Principles of Operating Systems

Lecture 4 - Process Synchronization

Ardalan Amiri Sani (ardalan@uci.edu)

[*lecture slides contains some content adapted from previous slides by Prof. Nalini Venkatasubramanian, and course text slides © Silberschatz, and Anderson textbook slides]*

Producer-Consumer Problem

- Paradigm for cooperating processes;
  - producer process produces information that is consumed by a consumer process.
- We need a buffer of items that can be filled by producer and emptied by consumer.
  - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
  - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
- Producer and Consumer must synchronize.

Bounded Buffer - message passing

Producer

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}
```

Consumer

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Bounded Buffer - Shared Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded Buffer - Shared Memory Solution: producer

```
item next_produced;
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer - Shared Memory Solution: consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;


    /* consume the item in next consumed */
}
```

Problem with this solution

- Shared memory solution to the bounded-buffer problem allows at most (n-1) items in the buffer at the same time.

Solution

- A solution that uses all N buffers is not that simple.
  - Modify producer-consumer code by adding a variable *counter*, initialized to 0, incremented each time a new item is added to the buffer
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded Buffer - Shared Memory Solution: producer

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

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

Bounded Buffer - Shared Memory Solution: consumer

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

Problem with this solution?

```
item next_produced;
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

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

        counter--;
    /* consume the item in next
consumed */
}
```

Consumer

Problem with this solution

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Access to shared data

- The statements

```
counter++;
counter--;
```

  must be executed *atomically*.
- Atomic operations
  - An indivisible operation that runs to completion without interruptions by other operations.

Race Condition

- **counter++** could be implemented with the following instructions in CPU:

  register1 = counter
  register1 = register1 + 1
  counter = register1

- **counter--** could be implemented with the following instructions in CPU:

  register2 = counter
  register2 = register2 - 1
  counter = register2

- Consider this execution interleaving with "count = 5" initially (**we expect count = 5 in the end too**):

  S0: producer execute **register1 = counter** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = counter** {register2 = 5}
  S3: consumer execute **register2 = register2 – 1** {register2 = 4}
  S4: producer execute **counter = register1** {counter = 6 }
  S5: consumer execute **counter = register2** {counter = 4 !!}

Race Condition

- If processes are working on separate data, we don't need to worry about race conditions:

Process A	Process B
x = 1;	y = 2;

- However, there can race conditions when we have shared data. Consider the following (Initially, y = 12):

Process A	Process B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?

Race Condition

- If processes are working on separate data, we don't need to worry about race conditions:

Process A	Process B
x = 1;	y = 2;

- However, there can race conditions when we have shared data. Consider the following (Initially, y = 12):

Process A	Process B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x? Answer = (13, 3, 5)
- Or, what are the possible values of x below?

Process A	Process B
x = 1;	x = 2;

  - What are the possible values of x?

Race Condition

- If processes are working on separate data, we don't need to worry about race conditions:

Process A	Process B
x = 1;	y = 2;

- However, there can race conditions when we have shared data. Consider the following (Initially, y = 12):

Process A	Process B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x? Answer = (13, 3, 5)
- Or, what are the possible values of x below?

Process A	Process B
x = 1;	x = 2;

  - What are the possible values of x? Answer = (1, 2)
- X's value is non-deterministic in the past two examples

Note

- We use processes with some shared memory for the discussions in this lecture
- We also assume the processes to be single-threaded, hence use “process” instead of “thread”
- Our discussions however apply to threads within the same process as well since these thread share all the process address space

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}} competing to access shared data
- Each process has a **critical section** segment of code
  - Process may be changing shared variables, updating shared table, writing to shared file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to achieve/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

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

Critical Section Problem - Requirements

- **Mutual Exclusion**
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**
  - If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- **Bounded Waiting**
  - A bound must exist on the 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 that request is granted.

Critical Section Problem - Assumptions

- Assume that each process executes at a nonzero speed in the critical section. That is, assume that each process finishes executing the critical section once entered
- No assumption concerning relative speed of the n processes.
- Assume that a process can get stuck in its remainder section indefinitely, e.g., in a non-terminating while loop.

Solution: Critical Section Problem -- Initial Attempt

- Only 2 processes, Pi and Pj
- General structure of process Pi (Pj)

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

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared Variable:
  - **int turn = i**;
  - (turn == i) means that P_i can enter its critical section and (turn == j) means that P_j can enter its critical section
- Process Pi

