Chapter 4 -- Threads

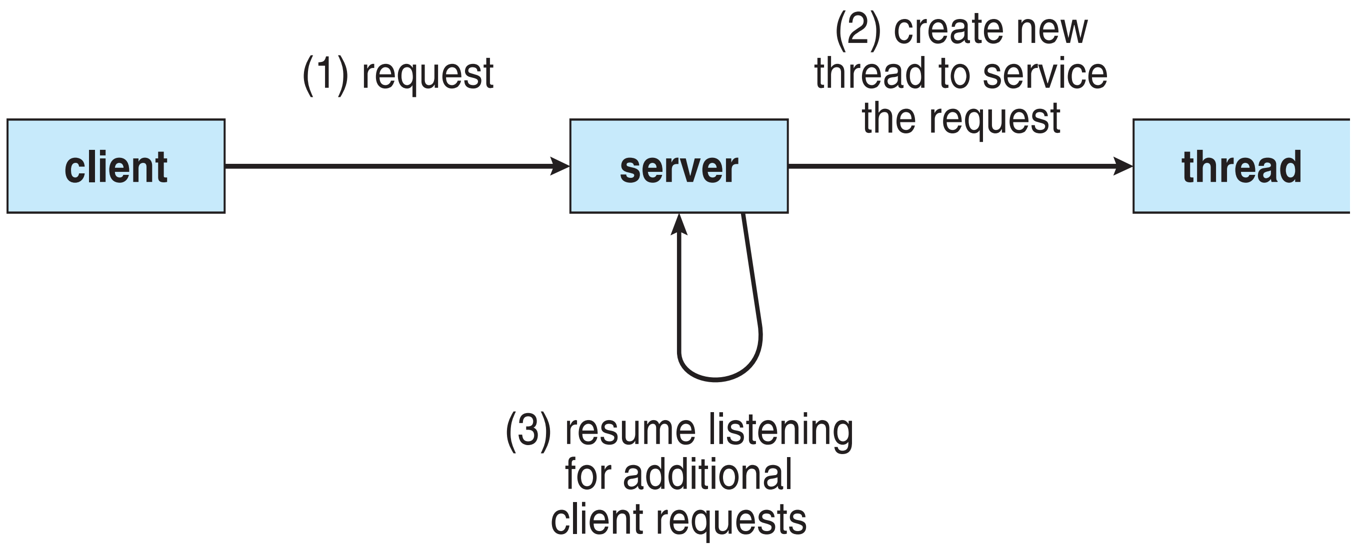
A/Prof Jian Zhang

School of Computing and Communication

University of Technology, Sydney

Ref: A. Silberschatz, P. B. Galvin & G. Gagne

Multithreaded Server Architecture



Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

* **Scalability:** In a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. In such scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system .

* A **page fault** occurs when a process accesses a **page** that is mapped in the virtual address space, but not loaded in physical memory

Multicore Programming

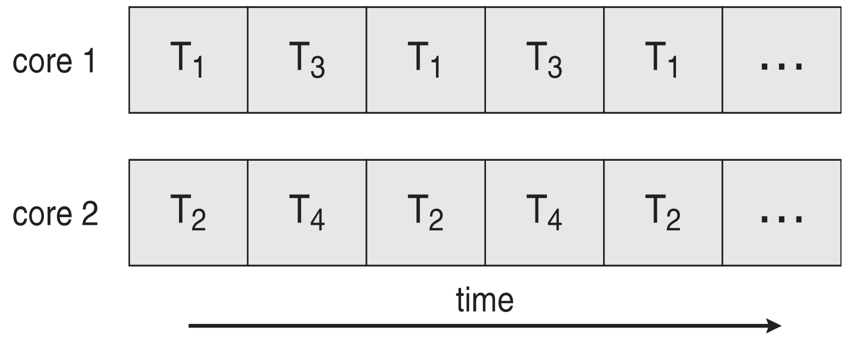
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- ......

Notice: the distinction between ***parallelism*** and ***concurrency*** in this discussion. A system is parallel if it can perform more than one task **simultaneously**. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism.

