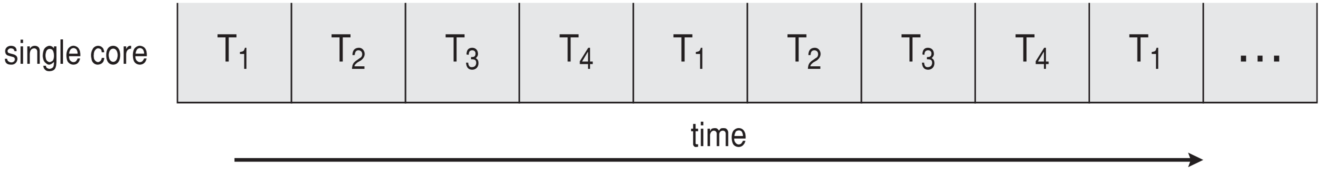
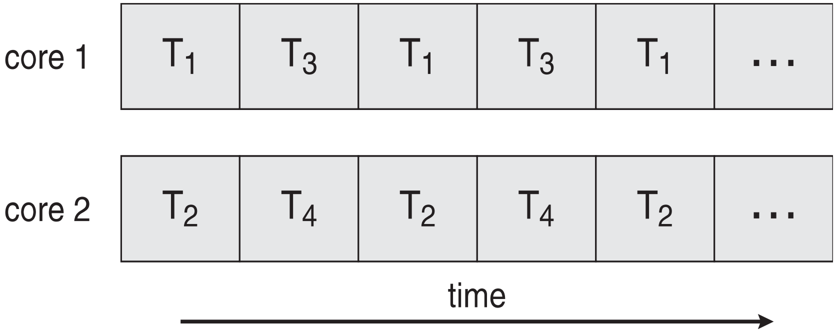
Concurrency vs. Parallelism

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



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



Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

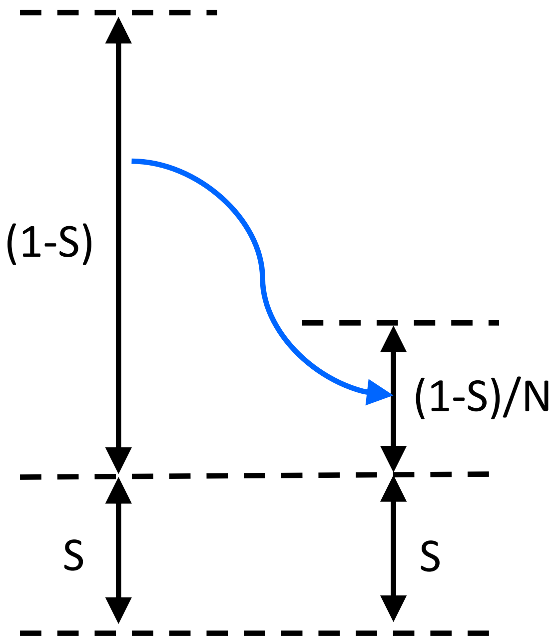
Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores



