



# Shared-Memory Programming: Pthread

National Tsing-Hua University  
2017, Summer Semester

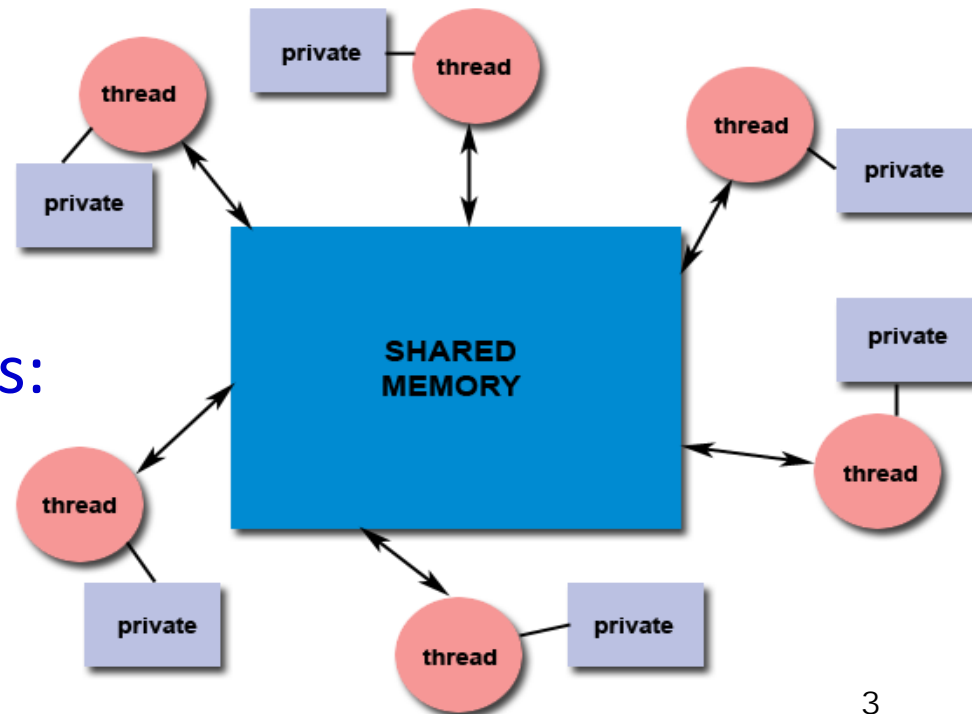


# Outline

- Shared-memory Programming
- Pthread
- Synchronization Problem & Tools

# Shared-Memory Programming

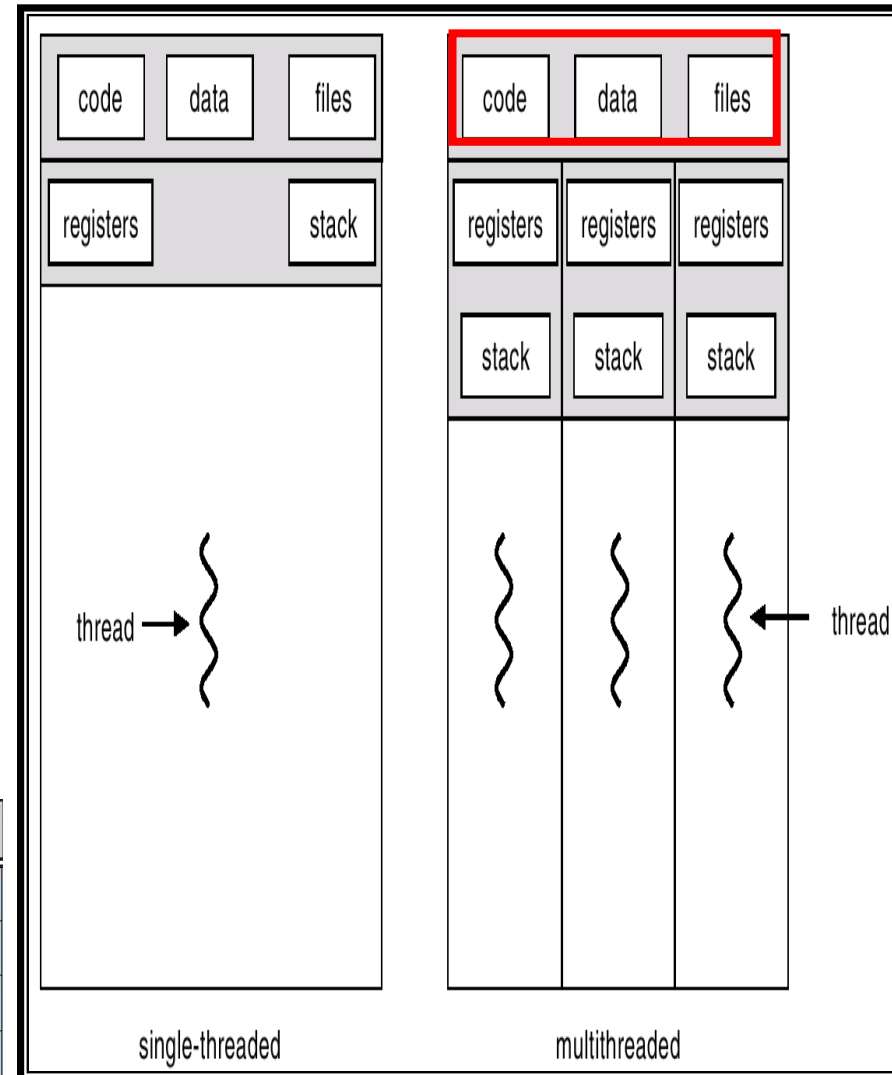
- **Definition:** Processes communicate or work together with each other **through a shared memory space** which can be accessed by all processes
  - **Faster & more efficient than message passing**
- **Many issues as well:**
  - **Synchronization**
  - **Deadlock**
  - **Cache coherence**
- **Programming techniques:**
  - **Parallelizing compiler**
  - **Unix processes**
  - **Threads (**Pthread**, Java)**



# Threads vs. Processes

- **Process (heavyweight process):** complete separate program with its own variables, stack, heap, and everything else.
- **Thread (lightweight process):** share the **same memory space** for global variables, resources
- In Linux:
  - Threads are created via **clone a process** with a flag to indicate the **level of sharing**

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.



# Why Thread?

## ■ Lower creation/management cost vs. Process

platform	fork()	pthread_create()	speedup
AMD 2.4 GHz Opteron	17.6	1.4	15.6x
IBM 1.5 GHz POWER4	104.5	2.1	49.8x
INTEL 2.4 GHz Xeon	54.9	1.6	34.3x
INTEL 1.4 GHz Itanium2	54.5	2.0	27.3x

## ■ Faster inter-process communication vs. MPI

platform	MPI Shared Memory BW (GB/sec)	Pthreads Worst Case Memory-to-CPU BW (GB/sec)	speedup
AMD 2.4 GHz Opteron	1.2	5.3	4.4x
IBM 1.5 GHz POWER4	2.1	4	1.9x
INTEL 2.4 GHz Xeon	0.3	4.3	14.3x
INTEL 1.4 GHz Itanium2	1.8	6.4	3.6x

# Outline

- Shared-memory Programming
- Pthread
  - What is Pthread
  - Pthread Creation
  - Pthread Joining & Detaching
- Synchronization Problem & Tools

