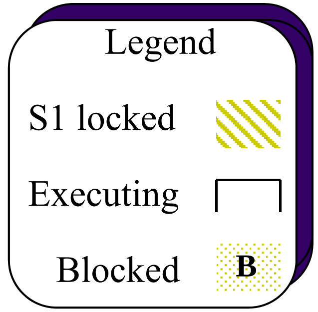
- What is priority inversion problem?
  - When the execution of a process is delayed by interference from lower-priority processes, we say the priority inversion has occurred.
  - If a process is blocked by a lower process, that is, a lower process has entered the critical section and is running there, it is not priority inversion. It is simply a case of blocking.
- Consequence of priority inversion
  - Time-critical or real-time applications with higher priority may have to wait too long to meet the deadline.
- Sources of priority inversion (examples)
  - non-preemptible regions of code
  - interrupts
  - synchronization and mutual exclusion

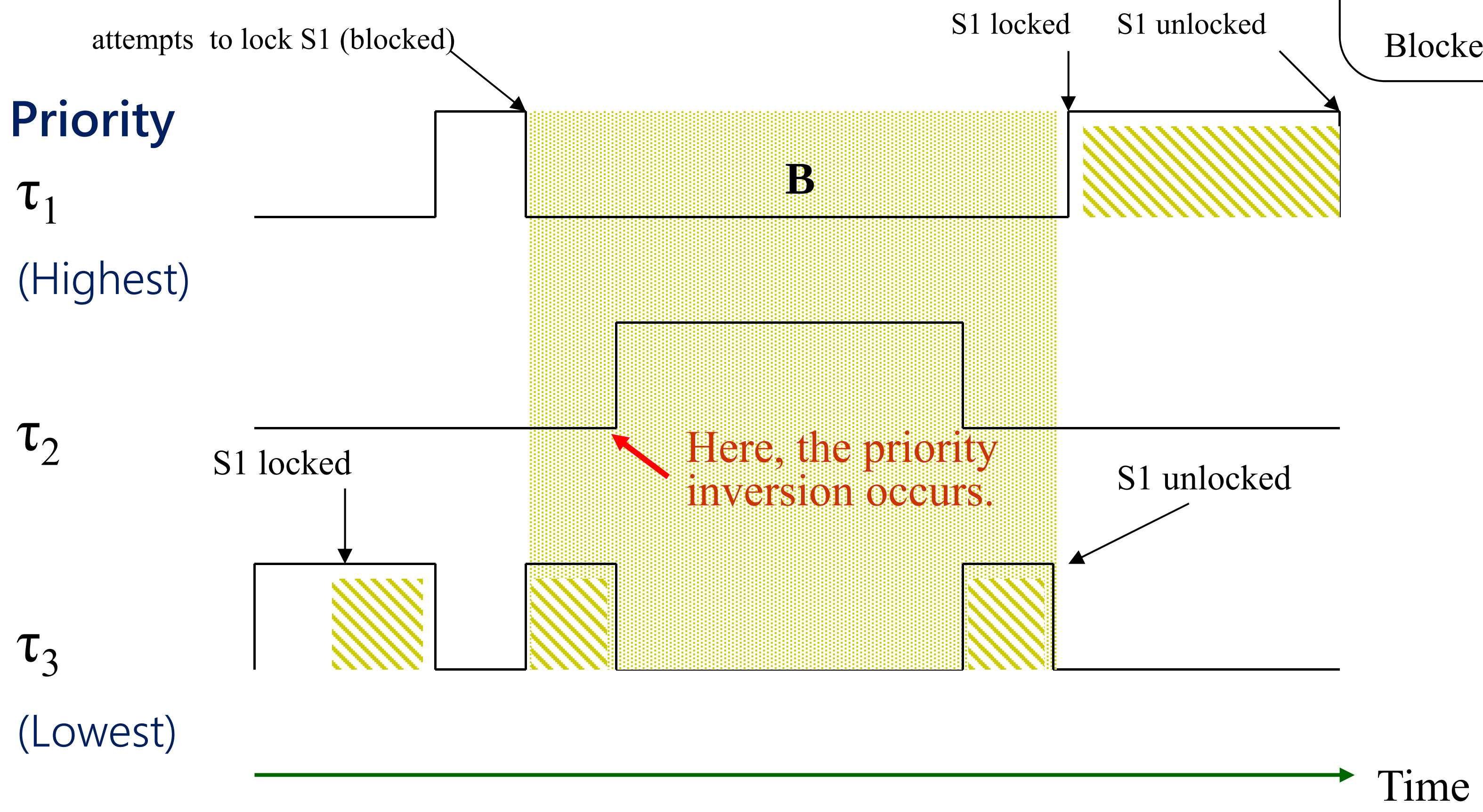
Priority Inversion in Synchronization

τ_1:{…wait(S1)…signal(S1)…}

τ_3:{…wait(S1)…signal(S1)…}

This means the two processes, τ_1 and τ_3, are sharing data, and access the same critical section





Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
      ...
     /* produce an item
      in next_produced */
      ...
   wait(empty);
   wait(mutex);
      ...
     /* add next_produced
       to the buffer */
      ...
   signal(mutex);
   signal(full);
} while (true);
```

- The structure of the consumer process

```
do {
   wait(full);
   wait(mutex);
      ...
     /* remove an item
       from buffer to
       next_consumed */
      ...
   signal(mutex);
   signal(empty);
      ...
     /* consume the item
      in next_consumed */
