운영체제의 기초: Process Synchronization

2023년 4월 13, 18, 20일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도교수
서울대학교 전기정보공학부 교수

Agenda

I. Why Process Synchronization?
II. Semaphore
III. Condition Variable
IV. Monitor

I. Why Process Synchronization?

Why Process Synchronization? (1)

- Processes interact with each other for good
- Why permit processes to cooperate?
  - Want to share resources
    - One computer, many users
    - One checking account file, many tellers
  - Want to do things faster
    - Read next block while processing current one
    - Divide jobs into sub-jobs, execute in parallel
  - Want to construct systems in modular fashion
    - UNIX example: **`tbl | eqn | troff`**

Why Process Synchronization? (2)

- Properties of interacting processes
  - Have shared resources and states
  - Non-deterministic
    - Outputs may vary depending execution ordering of processes
  - Their behavior is maybe irreproducible
    - Can’t stop and restart with no bad effects

Uncontrolled Task Interactions (1)

- Data sharing problem instance
  - Interrupt routines and task code may share one or more variables that they can use to communicate with each other
  - This may cause a data sharing problem – a sort of synchronization problem
  - Such a task is referred to as non-reentrant code
  - Example: nuclear reactor monitoring system

Uncontrolled Task Interactions (2)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void) {
   iTemperatures[0] = !! Read in value from hardware
   iTemperatures[1] = !! Read in value from hardware
}
void main (void){
   int iTemp0, iTemp1;

   while(TRUE){
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        if (iTemp0 != iTemp1)
             !! set off howling alarm;
   }
}
```

Uncontrolled Task Interactions (3)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void) {
   iTemperatures[0] = !! Read in value from hardware
   iTemperatures[1] = !! Read in value from hardware
}
void main (void){
   int iTemp0, iTemp;

   while(TRUE){
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        if (iTemp0 != iTemp1)
              !! set off howling alarm;
   }
}
```

If interrupt occurs between these two statements, iTemp0 and iTemp1 will differ and the system will set off the alarm, even though the two measured temperatures were always the same.

Uncontrolled Task Interactions (4)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void){
   iTemperatures[0] = !! Read in value from hardware
   iTemperatures[1] = !! Read in value from hardware
}

void main (void){
   while (TRUE){
        if (iTemperatures[0] != iTemperatures[1])
              !! set off howling alarm;
   }
}
```

The same bug as in previous page!

The problem is that the statement that compares iTemperatures[0] with iTemperatures[1] can be interrupted.

Uncontrolled Task Interactions (5)

- Solution
  - Disable interrupts

```
void main (void){
   int iTemp0, iTemp;

   while(TRUE){
        disable(); /*Disable interrupts using array*/
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        enable();
        if (iTemp0 != iTemp1)
              !! set off howling alarm;
   }
}
```

Task Reentrancy (1)

- To handle cooperating processes, we need the notion of non-interruptible operations
  - The operation cannot be interrupted in the middle
  - Examples:

    ```
    int A,B; A = B; // On most systems
    ```

    - On uniprocessors, code between interrupts
    - Test-and-set instruction in some architectures

- To provide non-interruptible operations, we need some hardware support

Task Reentrancy (2)

- Synchronization
  - Using atomic operations to ensure correct operation of interacting processes
  - Example of interacting processes that need synchronization
    - Two processes execute the following code

