Multitasking and Concurrency - Threads

Prof. Himanshu B. Dave
e-Infochips, Ahmedabad

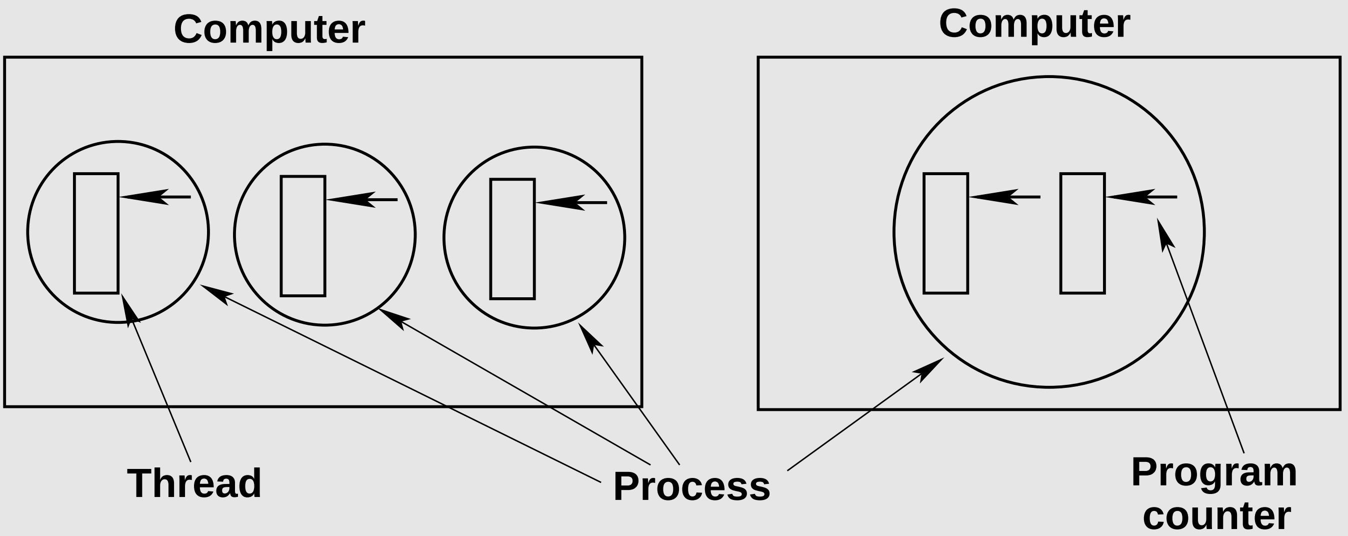
June, 2010 / e-Infochips, Ahmedabad

Contents

Need

- Consider a file server that has several "Threads of Control", or
- "Context of Execution" as Linus calls it
- The server will block occasionally for disk I/O
- While one thread is sleeping, another could continue its execution
- Net result: higher throughput, better performance

Processes, Threads and Machine



Threads

- sometimes called Light Weight Process (LWP)
- they are like mini-processes
- runs strictly sequentially, own PC, stack
- share CPU in time-shared manner
- only on multiple CPU (multiprocessor) can they really execute in parallel
- can create child threads, (but see below), block on system calls
- while one thread is blocked, other in the same process can run
- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction

- A thread does not maintain a list of created threads, nor does it know the thread that created it; in other words, the tree-like structure that we get with Processes is not there
- All threads within a process share the same address space
- thread : process :: process : machine.

