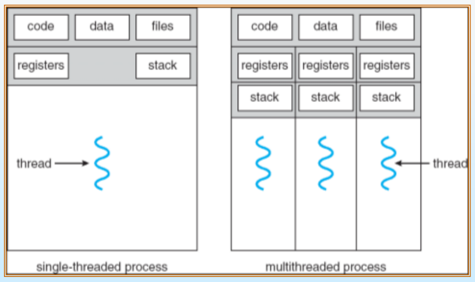
Threads

Concepts and Implementation

Threads

- Multithreading Models
- Threading Issues
- Why Threads
- Thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Single and Multithreaded Processes



Benefits

- Responsiveness
- Resource (memory, file, etc.) Sharing
- Economy (less synchronization overhead)
- Utilization of MP Architectures

Benefits-1

- A traditional OS process is one thread of execution only.
- Multiple threads allow multiple thread of computation in the same address space.
- Decomposing applications into concurrent threads, may reduce process or application blocking
- It may even simplify the programming
- Easier to create and destroy threads
- They may offer performance gain
- It is possible to utilize multiple CPUs

Benefits-2

- Performance gains from multiprocessing hardware (parallelism)
  - Different threads can run on different processors simultaneously with no special input from the user and no effort on the part of the programmer

Benefits -3

- An application that uses multiple processes can be replaced by an application that uses multiple threads to accomplish the same tasks, if possible.

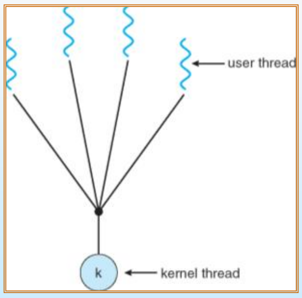
User/Kernel Threads

- User level thread management done using user-level threads library
  - Primary user level thread libraries:
    - UNIX and Linux: POSIX Pthreads
    - Windows: Win32 threads
    - JVM: Java threads
- Kernel level threads are managed by the operating system itself

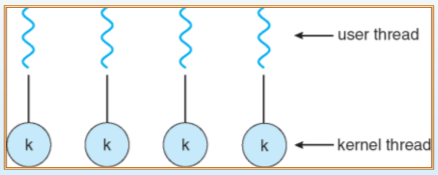
User and Kernel level Multithreading Models: Done by the kernel

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One Model



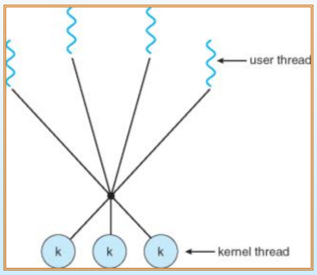
One-to-one Model



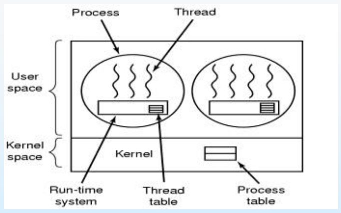
Many-to-Many Model

- 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 NT/2000 with the *ThreadFiber* package

Many-to-Many Model



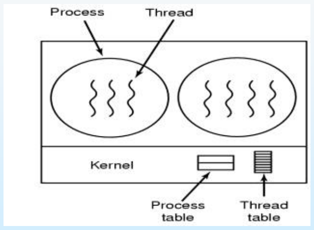
Implementing Threads in User Space



Implementing Threads in User Space

- A user-level threads package:
- Kernel knows nothing about threads.
- The user level library can manage a thread table to allow orderly execution. If a thread blocks it is state is saved in the table and a ready thread is allowed to execute…
- The application has its own thread scheduler…
- **Result: Efficient implementation is very hard to achieve, virtually impossible…**

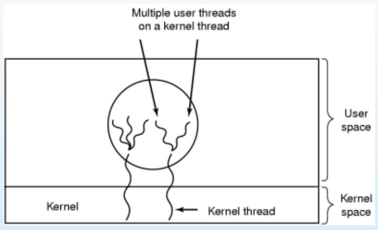
Kernel level threads



Kernel level threads

- A threads package managed by the kernel:
- Kernel has a thread table to keep record of all the threads in the system
- A thread library is part of the system call library.
- Kernel level thread management is much more expensive, as blocking is handled similar to multiple process case concept…

Hybrid Implementations



Multiplexing user-level threads onto kernel- level threads: using advantages of user/kernel level threads

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Pools

- Create a number of threads in a pool where they await to be activated.
- 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

