5. Process Synchronization

ECE30021/ITP30002 Operating Systems

Announcement

- Please prepare this chapter before the class.
  - Read the textbook in advance
  - Otherwise, this chapter would be fairly hard to understand.

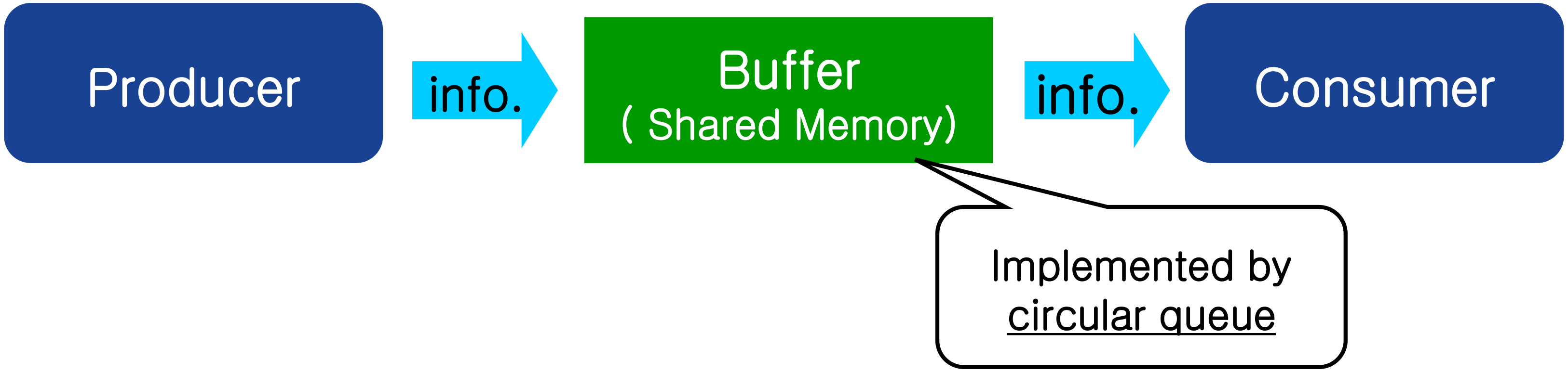
Agenda

- Background
- The critical-section problem
- Synchronization hardware
- Semaphores
- Classical problems of synchronization
- Monitors
- Synchronization examples
- Atomic transaction

Background

- Process communication method
  - Message passing
  - Shared memory -> confliction can occur !!

- Producer-consumer problem
  - An example of communication through shared memory



Concurrent Access of Shared Data

- Producer

```
while(true){
    while(counter==BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in+1)%BUFFERSIZE;
    counter++;
}
```

- Consumer

```
while(true){
    while(counter==0);
    nextConsumed = buffer[out];
    out = (out+1)%BUFFER_SIZE;
    counter--;
}
```

- Implementation of "**counter++**"

  $register_1$ = counter
  $register_1$ = $register_1$ + 1
  counter = $register_1$

- Implementation of "**counter--**"

  $register_2$ = counter
  $register_2$ = $register_2$ – 1
  counter = $register_2$

Switching can occur during modification operations !

Synchronization Problem

- Problematic situation
  - Initial value of counter is 5.
  - Producer increased counter, and consumer decreased counter concurrently.
  - Ideally, counter should be 5. But…

Producer	Consumer
register$_1$ = counter (*register1 = 5*)	
register$_1$ = register$_1$ + 1 (*register1 = 6*)	
	register$_2$ = counter (*register2 = 5*) register$_2$ = register$_2$ – 1 (*register2 = 4*)
counter = register$_1$ (*counter = 6*)	
	counter = register$_2$ (*counter = 4*)

counter can be 4 or 6 !

Process Synchronization

- Race condition
  - A situation, where several processes access and manipulate the same data concurrently
  - The outcome of execution depends on the particular order in which the access takes place.

- Synchronization: **the coordination of occurrences to operate in unison with respect to time.**

  Ex) ensuring only one process can access the shared data at a time

Agenda

- Background
- The critical-section problem
- Synchronization hardware
- Semaphores
- Classical problems of synchronization
- Monitors
- Synchronization examples
- Atomic transaction

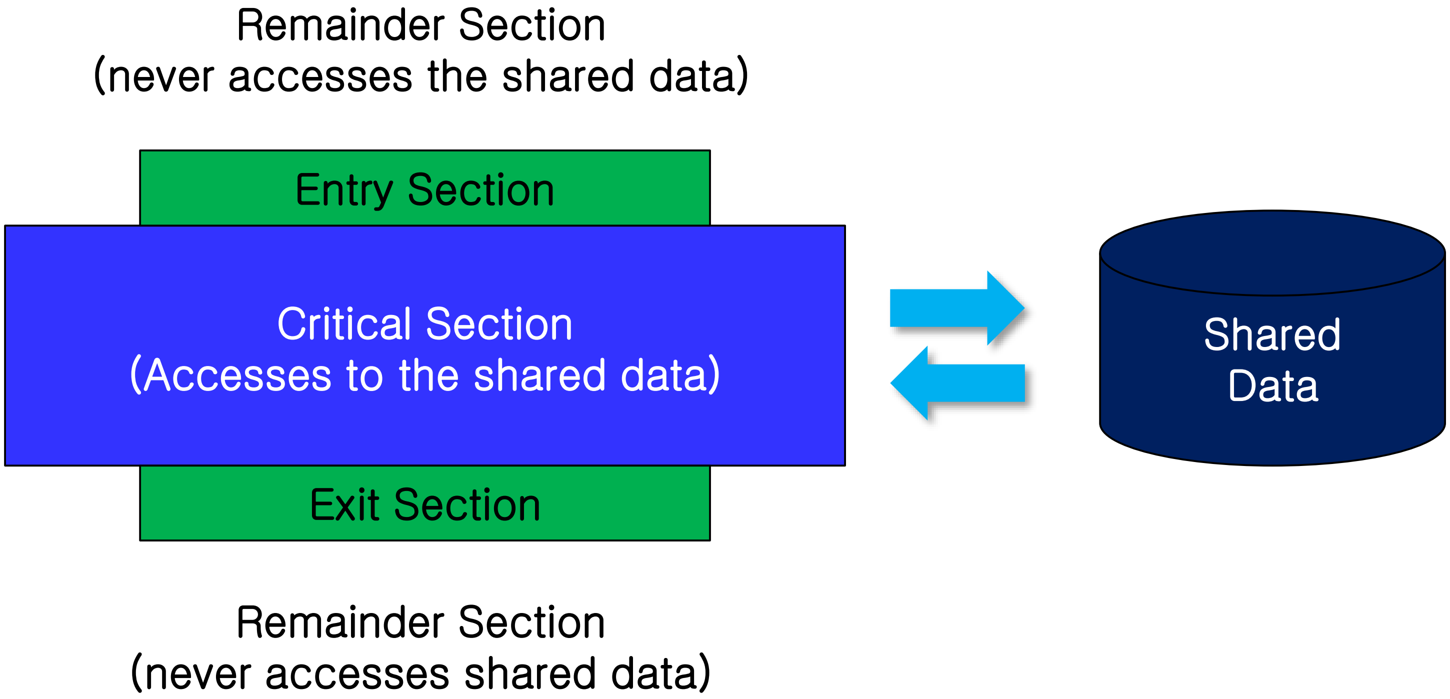
The Critical-Section Problem

- Critical section problem: designing a protocol that processes can use to cooperate
  - *n* processes P_0, …, P_{n-1} are running on a system concurrently. Each process want access a shared data
  - The code of each process is composed of
    - Critical section: a segment of code which may change shared resources (common var., file, …)
    - Remainder section: a segment of code which doesn't change shared resources

