

Threads

Thread:

a fundamental unit of CPU utilization

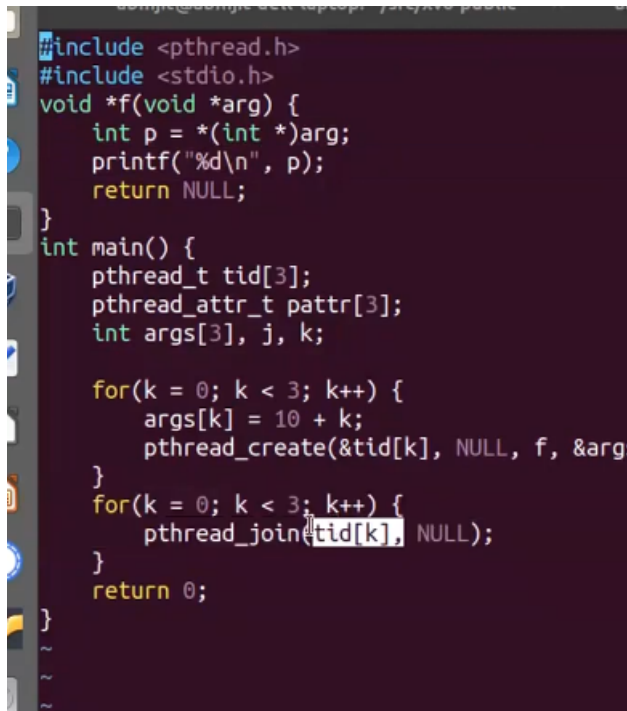
- separate control flow within prog
- set of instr exec “**concurrently**” with other part of code

Concurrently(progrss) \neq **Parallely**(execution)

parallelism is concurrency but concurrency is not parallelism.
Parallelism need extra h/w

Threads run within application

prog divide in multiple part
a thread run one these part



```
#include <pthread.h>
#include <stdio.h>
void *f(void *arg) {
    int p = *(int *)arg;
    printf("%d\n", p);
    return NULL;
}
int main() {
    pthread_t tid[3];
    pthread_attr_t pattr[3];
    int args[3], j, k;

    for(k = 0; k < 3; k++) {
        args[k] = 10 + k;
        pthread_create(&tid[k], NULL, f, &args[k]);
    }
    for(k = 0; k < 3; k++) {
        pthread_join(tid[k], NULL);
    }
    return 0;
}
```

Multiple task with appl can be implemented with separate threads

Process creation is heavy. Thread creation is light weighted
Kernel : multithreaded

Process = {Code + Data + Files + reg+ Stack}

Register + stack : not writen by user. Inserted by Compiler

Threads share **Code + data + files**

Have their **separate registers and stack**

Benefits of threads:

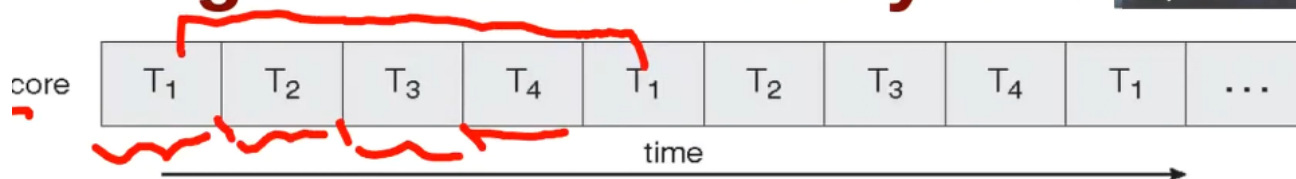
- Responsiveness

- Resource sharing

- economy : shared memory, global variable also shared

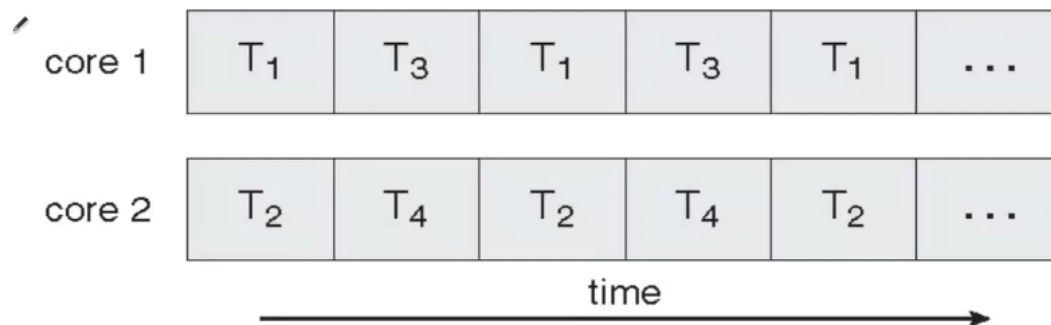
- scalability

Single vs Multicore system



Single core : Concurrency possible

Multicore : parallel execution possible



Multicore programming :

- have to face challenge

- dividing activity

- balan

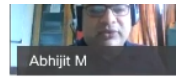
- data splitting

- data dependency

Process share the register by context switches.