```
do {
    while (turn == j);

        critical section

    turn = j;

        remainder section

} while (true);
```

Process Pj

```
do {
    while (turn == i);

        critical section

    turn = i;

        remainder section

} while (true);
```

Algorithm 1

- Satisfies mutual exclusion
  - The turn is equal to either i or j and hence one of Pi and Pj can enter the critical section
- Does not satisfy progress
  - Example: Pi finishes the critical section and then gets stuck indefinitely in its remainder section. Then Pj enters the critical section, finishes, and then finishes its remainder section. Pj then tries to enter the critical section again, but it cannot since turn was set to i by Pj in the previous iteration. Since Pi is stuck in the remainder section, turn will be equal to i indefinitely and Pj can't enter although it wants to. Hence no process is in the critical section and hence no progress.
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 2

- Shared Variables
  - **boolean** flag[2];
    flag[i] = false; flag[j] = false;
  - (flag[i] == true) means that Pi ready to enter its critical section
  - (flag[j] == true) means that Pj ready to enter its critical section
- Process Pi (Replace i with j and j with i for Pj)

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

Algorithm 2

- Satisfies mutual exclusion
  - If Pi enters, then flag[i] = true, and hence Pj will not enter.
- Does not satisfy progress
  - Example: There can be an interleaving of execution in which Pi and Pj both first set their flags to true and then both check the other process' flag. Therefore, both get stuck at the entry section
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 3

- Shared Variables
  - **boolean** flag[2];
    flag[i] = false; flag[j] = false;
  - (flag[i] == true) means that Pi ready to enter its critical section
  - (flag[j] == true) means that Pj ready to enter its critical section
- Process Pi (Replace i with j and j with i for Pj)

```
do {
        while (flag[j]);
        flag[i] = true;

            critical section

        flag[i] = false;

            remainder section
  } while (true);
```

Algorithm 3

- Does not satisfies mutual exclusion
  - Example: There can be an interleaving of execution in which both first check the other process' flag and see that it is false. Then they both enter the critical section.
- We don't need to discuss/consider progress and bounded wait when mutual exclusion is not satisfied

Algorithm 4

- Shared Variable:
  - **int turn = i**;
  - (turn == i) means that Pi can enter its critical section and (turn == j) means that Pj can enter its critical section
  - **boolean** flag[2];

    flag[i] = false; flag[j] = false;
  - (flag[i] == true) means that Pi ready to enter its critical section
  - (flag[j] == true) means that Pj ready to enter its critical section
- Process Pi (Replace i with j and j with i for Pj)

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

Algorithm 4

- Satisfies mutual exclusion
  - If one process enters the critical section, it means that either the other process was not ready to enter or it was this process' turn to enter. In either case, the other process will not enter the critical section
- Satisfies progress
  - If one process exits the critical section, it sets its ready flag to false and hence the other process can enter. Moreover, there is no interleaving in the entry section that can block both.
- Satisfies bounded wait
  - If a process is waiting in the entry section, it will be able to enter at some point since the other process will either set its ready flag to false or will set the turn to this process.

Algorithm 4

- Meets all three requirements, solves the critical section problem for 2 processes.
- This is called the “Peterson’s solution”.

Bakery Algorithm

- Critical section for n processes
  - Before entering its critical section, process receives a number. Holder of the smallest number enters critical section.
  - If processes Pi and Pj receive the same number,
    - if i < j, then Pi is served first; else Pj is served first (note that i and j cannot be the same numbers as process ID is unique).
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e. 1,2,3,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

- Notation -
  - Lexicographic order(ticket#, process id#)
    - $(a,b) < (c,d)$ if $(a<c)$ or if $((a=c)$ and $(b < d))$
    - $\max(a_0,\ldots a_{n-1})$ is a number, k, such that $k >= a_i$ for $i = 0,\ldots,n-1$
- Shared Data

  boolean choosing[n]; (all items initialized to false)

  int number[n]; (all initialized to 0)

Bakery Algorithm (cont.)

