CEng 332, Systems Programming and Support Environments: Threads

Onur Tolga Şehitoğlu

Computer Engineering, METU

April 10, 2008

Thread vs Process

- Memory space
  shared $\leftrightarrow$ distinct
- Context switch
  cheap $\leftrightarrow$ expensive
- Synchronization ve communication
  user mode/simple $\leftrightarrow$ kernel mod/complex

Threads of the same process

share:

- memory space
- files
- environment
- quota and resources
- signal disposition

differ:

- run-time stack, stack pointer
- register context
- instruction pointer
- signal mask

Thread Models

- User space threads
  - very fast, simple, portable
  - ideal for single processor
  - cannot utilize multiple processors/core
- Lightweight Processes (Linux 2.2-2.4)
  - Threads are simply processes sharing same memory space
  - Synchronization and signalling problem
  - Slow, utilizes multiple processors/core
- M:N threading model
  - total number of threads is larger than kernel threads
  - cooperation of kernel and user thread scheduler
  - complicated implementation
  - fast and utilizes multiple processors/core
- 1:1 threading model (Linux 2.6, Solaris 10)
  - Each thread works in a separate kernel thread (LWP)
  - enhanced kernel support for lwp synchronization and signalling
  - fast and utilizes multiple processors/core

Why to thread?

- single processor case: utilize CPU in I/O bound jobs (GUI, network)
- multiple processor case: parallelism
- dual core ready in desktop and laptops. 2 quad cores on servers. SMP systems with 64 processors exist in some archtectures.
- Intel working on 40 -80 cores CPU. Sun already have 16 core CPU with 16 threads per core.
- Single thread applications might be obsolote in 10 years?

Multiple processor and multicore

- Symmetric Multi Processor systems (single shared memory, multiple processors)
- Multi-Core processors (multiple processor cores sharing some cache memory in a single chip)
- Hyper-threading (multiple ALU or FPU's paralelized on a single core)
- Very low communication cost
- Efficient MIMD parallelism when data is local
- Cache utilization is still significant

POSIX Thread Library

- Started as user level library $\longrightarrow$ now kernel/user space interaction
- Standard in all Unix alike systems
- Microsoft OS's have compatibility libraries

Creating and terminating a thread

- `pthread_create(pthread_t * tidp, pthread_attr_t *attr, void *(*start)(void), void *arg)`
  Creates a new thread and puts the id on first parameter,
  new threads starts by calling `start` function , with `arg` parameter
- `pthread_exit(void *rval);`
  thread calls either this or returns from the start function as `return(rval)` to terminate
- `pthread_join(pthread_t id, void **rval)`,
  in order to get return value of a thread and wait for its termination
- `pthread_t pthread_self()`
  returns threads own identifier
- Thread identifiers are integers in most implementation but POSIX reserves it as a different type

Example: Hello Threads

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

void *helloworld(void *arg) {
        printf("Hello, I am: pid %u tid %u \n",
                getpid(),(unsigned)pthread_self());
}
int main(int argc,char *argv[])
{
        pthread_t tr[5];
        int i,r;

        for (i=0;i<5;i++)
                pthread_create(&tr[i],NULL,helloworld,NULL);
        for (i=0;i<5;i++)
                pthread_join(tr[i],NULL);

        return 0;
}
```

Example: Parallel Sort

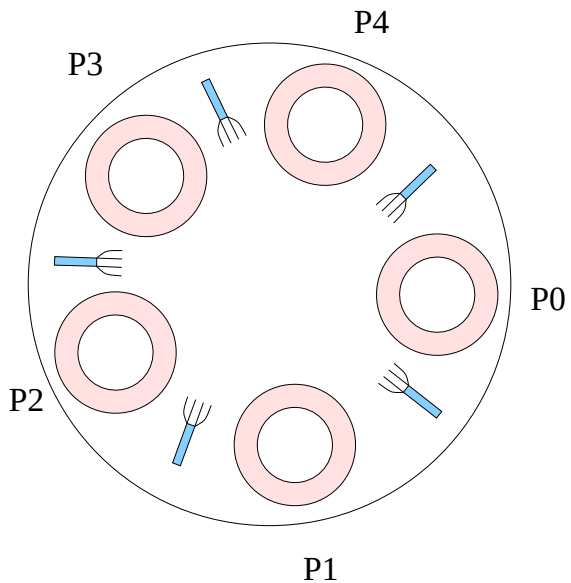
- Partition, sort partitions, merge: $O\left(\frac{N}{P}log(\frac{N}{P}) + N\right)$
- In distributed systems:
  $O(Nlog(N)) < comm. + O\left(\frac{N}{P}log(\frac{N}{P}) + N\right)$
- Merge Sort or Quick Sort

Senkronizasyon/Mutex

- Mutex: mutual exclusion on entrance to critical regions
- works like binary semaphores
- ```
  int pthread_mutex_init(pthread_mutex_t * mutex,
                         pthread_mutexattr_t *r attr);
  int pthread_mutex_destroy(pthread_mutex_t * mutex);
  ```
- ```
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  int pthread_mutex_trylock(pthread_mutex_t *mutex);
  int pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```

Example: Still Dining Philosophers

- Philosphers think for a while then eat
- They need both forks
- Two consecutive philosopher cannot eat together
- After eating for a while they stop and think again



Example: Still Dining Philosophers

```
pthread_mutex_t forks[5];
void philosopher(int *ip) {
        pthread_mutex_t *sol,*sag;
        int i=*ip, z;

        sol=i==0?forks+4:forks+(i-1);        sag=forks+i;
        for (z=0;z<1000;z++) {
                usleep(10000);
                pthread_mutex_lock(sol);
                pthread_mutex_lock(sag);
                printf("Philosopher %d is eating %d\n",i,z);
                usleep(10000);
                printf("Philosopher %d finished eating %d\n",i,z);
                pthread_mutex_unlock(sol);
                pthread_mutex_unlock(sag);
        }
}
int main() {
        pthread_t phils[5];
        int phids[5]={0,1,2,3,4}, i,n;

        for (i=0;i<5;i++) {
                pthread_mutex_init(&(forks[i]),NULL));
        for (i=0;i<5;i++) {
                pthread_create(&(phils[i]), NULL, philosopher, (void *) &(phids[i]));
        }
        for (i=0;i<5;i++) {
                pthread_join(phils[i],NULL);
        }
        return 0;
}
```

Two Phase Locking

- read locks are shared, write locks are exclusive
- threads doing operations with different roles
- ```
  int pthread_rwlock_init(pthread_rwlock_t * rwlock, ...
  attr);
  int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
  ```
- ```
  int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
  int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

  int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
  ```

Conditions and signalling

- Synchronization on condition variables
- Thread A: wait until condition holds
  Thread B: wake A when condition holds
- Condition is preserved by a mutex
- ```
  int pthread_cond_init(pthread_cond_t * cond, ...  * attr);
  int pthread_cond_destroy(pthread_cond_t *cond);

  int pthread_cond_wait(pthread_cond_t * cond,
                        pthread_mutex_t * mutex);
  int pthread_cond_timedwait(pthread_cond_t * cond,
         pthread_mutex_t * mutex, const struct timespec * timeout);

  int pthread_cond_signal(pthread_cond_t *cond);

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

- Producer:
  - If queue is not full, insert item, wake consumer
  - If queue is full wait for empty slot
- Consumer:
  - If queue is not empty, get item and wake consumer
  - If queue is empty, wait for a new item