User Threads and Kernel Threads

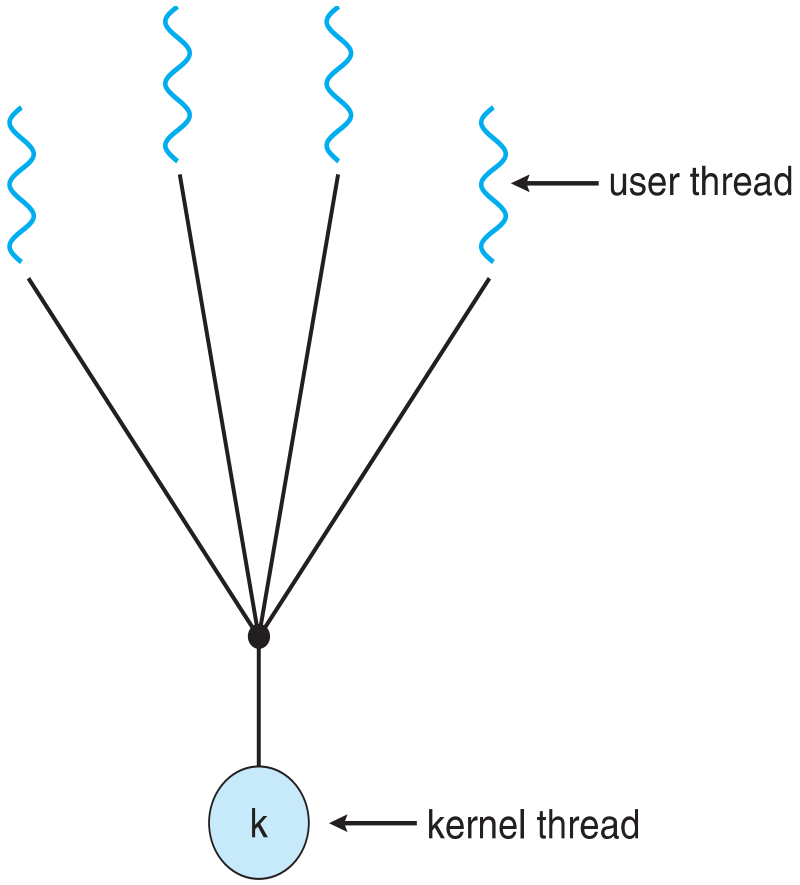
- **User threads** – Thread management takes place using a threads library without OS support.
  - Kernel would treat the process as single-threaded and any blocking system call by one of the threads would end up blocking all the threads of that process.
- **Kernel threads** - Supported and managed by the OS Kernel. Kernel support exists for most well-known Oses, e.g.:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
- **NOTE:** A kernel thread is not meant to indicate a thread executing kernel code, but rather a thread that is managed and supported by the kernel.

4.3 Multithreading Models

- Several thread management models exist:
  - Many-to-One model
  - One-to-One model
  - Many-to-Many model

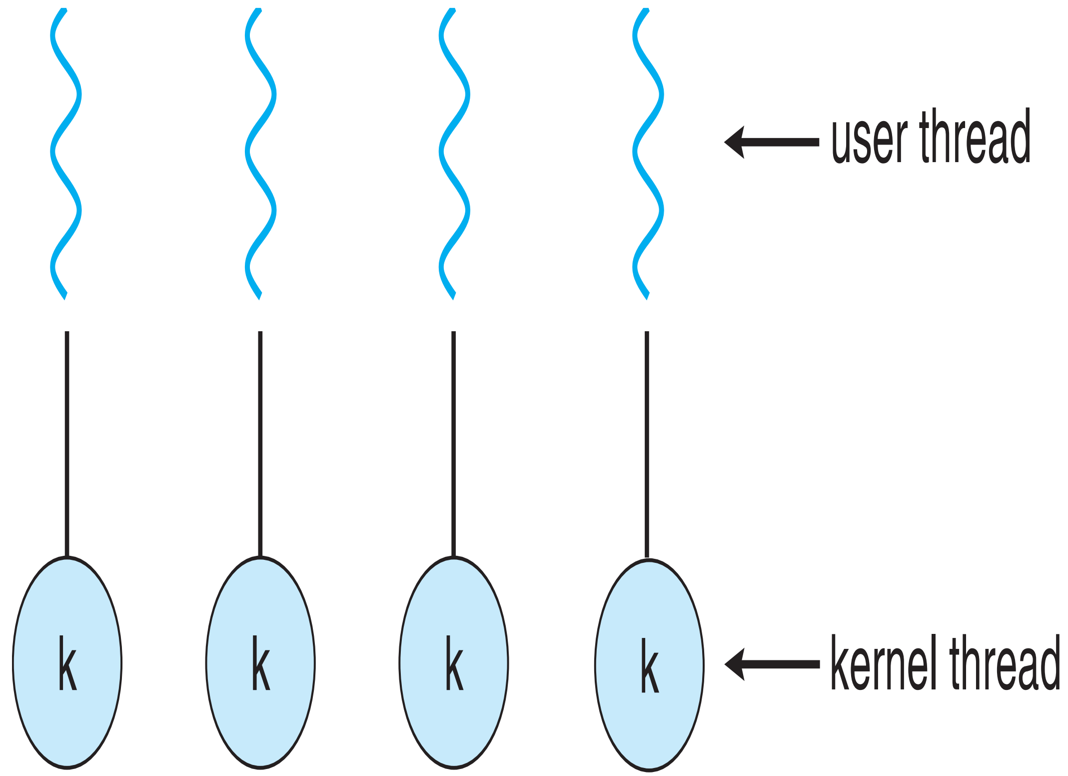
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread **blocking** causes all to block
- Multiple threads may **not run in parallel** on multicore systems 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