      ...
  } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1 (for write and read ops.)
  - Semaphore **mutex** initialized to 1 (for update of **read_count**)
  - Integer **read_count** initialized to 0
- Handling the semaphore **rw_mutex**
  - Writer: always competes with other writers and a reader to acquire the lock
  - Reader: first reader competes with a writer to acquire the lock, and last reader releases the lock; Other readers just perform reads

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
   wait(rw_mutex);
        ...
   /* writing is
      performed */
        ...
   signal(rw_mutex);
   /* Now, either writer
   or reader may be
   scheduled */
} while (true);
```

Writers may starve!

- The structure of a reader process

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
       wait(rw_mutex);
   signal(mutex);
     ...
  /* reading is performed */
     ...
   wait(mutex);
   read count--;
   if (read_count == 0)
       signal(rw_mutex);
   signal(mutex);
} while (true);
```

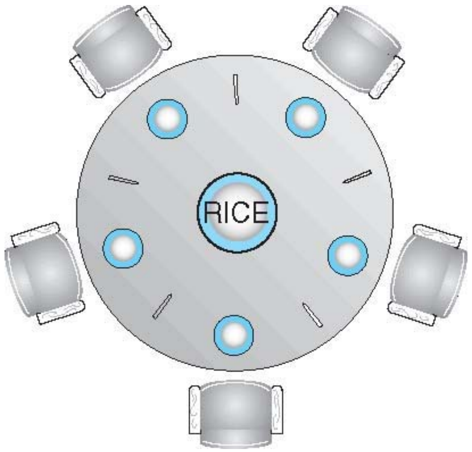
first reader

last reader

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
  - The example shown in the preceding slide implements this variation
- *Second* variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do  {
          wait ( chopstick[i] );
       wait ( chopStick[ (i + 1) % 5] );

              //  eat

       signal ( chopstick[i] );
       signal ( chopstick[ (i + 1) % 5] );