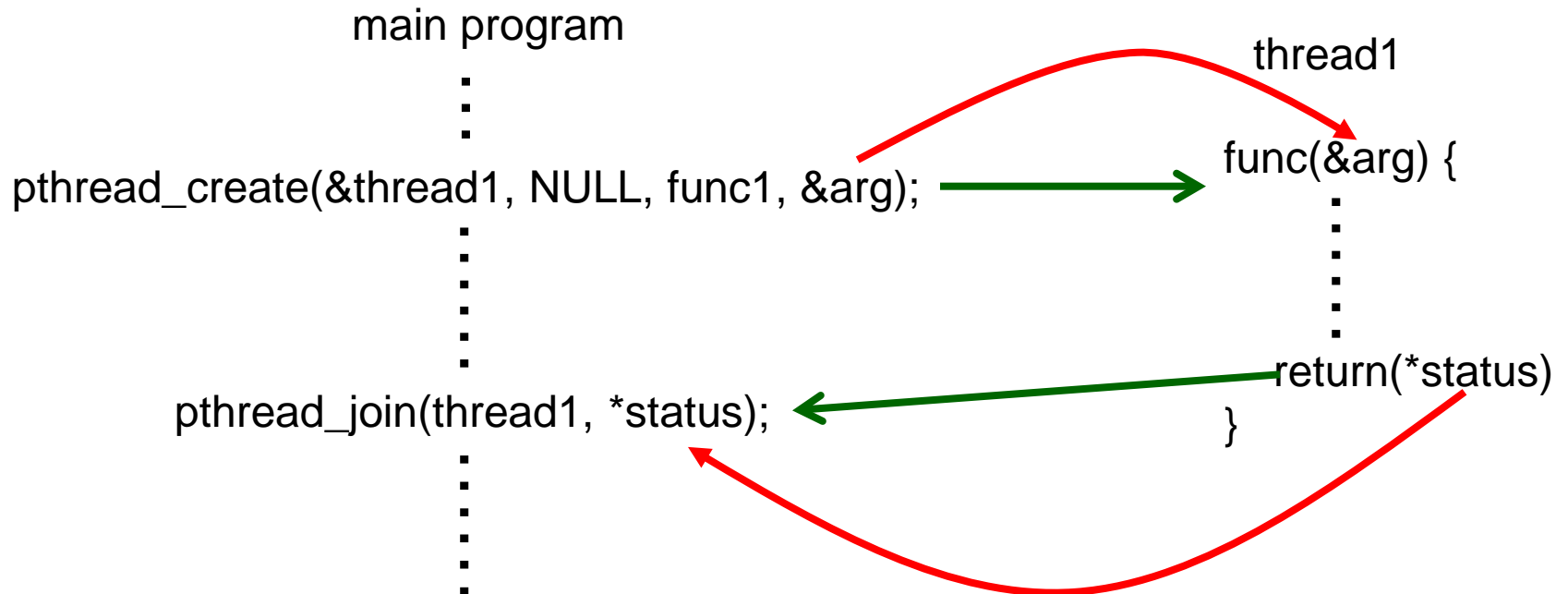
# What is Pthread?

- Historically, hardware vendors have implemented their own proprietary versions of threads
- **POSIX** (Potable Operating System Interface) standard is specified for portability across Unix-like systems
  - Similar concept as MPI for message passing libraries
- **Pthread** is the implementation of POSIX standard for thread
  - Same relation between MPICH and MPI

# Pthread Creation

## ■ pthread\_create(thread,attr,routine,arg)

- **thread**: An **unique identifier** (token) for the new thread
- **attr**: It is used to set **thread attributes**. NULL for the default values
- **routine**: The routine that the thread will execute once it is created
- **arg**: A **single argument** that may be **passed to routine**



# Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadId) {
    long* data = static_cast <long*> threadId;
    printf("Hello World! It's me, thread #%ld!\n", *data);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    for(long tid=0; tid<NUM_THREADS; tid++){
        pthread_create(&threads[tid], NULL, PrintHello, (void *)&tid);
    }
    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

