OPERATING SYSTEMS

OPERATING SYSTEMS

Threads

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

Contents

Thread Libraries

- A *thread library* provides the programmer with an API for creating and managing threads.
- There are two primary ways of implementing a thread library.
  - Provide a library entirely in user space with no kernel support.
    - All code and data structures for the library exist in user space.
    - Invoking a function in the library results in a local function call in user space and not a system call.
  - Implement a kernel-level library supported directly by OS.
    - Code and data structures for the library exist in kernel space.
    - Invoking a function in the API for the library typically results in a system call to the kernel.

Thread Libraries

- Three main thread libraries are in use today:
  - POSIX Pthreads
    - Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
    - Used on UNIX and Linux systems.
  - The Windows thread library
    - A kernel-level library available on Windows systems.
  - The Java thread API
    - It allows threads to be created and managed directly in Java programs.
    - In most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system.
    - This means that on Windows systems, Java threads are typically implemented using the Windows API.

Thread Libraries

- Asynchronous threading and Synchronous threading
  - Asynchronous threading and synchronous threading are two general strategies for creating multiple threads.
  - *Asynchronous threading*
    - Once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently.
    - Each thread runs independently of every other thread, and the parent thread need not know when its child terminates.
    - There is typically little data sharing between threads.
  - *Synchronous threading*
    - The parent thread creates one or more children and then waits for all of its children to terminate before it resumes — the so-called *fork-join* strategy (分支聚合策略).
    - The threads created by the parent perform work concurrently. Once each thread has finished its work, it terminates and joins with its parent. Only after *all* of the children have joined can the parent resume execution.
    - It involves significant data sharing among threads.

Thread Libraries

- POSIX Pthreads
  - *POSIX Pthreads* refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. It provide support either for ULT or KLT.
  - Pthreads is a *specification* for thread behavior. Operating-system designers may implement the specification in any way they wish.
  - Numerous systems implement the Pthreads specification.
    - UNIX-type systems, including Linux, Mac OS X, and Solaris.

Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

Some of the Pthreads function calls

Thread Libraries

- POSIX Pthreads
  - Windows doesn’t support Pthreads natively.
    - Some thirdparty implementations for Windows are available.
  - The C program shown in next slide demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread:

$$sum = \sum_{i=0}^{N} i$$

Thread Libraries

Pthreads – create & join

alg.13-1-pthread-create.c (1)

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

```
/* gcc -lpthread | -pthread */
int sum; /* shared by threads */
static void *runner(void *); /* thread function */
int main(int argc, char *argv[])
{
    if(argc < 2) {
        printf("usage: ./a.out <positive integer value>\n");
        return -1;
    }
    pthread_t ptid; /* thread identifier */
    pthread_attr_t attr; /* thread attributes structure */
    pthread_attr_init(&attr); /* set the default attributes */
      /* create the thread - runner with argv[1] */
    ret = pthread_create(&ptid, &attr, &runner, argv[1]);
    if (ret != 0) {
        perror("pthread_create()");
        return 1;
    }
    ret = pthread_join(ptid, NULL); /* join() waiting until thread tid returns */
    if (ret != 0) {
        perror("pthread_join()");
        return 1;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c (2)

```
/* The thread will begin control in this function */
static void *runner(void *param)
{
    int i, upper;

    upper = atoi(param);
    sum = 0;
    for (I = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c (2)

```
/* The thread will begin control in this function */
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
usage: a.out <positive integer value>
isscgy@ubuntu:/mnt/os-2020$ ./a.out 10
sum = 55
isscgy@ubuntu:/mnt/os-2020$ ./a.out 100
sum = 5050
isscgy@ubuntu:/mnt/os-2020$ ./a.out -10
sum = 0
isscgy@ubuntu:/mnt/os-2020$ ./a.out asd
sum = 0
isscgy@ubuntu:/mnt/os-2020$
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c
    - pthread.h must be included by any Pthreads programs.
    - When the program begins, a single thread of control begins in main(). After some initialization, main() creates a second thread that begins control in the runner() function. Both threads *share the global data* sum.
    - pthread_t tid declares the identifier tid for the thread we will create.
    - pthread_attr_t attr declares the attributes for the thread, set by pthread_attr_init(&attr). The default attributes are provided without explicitly setting.
    - pthread_create() creates a separate thread with the attributes attr. The thread identifier is passing to tid. The name of the function where the new thread will begin execution, the runner(), is also passed. Last, the integer parameter that was provided on the command line, argv[1], is passed.

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c
    - Now there are two threads: the *parent thread* in main() and the *child thread* performing the summation in the runner().
    - The program follows the *fork-join synchronous strategy*: after creating the summation thread, the parent thread will *wait* for it to terminate by calling the pthread_join() function.
    - The summation thread will terminate when it calls the function pthread_exit(). Once the summation thread has returned, the parent thread will resume, output the value of the shared data sum.
    - This example program creates only a single thread. A simple method for waiting on several threads using the pthread_join() function is to enclose the operation within a simple for loop.