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

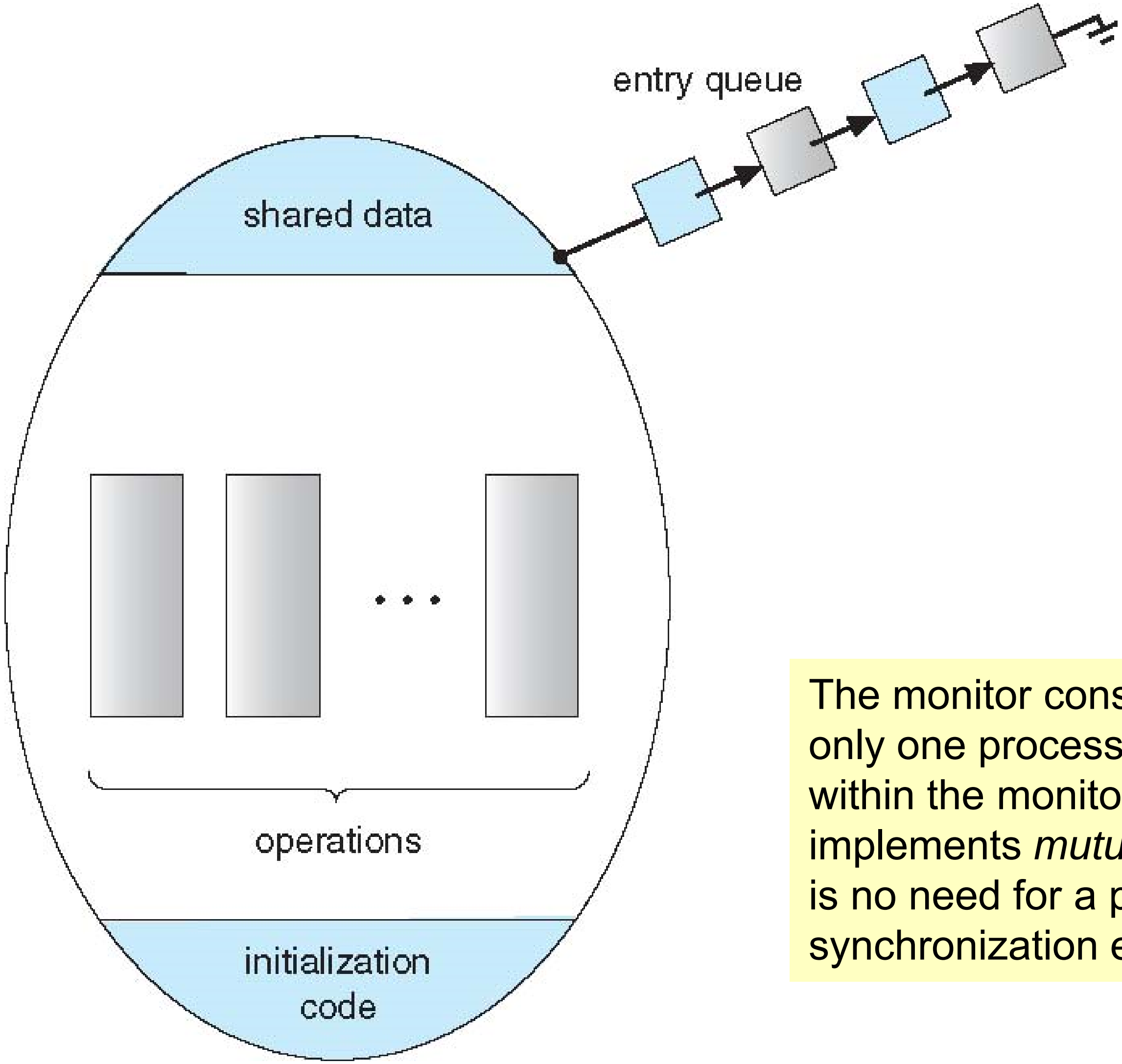
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - Access procedures can be considered critical section
  - Implemented by a compiler
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }          access
                                      procedures
    procedure Pn (…) {……}