# Pthread Joining & Detaching

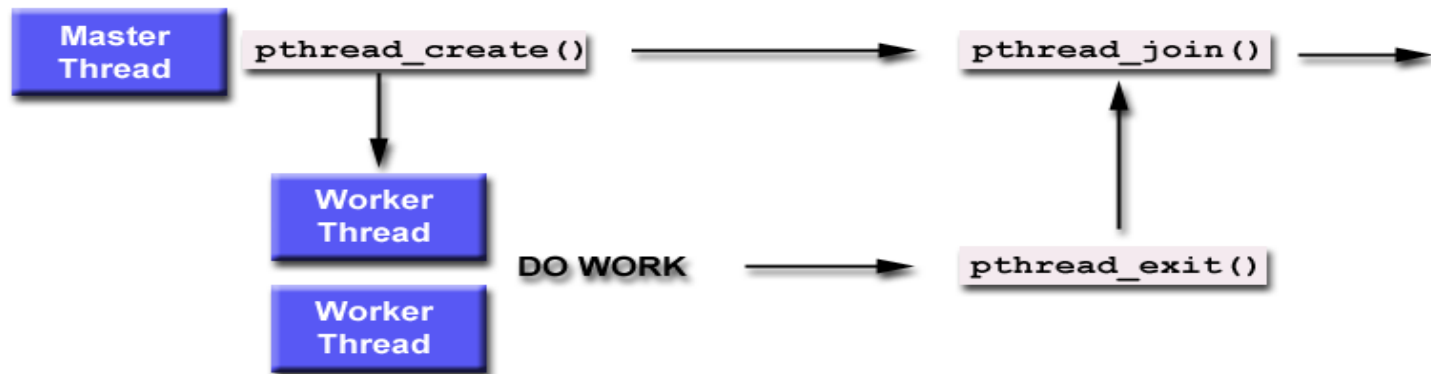
## ■ pthread\_join(threadId, status)

- Blocks until the specified *threadId* thread terminates
- One way to accomplish synchronization between threads
- Example: to create a pthread barrier

```
for (int i=0; i<n; i++) pthread_join(thread[i], NULL);
```

## ■ pthread\_detach(threadId)

- Once a thread is **detached**, it can **never** be joined
- Detach a thread could free some system resources



# Outline

- Shared-memory Programming
- Pthread
- Synchronization Problem & Tools
  - Pthread
    - ◆ Mutually exclusion Lock
    - ◆ Condition variable
  - POSIX Semaphore
  - JAVA Monitor
- Other issues

# Synchronization Problem

- The outcome of data content should **NOT** be decided by the **execution order among processes**

- **Instructions** of individual processes/threads may be **interleaved** in time

- E.g.: Assume variable **“counter”** is **shared by processes**

Process0

```
main() {
```

```
...
```

```
    counter++;
```

```
...
```

```
}
```

Process1

```
main() {
```

```
...
```

```
    counter--;
```

```
...
```

```
}
```

- The statement **“counter++”** & **“counter--”** may be implemented in machine language as:

```
move ax, counter
add  ax, 1
move counter, ax
```

```
move bx, counter
sub   bx, 1
move counter, bx
```

# Instruction Interleaving

- Assume counter is initially 5. One interleaving of statement is:

producer: move ax, counter      ➔ ax = 5

producer: add ax, 1      ➔ ax = 6

*context switch*

consumer: move bx, counter      ➔ bx = 5

consumer: sub bx, 1      ➔ bx = 4

*context switch*

producer: move counter, ax      ➔ counter = 6

*context switch*

consumer: move counter, bx      ➔ counter = 4

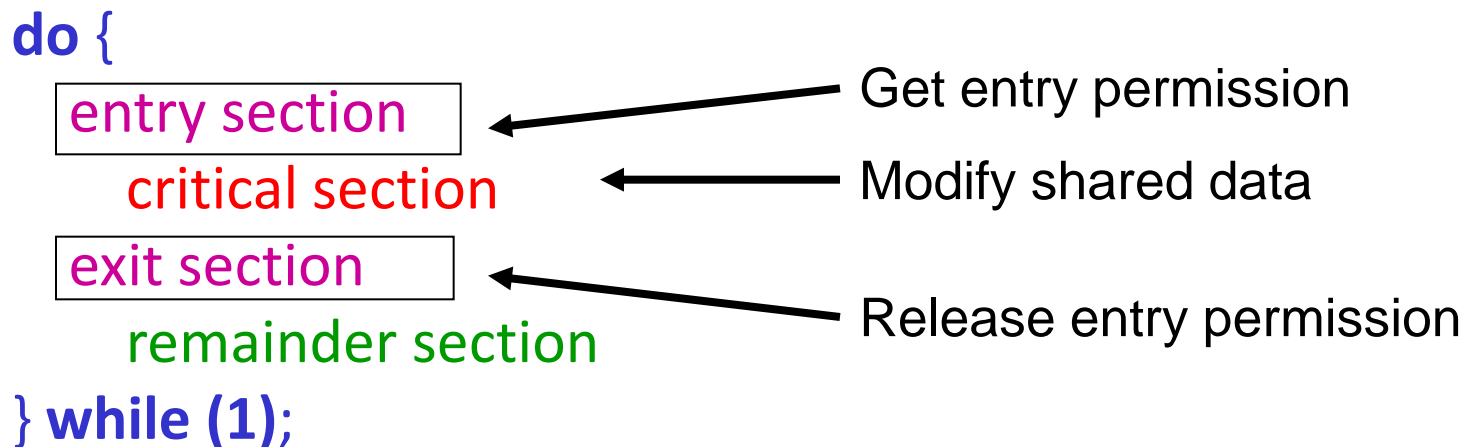
- The value of counter may be either 4, 5, or 6
- The ONLY correct result is 5!