Threads:

User vs Kernel Threads



- **User Threads: Thread management done by user-level threads library**
- **Three primary thread libraries:**
 - **POSIX Pthreads**
 - **Win32 threads**
 - **Java threads**
- **Kernel Threads: Supported by the Kernel**
- **Examples**
 - **Windows XP/2000**
 - **Solaris**
 - **Linux**
 - **Tru64 UNIX**
 - **Mac OS X**

User Threads :

typedef called as “threads”

scheduling done in user level library.

Needs timer handling functionality at user lvl exec of CPU

kernel not aware of these threads

Kernel threads :

implements concepts of thread

usr lvl lib to map kernel's concept to usr lvl library **as appl link** with usr libs.

KERNEL does Scheduling

Multithreading Models:

- Many-to-one
- One-to-one
- Many-to-many

If no kernel threads : only '1' proc.

Many-one mapping done by usr lvl thread lib.

Many-One model :

many usr lvl thread mapped to single kernel thread.

// actually each usr thread runs on CPU, each runs as part of one proc. each will have timer, interr, and scheduling.

e.g. solaris green

Problem:

if a one of proc calls read sys call, then kernel reads and it is block. And proc is in waiting Q. Now other proc can't be scheduled hence stopped.

One-One model:

each usr lvl thread map to kernel thread.

Usr thread internally calls sys. call of kernel to make thread

ex. Windows NT, Linux, solaris

Here if one is blocked then other may continue. But lot of overhead of kernel threads

Many-Manys:

lib in between creates as many threads needed.

Allows many usr lvl thrd to map to many kernel thrd
allows OS to create suff. number of kernel threads

eg.solaris prior to v9

Here blocking is handled as we have other threads to use if one is blocked.

Two level model:

many-many + one-one

similar to many-many but allow user to bound usr thrd to kernel thread.

Thread libraries:

provides Programmer API for creating and managing threads.

2 ways :

- lib in usr space
- kernel level lib supported by OS

pthreads:

- may be usr or kernel level
- POSIX std API for thrd create and syn
- API specifies behavior thrd lib and implementation is by development of lib
- solaris , linux, mac

Other lib :

- windows threading API
- java api

Issue with threads:

1. fork and exec : exactly which thread forks and execss
2. thread cancellation of target : exactly which get cancelled
 1. **Asynchronous** : immediately terminates
 2. **deferred Cancellation** : target periodically check if should cancelled

Signals

Single has to notify proc if a event has occurred

1. sig handling : synchronous and asynchronous

Used to handle proc sig

- sig is generated by event
- sig is delivered to proc
- sig handle

2. **Diff sig identify with diff. number**

3. Sys call kill() and signal() to enable process to deliver and rcv signals

#####

signal(signal , run_to_when_code) = used by proc to specify sig handler : code to be run on rcving sig

kill -\$SIGnum -\$pid : used by proc to send signal to proc

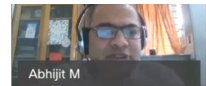
\$SIGNUM

9 = kill

```
#define SIGHUP      1
#define SIGINT    2
#define SIGQUIT    3
#define SIGILL     4
#define SIGTRAP    5
#define SIGABRT    6
#define SIGIOT     6
#define SIGBUS     7
#define SIGFPE     8
#define SIGKILL   9
#define SIGUSR1   10
#define SIGSEGV   11
#define SIGUSR2    12
#define SIGPIPE    13
#define SIGALRM   14
#define SIGTERM    15
#define SIGSTKFLT  16
#define SIGCHLD    17
#define SIGCONT    18
#define SIGSTOP    19
#define SIGTSTP    20
#define SIGTTIN    21
#define SIGTTOU    22
#define SIGURG     23
#define SIGXCPU    24
#define SIGXFSZ    25
#define SIGVTALRM  26
#define SIGPROF    27
#define SIGWINCH   28
#define SIGIO      29
#define SIGPOLL    SIGIO
/*
#define SIGLOST    29
*/
#define SIGPWR     30
#define SIGSYS     31
#define SIGUNUSED  31
```

there are RESTRICTION to sending SIGNAL (?)