```
if(BufferIsAvail) {
   BufferIsAvail = FALSE;
   UseBuffer();
   BufferIsAvail = TRUE;
}
```

Task Reentrancy (3)

Time	Proc 1	Proc 2
0	`if(BufferIsAvail)`	
1		`if(BufferIsAvail)`
2	`BufferIsAvail = FALSE`	
3		`BufferIsAvail = FALSE`
4	`UseBuffer();`	
5		`UseBuffer();`

Problem: Both processes issue `UseBuffer()`

Task Reentrancy (4)

- Lack of atomicity of “if” and “assignment”
- Mutual exclusion
  - Mechanisms which ensure that only one person or process is doing certain things at one time
- Critical section
  - A section of code, or collection of operations, in which only one process may be executing at a time

Task Reentrancy (5)

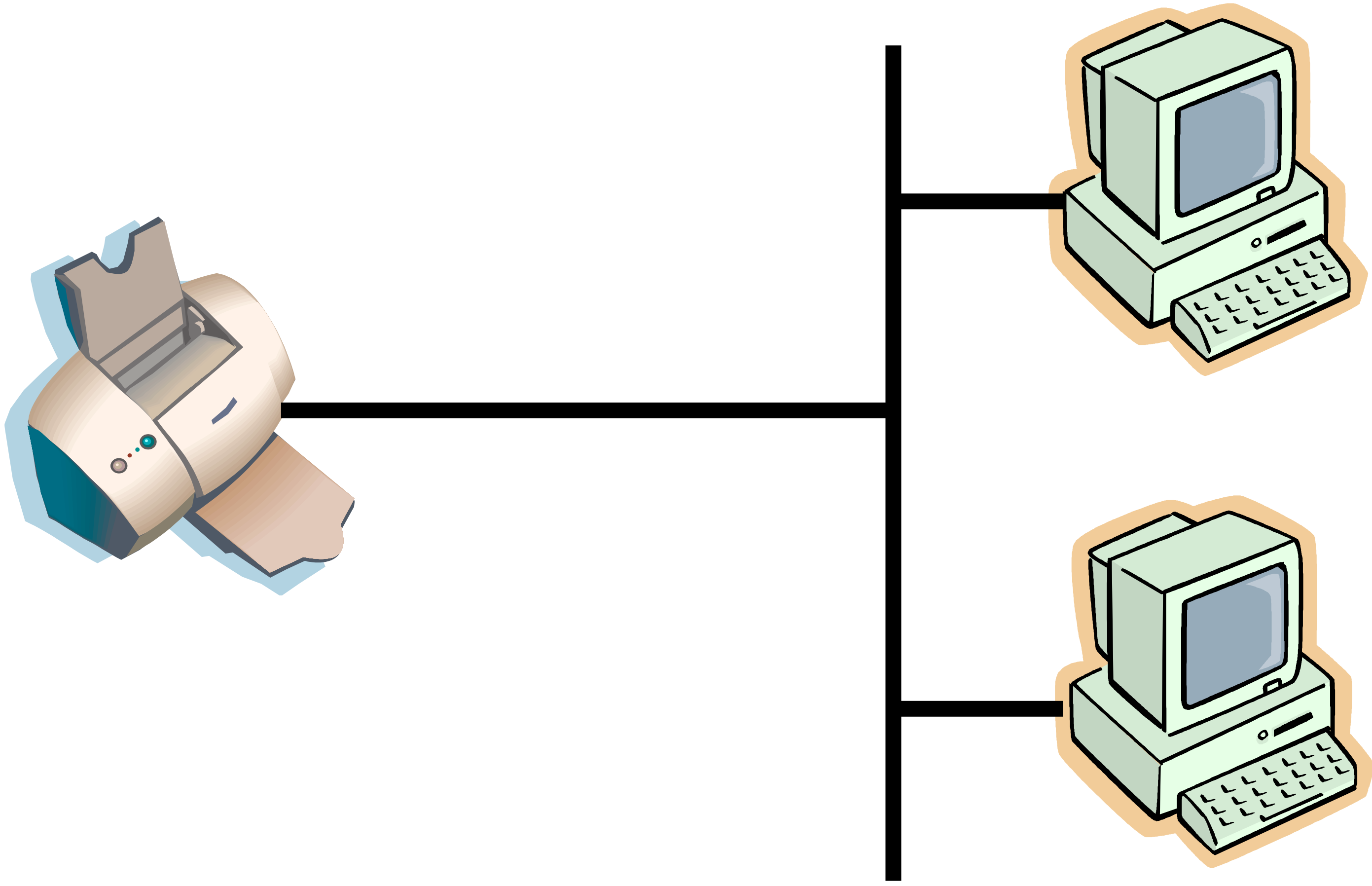
- Requirements for a mutual exclusion mechanism
  - Only one process is allowed in a critical section at a time
  - If several requests at once, it must allow one process to proceed
  - It must not depend on processes outside critical section