# Outline

- Shared-memory Programming
- Pthread
- Synchronization Problem & Tools
  - Pthread
    - ◆ Mutually exclusion Lock
    - ◆ Condition variable
  - POSIX Semaphore
  - JAVA Monitor
- Other issues

# Critical Section & Mutual Exclusion

- **Critical Section** is a piece of code that can only be accessed by one process/thread at a time
- **Mutual exclusion** is the problem to insure only one process/thread can be in a critical section
- E.g.: The design of entry section & exit section provides mutual exclusion for the critical section



# Locks

- Lock: the simplest mechanism for ensuring mutual exclusion of critical section

➤ Spinlock is one of the implementation:

```
while (lock == 1);           /* no operation in while loop */
lock = 1;                    /* enter critical section */
.
critical section
.
lock = 0;                     /* leave critical section */
```

- Locks are implemented in Pthreads by a special type of variables “mutex”
- **Mutex** is abbreviation of “mutual exclusion”

# Pthread Lock/Mutex Routines

- To use mutex, it must be declared as of type `pthread_mutex_t` and initialized with `pthread_mutex_init()`
- A mutex is destroyed with `pthread_mutex_destroy()`
- A critical section can then be protected using `pthread_mutex_lock()` and `pthread_mutex_unlock()`
- Example:

```
#include "pthread.h"
pthread_mutex_t  mutex;
pthread_mutex_init (&mutex, NULL);
pthread_mutex_lock(&mutex);
// enter critical section

    Critical Section

pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
// leave critical section
```

specify default attribute for the mutex

# Bounded-Buffer Problem

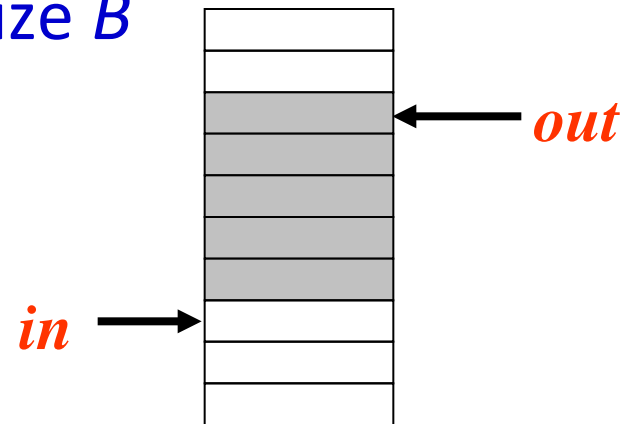
- A pool of  $n$  buffers, each capable of holding one item
- Producer:
  - grab an empty buffer
  - place an item into the buffer
  - waits if no empty buffer is available
- Consumer:
  - grab a buffer and retracts the item
  - place the buffer back to the free pool
  - waits if all buffers are empty

# Bounded-Buffer Problem

- **Producer** process produces information that is consumed by a **Consumer** process

- Buffer as a circular array with size  $B$

- next free:  $in$
- first available:  $out$
- empty:  $in = out$
- full:  $(in+1) \% B = out$



- The solution allows at most  $(B-1)$  item in the buffer
  - Otherwise, cannot tell the buffer is full or empty

