

# Chapter 4: Threads

---





# Pthreads Code for Joining 10 Threads

---

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

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





# Implicit Threading

---

- ❑ Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- ❑ Creation and management of threads done by compilers and run-time libraries rather than programmers
- ❑ Three methods explored
  - ❑ Thread Pools
  - ❑ OpenMP
  - ❑ Grand Central Dispatch
- ❑ Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package





# Thread Pools

- ❑ Create a number of threads in a pool where they await work
- ❑ Advantages:
  - ❑ Usually slightly faster to service a request with an existing thread than create a new thread
  - ❑ Allows the number of threads in the application(s) to be bound to the size of the pool
  - ❑ Separating task to be performed from mechanics of creating task allows different strategies for running task
    - ▶ i.e. Tasks could be scheduled to run periodically
- ❑ Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {  
    /*  
    * this function runs as a separate thread.  
    */  
}
```





# OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

**#pragma omp parallel**

Create as many threads as there are cores

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

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

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```





# Grand Central Dispatch

---

- ❑ Apple technology for Mac OS X and iOS operating systems
- ❑ Extensions to C, C++ languages, API, and run-time library
- ❑ Allows identification of parallel sections
- ❑ Manages most of the details of threading
- ❑ Block is in “^{}” - `^ { printf("I am a block"); }`
- ❑ Blocks placed in dispatch queue
  - ❑ Assigned to available thread in thread pool when removed from queue





# Grand Central Dispatch

---

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called **main queue**
    - ▶ Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - ▶ Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```





# Threading Issues

---

- ❑ Semantics of **fork()** and **exec()** system calls
- ❑ Signal handling
  - ❑ Synchronous and asynchronous
- ❑ Thread cancellation of target thread
  - ❑ Asynchronous or deferred
- ❑ Thread-local storage
- ❑ Scheduler Activations







# Semantics of `fork()` and `exec()`

---

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of `fork`
- `exec()` usually works as normal – replace the running process including all threads





# Signal Handling

- n **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- n 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
- n 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





# Signal Handling (Cont.)

---

- n Where should a signal be delivered for multi-threaded?
  - | Deliver the signal to the thread to which the signal applies
  - | Deliver the signal to every thread in the process
  - | Deliver the signal to certain threads in the process
  - | Assign a specific thread to receive all signals for the 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);
```





# Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - ▶ I.e. `pthread_testcancel()`
    - ▶ Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals





# Thread-Local Storage

---

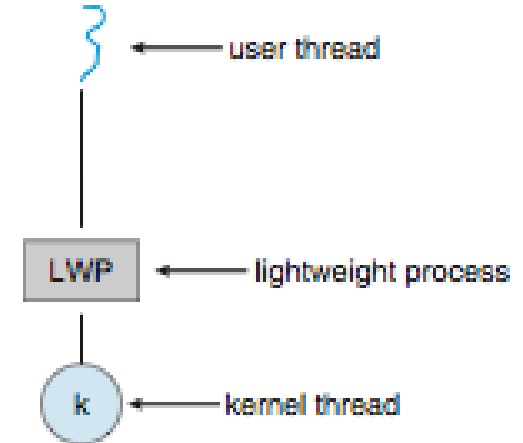
- ❑ **Thread-local storage (TLS)** allows each thread to have its own copy of data
- ❑ Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- ❑ Different from local variables
  - ❑ Local variables visible only during single function invocation
  - ❑ TLS visible across function invocations
- ❑ Similar to **static** data
  - ❑ TLS is unique to each thread





# Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





# Reading Assignment

---

□ Section 4.4.2 & 4.4.3

□ Section 4.7





# End of Chapter 4

---





# Interprocess Communication

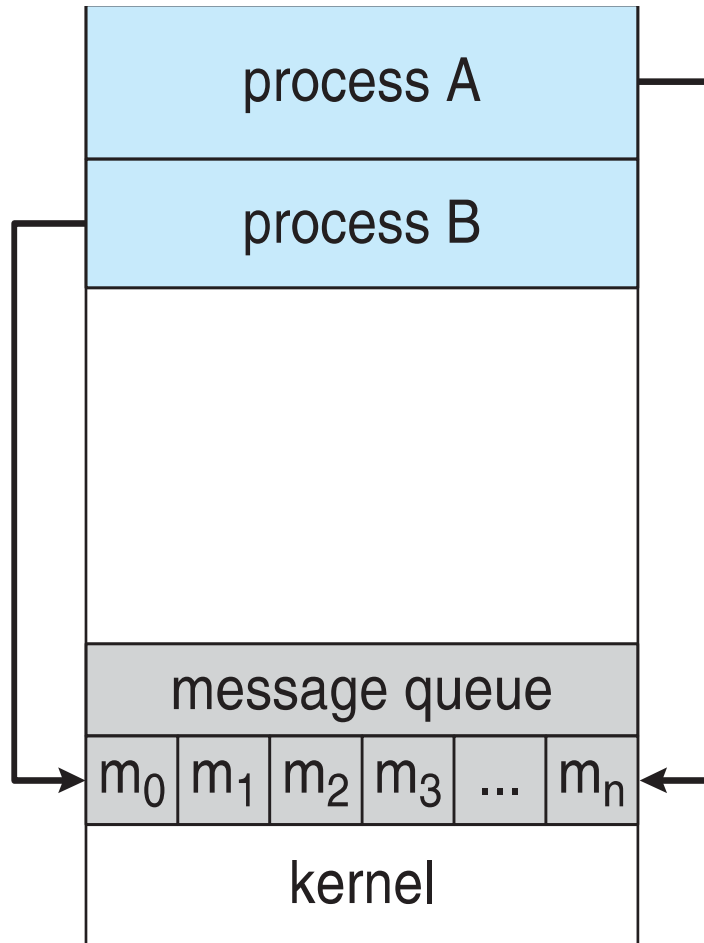
- ❑ Processes within a system may be **independent** or **cooperating**
- ❑ Cooperating process can affect or be affected by other processes, including sharing data
- ❑ Reasons for cooperating processes:
  - ❑ Information sharing
  - ❑ Computation speedup
  - ❑ Modularity
  - ❑ Convenience
- ❑ Cooperating processes need **interprocess communication (IPC)**
- ❑ Two models of IPC
  - ❑ **Shared memory**
  - ❑ **Message passing**



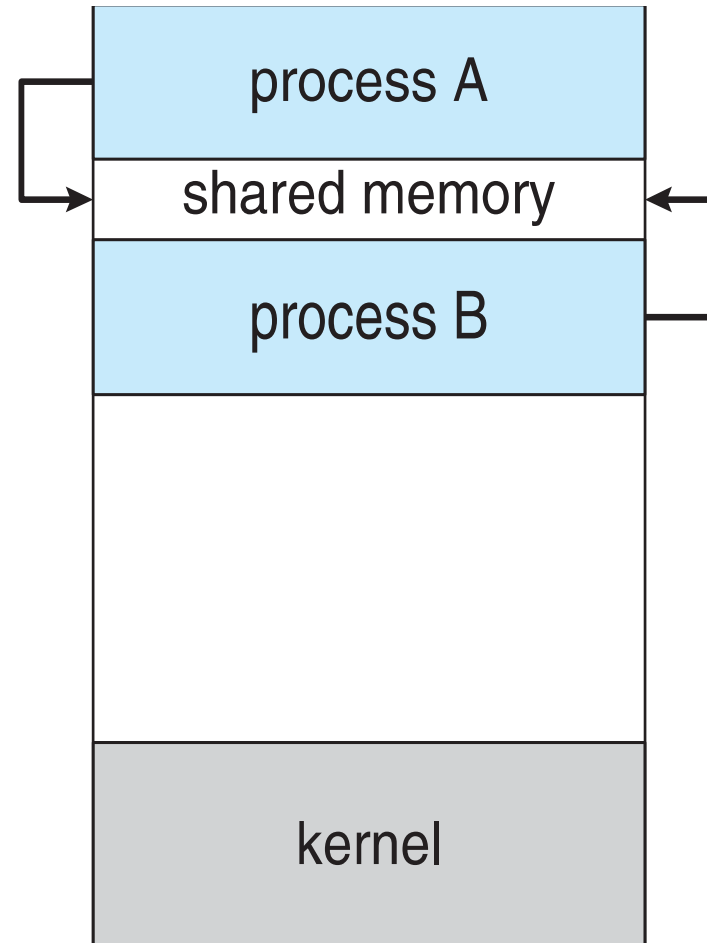


# Communications Models

(a) Message passing. (b) shared memory.



(a)



(b)





# Producer-Consumer Problem

---

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size





# Bounded-Buffer – Shared-Memory Solution

## □ Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0; //points to next free position
int out = 0; // points to the first full
             //position in the buffer
```

## □ Solution is correct, but can only use BUFFER\_SIZE-1 elements





# Bounded-Buffer – 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 – 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 */
}
```