    - Entry section: code section to request permission to enter critical section
    - Exit section: code section to notice it leaves the critical section

Critical Section

- A process modifies shared data only in the critical section
- At a time only one process can exist in its critical section.



Remainder Section
(never accesses the shared data)

Entry Section

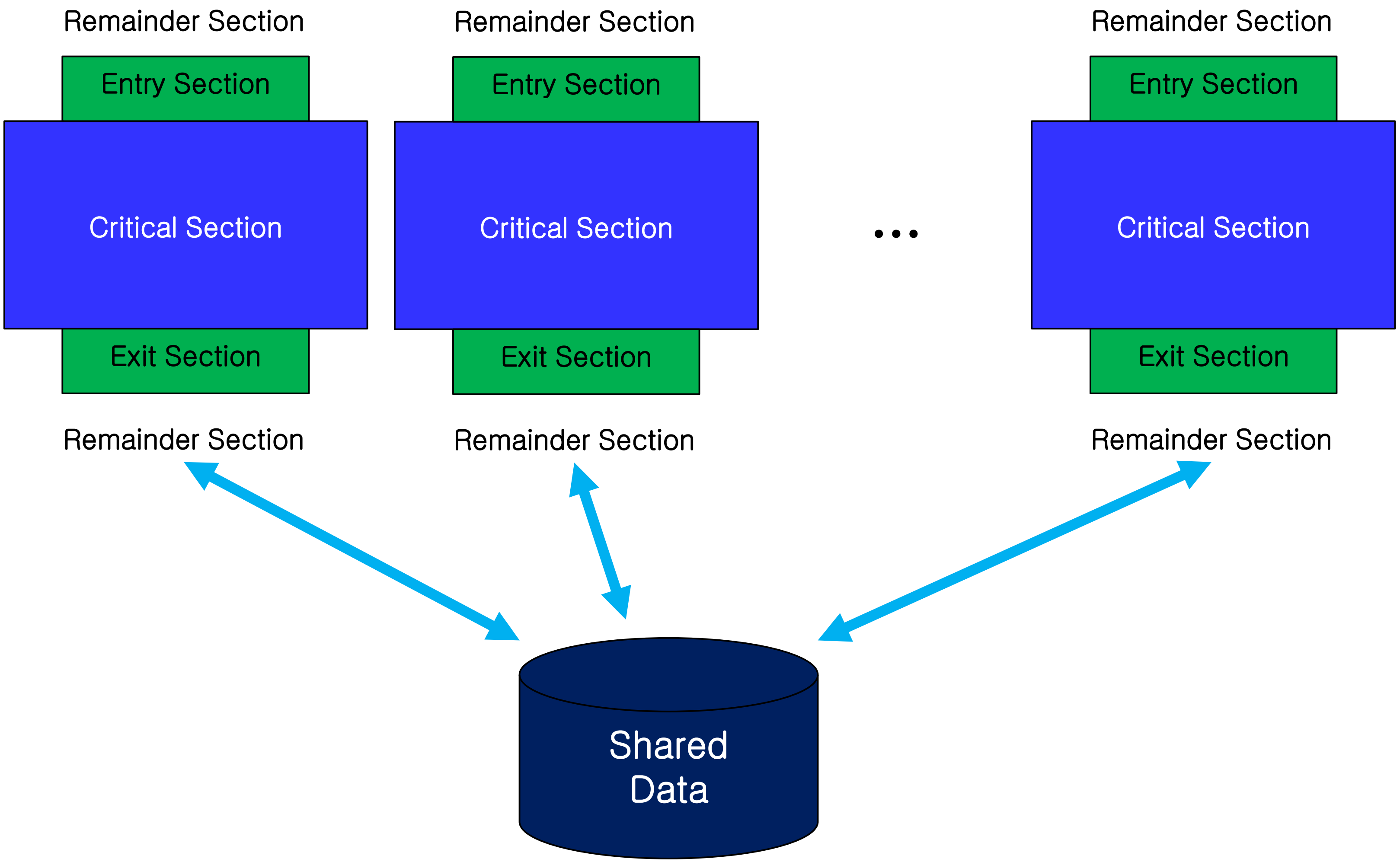
Critical Section
(Accesses to the shared data)

Exit Section

Shared
Data

Remainder Section
(never accesses shared data)

Critical Section



The Critical-Section Problem

- General structure of typical processes

```
do {
    entry section

        critical section

    exit section

        remainder section

} while(TRUE);
```

The Critical-Section Problem

- Requirements of critical-section problem
  - Mutual exclusion: If a process is in its critical section, no other processes can be executing in their critical sections

    -> Other processes should wait

  - Progress: If no process is executing in its critical section, and some processes wish to enter their critical sections, only processes not executing in their remainder section can participate in the decision of next process to enter its critical section next. This selection cannot be postponed indefinitely.

  - Bounded waiting: There exists a bound or limit on number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before it is granted.

The Critical-Section Problem

- Kernel is an example of critical-section problem
  - Kernel data structures such as list of open files,
- Approaches
  - Non-preemptive kernel: switching cannot occur when a process is executing in kernel mode
    - Free from race condition

    Ex) Windows 2000, Windows XP, early version of UNIX, Linux prior to v. 2.6.
  - Preemptive kernel accompanied with a solution of critical-section problem
    - Suitable for real-time programming
    - More responsive

    Ex) Linux v. 2.6, Solaris, IRIX

Peterson's Solution

- A S/W solution for critical section problem

  ➔ No guarantee to work correctly on some architectures due to **load/store** instructions, but helps to understand the problem

  - Two processes P_0 and P_1 (or P_i and P_j)
  - Data items

```
int turn;          // indicates process allowed to execute
                   // in its critical section
                   // initial value is 0

boolean flag[2];   // if flag[i] is true, Pi is ready to enter
                   // its critical section
```

Erroneous Algorithm 1

- Algorithm 1
  - Process P_i

```
do {
        while (turn != i);
                critical section
        turn = j;
                remainder section
} while (1);
```

Erroneous Algorithm 1

- Process P_0

```
do {
        while (turn != 0) ;
                critical section
        turn = 1;
                remainder section
} while (1);
```

- Process P_1

```
do {
        while (turn != 1) ;
                critical section
        turn = 0;
                remainder section
} while (1);
```

- Mutual exclusion is guaranteed, but progress is not guaranteed.