Per thread items

Per thread items:

- Program counter
- stack, SP
- Register set (How?)
- thread ID
- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno (pthread functions return "0" if OK)
- State

Thread Shared items

Shared items:

- Address space, static and global variables
- Open files (descriptors), Timers,
- Signals (Sent to each thread) and their handlers
- current working directory
- File Position Pointer
- Open/Close of Files
- lseek/read - conflict if TH1 lseek's and TH2 reads
- User and group id
- Child processes
- Semaphores

- Accounting information
- Page Tables

Benefits of threads - 1

Blocking IO:
Programs that do a lot of IO have three options:

- do the IO serially, waiting for each to complete before commencing the next
- use Asynchronous IO, dealing with all the complexity of asynchronous signals, polling or selects
- use synchronous IO, and just spawn a sperarate thread/process for each IO call. Threading can significantly improve both performance and code complexity

Benefits of threads - 2

Multiple Processors:

- use a threads library that supports multiple processors
- gain significant performance improvements by running threads on each processor
- particularly useful when the program is compute bound

Benefits of threads - 3

User Interface: By separating the user interface, and the program engine into different threads you can allow the UI to continue to respond to user input even while long operations are in progress.

Benefits of threads - 4

Servers:

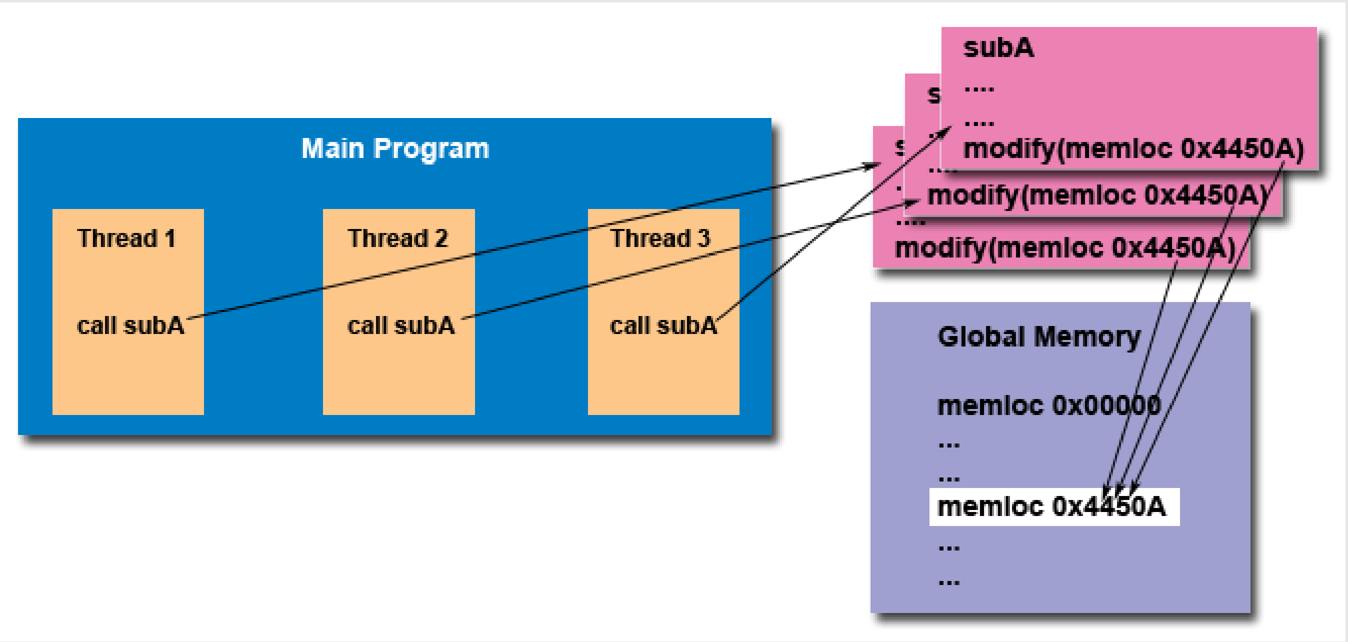
- Servers that serve multiple clients can be made more responsive by the appropriate use of concurrency.
- This has traditionally been achieved by using the `fork()` system call.
- However in some cases, especially when dealing with large caches, threads can help improve the memory utilisation, or
- even permit concurrent operation where `fork()` was unsuitable.