Task Reentrancy (6)

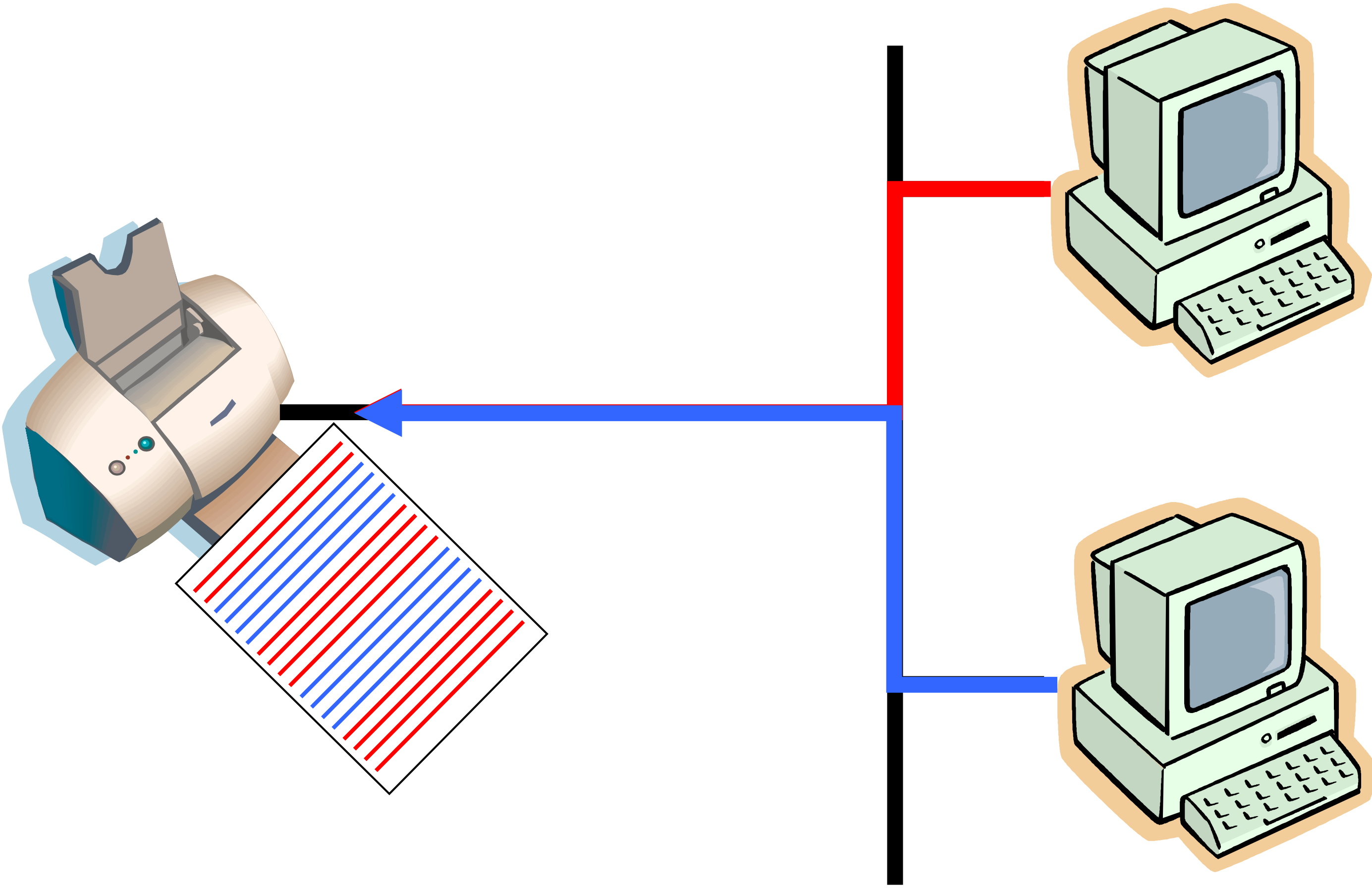
- Desirable properties for a mutual exclusion mechanism
  - Don’t make a process wait forever
  - Efficient
    - Don’t use up substantial amounts of resources when waiting
    - Example: busy waiting
  - Simple
    - Should be easy to use

II. Semaphore

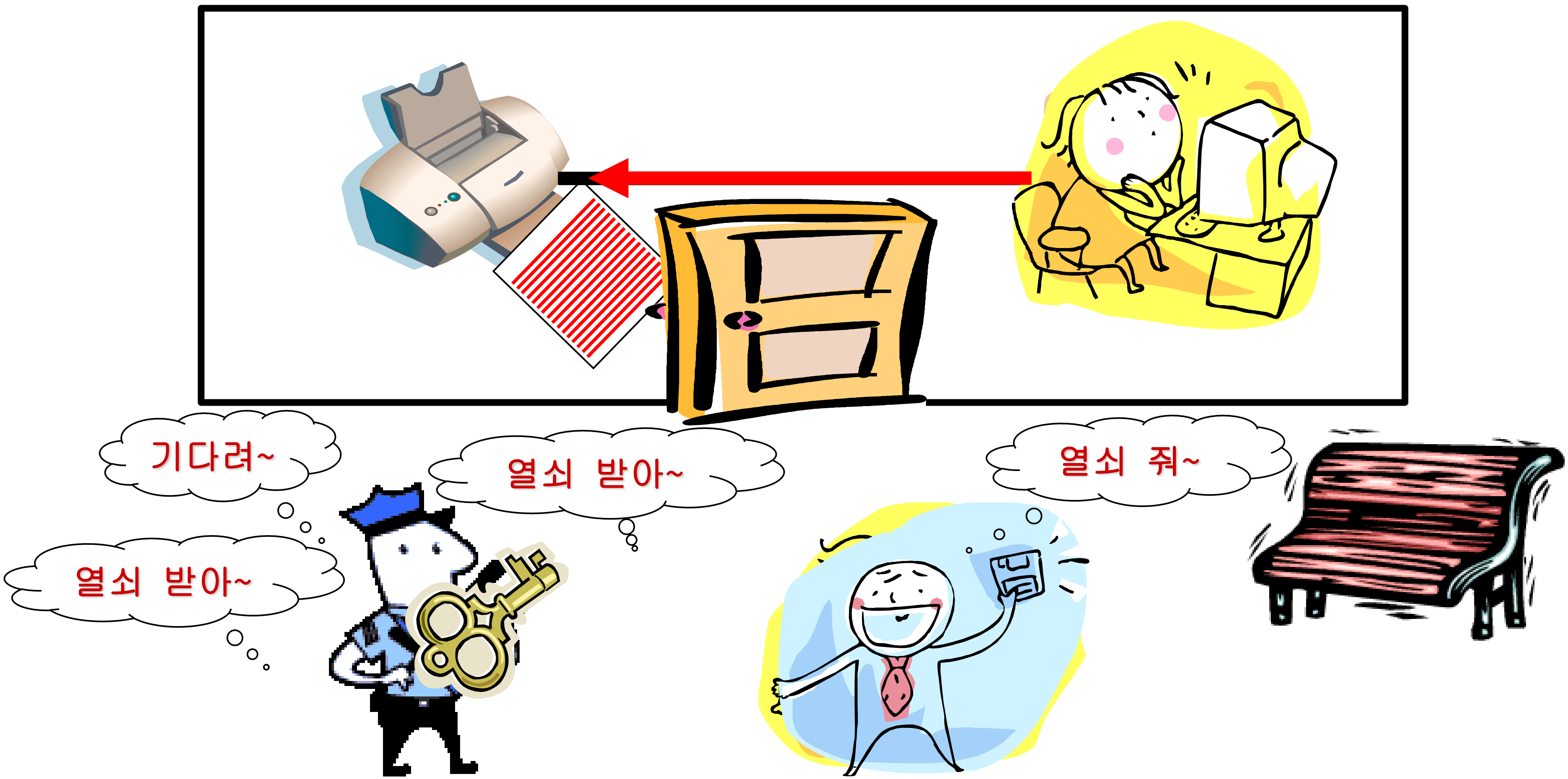
Motivations (1)



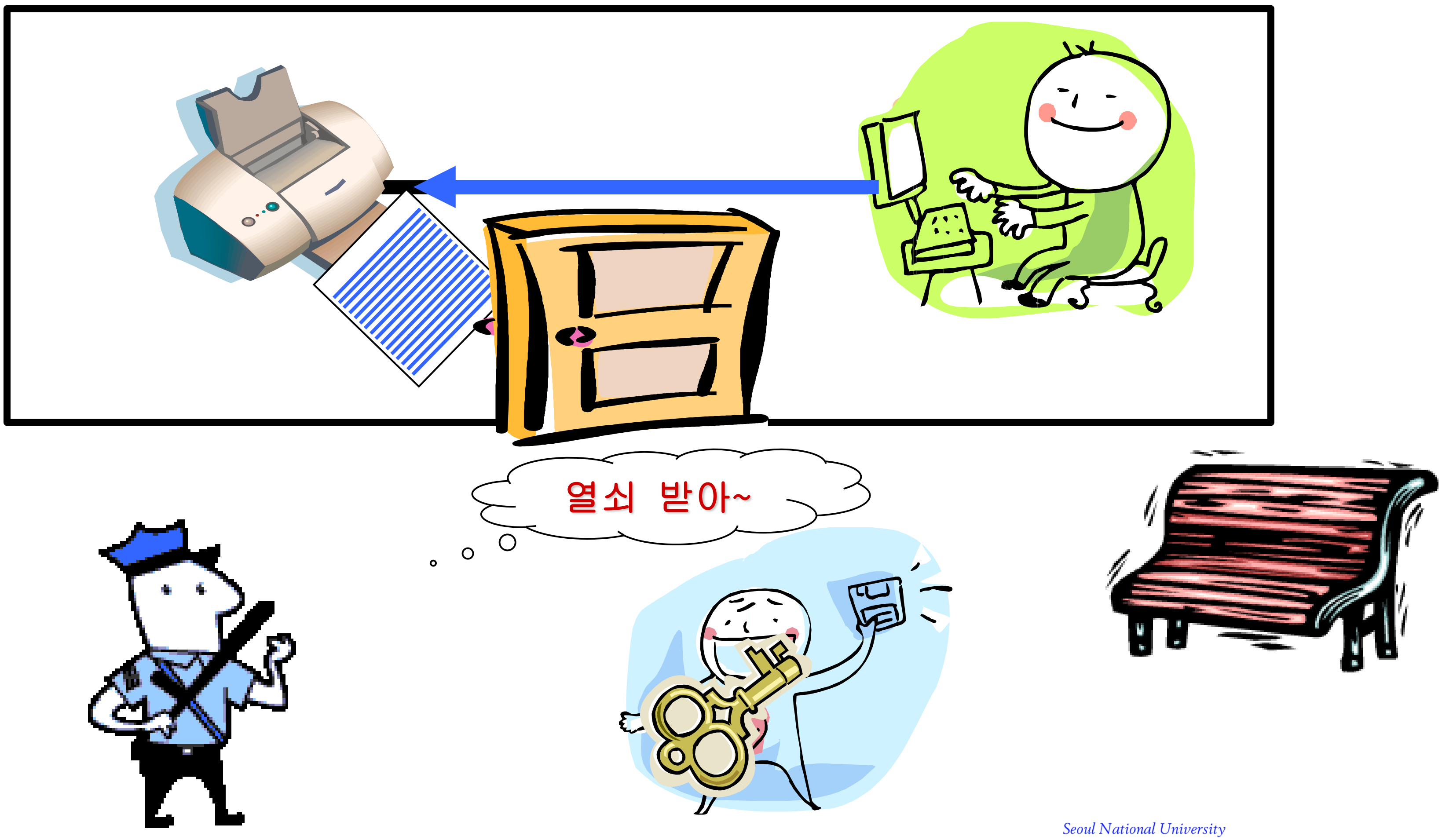
Motivations (2)



Operations (1)



Operations (2)



Basics

- One of key synchronization mechanisms
  - Synchronization variables that take on integer values
  - **`P(Semaphore)`**
    - An atomic operation that waits for semaphore to become positive and then decrements it by one
    - Also called wait()
  - **`V(Semaphore)`**
    - An atomic operation that increments semaphore by one
    - Also called signal()
  - They are simple and elegant
  - They do a lot more that just mutual exclusion

Usage