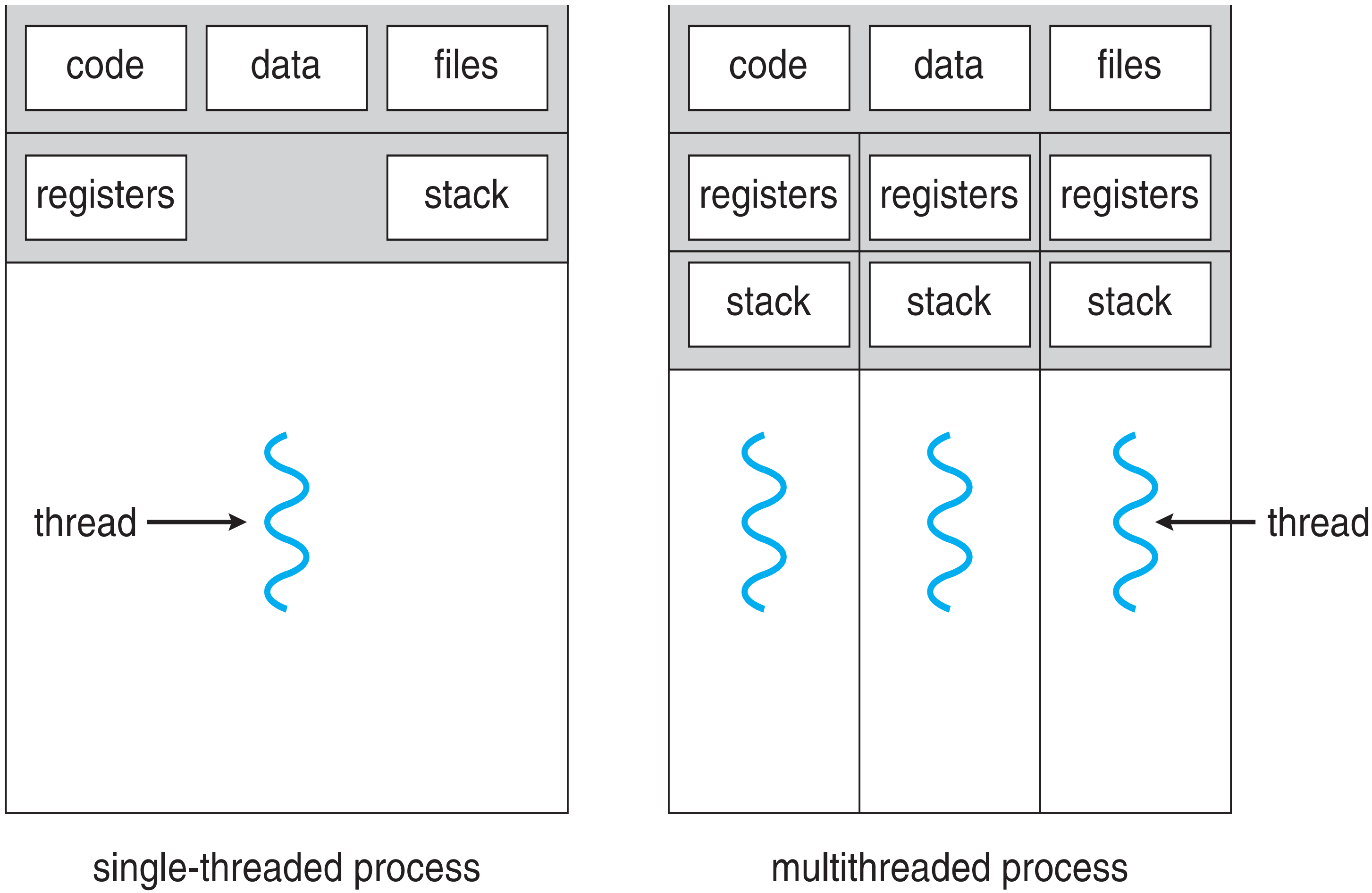
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