  Ex) P_1 is trying to enter its critical section, but P_0 is in its remainder section -> turn is not switched to 1

Erroneous Algorithm 2

- Algorithm 2
  - Process P_i

```
do {
        flag[i] = true;
        while (flag[j]) ;
                critical section
        flag[i] = false;
                remainder section
} while (1);
```

Erroneous Algorithm 2

- Process P_0

```
do {
        flag[0] = true;
        while(flag[1]);
            critical section
        flag[0] = false;
            remainder section
} while (1);
```

- Process P_1

```
do {
        flag[1] = true;
        while(flag[0]);
            critical section
        flag[1] = false;
            remainder section
} while (1);
```

- Mutual exclusion is guaranteed, but progress is not guaranteed.

    Ex) P_0 and P_1 enter simultaneously. Both of flag[0] and flag[1] can be true

Peterson’s Solution

- Peterson’s Solution
  - Process P_i

```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] and turn == j);
          critical section
     flag[i] = false;
          remainder section
} while (1);
```

  - Process P_j

```
do {
     flag[j] = true;
     turn = i;
     while (flag[i] and turn == i);
          critical section
     flag[j] = false;
          remainder section
} while (1);
```

- Satisfies the three conditions

Peterson's Solution

- Peterson's Solution
  - Process P_0

```
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] and turn == 1);
        critical section
    flag[0] = false;
        remainder section
} while (1);
```

  - Process P_1

```
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] and turn == 0);
        critical section
    flag[1] = false;
        remainder section
} while (1);
```

- Satisfies the three conditions

Peterson’s Solution

- Proof of mutual exclusion
  - If both P_i and P_j enter their critical section, it means
    - flag[0] = flag[1] = true
    - turn can be either 0 or 1, but turn cannot be both
- Proof of progress and bounded waiting
  - Blocking condition of P_i: flag[j] == true and turn == j
    - If P_j is not ready to enter critical section: flag[j] == false

      -> P_i can enter critical section
    - If P_j waiting in its while statement, turn is either i or j
    - If turn is i, P_i will enter critical section.
    - Otherwise, P_j will enter critical section.

      -> When P_j exits critical section, P_j sets flag[j] to false and P_i can enter critical section because P_j modifies turn to i.
    - Therefore, P_i will enter critical section (Progress) and waits at most one process (Bounded waiting).

Agenda

- Background
- The critical-section problem
- **Synchronization hardware**
- Semaphores
- Classical problems of synchronization
- Monitors
- Synchronization examples
- Atomic transaction

Synchronization Hardware

- Critical section problem requires lock
  - Race condition can be prevented by protecting critical section by *lock*

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

- H/W support makes it easier and improve efficiency

Interrupt Disable

- In single processor system, critical section problem can be solved by simply disabling interrupt
  - Non-preemptive kernel

- Problems
  - Inefficiency in multiprocessor environment
  - In some systems, clock is updated by interrupt

- This approach is taken by non-preemptive kernels

Atomic Instructions

- If H/W provides atomic instructions, locking is easer to implement
  - TestAndSet: set a variable to true and returns its previous value

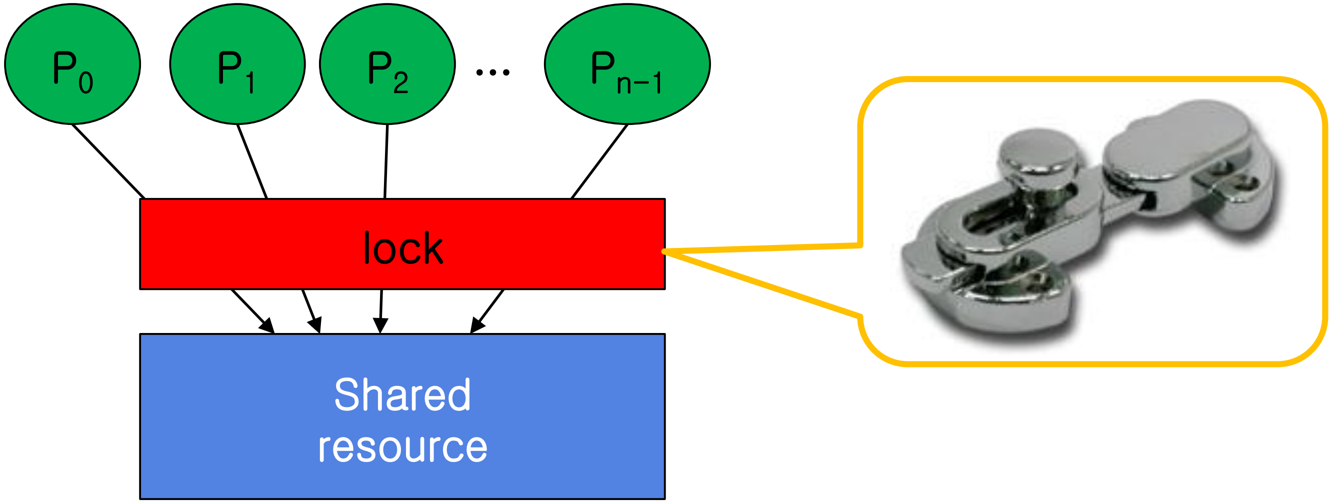
```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

  - Swap: exchanges two variables

```
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Mutual Exclusion by Lock variable

- A shared variable *lock*
    - Boolean lock = 0;
    - If *lock* is false, any process can enter its critical section
        - When a process enters its critical section, it sets *lock* to true
    - If *lock* is true, a process (P_i) is in its critical section.
        - Other processes should wait until P_i leaves its critical section and sets *lock* to false



Mutual Exclusion by Lock variable

- Initially, lock == false

- P_0

```
do {
    while(lock);
    lock = true;
        critical section
    lock = false;
        remainder section
} while(TRUE);
```

- P_1

```
do {
    while(lock);
    lock = true;
        critical section
    lock = false;
        remainder section
} while(TRUE);
```

- Note! Checking and locking must not be separated

Mutual Exclusion using TestAndSet

- Shared variable
  - boolean lock = false;
- Process P_i

```
do {
        while (TestAndSet(&lock)) ;
                critical section
        lock = false;
                remainder section
}
```