```
#define NUM_THREADS 10
pthread_t workers[NUM_THREADS];
    … …
for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c
    - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - If the second para of pthread_join() is not NULL, pthread_exit() transfers *retval back.

```
/* alg.13-1-pthread-create-1-1.c */
          … …
int main(int argc, char *argv[])
{
          … …
    pthread_create(&ptid, &attr, &runner, argv[1]);
          … …
    int *retptr;
    pthread_join(ptid, (void **)&retptr);
    printf("runner returned val = %d\n", *retptr);
          … …
}

static void *runner(void *param)
{
          … …
    int *retptr = (int *)malloc(sizeof(int)); /* allocated in process space */
    *retptr = 16;
    pthread_exit((void *)retptr);
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c
    - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - If the second para of pthread_join() is not NULL, pthread_exit() transfers *retval back.

```
/* alg.13-1-pthread-create-1-2.c */
int sum;
        … …
int main(int argc, char *argv[])
{
        … …
    pthread_create(&ptid, &attr, &runner, argv[1]);
        … …
    int *retptr;
    pthread_join(ptid, (void **)&retptr);
    printf("runner returned val = %d\n", *retptr);
        … …
}

static void *runner(void *param)
{
        … …

    pthread_exit((void *)&sum); /* also: return (void *)&sum; */
}:wq
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c
    - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - If the second para of pthread_join() is not NULL, pthread_exit() transfers *retval back.

```
/* alg.13-1-pthread-create-1-3.c */
/* without global variable sum defined */
int main(void)
{
    int sum;
    int *retptr = &sum;     … …

    pthread_create(&ptid, &attr, &runner, &sum);
        … …
    pthread_join(ptid, (void **)&sun);
    printf("runner returned val = %d\n", *retptr);
        … …
}

static void *runner(void *param)
{
    int *sum = (int *)param;
        … …
    pthread_exit((void *)sum); /* return the address in process stack segment */
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create.c
    - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - If the second para of pthread_join() is not NULL, pthread_exit() transfers *retval back.

```
/* alg.13-1-pthread-create-2.c */
        … …
#include <string.h>
int main(int argc, char *argv[])
{         … …
    pthread_create(&tid, &attr, &runner, argv[1]);
        … …
    char *retptr;
    pthread_join(tid, (void **)&retptr);
    printf("runner returned val = %s\n", retptr);
        … …
}

static void *runner(void *param)
{         … …
    char mag[] = "Hello world!";
    char *retptr = (char *)malloc((strlen(msg) + 1)*sizeof(char));
    strcpy(retptr, msg);
    pthread_exit((void *)retptr);
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create-3.c (1)

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

static void *ftn(void *arg)
{
    int *numptr = (int *)arg;
    int num = *numptr;
    char *retval = (char *)malloc(80*sizeof(char));  /* allocated in process heap */

    sprintf(retval, "This is thread-%d, ptid = %lu", num, pthread_self( ));
    printf("%s\n", retval);

    pthread_exit((void *)retval); /* or return (void *)retval; */
}

int main(int argc, char *argv[])
{
    int max_num = 5;
    int i, ret;

    printf("Usage: ./a.out total_thread_num\n");
    if(argc > 1)
        max_num = atoi(argv[1]);
    printf("main(): pid = %d, ptid = %lu.\n", getpid( ), pthread_self( ));
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create-3.c (2)

```
    pthread_t ptid[max_num];
    for(i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, ftn, (void *)&i);
        if(ret != 0) {
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
    }

    for(i = 0; i < max_num; i++) {
        char *retptr;  /* retptr pointing to address allocated by ftn() */
        ret = pthread_join(ptid[i], (void **)&retptr);
        if(ret!=0) {
            fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
            exit(1);
        }
        printf("thread-%d: retval = %s\n", i, retptr);
        free(retptr);
        retptr = NULL;  /* preventing ghost pointer */
    }

    return 1;
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create-3.c

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create-3.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 5
Usage: ./a.out total_thread_num
main(): pid = 18960, ptid = 140253726336832.
This is thread-2, ptid = 140253717825280
This is thread-3, ptid = 140253709432576
This is thread-4, ptid = 140253701039872
This is thread-0, ptid = 140253684254464
This is thread-0, ptid = 140253692647168
thread-0: retval = This is thread-2, ptid = 140253717825280
thread-1: retval = This is thread-3, ptid = 140253709432576
thread-2: retval = This is thread-4, ptid = 140253701039872
thread-3: retval = This is thread-0, ptid = 140253692647168
thread-4: retval = This is thread-0, ptid = 140253684254464
isscgy@ubuntu:/mnt/os-2020$
```