- **Parallelism on a multi-core system:**



Single and Multithreaded Processes



Single and Multithreaded Processes

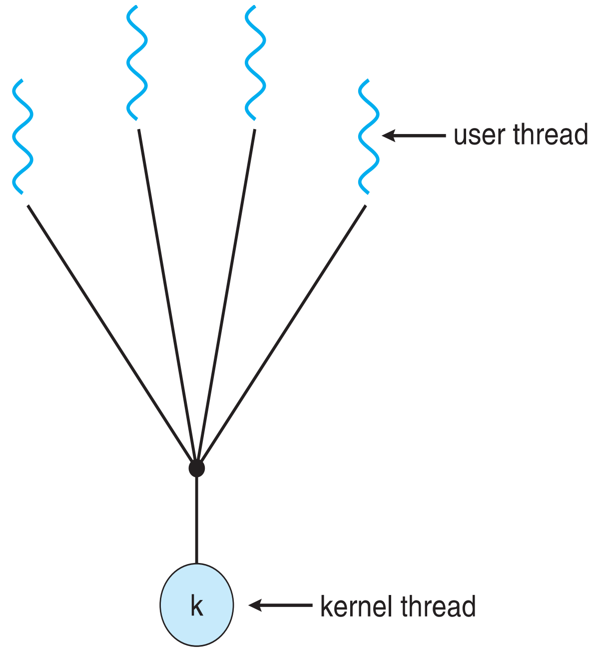
- A single-threaded: One by one process in programming.

  Concept: Sequential program such as the next step relies on the previous step closely.

  Example : Word process is better to have one thread to have editing, saving sequentially. Another example is a video paly program frame-by-frame!

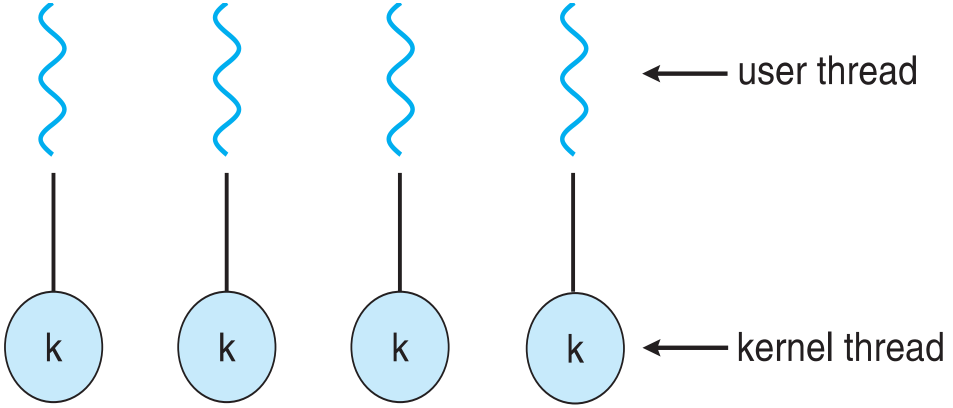
Many-to-One (Multithreading Models)

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muti-core system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



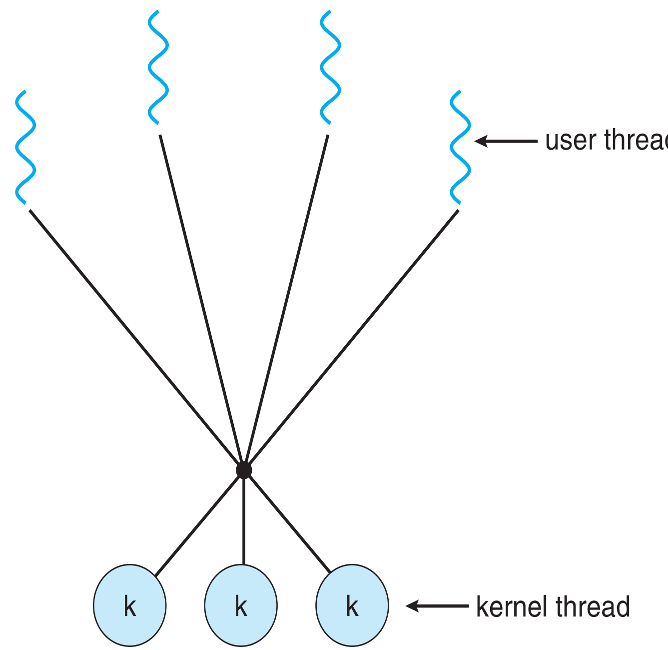
One-to-One (Multithreading Models)