- The while loop checks *lock* and sets it to true at the same time
- Case 1: *lock* == false
  - P_i passes while loop
  - Set *lock* to true
    - P_j should wait
- Case 2: *lock* == true
  - P_i waits at the while loop until other process turns lock to false

Mutual Exclusion using Swap

- Shared variable
  - boolean lock = false;
- Process *P*$_i$

```
do {
        key = true;
        while (key == true)
                Swap(&lock, &key);
                        critical section
        lock = false;
                        remainder section
}
```

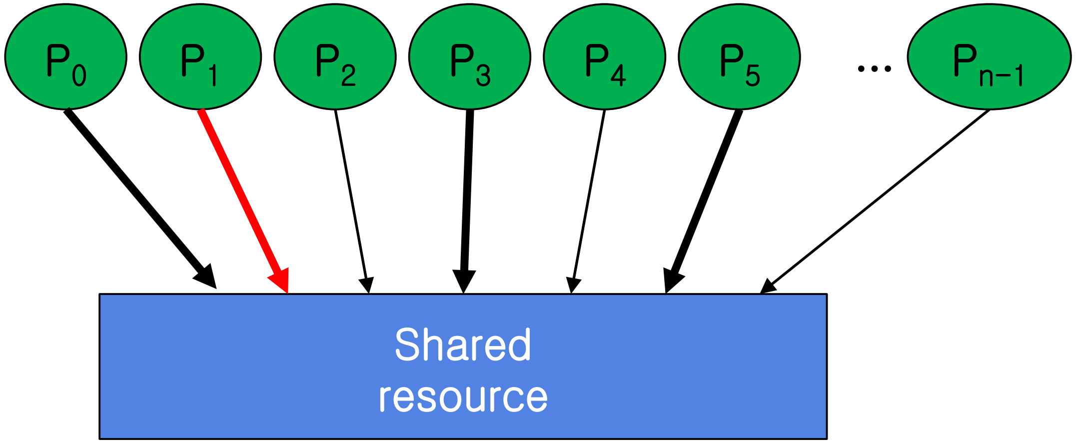
  - *key* is a local variable

- Swap fetches *lock* and sets it to true at the same time
- Case 1: *lock* == false
  - P$_i$ passes its while loop
  - Set *lock* to true
    - Other processes should wait
- Case 2: *lock* == true
  - P$_i$ waits at the while loop until other process sets *lock* to false

Bounded Waiting Mutual Exclusion

- Bounded waiting for n processes
  - Some processes want to enter their critical sections

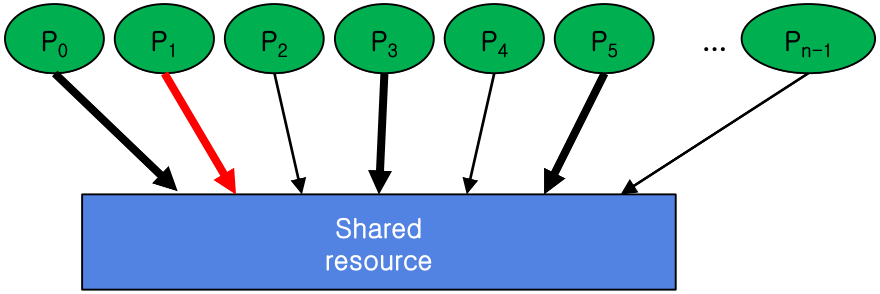
    Ex) P_0,P_1,P_3,P_5
  - One of them, (ex: P_1) is in its critical section



- Idea: Whenever a process leaves its critical section, it designates the next process to enter the critical section.
  - The next process: The first waiting process in right direction.

Bounded Waiting Mutual Exclusion

- Shared variables
  - boolean lock;
    - For mutual exclusion
  - boolean waiting[n];
    - if waiting[i] is true, it means P_i wants to enter critical section, but it didn't enter yet.



process	0	**1**	2	3	4	5	6	…	n-1
waiting	1	**0**	0	1	0	1	0	…	0

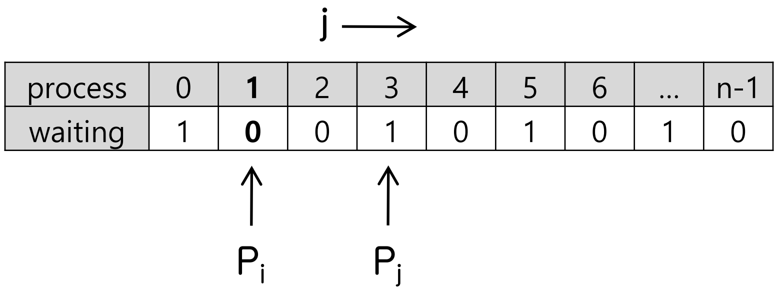
↑
P_i

- Algorithm

```
do{
        waiting[i] = TRUE;
        key = TRUE;
        while(waiting[i] && key)
            key = TestAndSet(&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);
```

Bounded Waiting Mutual Exclusion

- When P_i wants to enter its critical section, it turns *waiting[i]* to true and checks *waiting[i]* and *lock*
- When P_i enters its critical section, it turns *lock* to true and *waiting[i]* to false
- While P_i is in its critical section, other processes wait at their while loops



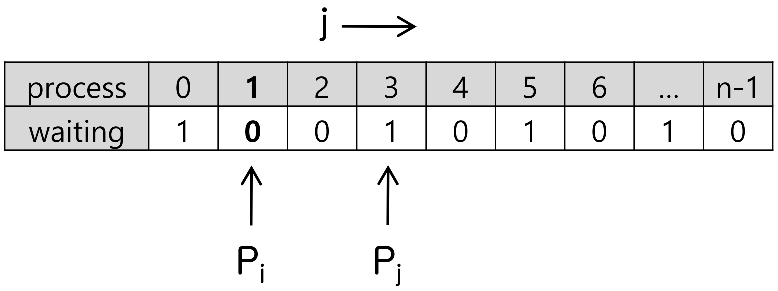
process	0	**1**	2	3	4	5	6	…	n-1
waiting	1	**0**	0	1	0	1	0	1	0

- Algorithm

```
do{
        waiting[i] = TRUE;
        key = TRUE;
        while(waiting[i] && key)
            key = TestAndSet(&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);
```

Bounded Waiting Mutual Exclusion

- When P_i leaves its critical section, it searches for the process P_j to give the next chance in right direction
- If there is a waiting process P_j, P_i releases by P_j setting *waiting[j]* to false
- Otherwise, P_i releases *lock*



process	0	**1**	2	3	4	5	6	…	n-1
waiting	1	**0**	0	1	0	1	0	1	0

- Algorithm

```
do{
        waiting[i] = TRUE;
        key = TRUE;
        while(waiting[i] && key)
            key = TestAndSet(&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);
```

Agenda

- Background
- The critical-section problem
- Synchronization hardware
- <u>Semaphores</u>
- Classical problems of synchronization
- Monitors
- Synchronization examples
- Atomic transaction