```
do {
    choosing[i] = true;
    number[i] := max(number[0], number[1],…,number[n-1]) + 1;
    choosing[i] = false;
    for (int j = 0; j < n, j++) {
        while (choosing[j]);
        while ((number[j] != 0) &&
                    ((number[j], j) < (number[i],i));
    }
        critical section
    number[i] = 0;
        remainder section
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software abstractions to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not

Solution to Critical-section Problem Using Locks

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

Synchronization Hardware

- Software-based solutions such as Peterson's solution and Bakery algorithm are not guaranteed to work on modern computer architectures due to how they perform basic machine-language instructions (e.g., out-of-order execution)
- Many systems provide hardware support for implementing the critical section.
- Our goal is to use synchronization hardware to implement **lock**
  - Key idea: Use atomic execution to implement acquire() and release()

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can ?

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work?

```
acquire()
    { disable interrupts }
release()
    { enable interrupts }
```

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work? Yes, but inefficient as no other process can run as long as one process is in its critical section.

```
acquire()
    { disable interrupts }
release()
    { enable interrupts }
```

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work?

```
acquire()
   {   disable interrupts;
       if (!lock) {lock=true;}
       enable interrupts;}
release()
   { disable interrupts;
     lock=false;
     enable interrupts;}
```

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work? No, does not satisfy mutual exclusion. Even if locked, the process still enters the critical section!

```
acquire()
   {   disable interrupts;
       if (!lock) {lock=true;}
       enable interrupts;}
release()
   { disable interrupts;
     lock=false;
     enable interrupts;}
```

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work?

```
acquire()
   {   disable interrupts;
       while (lock);
       lock=true;
       enable interrupts;}
release()
   { disable interrupts;
     lock=false;
     enable interrupts;}
```

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work? No, does not satisfy progress. If locked, the process will keep checking the lock forever and no other process can ever run!

```
acquire()
   {   disable interrupts;
       while (lock);
       lock=true;
       enable interrupts;}
release()
   { disable interrupts;
     lock=false;
     enable interrupts;}
```

Atomic execution

- We need atomic execution
  - **Atomic =** non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work?

```
acquire()
   {acquired = false;
    while (!acquired) {
      disable interrupts;
       if (!lock) {lock=true; acquired = true;}
       enable interrupts;}}
release()
   { disable interrupts;
     lock=false;
     enable interrupts;}
```

Atomic execution

- We need atomic execution
  - **Atomic =** non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will this work? Yes!

```
acquire()
   {acquired = false;
    while (!acquired) {
      disable interrupts;
       if (!lock) {lock=true; acquired = true;}
       enable interrupts;}}
release()
   { disable interrupts;
     lock=false;
     enable interrupts;}
```

Project note

- Pintos uses interrupt disabling for synchronization

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will it work on multiprocessor systems?

Atomic execution

- We need atomic execution
  - **Atomic** = non-interruptible and non-overlapping, as seen by others
- To do this, on uniprocessors, we can disable interrupts
  - Currently running code would execute without preemption
  - Will it work on multiprocessor systems?
  - Generally too inefficient on multiprocessor systems
    - Operating systems might use this but not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Either test memory word and set value
  - Or compare and swap contents of two memory words

test_and_set Instruction

Definition:

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

1. Executes atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

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);
```

Solution using test_and_set()

- Satisfies mutual exclusion
  - Only one process can hold the lock at any given time and hence only process can be in the critical section at any given time.
- Satisfies progress
  - If one process holds the lock and one process tries to acquire it, it will succeed and then can enter the critical section.
- Does not satisfy bounded wait
  - A process can end up trying to acquire the lock with no luck indefinitely.

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;
}
```

1. Executes atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” to be the value of the passed parameter “new_value” if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

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

Bounded-waiting critical section implementation with test_and_set

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

Spinlock

- The lock implementations we have seen using atomic instructions require **busy waiting**
- This lock therefore called a **spinlock**

Semantics of acquire() and release() in a spinlock

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

Problem...