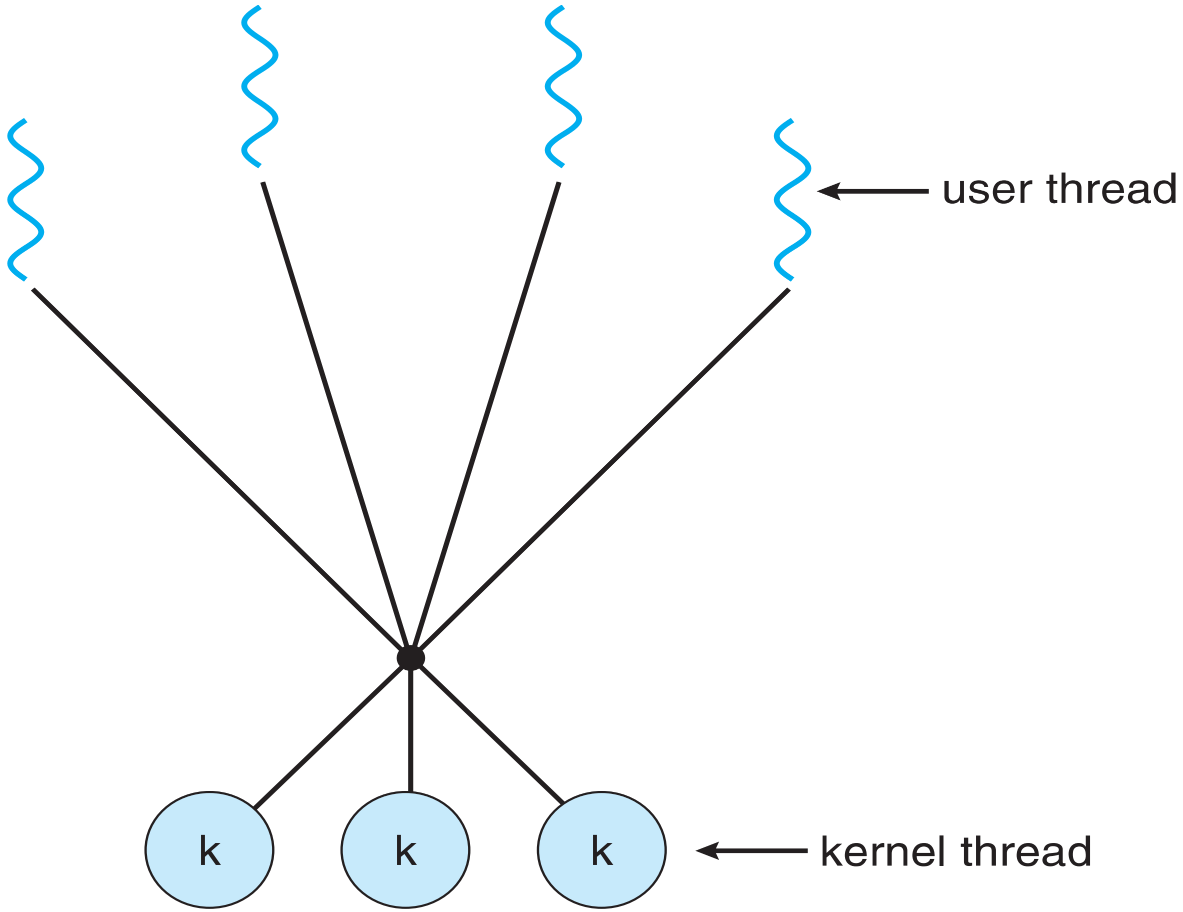
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

