Concurrent Programming

• So far, we have focused on sequential programming: all computational tasks are executed in sequence, one after the other.
• Next three lectures, we will focus on concurrent programming: multiple computational tasks are executed simultaneously, at the same time.

Communication Between Tasks

Interaction or communication between concurrent tasks can be done via:

- **Shared memory:**
  - all tasks have access to the same physical memory
  - they can communicate by altering the contents of shared memory
- **Message passing:**
  - no common/shared physical memory
  - tasks communicate by exchanging messages

Roadmap

• Concurrent Programming
  - Shared Memory vs Message Passing
  - Divide and Compute
  - Threads vs Processes
  - POSIX Threads

Motivation

• Increase the performance by running more than one tasks at a time.
  - divide the program to n smaller pieces, and run it n times faster using n processors
• To cope with independent physical devices.
  - do not wait for a blocked device, perform other operations at the background
**Serial vs Parallel**

**Divide and Compute**

\[ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \]

How many operations with sequential programming?

7

Step 1: \( x_1 + x_2 \)
Step 2: \( x_1 + x_2 + x_3 \)
Step 3: \( x_1 + x_2 + x_3 + x_4 \)
Step 4: \( x_1 + x_2 + x_3 + x_4 + x_5 \)
Step 5: \( x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \)
Step 6: \( x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \)
Step 7: \( x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \)

**Gain from parallelism**

In theory:
- dividing a program into \( n \) smaller parts and running on \( n \) processors results in \( n \) time speedup

In practice:
- This is not true, due to
  - Communication costs
  - Dependencies between different program parts
  - Eg. the addition example can run only in \( \log(n) \) time not \( 1/n \)

**Prevent Blocking**

- Do not wait for a blocked device, perform other operations at the background
  - During I/O perform computation
  - During continuous visualization, handle key strokes and I/O
    - Eg. video games
  - While listening to network, perform other operations
    - Listening to multiple sockets at the same time
  - Concurrent I/O, concurrent transfers
    - Eg. Web browsers

**Threads vs Processes**

**Process Spawning:**
- Process creation involves the following four main actions:
  - setting up the process control block,
  - allocation of an address space and
  - loading the program into the allocated address space and
  - passing on the process control block to the scheduler

**Thread Spawning:**
- All the threads created within one process share the resources of the process including the address space
- Scheduling is performed on a per-thread basis.
- The thread model is a finer grain scheduling model than the process model
- Threads have a similar lifecycle as the processes and will be managed mainly in the same way as processes are
**Threads vs Processes**

- Heavyweight Process = Process
- Lightweight Process = Thread

**Advantages (Thread vs. Process):**
- Much quicker to create a thread than a process
- Much quicker to switch between threads than to switch between processes
- Threads share data easily

**Disadvantages (Thread vs. Process):**
- Processes are more flexible
  - They don’t have to run on the same processor
- No security between threads: One thread can stomp on another thread’s data
- For threads which are supported by user thread package instead of the kernel:
  - If one thread blocks, all threads in task block.

**Synchronization**

- Mechanism that allows the programmer to control the relative order in which operations occur in different threads or processes.

**Synchronization - Threads**

```
Int sum = 0;

Thread 1:
int t;
lock(sum);
sum = sum + x;
t = sum;
....
unlock(sum);

Thread 2:
int t;
lock(sum);
sum = sum + y;
t = sum;
...
unlock(sum);
```

Use of semaphores for thread synchronization

**Synchronization - Processes**

```
x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8
```

1. Step 1: parallelism = 4
2. Step 2: parallelism = 2
3. Step 3: parallelism = 1

Wait for a message from other processes before continuing processing!

**On a single processor machine**

- You can have multiple threads
- You can also have multiple processes and have the effect of concurrency
  - timesharing

**Thread Creation**

- `pthread_create`
  // creates a new thread executing startRoutine
  int pthread_create(pthread_t *thread, 
  const pthread_attr_t *attr, 
  void *(*startRoutine)(void*), void *arg);

- `pthread_join`
  // suspends execution of the calling thread until the target
  // thread terminates
  int pthread_join(pthread_t thread, void **value_ptr);
Mutual Exclusion

• *pthread_mutex_lock*
  // blocks until mutex is available, and then locks it
  int pthread_mutex_lock(pthread_mutex_t *mutex);

*pthread_mutex_unlock*
  // unlocks the mutex
  int pthread_mutex_unlock(pthread_mutex_t *mutex);

---

Thread Example

```c
int main()
{

  pthread_t thread1, thread2; /* thread variables */
  pthread_create (&thread1, NULL, print_message_function, (void*)"hello ");
  pthread_create (&thread2, NULL, print_message_function, (void*)"world!");

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  exit(0);
}
```

Why use *pthread_join?*
To force main block to wait for both threads to terminate, before it exits. If main block exits, both threads exit, even if the threads have not finished their work.