    Initialization code (…) { … }
}
```

Schematic view of a Monitor

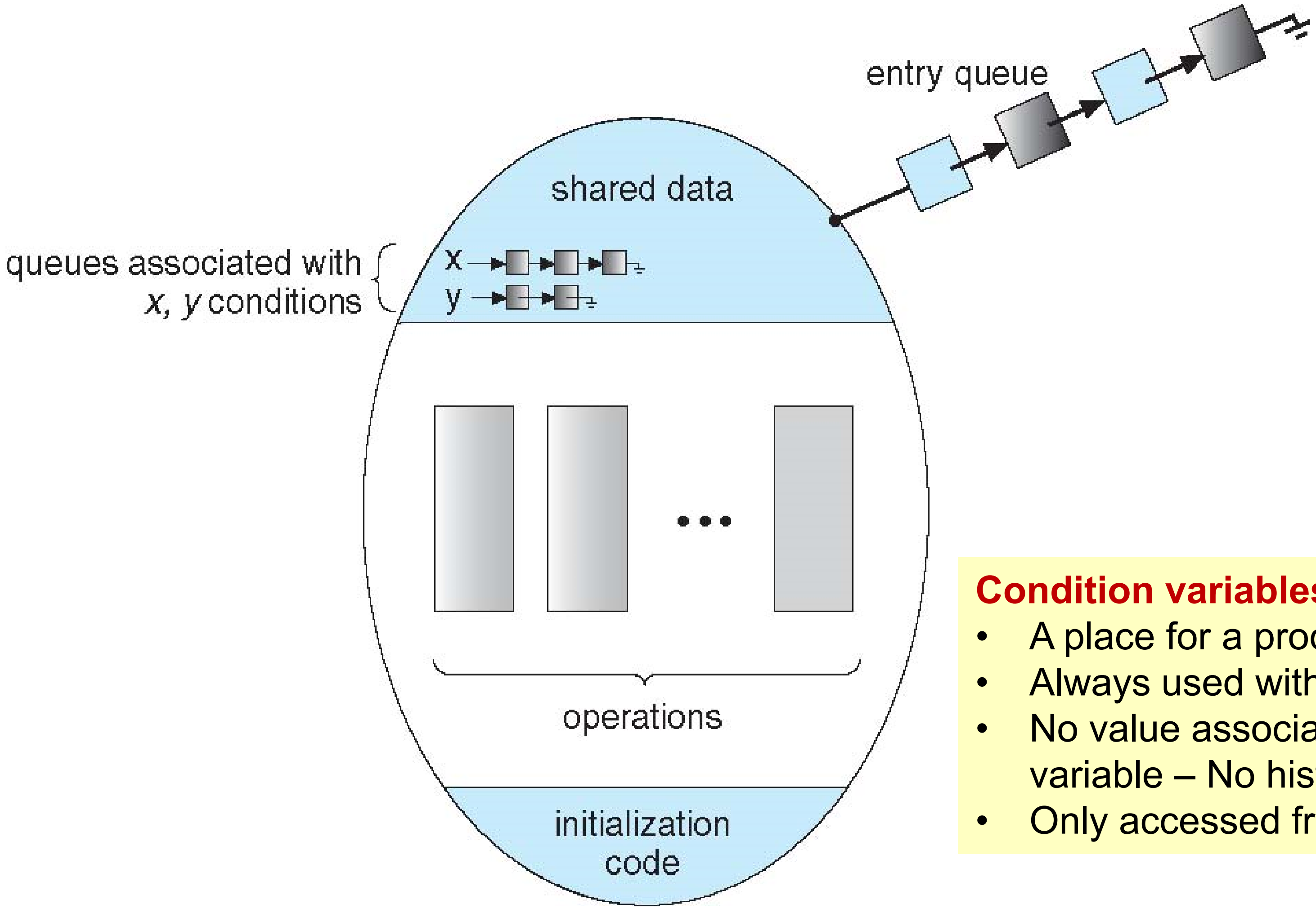


The monitor construct ensures that only one process at a time is active within the monitor. It effectively implements *mutual exclusion*. There is no need for a programmer to code synchronization explicitly.

Condition Variables

- Given two condition variables: condition x, y;
- Two operations on a condition variable:
  - x.wait ()
    - release monitor [lock], so some other process can enter
    - wait for another process to signal condition or until x.signal ()
  - x.signal ()
    - wake up at most one of waiting processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable – This is different from the semaphore case
- Condition variables implement synchronization mechanism
  - Support concurrent processing
  - What if a process is blocked within the monitor until some condition is satisfied?
  - While remaining blocked: Some other process becomes active based on the condition variables

Monitor with Condition Variables



Condition variables

- A place for a process to wait
- Always used with a monitor lock
- No value associated with condition variable – No history
- Only accessed from within monitor

Condition Variables Choices

- If process P invokes x.signal (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P must wait (Refer to "Semaphore implementation" slide)
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
- Implementation
  - Both options have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];      /* condition variables */

   void pickup (int i) {
     state[i] = HUNGRY;
     test(i);
     if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
     state[i] = THINKING;
     test((i + 4) % 5);    /* test right and left neighbors */
     test((i + 1) % 5);
   }

   void test (int i) {     /* test and wake up */
     if ( (state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY)
          && (state[(i + 1) % 5] != EATING) ) {
         state[i] = EATING ;
         self[i].signal () ;
     }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++) state[i] = THINKING;
   }
}
```

"Signal and continue"
Let another process waiting on `self[i]` ready for execution, and the calling process continue executing.

Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup (i);

EAT

DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1),
                     monitor lock
semaphore next;   // (initially  = 0)
                     for suspended processes
int next_count = 0; //# of suspended procs
```

- Each procedure *F* will be replaced by

```
wait(mutex);
     …
   body of F;
     …
if (next_count > 0)
     signal(next)
else
     signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation Using Semaphores – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem;
    (initially, x_sem = 0)
int x_count = 0;
```

- The operation **x.signal** can be implemented as:

```
if (x_count > 0) {
      next_count++;
     signal(x_sem);
      wait(next);
      next_count--;
}
```

- The operation **x.wait** can be implemented as:

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex);
wait(x_sem);
x_count--;
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems

- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted

- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux and Pthreads Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

- Pthreads
  - Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Summary

- 프로세스들이 공유하는 자원 혹은 데이터를 접근하는데 있어 동기화(synchronization)란 무엇인가?
- 병행성(竝行性, concurrency)이란 복수의 프로세스의 실행이 서로 중복되는 상태를 말함. 즉, 동시실행 상태.
- 병행성의 문제점: 서로 교차실행하며 데이터의 일관성(consistency)을 해칠 가능성이 있음 – race condition
- 임계영역(critical section)이란 공유 데이터를 접근(수정)하는 코드 블록으로, 한 프로세스만 실행 가능.
  - 세가지 요건을 만족해야함.

- We use the term "synchronization" when the processes access shared resources or shared data. What is it?
- Concurrency refers to the state in which executions of multiple processes are overlapped or interleaved.
- Concurrency problems: Interleaved execution may fail to maintain the consistency of data. – Race condition
- Critical section is a code segment in which a process may modify shared data. Only a single process is allowed to execute the c. s.
  - three requirements must be met

Summary (Cont.)

- Peterson's solution – locking을 사용하지 않음
- 하드웨어에 의한 동기화: `test&set`, `compare&swap`
- mutex lock: `acquire()`, `release()`
- 세마포어(semaphore)
  - 연산: `wait()`, `signal()`
  - 유형: counting semaphore, binary semaphore (≈ mutex)
  - 이슈: busy waiting, deadlock, starvation, 우선순위 전도(priority inversion)
- 세마포어를 적용할 수 있는 전형적인 동기화 예제
  - 유한한 버퍼 (bounded buffer) 문제
  - 읽는 프로세스와 쓰는 프로세스 (readers and writers) 문제
  - 식사하는 철학자 (dining philosophers) 문제

- Peterson's solution – no locking
- Hardware-based synchronization: `test&set`, `compare&swap`
- mutex lock: `acquire()`, `release()`
- Semaphore
  - operations: `wait()`, `signal()`
  - types: counting semaphore, binary semaphore (≈ mutex)
  - issues: busy waiting, deadlock, starvation, priority inversion
- Classical problems of sync with semaphores
  - bounded buffer problem
  - readers and writers problem
  - dining philosophers problem

Summary (Cont.)

- 모니터 (Monitors)
  - abstract data type을 이용함
  - 상호배제(mutual exclusion) 원칙: 모니터의 구조상 자동적으로 구현됨
  - 병행처리 정도(degree of concurrency)의 향상: 조건변수(condition variable) 이용
    - 모니터에서 작업하던 프로세스가 블록되어 일시중단되면 다른 프로세스가 모니터에 들어갈 수 없음. 즉 병행성 문제. 따라서 블록되는 경우를 유발하는 변수를 조건변수로 지정
  - 조건변수에 대한 연산: 프로세스의 실행 순서를 구현
    - x.wait(): x.signal()가 있을때까지 중단
    - x.signal(): x.wait()를 호출한 프로세스 중에서 하나가 계속 실행되게 함
- 동기화 구현 예
  - Solaris, Windows XP, Linux, Pthreads

- Monitors
  - Use abstract data type
  - Mutual exclusion principle: the monitor construct itself ensures mutual exclusion among processes
  - Degree of multiprogramming or concurrency: can be increased using condition variables
    - If a processing running within the monitor is blocked, no other processes can enter the monitor. Concurrency problem. For this reason, we designate as condition variable the variables that cause blocking
  - Operations on condition variables: implement the correct ordering of processes
    - x.wait(): suspended until x.signal()
    - x.signal(): resumes one of processes that invoked x.wait()
- Synchronization examples
  - Solaris, Windows XP, Linux, Pthreads