```
Task1(){

   P(S1)

      use pr;

   V(S1)

}
```

```
Task2(){

   P(S1)

      use pr;

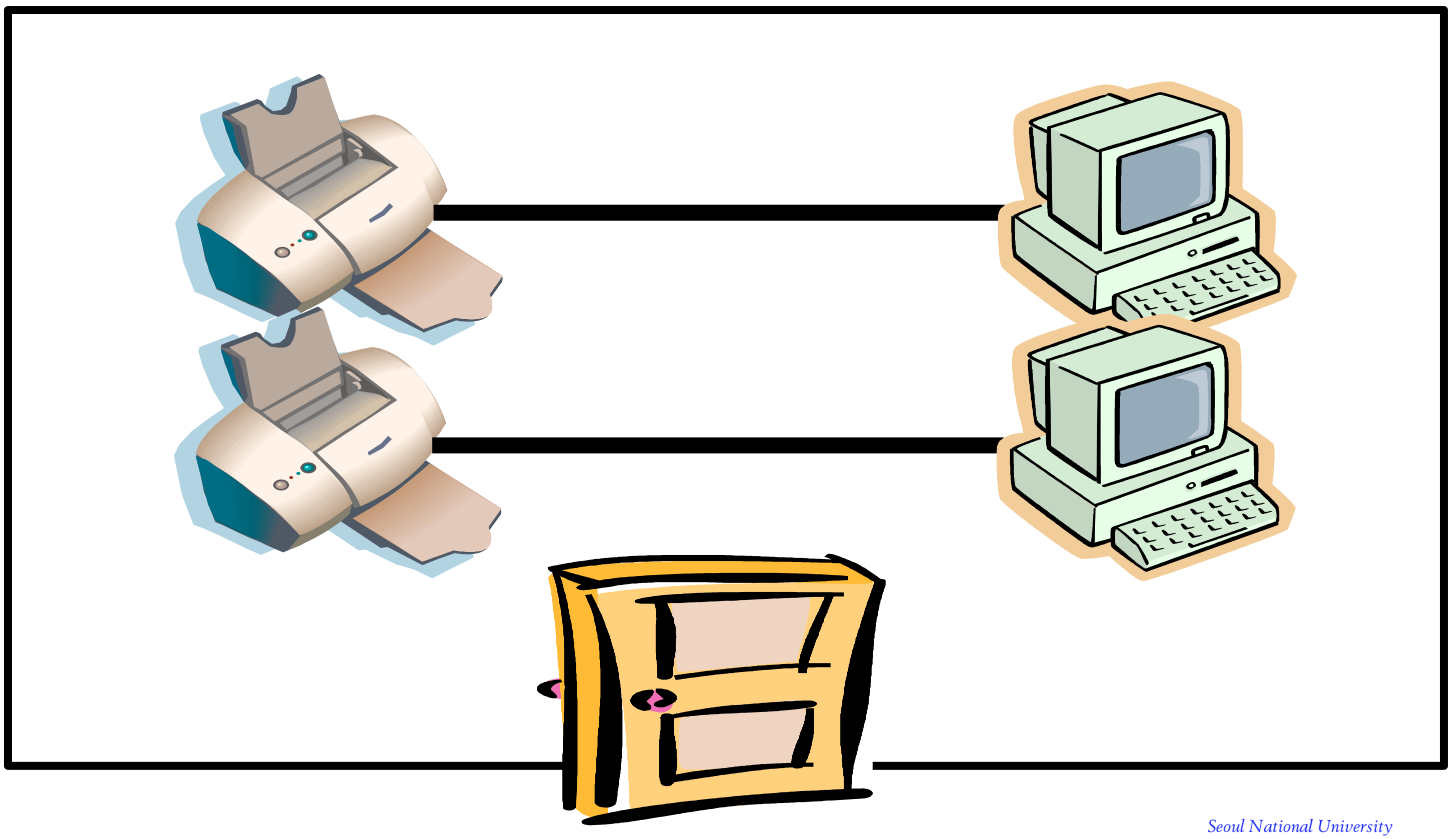
   V(S1)

}
```

```
semaphore S1 = 1;
```

Initialization (1)

- Binary semaphore versus counting semaphore



Initialization (2)

```
Task1(){
   P(S1)
      use pr;
   V(S1)
}
```

```
Task2(){
   P(S1)
      use pr;
   V(S1)
}
```

```
semaphore S1 = 2;
```

Basics

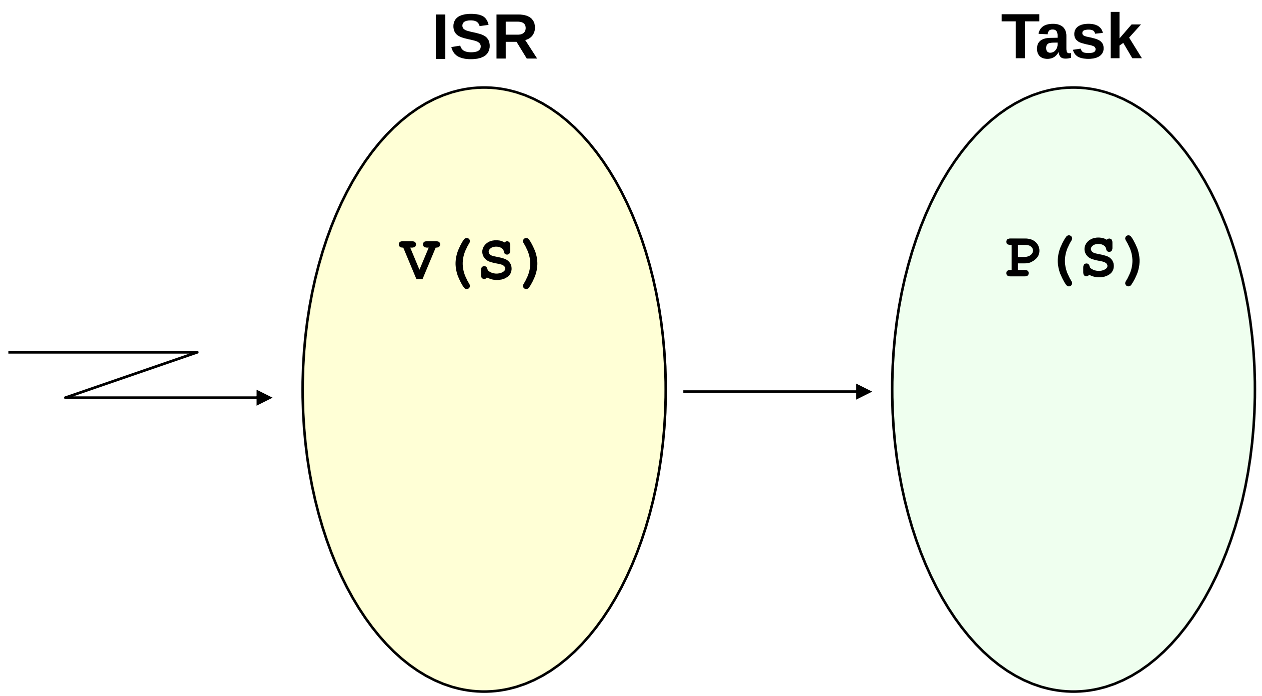
- Buffer example with semaphores:

```
P(BufferIsAvail);
 UseBuffer();
V(BufferIsAvail);
```

  - Note: **`BufferIsAvail`** must be set to one
  - What happens if **`BufferIsAvail`** is set to two? or zero?

- Roles of semaphores
  - Mutual exclusion
  - Scheduling

Scheduling

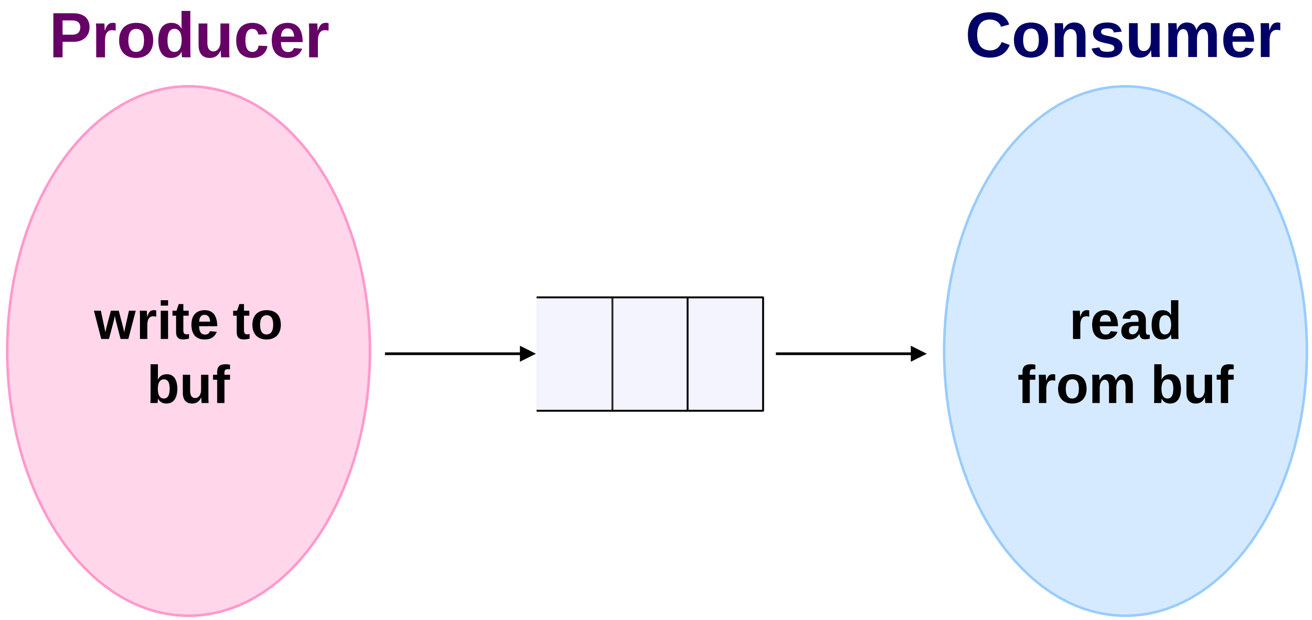



```
semaphore S = 0;
```

Producer/Consumer (1)

- Three key components
  - Producer: Creates copies of a resource
    - Example: User typing characters
  - Consumer: Uses up (destroys) copies of a resource
    - Example: Program reading users characters
  - Buffers: Memory used to hold info after the producer has created it and before the consumer has used it
- Operating rules
  - Allow producer to get ahead of consumer
  - Consumer and procedure don't operate in lock-step

Producer/Consumer (2)



Bounded buffer with blocking reads and writes

Producer/Consumer (3)

- What is "correct" for this example?
- Constraints
  - The consumer must wait for the producer to fill some of the buffer space if the buffer is empty
  - The producer must wait for the consumer to empty some of the buffer space if the buffer is full
- A separate semaphore is used for each constraint
  - **`buf_avail`** — Initialized to **`numBuffers`**
  - **`data_avail`** — Initialized to **`0`**

Producer/Consumer (4)

```
Producer(){
   P(buf_avail)
      buf = data;
   V(data_avail)
}
```

```
Consumer(){
   P(data_avail)
      data = buf;
   V(buf_avail)
}
```

```
Semaphore   buf_avail = 3,
           data_avail = 0;
```

Producer/Consumer (5)

- Note
  - V() is done when a resource is created
  - P() when destroyed

Disable Interrupts (1)

```
Task1(){
   P(S1)
      use pr1;
   V(S1)
}
```

```
Task2(){
   P(S1)
      use pr1;
   V(S1)
}
```

Disable Interrupts (2)

```
Task3(){
  P(S2)
      use pr2;
  V(S2)
}
```

```
Task1(){
  disable intr
      use pr1;
  enable intr
}
```

Turn all traffic lights in Seoul into red

Drawbacks (1)

```
Producer(){

  P(S1)

     buf = data;

  V(S2)

}
```

```
Consumer(){

  P(S2)

     data = buf;

  V(S1)

}
```

Semaphore naming issue

Drawbacks (2)

```
Task1(){
   P(S1)
      buf = data;
   V(S1)
}
```

```
Task2(){
   P(S2)
      data = buf;
   V(S1)
}
```

Is this a race condition or not?

Solution