- Synchronization involves waiting
  - **Busy Waiting,** uses CPU that others could use.
    - Waiting thread may take cycles away from thread holding lock (no one wins!)
    - OK for short times since it prevents a context switch.
    - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ problem!
  - Should *sleep* if waiting for a long time (in the next couple of slides, we will see one such lock implementation using semaphores)

Project note: timer_sleep()

- Better to call it timer_wait()
- Could be implemented by either busy waiting or sleeping
- Sleeping: be suspended/blocked until an event of interest happens

Semaphore

- Semaphore *S* - integer variable (non-negative)
  - used to represent number of abstract resources
- Can only be accessed via two atomic operations

  wait (*S*): **while** (*S* <= 0);
  S--;

  signal (S): *S*++;
  - *P* or *wait* used to acquire a resource, waits for semaphore to become positive, then decrements it by 1
  - *V* or *signal* releases a resource and increments the semaphore by 1, waking up a waiting P, if any
  - If *P* is performed on a *count* <= 0, process must wait for *V* or the release of a resource.

P():"*proberen*" (to test) ; V() "*verhogen*" (to increment) in Dutch

Example: Critical Section for n Processes

- Shared variables

  semaphore mutex;
  initially mutex = 1

- Process *Pi*

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

Semaphore as a General Synchronization Tool

- Execute *B* in *Pj* only after *A* execute in *Pi*
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Question: Is the mutex using semaphores a spinlock?

- Shared variables

  semaphore mutex;
  initially mutex = 1

- Process *Pi*

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

Question: Is the mutex using semaphores a spinlock? Depends on the implementation of the semaphore

- Shared variables

  semaphore mutex;
  initially mutex = 1

- Process *Pi*

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

Semaphore Implementation with no Busy waiting

We need to implement wait() and signal() in a way that allows processes to *block* and *resume.*

- Solution:
- With each semaphore there is an associated waiting queue
- 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
- ```
  typedef struct{
      int value;
      struct process *list;
  } semaphore;
  ```
  ```

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

A mutex using this semaphore is not a spinlock. It can sleep.

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 semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal (*S*) ;	*signal* (*Q*);
signal (*Q*);	*signal* (*S*);

- Deadlock results in starvation: indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Producer-Consumer with Bounded Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem

Producer-Consumer with Bounded Buffer Problem

- Shared data
  - ***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

- Producer process - creates filled buffers

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

Bounded Buffer Problem

- Consumer process - Empties filled buffers