Signal handling by OS



```

Process 12323 {
    signal(19, abcd);
}
  
```

```

OS: sys_signal {
    Note down that process 12323
    wants to handle signal number
    19 with function abcd
}
  
```

```

Process P1 {
    kill (12323, 19) ;
}
  
```

```

OS: sys_kill {
    Note down in PCB of process 12323 that
    signal number 19 is pending for you.
}
  
```

When process 12323 is scheduled, at that time the OS will check for pending signals, and invoke the appropriate signal handler for a pending signal.

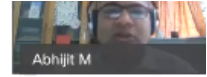
No action when kill is called. Only when it is getting scheduled, then only it runs.

#####

```

#include <stdio.h>
#include <signal.h>
int *p = 1234, i = 1234;
void seghandler(int signo) {
    printf("Seg fault occurred \n");
    return;
}
void inthandler(int signo) {
    printf("INT signal received\n");
    return;
}
int main() {
    signal(SIGINT, inthandler);
    getchar();
    signal(SIGSEGV, seghandler);
    *p = 100;
    return 0;
}
  
```

Issues with threads



- **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 Pool:

- number of threads in a pool

- **Advantage:**

- slightly faster than creating

- allow no of threads in appl to bound to size of pool.(limited means better kernel performance)

Thread Specific Data . Thread Local Storage (TLS):

not local but **global to all functions of thread**,

Similar to “Static” data

a facility for a private data to thread

each thread will have a copy of data

useful when don't have control over creation. In thread pools

in gcc : `static __thread int threadID;`

Scheduler Activations:

function in a thread library

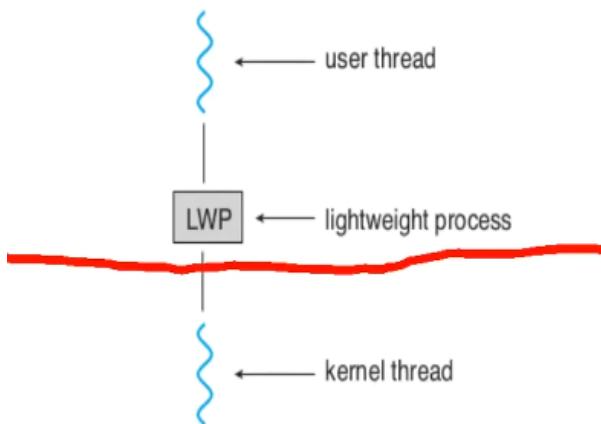
many-many and Two level require communication to **maintain number of kernel allocated to application**

scheduler activation provide:

upcalls : communcion mech from kernel to lib. For any call from **lower to upper** have to use scheduler activation.

This allows application to main correct number of kernel threads.

Issues with threads



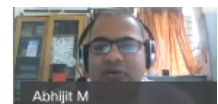
- **Scheduler Activations: LWP approach**
 - An intermediate data structure LWP
 - appears to be a virtual processor on which the application can schedule a user thread to run.
 - Each LWP attached to a kernel thread
 - Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed

- Kernel has to call scheduler activation, before events such as blocking a thread or wait is over.
- this will help appl. To relinquish LWP or req a new

Linux threads:

- called as tasks
- created through clone()
- allows child to share addr space of parent
- struct task_struct points to data struc. (shared or unique)

Linux threads



- **fork() and clone() system calls**
- **Doesn't distinguish between process and thread**
- **Uses term task rather than thread**
- **clone() takes options to determine sharing on process create**
- **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.

1-1 mapping kernel threads


```
*Untitled Document 1
1-1 mapping

typedef struct thread__t {
    void *stack;
    ucontext__t context;
    ttid tid;
    int kernel-tid;

}thread__t;

thread__t thread__create(...., f, ....) {
    stack = create something to hold the stack;
    tid = clone(f, stack, );
}
```

Many – one kernel

```
*Untitled Document 1
1-1 mapping

typedef struct thread__t {
    void *stack;
    ucontext__t context;
    ttid tid;
    int kernel-tid;

}thread__t;

thread__t thread__create(...., f, ....) {
    stack = create something to hold the stack;
    tid = clone(f, stack, );
}
```