Semaphores

- Lock using atomic instructions is complicated
  - Higher-level tool is required
- Semaphore: an integer variable accessed only through two atomic operations: wait() and signal()
  - Initial value of S is 1 or a positive integer.
  - Wait

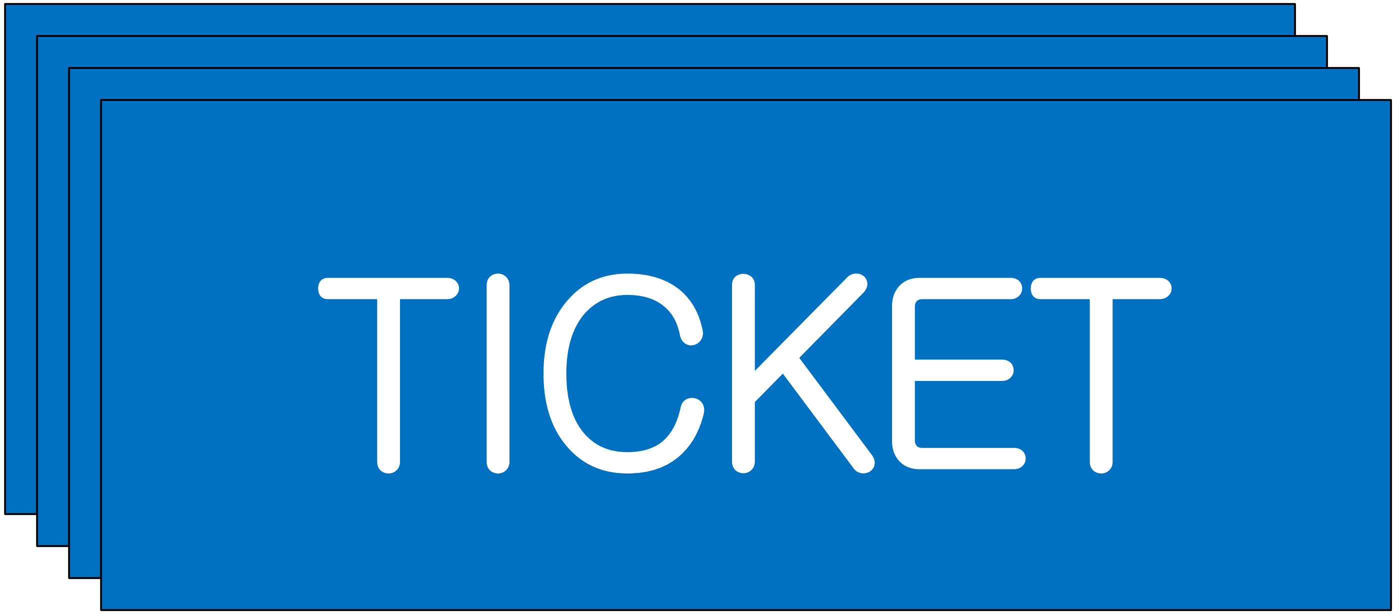
```
wait(S) {
   while(S <= 0);              // waits for the lock
   S--;                        // holds the lock
}
```

  - Signal

```
signal(S) {
   S++;                        // releases the lock
}
```

Semaphores

- S is similar to the number of reusable tickets to enter the critical section.



Semaphores

- Mutual-exclusion implemented with semaphore

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
}
```

- If initially value of S is 1,
  - wait(S) acquires lock
  - signal(S) releases lock

- Wait

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

- Signal

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

Types of Semaphores

- Binary semaphore (mutex locks)
  - S can be 0 or 1
  - Used to ensure mutual exclusion
- Counting semaphore: unrestricted range
  - S can be any integer number
  - Initialized to # of available resources
    - If initial value of S is a positive number k, first k processes can enter their critical sections.
    - Other processes cannot enter their critical sections until one of them leaves the critical section.

Implementation of Semaphore

- Problem of previous definition of wait(): spinlock

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

- Alternative implementation: block instead of spinlock
  - If a process invokes wait(S), put the process into a waiting queue of S and block itself.

Implementation of Semaphore

- New definition of semaphore

```
typedef struct {
    int value;
    struct process *list;     // waiting queue
} semaphore;
```

- Wait

```
wait(semaphore *S){
    S->value--;
    if(S->value < 0){
      add this process to S->list;
      block(); // suspend
    }
}
```

- Signal

```
signal(semaphore *S)
{
    S->value++;
    if(S->value <= 0){
      remove a process P from S->list;
      wakeup(P);    // resume
    }
}
```

Implementation of Semaphore

- Critical aspect of semaphore: atomicity
  - Atomicity is often enforced by mutual exclusion
  - Single processor environment: disable interrupt
  - Multiprocessor environment
    - Disable interrupt of all processors -> performance degradation
    - Spinlock (but much short than previous algorithms)

Deadlock and Starvation

- Deadlock