```
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);
```

Discussion

- ASymmetry
  - Producer does: P(empty), V(full)
  - Consumer does: P(full), V(empty)

Discussion

- ASymmetry
  - Producer does: P(empty), V(full)
  - Consumer does: P(full), V(empty)
- Is order of P's important between P(mutex) and P(empty or full)?

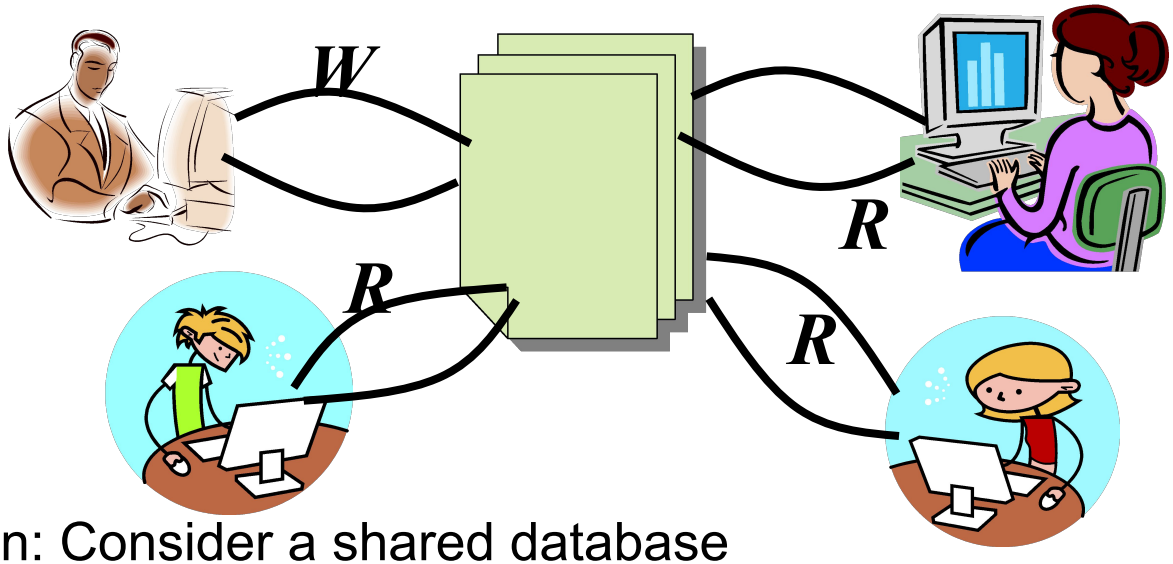
Discussion

- ASymmetry
  - Producer does: P(empty), V(full)
  - Consumer does: P(full), V(empty)
- Is order of P's important between P(mutex) and P(empty or full)?
  - Yes! Can cause deadlock if reordered
- Is order of V's important?

Discussion

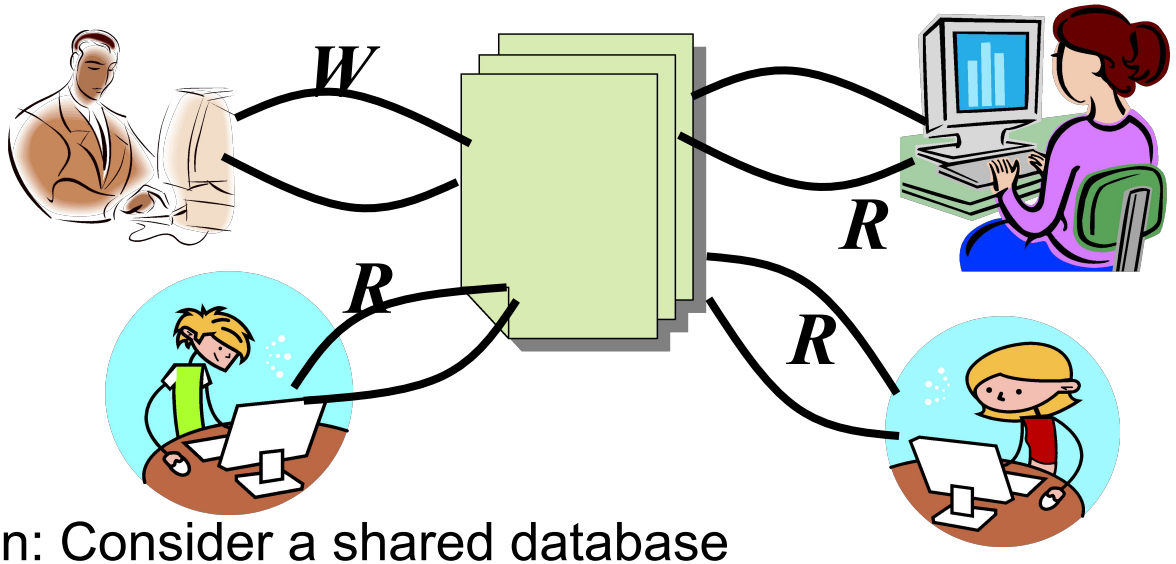
- ASymmetry
  - Producer does: P(empty), V(full)
  - Consumer does: P(full), V(empty)
- Is order of P's important between P(mutex) and P(empty or full)?
  - Yes! Can cause deadlock if reordered
- Is order of V's important?
  - No, except that it might affect scheduling efficiency

Readers-Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?

Readers-Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - We'd like to have many readers at the same time
    - Only one writer at a time

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0
- The structure of a writer process

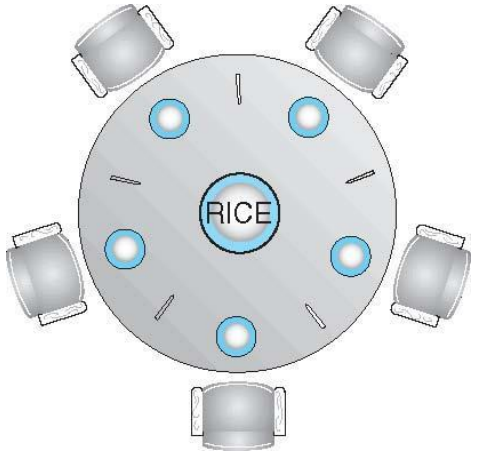
```
do {
     wait(rw_mutex);
          ...
        /* writing is performed */
          ...
     signal(rw_mutex);
} while (true);
```

Readers-Writers Problem

- 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);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from their bowls
  - 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

- 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);
```

Dining Philosophers Problem

- 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?