---

Thread Example (cont.)

```c
void print_message_function ( void *ptr )
{
  char *cp = (char*)ptr;
  for (i=0;i<NUM;i++)
    printf("%s ", cp);
  fflush(stdout);
}
pthread_exit(0); /* exit */
```

---

Interthread Cooperation

```c
void* print_count ( void *ptr )
{
  int i;
  for (i=0;i<NUM;i++)
    printf("counter = %d 
", counter);
  sleep(1);
}
```

```c
void* increment_count ( void *ptr )
{
  int i;
  for (i=0;i<NUM;i++)
    counter++;
  sleep(1);
}
```

---

Interthread Cooperation (cont.)

```c
int total_words;
main(int ac, char *av[])
{
  int i;
  if ( ac != 3 )
    printf("usage: %s file1 file2
", av[0]);
  exit(1);
  total_words = 0;
  pthread_create(&t1, NULL, count_words, (void *) av[1]);
  pthread_create(&t2, NULL, count_words, (void *) av[2]);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("%5d: total words
", total_words);
}
```

---

2-Thread Word Counter

```c
int main()
{
  pthread_t thread1, thread2; /* thread variables */
  pthread_create (&thread1, NULL, print_message_function, (void *)"hello ");
  pthread_create (&thread2, NULL, print_message_function, (void *)"world!");

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  exit(0);
}
```
void *count_words(void *f)
{
    char *filename = (char *) f;
    FILE *fp;
    int c, prevc = '\0';
    if ( (fp = fopen(filename, "r")) != NULL ) {
        while ( (c = getc(fp)) != EOF ) {
            if ( !isalnum(c) && isalnum(prevc) )
                total_words++;
            prevc = c;
        }
        fclose(fp);
    } else
        perror(filename);
    return NULL;
}

void *count_words(void *a)
{
    struct arg_set *args = (struct arg_set *) a;
    FILE *fp;
    int c, prevc = '\0';
    if ( (fp = fopen(args->fname, "r")) != NULL ) {
        while ( (c = getc(fp)) != EOF ) {
            if ( !isalnum(c) && isalnum(prevc) )
                args->count++;
            prevc = c;
        }
        fclose(fp);
    } else
        perror(args->fname);
    return NULL;
}

struct arg_set { /* two values in one arg */
    char *fname; /* file to examine */
    int count; /* number of words */
};

void *count_words(void *a)
{
    struct arg_set *args = (struct arg_set *) a;
    FILE *fp;
    int c, prevc = '\0';
    if ( (fp = fopen(args->fname, "r")) != NULL ) {
        while ( (c = getc(fp)) != EOF ) {
            if ( !isalnum(c) && isalnum(prevc) )
                args->count++;
            prevc = c;
        }
        fclose(fp);
    } else
        perror(args->fname);
    return NULL;
}

void *count_words(void *f)
{
    char *filename = (char *) f;
    FILE *fp;
    int c, prevc = '\0';
    if ( (fp = fopen(filename, "r")) != NULL ) {
        while ( (c = getc(fp)) != EOF ) {
            if ( !isalnum(c) && isalnum(prevc) )
                total_words++;
            prevc = c;
        }
        fclose(fp);
    } else
        perror(filename);
    return NULL;
}

void *count_words(void *a)
{
    struct arg_set *args = (struct arg_set *) a;
    FILE *fp;
    int c, prevc = '\0';
    if ( (fp = fopen(args->fname, "r")) != NULL ) {
        while ( (c = getc(fp)) != EOF ) {
            if ( !isalnum(c) && isalnum(prevc) )
                args->count++;
            prevc = c;
        }
        fclose(fp);
    } else
        perror(args->fname);
    return NULL;
}

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J. Kim (KAIST), and J. Schaumann (SIT).