```
Task1(){
  mutex_lock(S1)
     buf = data;
  mutex_unlock(S1)
}
```

```
Task2(){
  mutex_lock(S2)
     data = buf;
  mutex_unlock(S1)
}
```

This is surely a race condition

Implementation (1)

- Uniprocessor solution:

```
struct Semaphore {
        int cnt;
        Queue queue;
}
```

- Key idea
  - Use disable interrupt primitive to get mutual exclusion

Implementation (2)

```
P(S) {
  disableInterrulpts();
  if(S.cnt-- > 0)
        enableInterrupts();
  else
        sleep(S.queue);
}
sleep(Q) {
   // cur_p is current proc
   enqueue(cur_p, Q);
   enableInterrupts();
   yield_cpu();
}
```

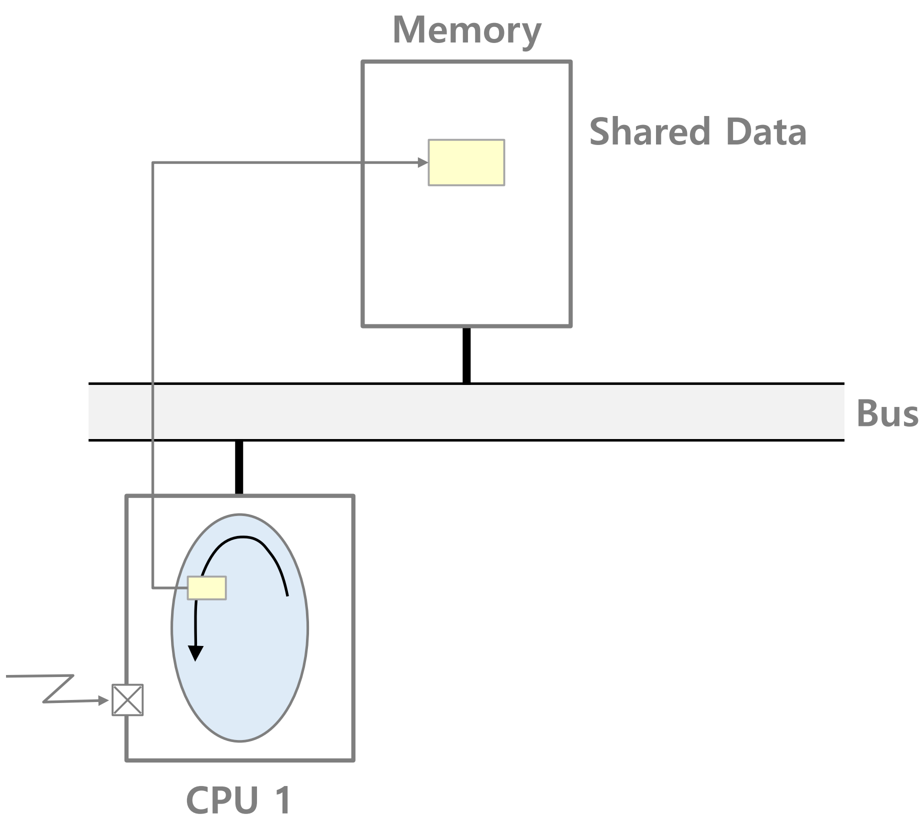
```
V(S) {
  disbleInterrupts();
  if(S.cnt++ >= 0)
        enableInterrpts();
  else
        wakeup(S.queue);
}
wakeup(Q) {
   p = dequeue(Q);
   enableInterrupts();
   reschedule(p);
}
```

Implementation (3)

- Semaphore implementation of the previous slide
  - Works only for FIFO semaphore queue
  - Works only for a single core processor

Implementation (4)

- Guaranteeing atomicity on a single processor



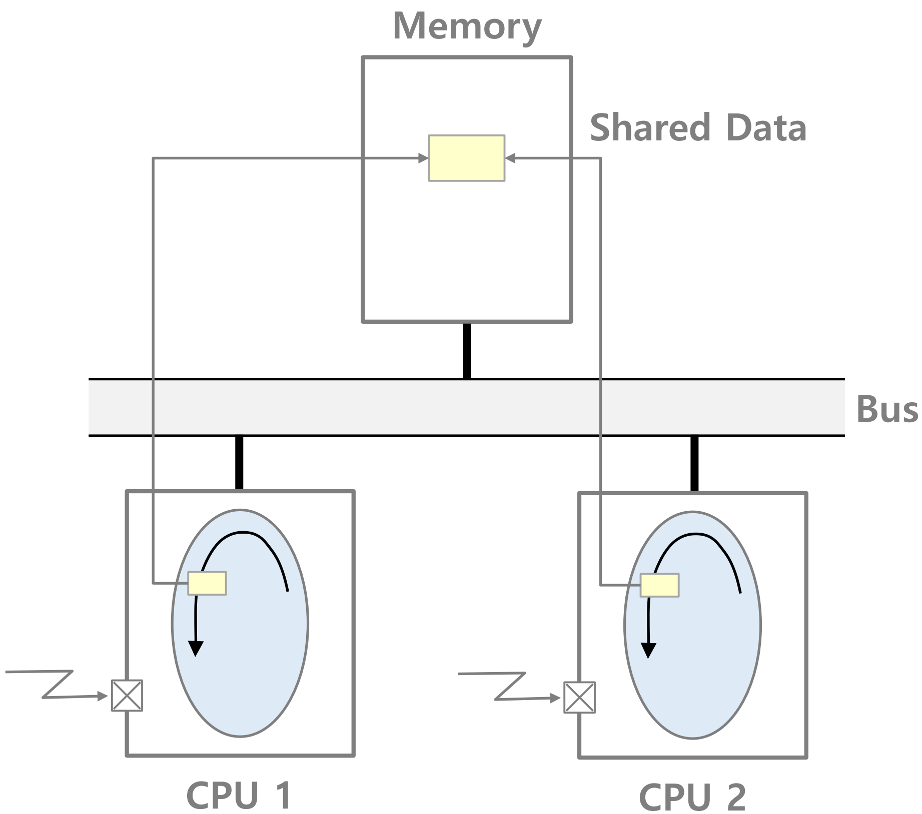
Implementation (5)

- Multiprocessor solution:
  - Mutual exclusion is harder

- Possibilities
  - Prevent other processors from accessing main memory
  - Use *atomic hardware support* for memory operations
    - Memory accesses via *unified read-and-write bus transactions*
    - Atomic *read-modify-write instruction*

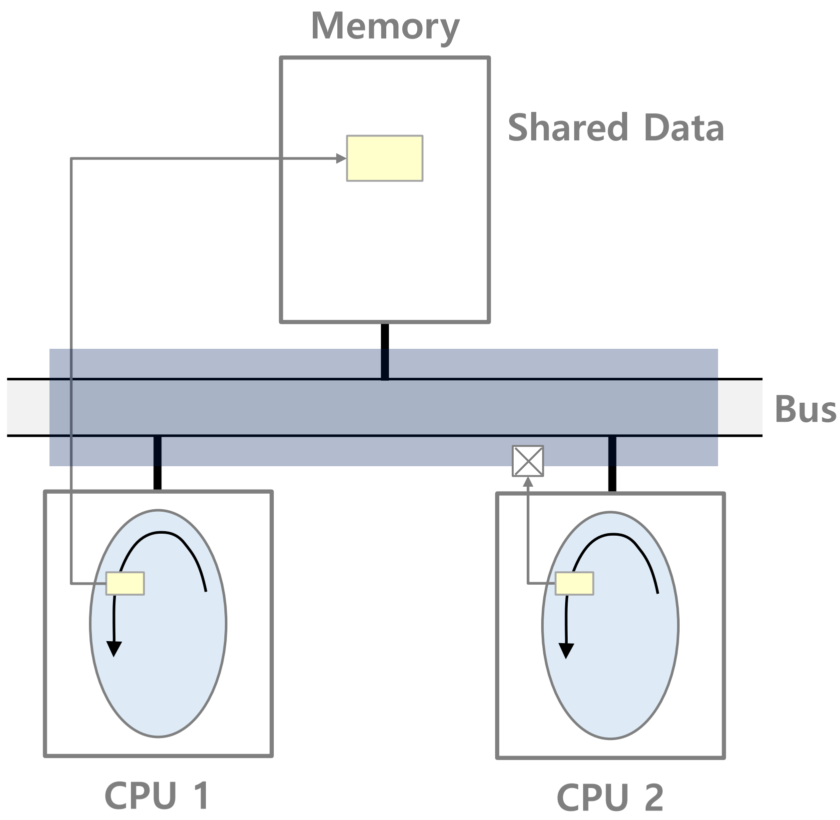
Implementation (6)

- Broken atomicity on a multiprocessor



Implementation (7)

- Guaranteeing atomicity on a multiprocessor



Implementation (8)

- Multiprocessor-safe primitives:
  - Utilize test-and-set (TAS) instruction
  - Change `disableInterrupt();` to
    ```
    disableInterrupt();
    while (TAS(lockMem) ! = 0) continue;
    ```
  - Change `enableInterrupt();` to
    ```
    lockMem = 0;
    enableInterrupt();
    ```
  - Note:
    - Multiprocessor solution does some busy-waiting

Implementation (9)

- Important point
  - Implement some mechanism once, very carefully and then always write programs that use that mechanism
    - Layering is very important

pthread Mutex (1)

- Header file
  - `#include <pthread.h>`
- Semaphore declaration
  - `pthread_mutex_t mutex;`
- Semaphore functions
  - ```
    int pthread_mutex_init(
              pthread_mutex_t *mutex,
              const pthread_mutexattr_t *attr);
    ```
  - `int pthread_mutex_lock(*mutex);`
  - `int pthread_mutex_unlock(*mutex);`
  - `int pthread_mutex_destroy(*mutex);`
    ```