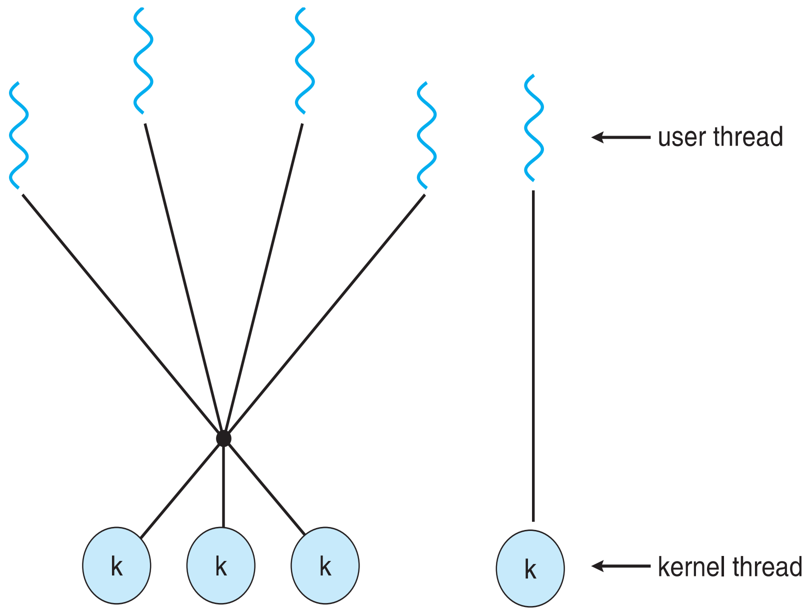
- Allows many user level threads to be mapped to many kernel threads;

  # kernel threads ≤ # user threads
- Allows the operating system to create a sufficient number of kernel threads
- Examples:
  - Windows with the *ThreadFiber* package
  - Solaris prior to version 9



Many-to-Many variant: The Two-level Model

- Similar to the many-to-many model in that it allows many user threads to map to many kernel threads.
- But it also allows a one-to-one relationship on some user/kernel thread pairs as shown on the diagram.
- Examples
  - HP-UX
  - Tru64 UNIX
  - Solaris prior to version 8



4.4 Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads and is responsible for implementing a thread management model
- Three primary thread libraries:
  - POSIX **Pthreads** (whether user/kernel thread is platform dependent, but same interface)
  - Windows threads (kernel threads)
  - Java threads (user threads)
- Two primary ways of implementing
  - Library entirely in user space (i.e. with no kernel support)
  - Kernel-level library supported by the OS

4.4.1 Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- May be implemented either as user-level or kernel-level.
- ***Specification***, not ***implementation***
  - Different implementation for different OS platforms, but all have the same interface.
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX-like 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) {
    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;
  }
```

```
  /* 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);
}
```

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);
```