Dining Philosophers Problem

- 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? Can result in a deadlock

Dining Philosophers Problem

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations can result in deadlock and/or starvation:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

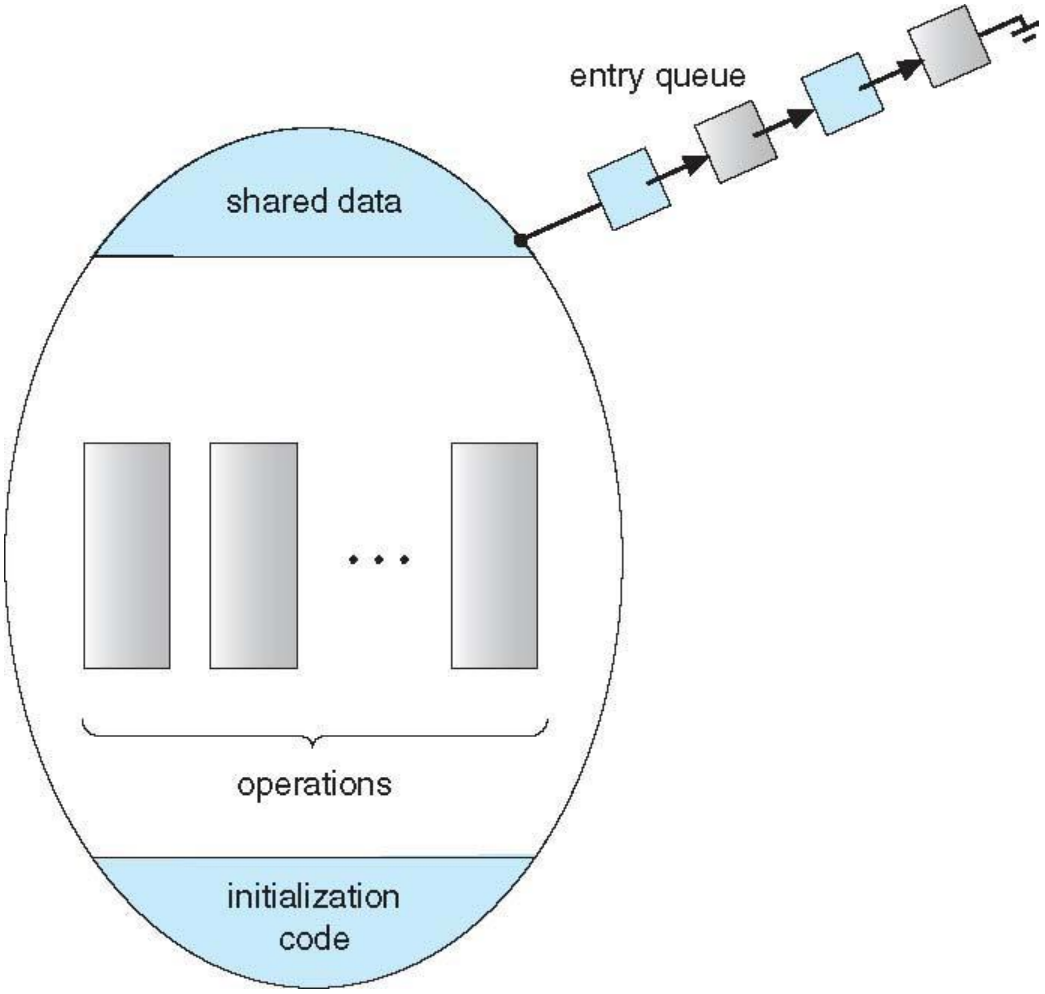
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization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

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

  procedure Pn (…) {……}

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

Schematic view of a Monitor



Monitor Solution to Dining Philosophers

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

   int pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          If (state[i] == EATING) {
              return 0;
           } else {
              return 1;
           }
   }

   void putdown (int i) {
           state[i] = THINKING;
   }
```

Solution to Dining Philosophers (Cont.)

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

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

Solution to Dining Philosophers (Cont.)

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

```
While (DiningPhilosophers.pickup(i));

        EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

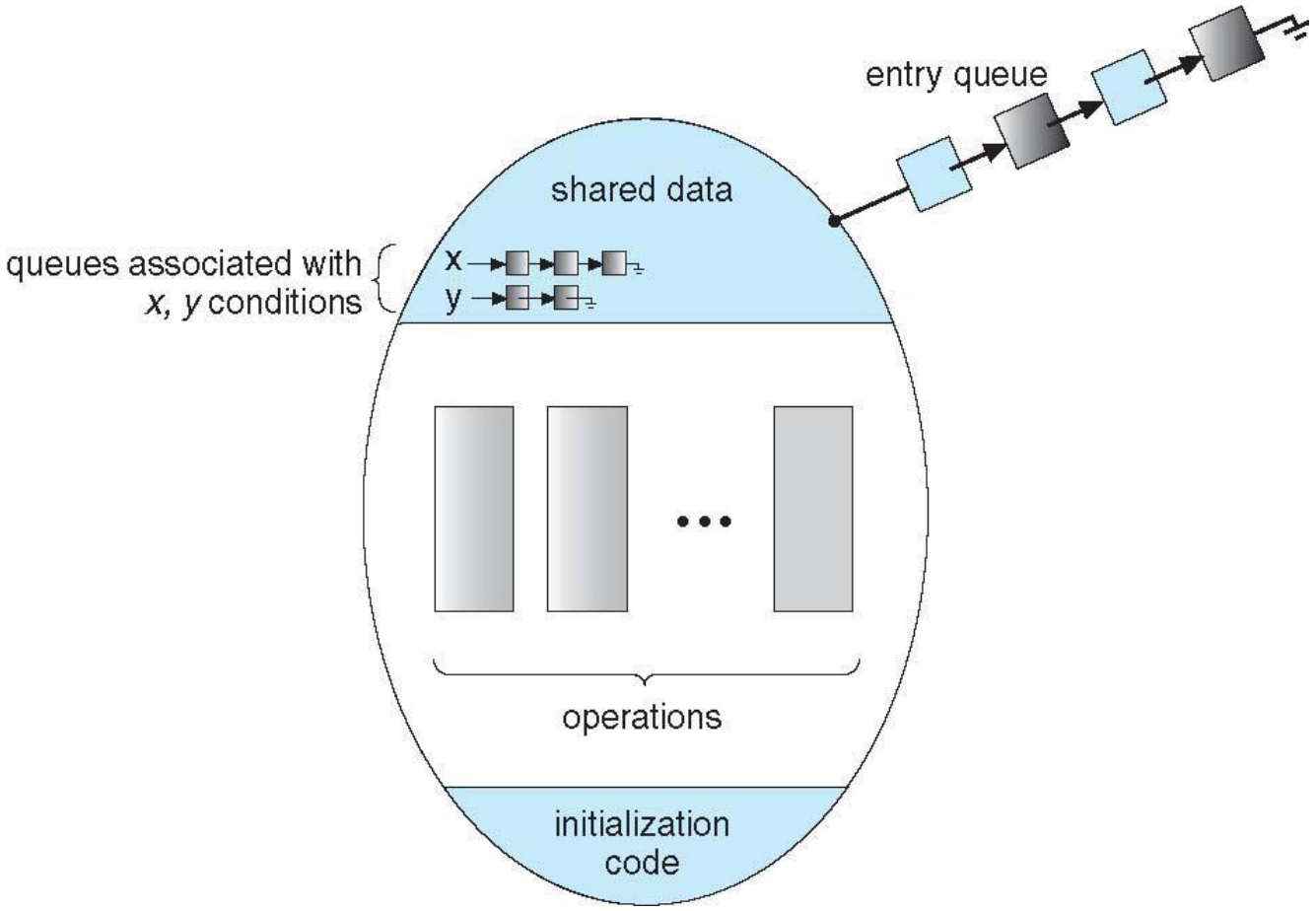
What’s wrong with the previous solution?

- Busy waiting! The philosophers keep checking to see if they can eat or not.

Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Monitor Solution to Dining Philosophers without busy waiting

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

   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 neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
    }
```

Monitor Solution to Dining Philosophers without busy waiting (Cont.)

```
    void test (int i) {
            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;
        }
}
```

Monitor Solution to Dining Philosophers without busy waiting (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

Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel in the monitor. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – monitor implementer can decide