pthread Mutex (2)

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_t tid[2];
int counter = 0;
pthread_mutex_t mutex;
```

pthread Mutex (3)

```
void *ThreadCode(void *argument)
{
   pthread_mutex_lock(&mutex);

   unsigned long i = 0;
   counter++;

   printf("\n Job %d started.\n", counter);
   for(i=0; i<0xFFFFFFFF; i++);
   printf("\n Job %d finished.\n", counter);

   pthread_mutex_unlock(&mutex);

   return NULL;
}
```

pthread Mutex (4)

```
int main(void)
{
   int i = 0, err;

   /* Since mutex is a binary semaphore, it gets 1 */
   if (pthread_mutex_init(&mutex, NULL) != 0){
      printf("\n mutex_init failed.\n");
      return 1;
   }
   while (i < 2){
      err = pthread_create(&(tid[i]), NULL, ThreadCode, NULL);
      if (err != 0)
            printf("\ncan't create thread: [%s].\n", strerror(err));
      i++;
   }
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
   pthread_mutex_destroy(&mutex);

   return 0;
}
```

`pthread` Mutex (5)

```
$ gcc mutex.c -o mutex -lpthread
$ ./mutex
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
```

POSIX Semaphores (1)

- Header file
  - `#include <semaphore.h>`
- Semaphore declaration
  - `sem_t sem;`
- Semaphore functions

```
init sem_init(sem_t *sem, int pshared,
                      unsigned int value);
```

```
int sem_wait(sem_t *sem);
```

```
int sem_post(sem_t *sem);
```

```
int sem_destory(sem_t *sem);
```

POSIX Semaphores (2)

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

pthread_t tid[2];
int counter = 0;
sem_t sem;
```

POSIX Semaphores (3)

```
void *ThreadCode(void *argument)
{
   sem_wait(&sem);

   unsigned long i = 0;
   counter++;

   printf("\n Job %d started.\n", counter);
   for(i=0; i<0xFFFFFFFF; i++);
   printf("\n Job %d finished.\n", counter);

   sem_post(&sem);

   return NULL;
}
```

POSIX Semaphores (4)

```
int main(void)
{
   int i = 0, err;

   /* Since sem is a binary semaphore, it gets 1 */
   if (sem_init(&sem, 0, 1) != 0){
      printf("\n sem_init failed.\n");
      return 1;
   }
   while (i < 2){
      err = pthread_create(&(tid[i]), NULL, ThreadCode, NULL);
      if (err != 0)
            printf("\ncan't create thread: [%s]", strerror(err));
      i++;
   }
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
   sem_destroy(&sem);

   return 0;
}
```

POSIX Semaphores (5)

```
$ gcc semaphore.c -o semaphore -lpthread
$ ./semaphore
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
```

III. Condition Variable

What is Condition Variable?

