Operating Systems
CS F372

Threads

BIJU K RAVEENDRAN

Threading Issues

- The fork and exec system calls
  - If one thread in a system calls fork()
    - The new process duplicates all threads
    - The new process duplicates only the calling thread.
- Cancellation
  - Task of terminating the thread before it has completed.
  - Cancellation of target thread (the thread that is to be cancelled) may occur in 2 different scenarios
    - Asynchronous cancellation
      - terminates the target thread immediately
    - Deferred cancellation
      - allows the target thread to periodically check if it should be cancelled

```
#include<pthread.h>                    #include<unistd.h>
  #include<stdio.h>                    #include<sys/types.h>
  #include<asm/unistd.h>
void *runner(void *param);
int main(int argc,char *argv[])
{ pthread_t tid,tid1;
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_create(&tid,&attr,runner,argv[1]);
  pthread_create(&tid1,&attr,runner,argv[2]);
  printf("1st thread ID=%u & 2nd thread ID=%u\n",tid,tid1);
   if(!fork())
   {   printf("Child PID=%d, PPID=%d\n",getpid(),getppid());
       printf("Child TID=%d, PID=%d\n",syscall(__NR_gettid),getpid());
   }
```

```
else
{
        wait(NULL);
        printf("Parent:PID=%d,PPID=%d\n",getpid(),get
ppid());
        printf("Parent:TID=%d, PID=%d\n",
                syscall(__NR_gettid),getpid());
}
pthread_join(tid,NULL);
pthread_join(tid1,NULL);
return 0;
}
```

```
// runner function
void *runner ( void *param )
{       int upper=atoi(param);
        int i;
        int sum=0;
        if (upper>0)
        {       for ( i=1; i <= upper; i++ )
                {       sum = sum + i;   }
        }
        printf("From thread:Thread ID=%u,SUM=%d\t PID=%d,
        PPID=%d\n",pthread_self(),sum,getpid(),getppid());
        printf("From thread:TID=%d,PID=%d\n",
        syscall(__NR_gettid),getpid());
        pthread_exit(0);
}
```

Windows Multi-threaded C Program

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

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

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

   Param = atoi(argv[1]);
   /* 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 */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

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

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Linux Threads

- Linux 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

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.

Processes & Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent
- A distinction is only made when a new thread is created by the `clone` system call
  - `fork` creates a new process with its own entirely new process context
  - `clone` creates a new process with its own identity, but that is allowed to share the data structures of its parent
- Using `clone` gives an application fine-grained control over exactly what is shared between two threads

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
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

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
  - Signal may be handled by
    - A default signal handler
    - A user defined signal handler

Every signal has **default handler** that kernel runs when handling signal - **User-defined signal handler** can override default

Signal Handling

- Options:
  - 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 Pools

- Create a number of threads in a pool where they await work
- Unlimited threads could exhaust system resources and one solution for this is pooling.
  - Create a number of threads at process startup and place them into a pool. When the server require the thread it is allotted and after completion of task it will be back on pool.
- 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

Thread Specific Data

- Allows each thread to have its own copy of data like local variables
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

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

Thread Cancellation

- 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
  - Then cleanup is invoked
- On Linux systems, thread cancellation is handled through signals

Windows Threads

- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Thread Data Structures