Thread libraries

- user-space:
  - slightly faster,
  - can NOT use SMP,
  - a thread can monopolize CPU,
  - time-slice I/O blocked thread can starve others
- kernel-space:
  - each process - a table of threads,
  - less likely to hog time-slice,
  - can easily use SMP.
- LTL, NPTL, GNU thread library (`pth`)
- POSIX thread library - `pthreads`

- based on `clone()` – see man page

Thread Unsafe functions

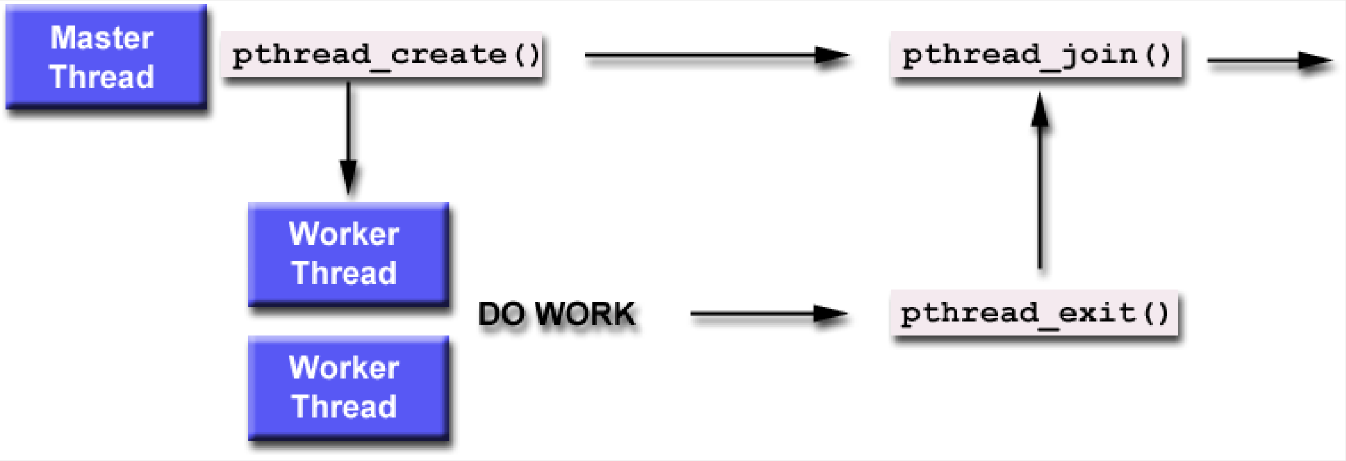


Basic thread management: create etc.

- `pthread_create()` - create a new thread
- `pthread_exit()` - terminate the calling thread
- `pthread_self()` - return identifier of current thread
- `pthread_join()` - wait for termination of another thread, like `wait()` for a process
- `pthread_detach()` - put a running thread in the detached state, make it unnecessary for the parent thread to wait when the caller exits

C programs should be compiled as:
`gcc file.c -o file -lpthread`.

Use of pthread_join()



Mutex calls: operations on mutexes

- pthread_mutex_init() - create a mutex
- pthread_mutex_destroy() - delete a mutex
- pthread_mutex_lock() - lock a mutex, if not already locked, *wait otherwise*
- pthread_mutex_trylock() - try to lock a mutex, *fail* if locked
- pthread_mutex_unlock() - unlock a mutex

Other pthread operations

- `pthread_attr_create()`, etc - template operation - setting attributes of the threads;
- `pthread_cond_init()`, etc. - condition variables - threads can sleep on a condition variable;
- `pthread_setspcific()`, etc. - per thread global variables - can be used by functions inside a thread, but not outside it;
- `pthread_cancel()`, etc. - thread killing - killing of one thread by another

Some Simple Threads Examples

- `chap5.c` : creating a simple thread
- `ex1.c` : creating two threads
- `pthread1.c` : creating two threads, with and w/o joins
- `/concurrency_ei/resource/examples/raceexample.c`
- `/concurrency_ei/resource/examples/mutexex.c`

Exercise:
Write a C program to create n threads, (n is a reasonable input value). Each thread will announce its number and thread ID.