Thread Specific Data

- Each thread can have its own copy of data
- This is useful when there is no control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Scheduler maintains an appropriate number of kernel threads allocated to an application
- Scheduler activations provide **upcalls**: a communication mechanism from the kernel to the thread library
  - This communication allows an application to maintain the correct number kernel threads

What is Pthreads

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
  - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
  - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
  - Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

Pthreads

- Some useful links:

  standards.ieee.org/findstds/standard/1003.1-2008.html
  www.opengroup.org/austin/papers/posix_faq.html
  www.unix.org/version3/ieee_std.html
- 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 are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library

The Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Per process: Items shared by all threads in a process
- Per thread: Items private to each thread

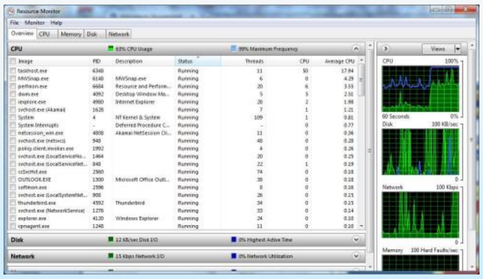
Common Reason for Threads (1)

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. The difference could be up to 20:1
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications :
  - Overlapping CPU work with I/O:
    - Aprogram may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.

Common Reason for Threads (2)

- Priority/real-time scheduling:
  - tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling:
  - interleaving. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
  - A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Example: WINDOWS: Start button/ All apps/Windows Administrative Tools/Resource Monitor.



CPU 🟩 63% CPU Usage 🟦 99% Maximum Frequency

Image	PID	Description	Status	Threads	CPU	Average CPU
taskhost.exe	6348		Running	11	50	17.84
MWSnap.exe	6148	MWSnap.exe	Running	6	0	4.29
perfmon.exe	6684	Resource and Perform...	Running	20	6	3.55
dwm.exe	4092	Desktop Window Ma...	Running	5	5	2.51
iexplore.exe	4980	Internet Explorer	Running	28	2	1.98
svchost.exe (Akamai)	1628		Running	7	1	1.21
System	4	NT Kernel & System	Running	109	1	0.81
System Interrupts	-	Deferred Procedure C...	Running	-	0	0.77
netsession_win.exe	4808	Akamai NetSession Cli...	Running	11	0	0.36
svchost.exe (netsvcs)	948		Running	48	0	0.28
policy.client.invoker.exe	1992		Running	4	0	0.26
svchost.exe (LocalServiceNo...	1464		Running	20	0	0.25
svchost.exe (LocalServiceNet...	848		Running	22	1	0.19
ccSvcHst.exe	2560		Running	74	0	0.18
OUTLOOK.EXE	1300	Microsoft Office Outl...	Running	38	0	0.18
softmon.exe	2596		Running	8	0	0.18
svchost.exe (LocalSystemNet...	908		Running	26	0	0.15
thunderbird.exe	4592	Thunderbird	Running	34	0	0.15
svchost.exe (NetworkService)	1276		Running	33	0	0.14
explorer.exe	4120	Windows Explorer	Running	24	0	0.10
vpnagent.exe	1248		Running	11	0	0.10

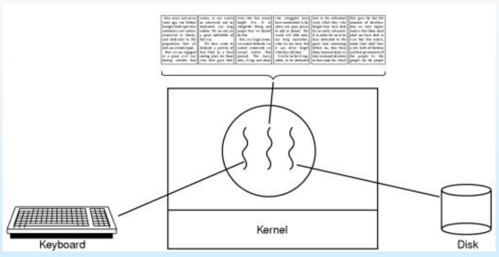
Disk 🟩 12 KB/sec Disk I/O 🟦 0% Highest Active Time

Network 🟩 15 Kbps Network I/O 🟦 0% Network Utilization

Suitability for thread implementation

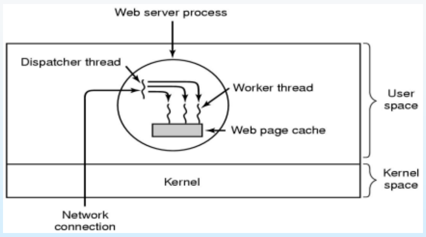
- Work that can be executed, or data that can be operated on, by multiple tasks simultaneously:
- Block for potentially long I/O waits
- Use many CPU cycles in some places but not others
- Must respond to asynchronous events
- Some work is more important than other work (priority interrupts)

