Programming Shared Address Space Platforms

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Thank Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar for providing slides.

Topic Overview

- Thread Basics
- Unix Multi-process Codes
- The POSIX Thread API
- Synchronization Primitives in Pthreads

Overview of Programming Models

- Programming models provide support for expressing concurrency and synchronization.
- Process based models assume that all data associated with a process is private, by default, unless otherwise specified.
- Lightweight processes and threads assume that all memory is global.

The Unix address space

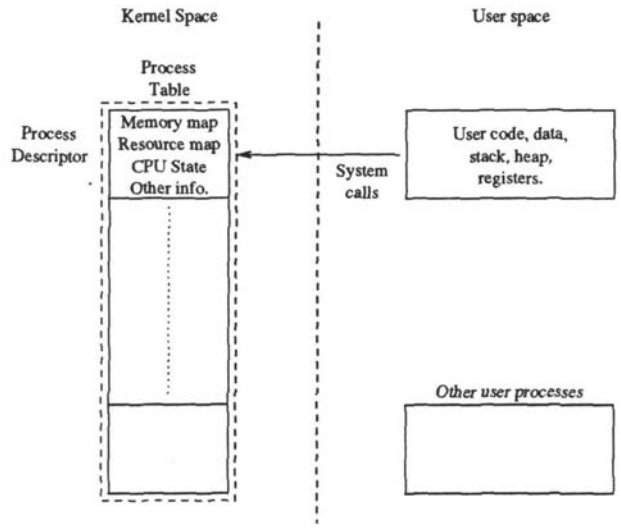

Figure : The process structure of a typical multiprocessing operating system.

- Process is an address space + current state.
- Current state { a program counter, register values, a subroutine call stack.
- The address space in UNIX is partitioned into the user space and kernel space. All accesses to the kernel space must make via system calls, which are to protect one process from another.

Unix Multi-process Code

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
  pid_t pid;
  int i,j;
  if((pid=fork())<0){ //returns 0 to the child and pid of child to
                      parent and return -1 on error.
    printf("fork gailed\n");
    exit(1);
  }
  if(pid==0){
    printf("I am the child\n");
    for(i=0;i<10;i++)printf("waiting\n");
    exit(2);
  }
  else{
    printf("I am the parent with child %d \n",pid);
    while(wait(NULL)!=pid); //Wait(NULL) return the pid of the child.
    printf("Child is now done\n");
    exit(0);
  }
}
```

Thread Basics

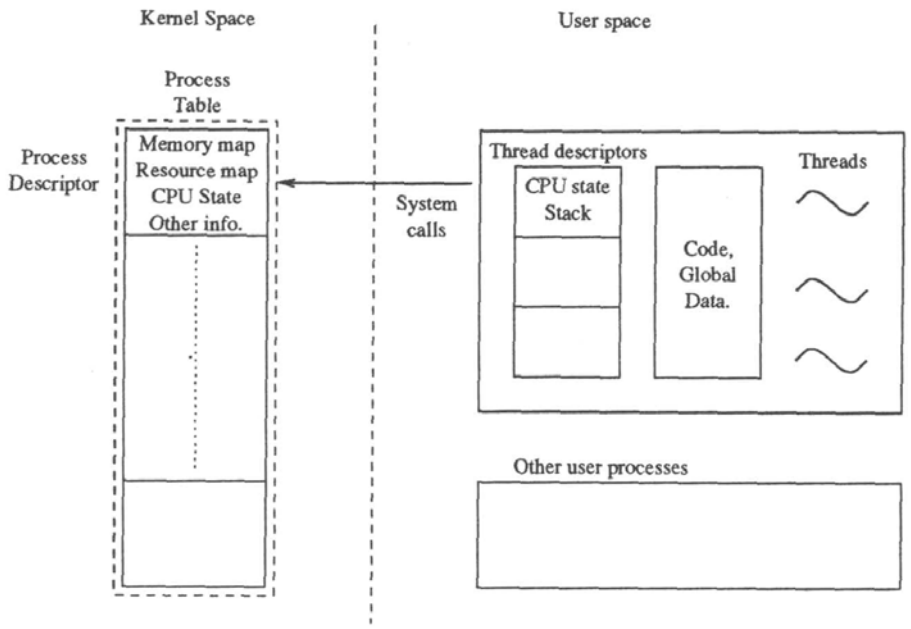

Figure : The two-level structure. Each process has several threads associated with it.

- All memory in the logical machine model of a thread is globally accessible to every thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.

Thread Basics

- This implies a logical machine model with both global memory (default) and local memory (stacks).
- It is important to note that such a flat model may result in very poor performance since memory is physically distributed in typical machines.
- Threads are process share the same address space.
- Threads allow a concurrent programming model within a single process, with very rapid switching of control of cpu from one thread to another possible because little or no memory management is involved.
- Individual threads are not visible outside a process.

Thread Basics

- Threads provide a natural implementation of nonblocking communication operations.
- Threads can increase the convenience and efficiency of implementation of collective operations.
- Threads are becoming the parallel programming model of choice.

Overview of Programming Models

- A *thread* is a single stream of control in the flow of a program. A program like:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(
                dot_product(get_row(a, row),
                            get_col(b, col)));
```

In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

Thread Basics

Why threads ?

- Software Portability
- Latency Hiding: mask memory latency
- Scheduling and load Balancing
- Synchronization and shared variables
- Ease of programming
- POSIX Threads (Pthreads) IEEE specified a standard thread at 1995.

Pthread Sum

```
#define NUMTHREADS 4
#define NUM_ELEMENTS 16
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
struct array_descriptor{
  int thread_number;
  int *array;
  int length;
};
void *initialize_and_sum_array(void *start);
int array[NUM_ELEMENTS],partial_sum[NUMTHREADS];
main()
{
  pthread_t threads[NUMTHREADS];
  struct array_descriptor desc[NUMTHREADS];
  int i,j,global_sum,index;
  for(i=0;i<NUMTHREADS;i++) partial_sum[i]=0;
  global_sum=0;

  pthread_init();
```

Pthread Sum

```
  /* setup array here and spawn threads*/
  for(i=0;i<NUMTHREADS;i++){
    desc[i].thread_number=i;
    desc[i].array=&(array[i*NUM_ELEMENTS/NUMTHREADS]);
    desc[i].length=NUM_ELEMENTS/NUMTHREADS;
    pthread_create(&threads[i],NULL,initialize_and_sum_array,
   (void *)&desc[i]);
  }

  /*wait for everybody to end now */
  for(i=0;i<NUMTHREADS;i++){
    index=pthread_join(threads[i],NULL);
    printf("index %d\n",index);
  }

  for(i=0;i<NUMTHREADS;i++){
    global_sum+=partial_sum[i];
    printf("%d \n",partial_sum[i]);
  }
  printf("sum  %d \n",global_sum);
}
```

Pthread Sum

```
void *initialize_and_sum_array(void *s)
{
  int *array,length;
  int i,thread_no;

  thread_no=((struct array_descriptor *)s)->thread_number;
  array=((struct array_descriptor *)s)->array;
  length=((struct array_descriptor *)s)->length;
  for(i=0;i<length;i++)
    array[i]=thread_no*NUM_ELEMENTS/NUMTHREADS+i;
  for(i=0;i<length;i++)
    partial_sum[thread_no]+=array[i];
}
```