4.4.2 Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Windows Multithreaded C Program (Cont.)

```
    /* create the thread */
    ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

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

Windows Multithreaded C Program (Cont.)

The creating thread can use the arguments to [CreateThread] to specify the following:

- The security attributes for the handle to the new thread, including:
  - An **inheritance flag** that determines whether the handle can be inherited by child processes.
  - A **security descriptor**, which the system uses to perform access checks on all subsequent uses of the thread's handle before access is granted.
- The initial **stack size** of the new thread.
  - The thread's stack is allocated automatically in the memory space of the process
  - The system increases the stack as needed and frees it when the thread terminates.
- A creation flag that enables you to create the thread in a **suspended** state. When suspended, the thread does not run until the [ResumeThread] function is called.

4.5 Implicit Threading

- Growing in popularity; As the number of threads increases, program correctness becomes 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) and `java.util.concurrent` package

4.5.1 Thread Pools

- **At process startup**, create a **predefined** number of threads in a pool where they await work.
- Threads wait for work to be dispatched to them. When work is dispatched to the thread it performs it and when done returns back to the pool.
- If more work needs to be dispatched with no threads available in the pool, the main process waits till a thread becomes available to the pool.
- Advantages:
  - Servicing a request with an existing thread is **faster than creating** a new thread
  - Allows the number of threads in the application(s) to be **bound** to the size of the pool (thus preventing the creation of too many threads)
  - Separating tasks to be performed **from mechanics of creating task allows** different strategies for running tasks
    - e.g. Tasks may be scheduled to run as **one-shot** after a delay time
    - or may be scheduled to run **periodically**
- Works very well for tasks that have a finite duration, i.e. tasks that start, do some work, then exit.

Thread Pools – cont.

- **The number of threads** in a pool may be **pre-determined** based on the number of CPUs, memory or the expected number of client requests.
- Alternatively, more sophisticated systems may adjust the number of threads **dynamically.**
- Windows API supports thread pools – The user may call an API for dispatching work to a thread using the function:

```
BOOL QueueUserWorkItem(LPTHREAD_START_ROUTINE
    Function, PVOID Param, ULONG Flags );
```

- The function may take the form:

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

4.5.2 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 using:

  `#pragma omp parallel`

- Creates as many threads as there are cores

```
#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;
}
```

OpenMP cont.

- The following runs a for-loop in parallel

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

- OpenMP allows developers to control the level of parallelism by allowing:
  - Manual setting of the **number of threads**.
  - Specify certain data as **shared** or private
- OpenMP is available for open-source as well as commercial compilers:
  - Linux, Windows and Mac OS X.

4.6 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

4.6.1 Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIX systems have two versions of fork
  - In Linux, if a process has multiple threads and one of them calls `fork()`, the child thread will have a replica of the parent's code, data, stack, heap, file and other resources BUT will only have one thread running, the one that called the `fork()` function.
- **`exec()`** usually works as normal – replace the running process **including all its threads**

4.6.2 Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- Synchronous signals are those caused by the running process (e.g. divide by zero, or memory illegal memory access). Asynchronous signals are caused outside the program (e.g. upon an expiration of a timer after the process issues an `alarm()` call).
- A **signal handler** is a function that is used to process/handle signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Each signal is handled by one of two signal handlers:
     1. default
     2. user-defined (provided by the process)
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, the signal is delivered to process's main thread → no issues

Signal Handling (Cont.)

- 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
- The standard linux function for delivering signals is:

```
Kill (pid_t pid, int signal)
```

- Most Unix versions allow a **thread to specify which signals it accepts** (else the signal is blocked), and a signal is **delivered ONLY ONCE to the first** thread that accepts it.
- POSIX allows signal delivery to a specified thread

```
pthread_kill(pthread_t tid, int signal);
```

- Windows (which doesn't have signals) uses Asynchronous procedure calls (APC)
  - Allows a thread X to specify an APC function to another thread Y.

```
DWORD QueueUserAPC(PAPCFUNC pfnAPC, HANDLE hThread, ULONG_PTR dwData );
```

  - Windows calls the APC function once thread Y blocks for an event, semaphore or any synchronization object.

4.6.3 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 and thus terminates orderly.
- Difficulty: When a thread is allocated resources or shared data. When canceling a thread asynchronously, the OS may reclaim only system resources. Also a thread may be in the middle of updating some shared data.

Thread Cancellation - cont.

- Pthread code to create and cancel a thread:

```
pthread tid;
/* create the thread */
pthread_create(&tid, 0, worker, NULL);
…
/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- In pThreads, 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.
  - `pthread_setcanceltype(..)` – to set thread's mode
  - `pthread_setcancelstate(..)` – to set thread's state
- Default is deferred and enabled
  - Cancellation only occurs when thread reaches **cancellation point,**
    - By calling `pthread_test_cancel()` to establish a cancellation point (after it frees the assigned resources or gracefully stops manipulating shared data).
    - After calling pthread_join()
    - After calling sigwait() or pthread_cond_wait()
- On Linux systems, thread cancellation is handled through signals