Thread Usage (1)



A word processor with three threads: formatting, user interaction, backup threads

Thread Usage: web server

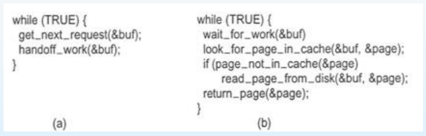


- A multithreaded Web server: maintains a cache to hold the frequently requested pages

Thread Usage (3)

Web Dispatcher Thread

Worker thread: client



```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
          (a)
```

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
     read_page_from_disk(&buf, &page);
  return_page(&page);
}
          (b)
```

- Rough outline of code for the web server
  - (a) Dispatcher thread: handles the incoming web page requests
  - (b) Worker thread, when woken up checks to see if the requested page in the cache, if not it reads from the disk before delivering

    One worker would have been inefficient as it has to block until IO is complete. Thus multiple worker implementation should be preferred.

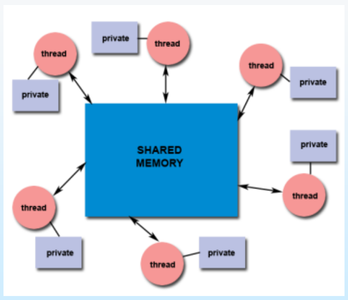
common models for threading

- ***Manager/worker:*** a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks.
- ***Pipeline:*** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
- ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work just like others.

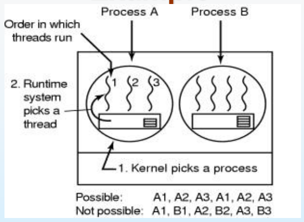
Shared Memory Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.

Multi-thread appearance



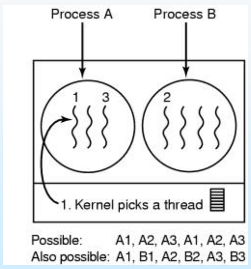
User Level Thread Scheduling Example


Process A
Process B
Order in which threads run
2. Runtime system picks a thread
1 2 3
1. Kernel picks a process
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Kernel Level Thread Scheduling Example

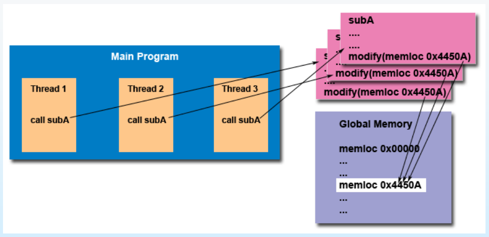


Process A
Process B
1
3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec per CPU burst

Thread-Safeness



- Suppose that your application creates several threads, each of which makes a call to the same library routine
- There is problem: This not thread safe

Thread Limits

- Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways not specified by the standard.
- Because of this, a program that runs fine on one platform, may fail or produce wrong results on another platform.
- For example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing your program.

pthread Example

```
/******************************************************************************
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* AUTHOR: Blaise Barney
* LAST REVISED: 08/09/11
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   5

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

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Passing Arguments to Threads

- The pthread_create() routine permits the programmer to pass one argument to the thread start routine.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine.
- All arguments must be passed by reference and cast to (void *).

Thread Multiple Argument Passing Example

```
/******************************************************************************
* FILE: hello_arg2.c
* DESCRIPTION:
*   A "hello world" Pthreads program which demonstrates another safe way
*   to pass arguments to threads during thread creation.  In this case,
*   a structure  is used to pass multiple arguments.
* AUTHOR: Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS                     8

char *messages[NUM_THREADS];

struct thread_data
{
   int                                        thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;

   sleep(1);
   my_data = (struct  thread_data *) threadarg;
   taskid  = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;

sum=0;
messages[0]  = "English: Hello World!";
messages[1]  = "French: Bonjour, le monde!";
messages[2]  = "Spanish: Hola al mundo";
messages[3]  = "Klingon: Nuq neH!";
messages[4]  = "German: Guten Tag, Welt!";
messages[5]  = "Russian:  Zdravstvytye,  mir!";
messages[6]  = "Japan: Sekai e konnichiwa!";
messages[7]  = "Latin: Orbis, te saluto!";

for(t=0;t<NUM_THREADS;t++) {
  sum = sum  + t;
  thread_data_array[t].thread_id = t;
  thread_data_array[t].sum = sum;
  thread_data_array[t].message = messages[t];
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
       &thread_data_array[t]);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
pthread_exit(NULL);
}
```