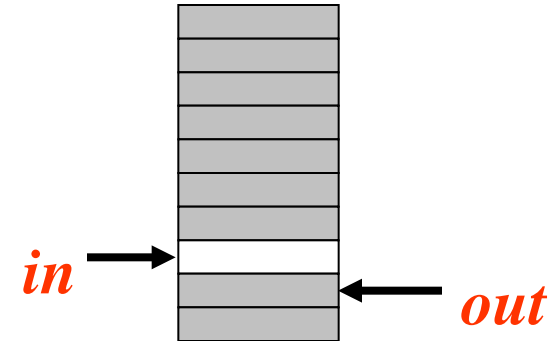
# Shared-Memory Solution

```
/*producer*/  
while (1) {  
    while (((in + 1) % BUFFER_SIZE) == out)  
        ; //wait if buffer is full  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

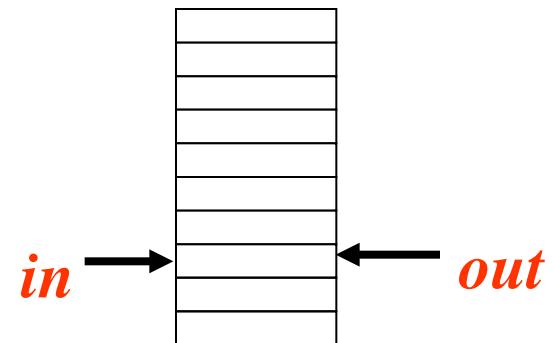
“*in*” only modified by producer

```
/*consumer*/  
while (1) {  
    while (in == out); //wait if buffer is empty  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
}
```

“*out*” only modified by consumer



```
/* global data structure */  
#define BUFSIZE 10  
item buffer[BUFSIZE];  
int in = out = 0;
```



# Using Mutex Lock

```
/*producer*/
```

```
while (1) {  
    nextItem = getItem( );  
    while (counter == BUFFER_SIZE) ;  
    buffer[in] = nextItem;  
    in = (in + 1) % BUFFER_SIZE;  
    mutex_lock(mutex);  
    counter++;  
    mutex_unlock(mutex);  
}
```

```
/*consumer*/
```

```
while (1) {  
    while (counter == 0) ;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    mutex_lock(mutex);  
    counter--;  
    mutex_unlock(mutex);  
}
```

# Condition Variables (CV)

- CV represent some **condition** that a thread can:
  - Wait on, until the condition occurs; or
  - Notify other waiting threads that the condition has occurred
- Three operations on condition variables:
  - **wait()** --- **Block** until another thread calls **signal()** or **broadcast()** on the CV
  - **signal()** --- Wake up **one thread** waiting on the CV
  - **broadcast()** --- Wake up **all threads** waiting on the CV
- In Pthread, CV **type** is a **pthread\_cond\_t**
  - Use **pthread\_cond\_init()** to initialize
  - **pthread\_cond\_wait (&theCV, &somelock)**
  - **pthread\_cond\_signal (&theCV)**
  - **pthread\_cond\_broadcast (&theCV)**

# Using Condition Variable

## ■ Example:

- A threads is designed to **take action when x=0**
- Another thread is responsible for decrementing the counter

```
pthread_cond_t  cond;  
pthread_cond_init (cond, NULL);
```

```
pthread_mutex_t  mutex;  
pthread_mutex_init (mutex, NULL);
```

```
action() {  
    pthread_mutex_lock (&mutex)  
    if (x != 0)  
        pthread_cond_wait (cond, mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    pthread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        pthread_cond_signal (cond);  
    pthread_mutex_unlock (&mutex);  
}
```

- All condition variable operation **MUST** be performed while a mutex is **locked!!!**

# Semaphore

- A tool to generalize the synchronization problem
  - **Deadlock** may occur if not use appropriately !
- More specifically...
  - a record of **how many units** of a particular resource are available
    - ◆ If #record = 1 ➔ **binary semaphore, mutex lock**
    - ◆ If #record > 1 ➔ **counting semaphore**
  - accessed only through 2 *atomic* ops: **wait** & **signal**
- **Spinlock** implementation:
  - Semaphore is an **integer variable**

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

# Semaphore Example

- shared data:

semaphore  $S$  ; // initially  $S = 1$

- Process  $P_i$ :

do {

    wait ( $S$ ) ;

    critical section

    signal ( $S$ );

    remainder section

} while (1) ;

# POSIX Semaphore

- Semaphore is part of **POSIX** standard BUT it is **not** belonged to Pthread
  - It can be used with or **without** thread
- POSIX Semaphore routines:
  - **sem\_init**(sem\_t \*sem, int pshared, unsigned int value)
  - **sem\_wait**(sem\_t \*sem)
  - **sem\_post**(sem\_t \*sem)
  - **sem\_getvalue**(sem\_t \*sem, int \*valptr)
  - **sem\_destroy**(sem\_t \*sem)

Initial value of the semaphore

Current value of the semaphore

■ Example:

```
#include <semaphore.h>
sem_t sem;
sem_init(&sem);
sem_wait(&sem);
    // critical section
sem_post(&sem);
sem_destroy(&sem);
```

# Readers-Writers Problem

- A set of shared data objects
- A group of processes
  - reader processes (read shared objects)
  - writer processes (update shared objects)
  - **a writer process has exclusive access to a shared object**
- *Different variations involving priority*
  - *first RW problem*: no reader will be kept waiting unless a writer is updating a shared object
  - *second RW problem*: once a writer is ready, it performs the updates as soon as the shared object is released
    - ➔ writer has higher priority than reader
    - ➔ once a writer is ready, no new reader may start reading

# First Reader-Writer Algorithm

```
// mutual exclusion for write
semaphore wrt_sem=1
// mutual exclusion for readcount
semaphore mutex=1
int readcount=0;
```

```
Writer(){
    while(TRUE){
        wait(wrt_sem);
        // Writer Code

        signal(wrt_sem);
    }
}
```

Acquire write lock  
if reads haven't

```
Reader(){
    while(TRUE){
        wait(mutex);
        readcount++;
        if(readcount==1)
            wait(wrt_sem);
        signal(mutex);

        // Reader Code

        wait(mutex);
        readcount--;
        if(readcount==0)
            signal(wrt_sem);
        signal(mutex);
    }
}
```

release write lock if  
no more reads

- Readers share a single wrt lock
- Writer may have starvation problem

# Semaphore Drawback

- Although semaphores provide a convenient and effective synchronization mechanism, its correctness is depending on the programmer
  - All processes access a shared data object must execute `wait()` and `signal()` in the right order and right place
  - This may not be true because honest programming error or uncooperative programmer

# Synchronized Tools in JAVA

## ■ Synchronized Methods (Monitor)

- Synchronized method uses the method receiver as a lock
- Two invocations of synchronized **methods cannot interleave on the same object**
- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread exits the object

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() { c++; }  
    public synchronized void decrement() { c--; }  
    public synchronized int value() { return c; }  
}
```

# Synchronized Tools in JAVA

## ■ Synchronized Statement (Mutex Lock)

- Synchronized blocks uses the **expression** as a lock
- A synchronized Statement can only be executed once the thread has obtained a **lock for the object or the class that has been referred to in the statement**
- useful for improving concurrency **with fine-grained**

```
public void run()
{
    synchronized(p1)
    {
        int i = 10; // statement without locking requirement
        p1.display(s1);
    }
}
```

# The Big Picture

- Getting **synchronization** right is hard!
- How to pick between locks, semaphores, convars, monitors???
- **Locks** are very **simple** for many cases
  - But may not be the most efficient solution
- **Condition variables** **allow threads to sleep** while holding a lock
  - Be aware whether they use Mesa or Hoare semantics
- **Semaphores** provide **general functionality**
  - But also make it really easy to mess up or cause deadlock
- **Monitors** are a **“pattern”** for using locks and condition variables

# Reference

- Textbook:

- Parallel Computing Chap8

- Pthread Tutorial

- <https://computing.llnl.gov/tutorials/pthreads/>

- Synchronization Tools:

- <http://www.eecs.harvard.edu/~mdw/course/cs61/mediawiki/images/7/7e/Lectures-semaphores.pdf>

- Pthread API:

- <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

- JAVA Synchronized methods

- <http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>