4.6.4 Thread-Local Storage

- Generally threads within the same process share their data, but sometimes a thread may have some data that is not to be shared (e.g. tabbed webpages each with a thread) -> it needs **Thread-local storage** (**TLS**).
- Local variables within a function do not directly achieve that, since they do not propagate across function calls.
- Most thread libraries (pthreads and Windows API) provide some form of support.
- For pthreads use the `__thread` specifier prior to variable declaration, e.g.

```
__thread int i;
```

  This requires significant support from the compiler (gcc), linker (ld), dynamic linker (ld.so) and system libraries (libc.so and libpthread.so)

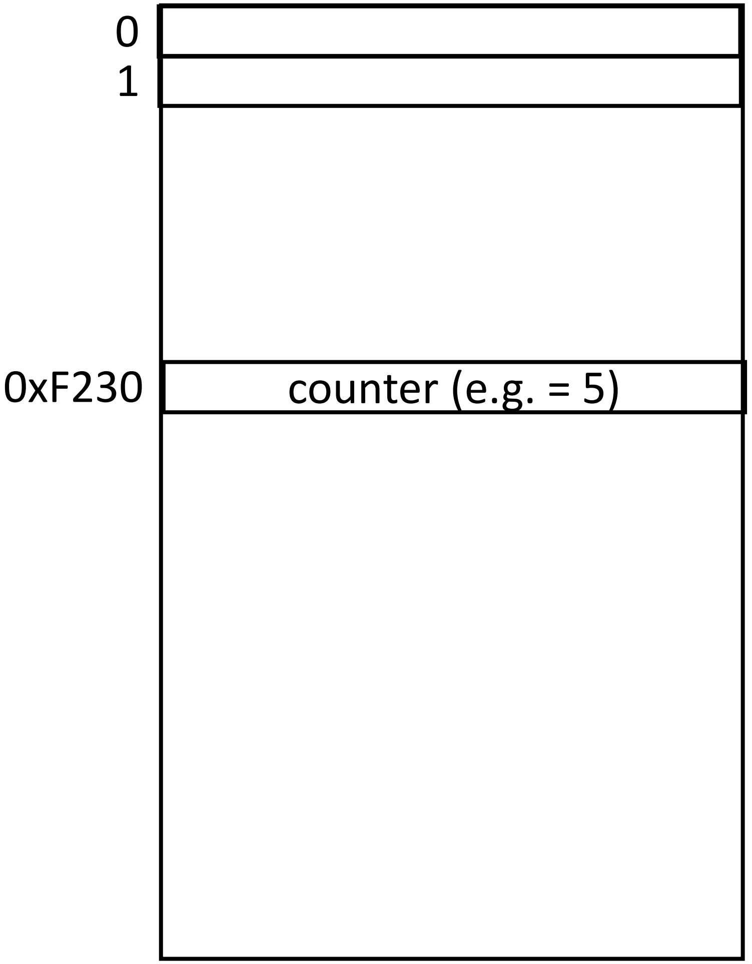
Race Conditions

```
int counter=0;
void thread1()
{
    .
    .

    counter++;

    .
    .
}
```