```
P0              P1
wait(S);        wait(Q);
wait(Q);        wait(S);
  ⋮               ⋮
signal(S);      signal(Q);
signal(Q);      signal(S);
```

- Starvation (infinite blocking): a situation in which a process waits indefinitely within the semaphore

  Ex) waiting list is implemented by LIFO order

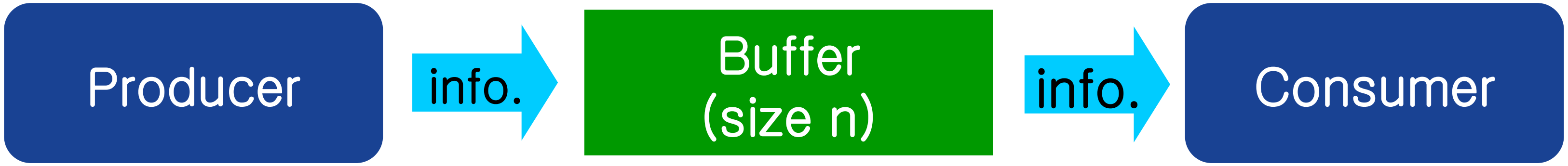
Agenda

- Background
- The critical-section problem
- Synchronization hardware
- Semaphores
- <u>Classical problems of synchronization</u>
- Monitors
- Synchronization examples
- Atomic transaction

Classical Problems of Synchronization

- The bounded-buffer problem
- The readers-writers problem
- The dining-philosophers problem

The Bounded-Buffer Problem



Producer

info.

Buffer
(size n)

info.

Consumer

- If the buffer is full, the producer must wait until the consumer deletes an item.
  - Producer needs an empty space
  - # of empty slot is represented by a semaphore *empty*
- If the buffer is empty, the consumer must wait until the producer adds an item.
  - Consumer needs an item
  - # of item is represented by a semaphore *full*

The Bounded-Buffer Problem

- Producer-consumer problem with bounded buffer
  - Semaphores: full = 0, empty = n, mutex = 1;
  - Producer

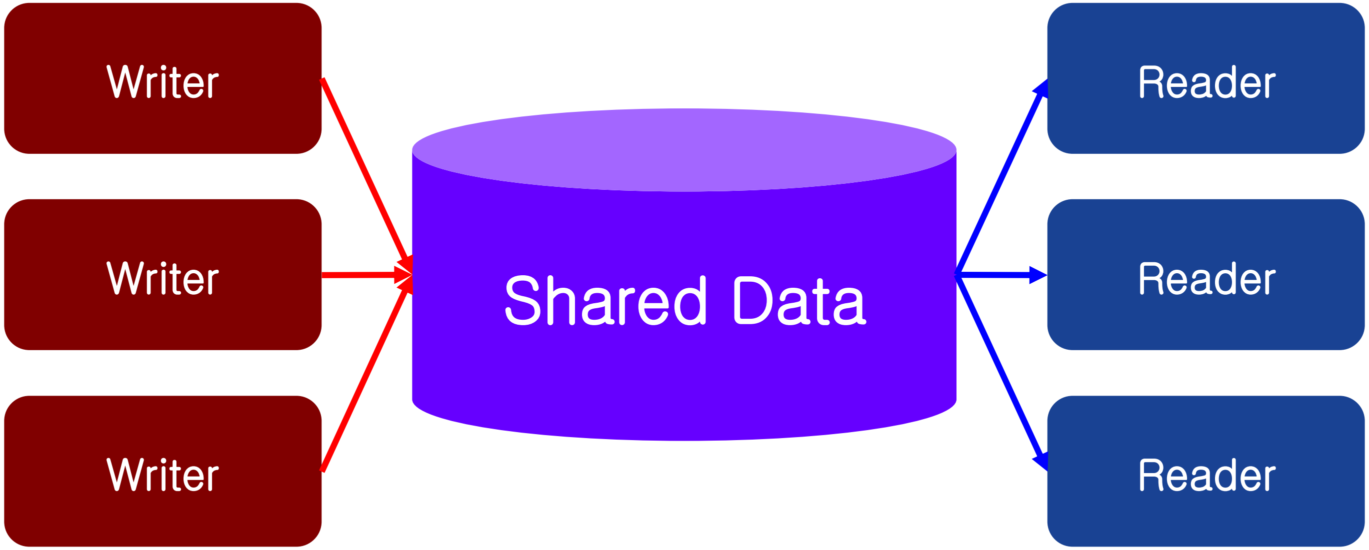
```
do {
            …
   produce an item in nextp
            …
   wait(empty);
   wait(mutex);
            …
   add nextp to buffer
            …
   signal(mutex);
   signal(full);
} while (1);
```

  - Consumer

```
do {
   wait(full);
   wait(mutex);
            …
   remove an item from buffer to nextc
            …
   signal(mutex);
   signal(empty);
            …
   consume the item in nextc
            …
} while (1);
```

The Readers–Writers Problem

- There are multiple readers and writers to access a shared data
  - Readers can access database simultaneously.
  - When a writer is accessing the shared data, no other thread can access it.



The Readers-Writers Problem

- Behavior of a writer
  - If a thread is in the critical section, all writers must wait.
  - The writer can enter the critical section only when no thread is in its critical section.
    - It should prevent all threads from entering the critical section.

- Behavior of a reader
  - If no writer is in its critical section, the reader can enter the critical section.
  - Otherwise, the reader should wait until the writer leaves the critical section.
  - When a reader is in its critical section, any reader can enter the critical section, but no writer can.
    - Condition for the first reader is different from the following readers.

The Readers-Writers Problem

- Shared data
  - semaphore mutex=1, wrt=1;
  - int readcount = 0;
    - # of readers in critical section

- Writer

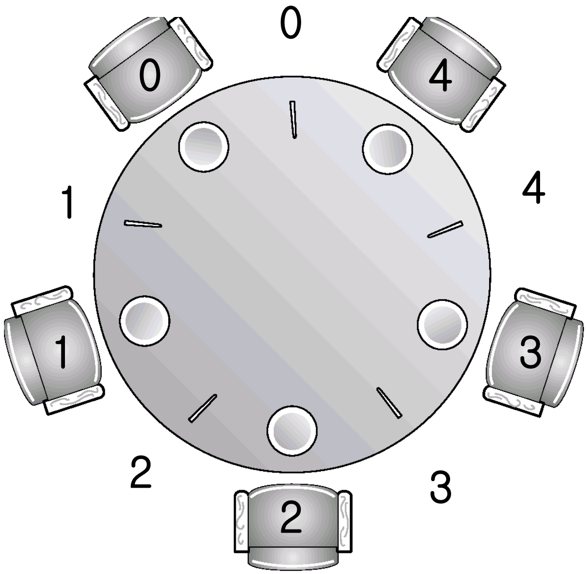
```
wait(wrt);
   …
writing is performed
   …
signal(wrt);
```

- Reader

```

readcount++;
if (readcount == 1)
        wait(wrt);

          …
reading is performed
          …

readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex);
```

The Dining Philosophers Problem

- Problem definition
  - 5 philosophers sitting on a circular table
    - Thinking or eating
  - 5 bowls, 5 single chopsticks
  - No interaction with colleagues
  - To eat, the philosopher should pick up two chopsticks closest to her
  - A philosopher can pick up only one chopstick at a time
  - When she finish eating, she release chopsticks



- Solution should be deadlock-free and starvation free

The Dining Philosophers Problem

- A possible solution, but deadlock can occur

```
do {
    wait(chopstick[i]);              // pick up left chopstick
    wait(chopstick[(i+1) % 5]);      // pick up right chopstick
     ...
    eat
     ...
    signal(chopstick[i]);            // release left chopstick
    signal(chopstick[(i+1) % 5]);    // release right chopstick
    ...
    think
    ...
} while (TRUE);
```

Agenda

- Background
- The critical-section problem
- Synchronization hardware
- Semaphores
- Classical problems of synchronization
- Monitors
- Synchronization examples
- Atomic transaction

Monitors

- Motivation: semaphore is still too low-level tool.

Example)

correct