- Condition variable
  - An event having two operations that are performed on itself
    - Two operations: `wait(c)`, `signal(c)`
    - Note that a condition variable has no value!
      - One cannot store a value into or retrieve a value from a condition variable
    - A thread waits for an event to occur using `wait(c)`
      - Condition variable "`c`" corresponds to the event
      - Condition variable is associated with a waiting queue
    - A thread wakes up another thread waiting on an event using `signal(c)`
      - Condition variable "`c`" corresponds to the event

POSIX Condition Variable (1)

- Header file
  - `#include <pthread.h>`
- Semaphore declaration
  - `pthread_cond_t cond;`
- Semaphore functions

```
int pthread_cond_init(
            pthread_cond_t *cond,
            const pthread_condattr_t *attr);
```

```
int pthread_cond_wait(*cond, *mutex);
```

```
int pthread_cond_signal(*cond);
```

```
int pthread_cond_destroy(*cond);
```

POSIX Condition Variable (2)

```
#include <stdio.h>
#include <pthread.h>

/* static initializers */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int data_produced = 0;
int count = 0;
```

POSIX Condition Variable (3)

```
void *consumer(void)
{
    while (1) {
        pthread_mutex_lock(&mutex);
        while (data_produced == 0)
            pthread_cond_wait(&cond, &mutex);
        printf("Consumed %d\n", count);
        data_produced = 0;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }

    return 0;
}
```

POSIX Condition Variable (4)

```
void *producer(void)
{
   while (1) {
      pthread_mutex_lock(&mutex);
      while (data_produced == 1)
         pthread_cond_wait(&cond, &mutex);
      printf("Produced %d\n", count++);
      data_produced = 1;
      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mutex);
   }

   return 0;
}
```

POSIX Condition Variable (5)

```
int main(void)
{
   pthread_t tid[2];

   pthread_create(&(tid[0]), NULL, consumer, NULL);
   pthread_create(&(tid[1]), NULL, producer, NULL);

   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);

   return 0;
}
```

Why Condition Variable Needs Mutex?

- Reason
  - `wait()` involves unlocking, blocking, wake-up, and locking
  - `wait()` needs an *atomic* operation that is able to (1) do the wait and (2) unlock the mutex atomically
    - Calling `unlock()` and `wait()` in series does not work
      - If `signal()` is issued after `unlock()` and before `wait()`, `signal()` will be missed and `wait()` will stay blocked
    - Neither does calling `wait()` and `unlock()`
      - The thread gets blocked with the mutex being locked

IV. Monitor

Why Monitors?

- Motivations
  - Semaphore is an unstructured construct
    - Prone to synchronization bugs – race condition
    - Too low-level
  - Great if we have a structured construct having a higher-level abstraction
    - High-level mutual exclusion semantics
      - Only one thread is active in the construct at any given time
    - Locks are hidden
      - Automatically locks and unlocks to enter or exit from a critical section
  - Too good to be true?!
    - No – the answer to this wish is "monitor"

What is Monitors? (1)

- Key ideas
  - Monitor
    - A synchronization tool that automatically locks and unlocks a mutex lock (AKA monitor lock) when in a critical section
      - The mutex lock is added implicitly to the code, never seen by user
    - Has condition variables for dealing with diverse scheduling situations (thread cooperation)
    - Mostly associated with an abstract data type (ADT)
  - ADT
    - A class of objects whose logical behavior is defined by a set of values and a set of operations
    - Reminds us of a class in an object-oriented language

What is Monitors? (2)

- Definition
  - A monitor is
    - A programming language construct (ADT or class) that supports controlled access to shared data
  - A monitor encapsulates:
    1. Shared data structures
    2. Procedures that operate on the shared data
    3. Synchronization between concurrent threads that invoke those procedures
  - Implication from ADT
    - Data can only be accessed from within the monitor, using the provided procedures
    - Leads to the protection of the data from unstructured access

What is Monitors? (3)

- Evolution
  - Brinch Hansen (1973)
    - Requires Signal to be the last statement
  - C. A. R. Hoare (1974)
    - CACM, vol. 17, no. 10. 10 October 1974, pp. 549-557
    - Requires relinquishing CPU to signaler
  - Mesa language (1977)
    - Monitor in language, but signaler keeps mutex and CPU
    - Waiter simply put on ready queue, with no special priority
  - Pthreads (1995)
    - Mutex lock primitives and condition variables
  - Java threads (1995)
    - Use most of the Pthreads primitives

Condition Variables in Monitor (1)

- Three operations
  - `wait(condition)`
    - Release monitor lock, put thread to sleep
    - Reacquire lock when waken
  - `signal(condition)`
    - Wake up one thread waiting on the condition variable
    - If nobody waiting, do nothing
  - `broadcast(condition)`
    - Wake up all threads waiting on the condition variable

Condition Variables in Monitor (2)

- Semantic variations on the wait/signal mechanism
  - Who gets the monitor lock after a signal?
    - "Hoare semantics"
      - On signal, the signaler releases the monitor lock
      - The awakened thread acquires the monitor lock
      - Re-enters the monitor (need not check) and resumes
    - "Mesa semantics"
      - On signal, the signaler keeps the monitor lock
      - The awakened thread waits for the monitor lock
      - Must check again and be prepared to sleep

Code Example using Monitor (1)

```
procedure producer()
begin
   while (true) do
   begin
      data = produceData();
      ProducerConsumer.insert(data);
   end
end

procedure consumer()
begin
   while (true) do
   begin
      data = ProducerConsumer.remove();
      consumeData(data);
   end
end
```

Code Example using Monitor (2)

```
monitor ProducerConsumer
   integer count;
   condition full, empty;
   procedure insert(integer data)
   begin
      if (count = N) then full.wait();
      enqueue(data);
      count++;
      if (count = 1) then empty.signal();
   end
   function integer remove()
   begin
      if (count = 0) then empty.wait();
      remove = dequeue();
      count--;
      if (count = N-1) then full.signal();
   end
   count = 0;
end monitor
```

Monitor Implementation (1)

```
monitor ResourceManager
   boolean busy;
   condition x;

   procedure acquire()
   begin
      if (busy) then x.wait();
      busy = true;
   end

   procedure release()
   begin
      busy = false;
      x.signal();
   end

   busy = false;
end monitor
```

Monitor Implementation (2)

```
P(monitor_lock);
   Body of the Function;
if (sig_lock_cnt > 0) {
  V(sig_lock);
} else {
  V(monitor_lock);
}
```

```
/* x.wait */
x_cnt++;
if (sig_lock_cnt > 0) {
  atomically {
    V(sig_lock);
    P(x_lock);
  }
} else {
  atomically {
    V(monitor_lock);
    P(x_lock);
  }
}
x_cnt--;
```

```
/* x.signal */
if (x_cnt > 0) {
  sig_lock_cnt++;
  atomically {
    V(x_lock);
    P(sig_lock);
  }
  sig_lock_cnt--;
}
```

monitor_lock	Semaphore for the monitor
sig_lock	Semaphore for signalers
x_lock	Semaphore for condition x

What is Monitors? (4)

- Disadvantages
  - May be less efficient than lower-level synchronization
  - Not available from all programming languages