e.g. if #counter = 0xF230



Memory

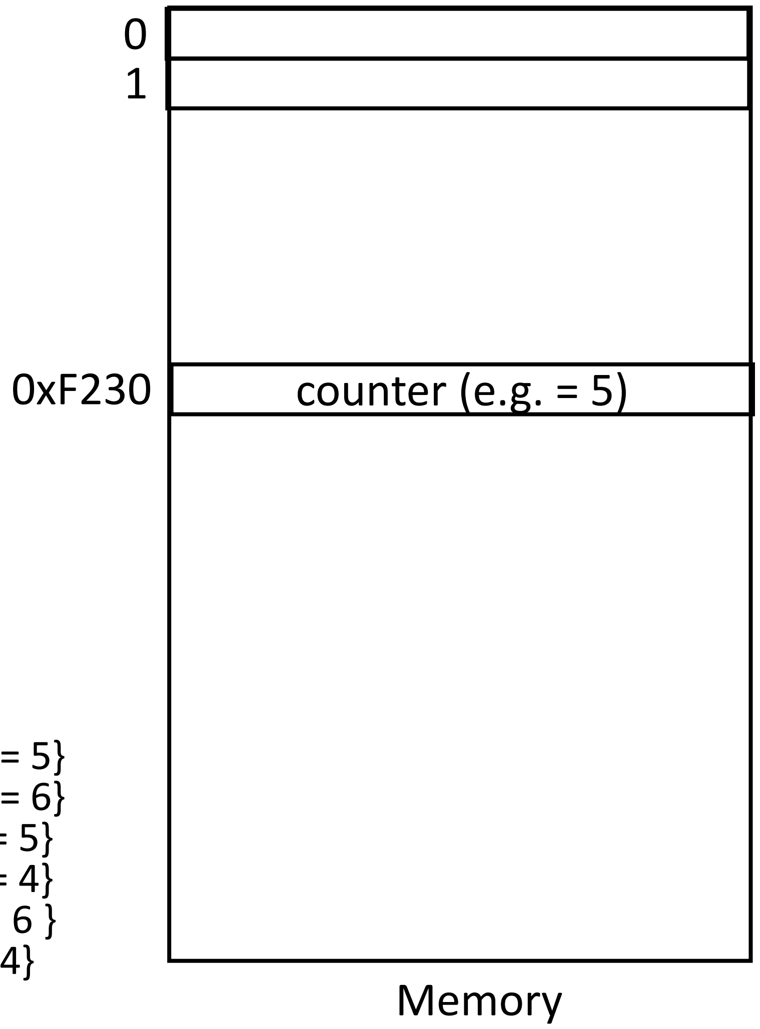
```
void thread2()
{
    .
    .

    counter--;

    .
    .
}
```

Race Conditions

e.g. if #counter = 0xF230



`counter++` could be implemented **in machine code** as

```
mov r2, #counter

ld  r1,[r2]          register1 = counter
inc r1               register1 = register1 + 1
st  r1,[r2]          counter = register1
```

- **`counter--`** could be implemented **in machine code** as

```
mov r3,#counter

ld  r2,[r3]          register2 = counter
dec r2               register2 = register2 - 1
st  r2,[r3]          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}

Race Conditions – cont.

- A **race condition** or **race hazard** is the behavior of a software (or hardware) system where the output is dependent on the sequence or relative timing of the executing threads.
- A **critical race condition** occurs when the order of operations on shared variables causes them to have unexpected or **erroneous** values.
- A **non-critical race condition** occurs when the order of operations on shared variables does not result in an unexpected or erroneous value.
- Critical race conditions result in invalid execution and bugs. Failure to obey mutual exclusion opens up the possibility of corrupting the shared variables.

Race Conditions - cont.

- A critical race condition occurs when **multiple threads** are performing **non-atomic read-modify-write** concurrently or in parallel.

- It is not necessary for two threads writing to a shared variable concurrently, to result in a critical race condition. A read-modify-write needs to exist to cause a critical race condition.

- Examples of read-modify-write operations:
  - Increment and decrement (e.g. counter++, counter--)
  - test-and-set
  - compare-and-swap
  - accumulate operations (e.g. counter+=4)

Race Conditions – cont.

- Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads.
- Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger. Thus, a bugs that is due to a race condition is often referred to as a "Heisenbug".
- Thus, it is better to avoid race conditions in the first place and there is no alternative to proper and careful software design.

5.2 Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code (the section that manipulates shared variables using read-modify-write operations)
  - A Process may be changing common variables, updating a table, writing file, etc
  - To avoid race conditions, when one process is in critical section, no other should be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section. This happens in the **entry section**. It may follow critical section with an **exit section**, then **remainder section**

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

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist 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.
   - Only processes that are in their entry section can participate in the selection.
3. **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 its request is granted)
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

5.3 Peterson's Solution

- Good algorithmic description of solving the problem
- **Two process solution** (i.e. works for two processes only)
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted (a reasonable assumption)
- The two processes share two variables:
  - `int turn;`
  - `bool req[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `req` array is used to indicate if a process is ready to enter the critical section. `req[i] = true` implies that process P_i is ready!

Algorithm for Process P_i

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

Algorithm for Process P_i