```
wait(mutex);
…
   critical section
…
signal(mutex);
```

wrong

```
signal(mutex);
…
   critical section
…
wait(mutex)
```

wrong

```
wait(mutex);
…
   critical section
…
wait(mutex);
```

- Monitor: a high-level language construct to support synchronization
  - Private data: accessible only through the methods
  - Public methods: mutual exclusion is provided

Monitors

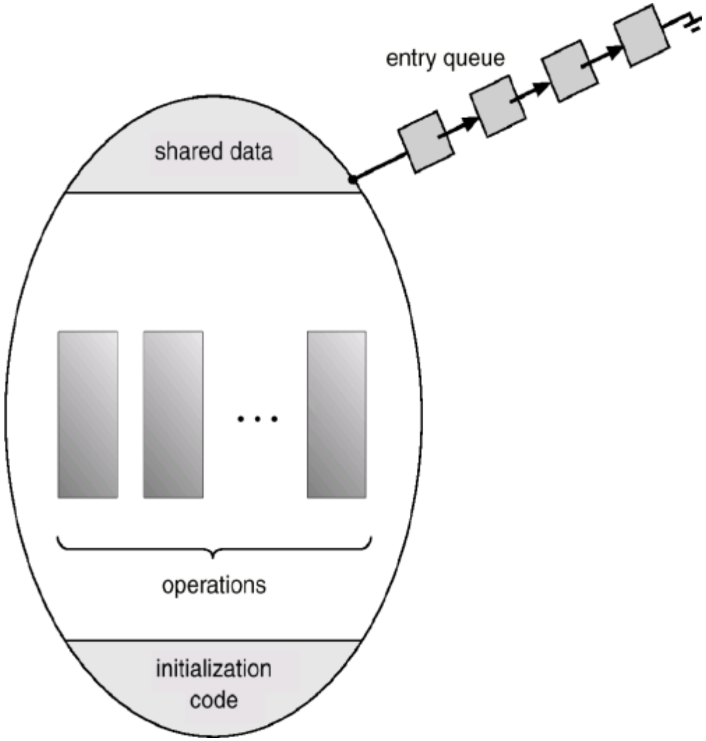
- Syntax of monitor

```
monitor monitor-name
{
    // shared variable declarations

    procedure body P1 (…) {
      …
    }
    procedure body P2 (…) {
      …
    }
    …
    procedure body Pn (…) {
      …
    }
    initialization code (…) {
      …
    }
}
```

Only one process can be active within the monitor at a time

Monitors

- Only one process can be active within the monitor at a time.
  - The programmer doesn't have to concern about synchronization.



Monitor in Java

- Synchronized method

Ex)

```
class Producer {
    private int product;
    …
    private synchronized void produce();      // mutually exclusive
}
```

Cf. C#: System.Threading.Monitor class

Condition Variables

- Condition: additional synchronization mechanism
  - Variable declaration

    `condition x, y;`

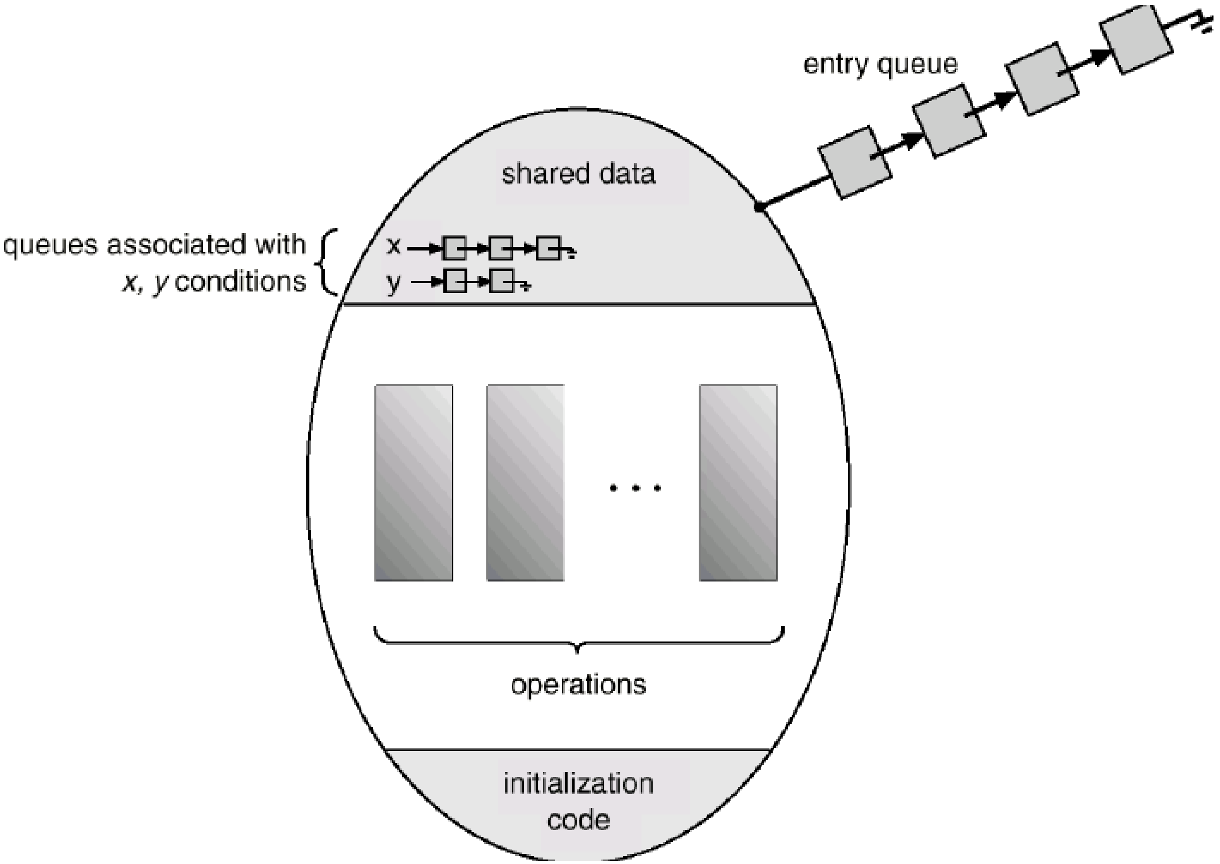
  - Wait

    ```
    x.wait();              // invoking process is suspended until
                           // other process call x.signal()
    ```

  - Signal

    ```
    x.signal();            // resumes exactly one suspended process
                           // if no process is waiting, do nothing
    ```

Condition Variables



Condition Variables

- Problem
  - Process P wakes up another process Q by invoking x.signal().
  - Both P and Q are executing in monitor.

- Solution
  - Signal and wait: P waits until Q leaves monitor or waits
  - Signal and continue: Q waits until P leaves monitor or waits

Dining Philosophers Solution Using Monitor

- Data structures
  - enum { thinking, hungry, eating } state[5];
  - condition self[5];

- Process of i-th philosopher implemented by a monitor dp

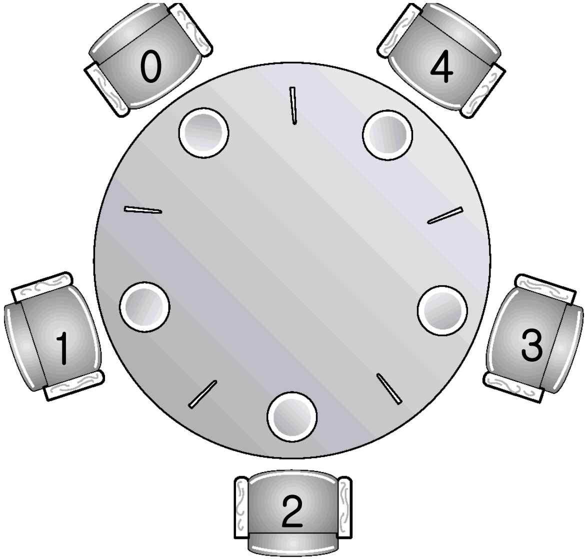
```
dp.pickup(i);                    // entry section
   …
   Eat                           // critical section
   …
dp.putdown(i);                   // exit section
```

Dining Philosophers Solution Using Monitor