- Each user-level thread maps to kernel thread
- Creating a user-level thread and also creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model (Multithreading Models)

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Consider the following code segment: Check point

```
pid t pid;
pid = fork();
if (pid == 0) { /* child process */
     //fork();
     thread create( . . .);
}
fork();
```

a. How many unique processes are created?
b. How many unique threads are created?

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- **Pthreads** may be provided either as user- or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {  // Argument number should be 2
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }                            // convert string to integer
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Program state shared across threads in a multithreaded process

Program state -- Global variables

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
```

Program state – Heap memory

Each thread gets a stack, while there's typically only one heap for the application

Conclusion: each thread has its separate set of register values and a separate stack memory.

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Linux Threads

- Linux (OS, system call) refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior
- **`struct task_struct`** points to process data structures (shared or unique)

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.

Chapter 5 – Process Synchronization

A/Prof Jian Zhang
School of Computing and Communication
University of Technology, Sydney

Ref: A. Silberschatz, P. B. Galvin & G. Gagne

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race Condition

- **`counter++ in Producer`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter-- in Consumer`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

  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}

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

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

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- A single-processor system – to disable interrupts!
  - **It can disable the timer interrupt and prevent context switching from taking place**
  - Currently running code would execute without preemption
  - Generally, **too inefficient on multiprocessor systems**
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
    - Function call -- test_and_set(&lock)
  - Or swap contents of two memory words
    - Function call -- compare and swap(int *value, int expected, int new value)

Mutex Locks

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

*spin in place" waiting on the lock to be released by the first process, thus the name spin lock.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable *
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

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

* As usual synchronization primitive, a `semaphore` is based on a variable. This variable may be incremented or decremented and it's state will represent ability to acquire lock.

Example discussion
Check point

- Semaphores as a server to limit the number of concurrent connections
  - Set up the number of allowable open socket connections
  - Acquire method is to set up a connection, Release method is to release the connection
  - When the connection reaches the limited number (set by semaphores), semaphore will block the acquire method until a new connection (released) is available

5.23 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test and set() instruction. The solution should exhibit minimal **busy waiting**.

```
int guard = 0;
int semaphore value = 0;
wait()
{
        while (TestAndSet(&guard) == 1);
        if (semaphore value == 0) {
                atomically add process to a queue of processes
                waiting for the semaphore and set guard to 0;
        }else {
                semaphore value--;
                guard = 0;
        }
}

signal()
{
        while (TestAndSet(&guard) == 1);
        if (semaphore value == 0 && there is a process on the wait queue)
                //wake up the first process in the queue
                //of waiting processes (busy waiting)
        else
        semaphore value++;
        guard = 0;
        }
```

Busy waiting -- Check point

busy waiting: While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). It is waiting for a condition to be satisfied in a tight loop without relinquishing the processor

Busy waiting wastes CPU cycles that some other process might be able to use productively.

Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future

Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**`synch`**" initialized to 0

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

- Can implement a counting semaphore ***S*** as a binary semaphore

Deadlock and Starvation

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

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

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock while needed by higher-priority process. it is possible that a third task *M* of medium priority will jump in ….
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

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

The bounded-buffer problem was introduced in Section 5.1

- The producer and consumer processes share the following data structures:
  - Int ***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

Examples are in Figures 5.9 (the procedure) and 5.10 (the consumer)

The bounded-buffer:

```
#define BUFFER_SIZE 10
        typedef struct {
                  . . .}
     item; item buffer[BUFFER_SIZE];
     int in = 0; int out = 0;
```

* Solution is correct, but can only use BUFFER_SIZE-1 elements

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

*They may not function correctly when executed concurrently – **counter == ???**

Classical Problems of Synchronization

- do {`mutex=1; empty =1; full=0`
- . . .
- /* produce an item in next produced */
- . . .
- wait(empty); // value n - 1
- wait(mutex); // =1 - 1
- . . .
- /* add next produced to the buffer */
- . . .
- signal(mutex); //release Mutex
- signal(full); // full: = + 1
- } while (true);
- **Figure 5.9** The structure of the producer process.

- do {`mutex=1; empty =1; full=0`
- ….
- wait(full);// full= -1
- wait(mutex); //= 1-1
- . . .
- /* remove an item from buffer to next consumed */
- . . .
- signal(mutex); // release Mutex
- signal(empty); //value ++1
- . . .
- /* consume the item in next consumed */
- . . .
- } while (true);
- **Figure 5.10** The structure of the consumer process

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- ***Abstract data type -- ADT***, internal variables only accessible by code within the procedure
- ADT -- encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
- 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 (…) { … }
  }
}
```

Condition Variables

An additional synchronization mechanisms – Condition construct

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 exactly one of suspended processes (if any) that invoked **`x.wait()`**

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. If Q is resumed, then P must wait
- Otherwise, both P and Q would be active simultaneously within the monitor. The 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 – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Linux Synchronization

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

```
/*Ruuner + Mutex + Semaphore*/
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int sum;/*this data is shared by the threads*/
/* The mutex lock */
pthread_mutex_t mutex;
/* the semaphores */
sem_t one, two;
pthread_t tid1,tid2;       //Thread ID
pthread_attr_t attr; //Set of thread attributes

void *runnerOne(void *param);/*thread one call this function*/
void *runnerTwo(void *param);/*thread two call this function*/
void initializeData();

int main(int argc, char*argv[])
{
 // pthread_t tid;/*the thread identifier*/
 // pthread_attr_t attr;/*set of thread attributes*/

  if(argc!=2){
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
  }
  if(atoi(argv[1])<0){
        fprintf(stderr,"%d must be >=0\n",atoi(argv[1]));
        return -1;
  }
```

```
initializeData();

/*get the default attributes*/
  pthread_attr_init(&attr);

pthread_create(&tid1,&attr,runnerOne,argv[1]); /*create the thread*/
printf("sum=%d\n",sum);

pthread_create(&tid2,&attr,runnerTwo,argv[1]); /*create the thread*/

sem_post(&two); /* unlcok the 'two'semaphore */

//sem_post(&one); /* unlcok the 'one'semaphore */

               /*wait for the thread to exit*/
               pthread_join(tid1,NULL);
               pthread_join(tid2,NULL);
               printf("sum=%d\n",sum);
         }/* end of Main */

void initializeData() {
   /* Create the mutex lock */
   pthread_mutex_init(&mutex, NULL);

   /* Create the 'one' semaphore and initialize the value = 0 the subroutine
call is blocked*/   sem_init(&one, 0, 0); //

   /* Create the 'two' semaphore and initialize the value = 0 the subroutine
call is blocked */
   sem_init(&two, 0, 0);

   /* Get the default attributes */
   pthread_attr_init(&attr);
}
```

```c
/*The thread one will begin control
in this function*/

void *runnerOne(void *param)
{
  /* acquire the 'one' semaphore*/
  sem_wait(&one);
  /* acquire the mutex lock */
  pthread_mutex_lock(&mutex);

  int i,upper=atoi(param);
  sum=0;


 for(i=0;i<=upper;i++)
    sum+=2*i;

  printf("thread one\n");
  /* release the mutex lock */
  pthread_mutex_unlock(&mutex);
  /* release and unlock the 'two'
semaphore */
  sem_post(&two);

}
```

```c
/*The thread two will begin control
in this function*/

void *runnerTwo(void *param)
{
  /* acquire the 'two' semaphore */
  sem_wait(&two);
  /* acquire the mutex lock */
  pthread_mutex_lock(&mutex);

  int i,upper=atoi(param);
  sum=10;

  for(i=0;i<=upper;i++)
    sum+=i;
  printf("thread two\n");
  /* release the mutex lock */
  pthread_mutex_unlock(&mutex);
  /* release and unlock the 'one'
semaphore */
  sem_post(&one);
}
```