```
do {
    req[0] = true;
    turn = 1;
    while (req[1] && turn == 1);
    critical section
    req[0] = false;
    remainder section
 } while (true);
```

```
do {
    req[1] = true;
    turn = 0;
    while (req[0] && turn == 0);
    critical section
    req[1] = false;
    remainder section
 } while (true);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`reg[j]==false`** or **`turn==i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

- **<u>Note:</u>** The two threads are setting the turn variable. This is ***<u>not</u>*** a read-modify-write and does not result in a critical race condition.

5.4 Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All H/W solutions described in this section are based on idea of **locking**
  - Protecting critical regions via locks
- **Uniprocessors** – could **disable interrupts**
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems since it requires sending a disable interrupts message to all cores.
    - Operating systems using this approach are not broadly scalable
- Modern machines provide special **atomic hardware instructions**
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

Process A

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

Process B

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

test_and_set Instruction

Definition:

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

1. Executed atomically (**it is a single machine instruction**)

it is a single machine instruction

1. Returns the original value of the lock variable (`*target`)
2. Set the new value of lock variable (*target) to “TRUE”.

Using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- A possible solution to critical section problem?

```
do {
   /* Wait till lock is false i.e. not locked, then acquire it */
   while (test_and_set(&lock));


   /* critical section */

   . . .

   /* release the lock at the end (i.e. make it false) */

   lock = false;


   /* remainder section */

   . . .


} while (true);
```

fetch_and_add Instruction

Definition:

```
int fetch_and_add (int *target, int inc)
{
    int rv = *target;
    *target = *target + inc;
    return rv:
}
```

1. Executed atomically (**it is a single machine instruction**)
2. Returns the original value of the lock variable (*`target`)
3. Set the new value of (*target) to (*target) + inc.

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int rv = *value;

    if (*value == expected)
       *value = new_value;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of the lock variable `(*value)`
3. Set the variable "value" the value of the passed parameter "new_value" but only if "*value" =="expected". That is, the swap takes place only under this condition.

Using compare_and_swap

- Shared integer “lock” initialized to 0;
- A possible solution to critical section problem?

```
do {
   /* Wait for value to be zero (i.e. lock is released), then acquire lock */
   while (compare_and_swap(&lock, 0, 1) != 0);

   /* critical section */
   . . .
   /* release the lock when done with CS */
   lock = 0;

   /* remainder section */
   . . .
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

- **Previous algorithms didn't satisfy the bounded wait requirement.**
- This algorithm uses common data structures:
  `bool waiting[n];`
  `bool lock;`
- The variable `Key` is not shared
- Proof of mutual exclusion:
  - P_i can enter its critical section only if either `waiting[i]== false` OR `key==false`.
  - The value of `key` can become false only if `test_and_set()` is executed. The first process to execute it will find `key == false`; all others must wait.
  - The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

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

  /* critical section */
  . . .
  /* Select next process to run
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = false;
  else
    waiting[j] = false;
  /* remainder section starts below*/
  . . .
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

- Proof of progress:
  - Since a process exiting the critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.
- Proof of bounded wait:
  - When a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1$, $i + 2$, ..., $n - 1$, 0, ..., $i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] ==true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

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

 /* critical section */
  . . .
 /* Select next process to run
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

 if (j == i)
    lock = false;
  else
    waiting[j] = false;
  /* remainder section starts below*/
  . . .
} while (true);
```