The Important function used

```
int pthread_create(pthread_t * thread,
          const pthread_attr_t * attr,
          void * (*start_routine)(void *), void *arg);
```

`thread` - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)

`attr` - Set to NULL if default thread attributes are used

`void * (*start_routine)` - pointer to the function to be threaded. Function has a single argument: pointer to void

`*arg` - pointer to argument of function. To pass multiple arguments, send a pointer to a structure

To set thread attributes

Define members of the struct pthread_attr_t defined in bits/pthreadtypes.h
Attributes include:

- detached state (joinable? Default: PTHREAD_CREATE_JOIN-ABLE. Other option: PTHREAD_CREATE_DETACHED)
- scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHR
PLICIT_SCHED,SCHED_OTHER)
- scheduling parameter
- inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
- scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
- guard size

- stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACK ADDR)
- stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h)

Other functions used

```
void pthread_exit(void *retval);
```

`retval` - Return value of thread
This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using `pthread_join()`.
Note: the return pointer `*retval`, must not be of local scope otherwise it would cease to exist once the thread terminates.

A Note on Condition Variables

Mutexes allow you to avoid data races, unfortunately while they allow you to protect an operation, they don't permit you to wait until another thread completes an arbitrary activity.
Condition Variables solve this problem.
There are six operations which you can do on a condition variable:

- Initialisation:
  `int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);`
  Again to use the default attributes, just pass NULL as the second parameter.

- Waiting:
  `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);`
  This function always blocks. In pseudo-code:

```
pthread_cond_wait (cond, mut)
begin
   pthread_mutex_unlock (mut);
   block_on_cond (cond);
   pthread_mutex_lock (mut);
end
```

Note that it releases the mutex before it blocks, and then re-acquires it before it returns. This is very important. Also note that re-acquiring the mutex can block for a little longer, so the the condition which was signalled will need to be rechecked after the function returns.

Signalling:

```
int pthread_cond_signal (pthread_cond_t *cond);
```

This wakes up at least one thread blocked on the condition variable. Remember that they must each re-acquire the mutex before they can return, so they will exit the block one at a time.

- Broadcast Signalling:
  `int pthread_cond_broadcast (pthread_cond_t *cond);`
  This wakes up all of the threads blocked on the condition variable. Note again they will exit the block one at a time.

- Waiting with timeout
  `int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mut, const struct timespec *abstime);`
  Identical to `pthread_cond_wait()`, except it has a timeout. This timeout is an absolute time of day.

```
struct timespec to {
        time_t tv_sec;
        long tv_nsec;
};
```

  If a abstime has passed, then `pthread_cond_timedwait()` returns ETIMEDOUT.

- Deallocation: `int pthread_cond_destroy (pthread_cond_t *cond);`

A Concurrent Programming API

We present here a simple API called `parbegin`for concurrent proramming.

- uses a pre-processor `par2.pl` (written in Perl)
- `initiate`: create and start several threads, `join` is auo-generated. See example `/concurrency_ei/parbegin/test1.c`
- `Initiate`: similar to the above, but w/o auto-generated `join`.
- `Join`: join threads generated by immediately preceeding `Initiate`
- `Initall`: create `n` threads and corresponding `joins`
- `shared`: *value-type variable* - creates a shared variable, protected by a mutex and a condition
- `region`: *variable* `do` - creates a critical region, locked by the mutex of the variable; see examples: `/concurrency_ei/parbegin/test4.c, test5.c`

- semaphore: defines a semaphore, see test7.c
- Inits, Wait, Signal: semaphore functions, see test7.c
- await: wait for a condition variable,
  see /concurrency_ei/parbegin/test8.c, prodcons.c
- monitor: start definition of a monitor
- condition: declare a condition variable
- condwait:conditional critical region within monitor functions is started
- condsignal: region is ended
- endmon: end definition of a monitor
  see /concurrency_ei/examples/monitorPC_par2.c

A brief note on this API and how to use the pre-processor will be made available.

Some Example Programs

/unpv22e/

- mutex/prodcons2.c
- mutex/prodcons6.c
- pxsem/prodcons1.c
- pxsem/prodcons2.c
- pxsem/mycat2.c