```
monitor diningPhilosophers {
     int state[5];
     static final int THINKING = 0;
     static final int HUNGRY = 1;
     static final int EATING = 2;
     condition self[5];

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

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

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

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

Deadlock-free, but not starvation-free

Dining Philosophers Solution Using Monitor



Implementing a Monitor Using Semaphores

- Functions to implement
  - External procedures
  - wait() of condition variable
  - signal() of condition variable

- Two semaphores are required:
  - semaphore *mutex*; // Mutual exclusiveness
  - semaphore *next*; // signaling process should wait // until the resumed process leaves monitor
  - int next_count = 0;

Implementing a Monitor Using Semaphores

- External procedure F

```
wait(mutex);
    …
    body of $F$;
    …
signal(mutex);
```

Implementing a Monitor Using Semaphores

- Condition variables
  - Required data
    - semaphore x_sem; // (initially = 0)
    - int x_count = 0;
- x.wait()

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

- x.signal()

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

Agenda

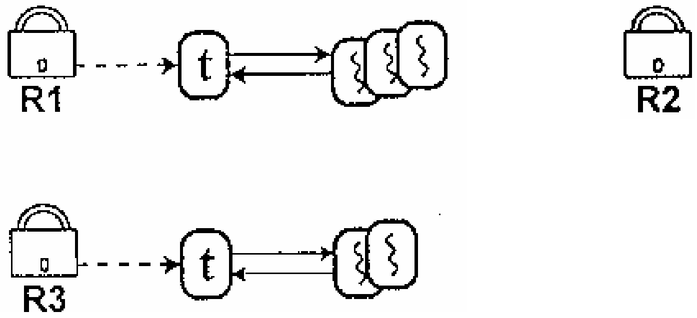
- Background
- The critical-section problem
- Synchronization hardware
- Semaphores
- Classical problems of synchronization
- Monitors
- Synchronization examples
- Atomic transaction

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthread

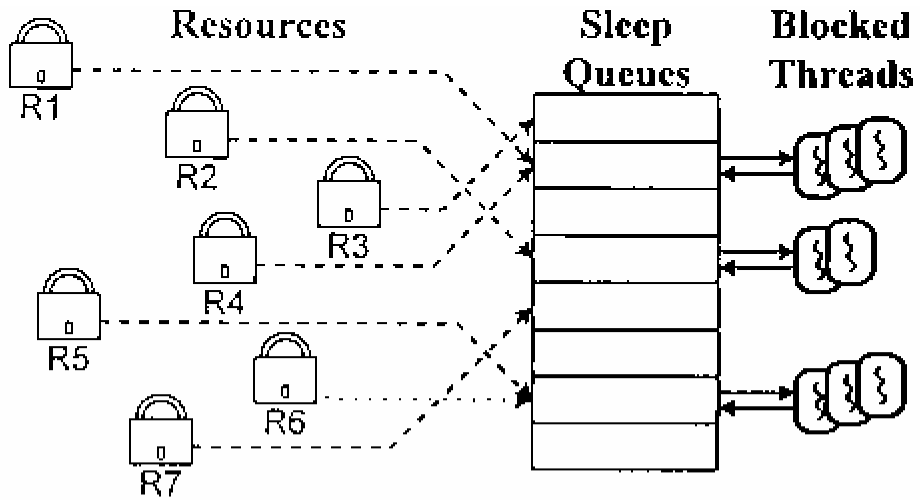
Synchronization in Solaris

- Adaptive mutexes
  - lock is held by a thread running on other CPU -> spin
  - lock is held by a thread not running -> sleep
- Condition variables
- Semaphores
- Reader-writer locks
- Turnstiles
  - A queue structure containing threads blocked on a lock

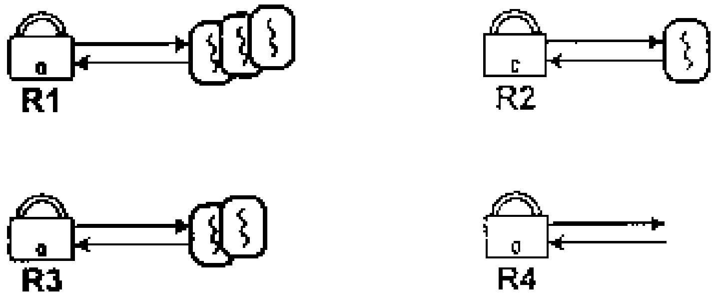


Sleeping Queue in Traditional UNIX

- Global sleeping queue
  - Inefficiency



- Sleeping queue for each resource
  - Wasting memory



Synchronization in Windows XP

- Uses *interrupt masks* to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Also provides *dispatcher objects* which may act as with ***mutexes*** and ***semaphores***.
- Dispatcher objects may also provide *events.* An event acts much like a *condition variable*.

Synchronization in Linux

- From v.2.6., Linux kernel is fully preemptive
  - preempt_disable()/preempt_enable()

Single processor	Multiprocessor
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

Synchronization in Pthread

- Mutex locks
- Condition variables
- Read-write locks

Agenda

- Background
- The critical-section problem
- Synchronization hardware
- Semaphores
- Classical problems of synchronization
- Monitors
- Synchronization examples
- <u>Atomic transaction</u>

Atomic Transactions

- Collection of instructions that performs a single logical function
  - Major issue: preservation of atomicity
- Result of a transaction may be either
  - Committed: successfully executed
  - Aborted: otherwise
    - If the system state was already modified, it should roll back
- Problem: How to ensure atomicity ?

Log-Based Recovery

- Recode information describing all modification
  - Transaction name
  - Data item name
  - Old value
  - New value

- Recovery algorithm
  - undo(T_i)
  - redo(T_i)

Checkpoints

- Motivation: searching log and modification is time-consuming

- Checkpoint: save all modified data to more stable storage
  - Output all log records in volatile storage onto stable storage
  - Output all modified data in volatile storage to stable storage
  - Output a log record <checkpoint> onto stable storage