Thread Libraries

Pthreads – create & join

alg.13-1-pthread-create-3-1.c

i is not consistent to ptid[i]

```
pthread_t ptid[max_num];
for(i = 0; i < max_num; i++) {
    ret = pthread_create(&ptid[i], NULL, ftn, (void *)&i);
    if(ret != 0) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(1);
    }
    sleep(1);
}

for(i = 0; i < max_num; i++) {
    char *retptr;  /* retptr pointing to address allocated by ftn() */
    ret = pthread_join(ptid[i], (void **)&retptr);
    if(ret!=0) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(1);
    }
    printf("thread-%d: retval = %s\n", i, retptr);
    free(retptr);
    retptr = NULL;  /* preventing ghost pointer */
}

return 1;
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create-3-1.c

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create-3-1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 5
Usage: ./a.out total_thread_num
main(): pid = 18977, ptid = 139918462719808.
This is thread-0, ptid = 139918454208256
This is thread-1, ptid = 139918443620096
This is thread-2, ptid = 139918435227392
This is thread-3, ptid = 139918357559040
This is thread-4, ptid = 139918349166336
thread-0: retval = This is thread-0, ptid = 139918454208256
thread-1: retval = This is thread-1, ptid = 139918443620096
thread-2: retval = This is thread-2, ptid = 139918435227392
thread-3: retval = This is thread-3, ptid = 139918357559040
thread-4: retval = This is thread-4, ptid = 139918349166336
isscgy@ubuntu:/mnt/os-2020$
```

```
    }

    return 1;
}
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create-4.c

```
int main(int argc, char *argv[])
{
    int max_num = 5;
    int i, ret;

    printf("Usage: ./a.out total_thread_num\n");
    if(argc > 1) {
        max_num = atoi(argv[1]);
    }

    int thread_num[max_num];
    for (i = 0; i < max_num; i++) {
        thread_num[i] = i;
    }

    printf("main(): pid = %d, ptid = %lu.\n", getpid( ), pthread_self( ));

    pthread_t ptid[max_num];
    for(i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, ftn, (void *)&thread_num[i]);
        if(ret != 0) {
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
    }
```

Thread Libraries

- Pthreads – create & join
  - alg.13-1-pthread-create-4.c

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create-4.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 5
Usage: ./a.out total_thread_num
main(): pid = 19020, ptid = 140096962860864.
This is thread-0, ptid = 140096954349312
This is thread-1, ptid = 140096945956608
This is thread-3, ptid = 140096858744576
This is thread-4, ptid = 140096850351872
This is thread-2, ptid = 140096867137280
thread-0: retval = This is thread-0, ptid = 140096954349312
thread-1: retval = This is thread-1, ptid = 140096945956608
thread-2: retval = This is thread-2, ptid = 140096867137280
thread-3: retval = This is thread-3, ptid = 140096858744576
thread-4: retval = This is thread-4, ptid = 140096850351872
isscgy@ubuntu:/mnt/os-2020$
```

```
        if(ret != 0) {
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
    }
```

Thread Libraries

- Pthreads – sharing memory
  - alg.13-2-pthread-shm.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#define MSG_SIZE 1024
```

```
struct msg_stru {
    char msg1[MSG_SIZE], msg2[MSG_SIZE], msg3[MSG_SIZE];
}; /* global variable */

static void *runner1(void *);
static void *runner2(void *);

int main(void)
{
    pthread_t tid1, tid2;
    pthread_attr_t attr = {0};
    struct msg_stru msg; /* storage in main-thread stack */

    sprintf(msg.msg1, "message 1 by parent");
    sprintf(msg.msg2, "message 2 by parent");
    sprintf(msg.msg3, "message 3 by parent");
    printf("\nparent say:\n%s\n%s\n%s\n", msg.msg1, msg.msg2, msg.msg3);

    pthread_attr_init(&attr);
    if(pthread_create(&tid1, &attr, &runner1, (void *)&msg) != 0) {
        perror("pthread_create()"); return 1;
    }
    if(pthread_create(&tid2, &attr, &runner2, (void *)&msg) != 0) {
        perror("pthread_create()"); return 1;
    }
```

Thread Libraries

- Pthreads – sharing memory
  - alg.13-2-pthread-shm.c (2)

```
    if(pthread_join(tid1, NULL) != 0) {
        perror("pthread_join()"); return 1;
    }
    if(pthread_join(tid2, NULL) != 0) {
        perror("pthread_join()"); return 1;
    }

    printf("\nparent say:\n%s\n%s\n%s\n", msg.msg1, msg.msg2, msg.msg3);
    return 0;
}

static void *runner1(void *param)
{
    struct msg_stru *ptr = (struct msg_stru *)param;
    sprintf(ptr->msg1, "message 1 changed by child1");
    pthread_exit(0);
}

static void *runner2(void *param)
{
    struct msg_stru *ptr = (struct msg_stru *)param;
    sprintf(ptr->msg2, "message 2 changed by child2");
    pthread_exit(0);
}
```

Thread Libraries

- Pthreads – sharing memory
  - alg.13-2-pthread-shm.c (2)

```
    if (pthread_join(tid1, NULL) != 0) {
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-2-pthread-shm.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out

parent say:
message 1 by parent
message 2 by parent
message 3 by parent

parent say:
message 1 changed by child1
message 2 changed by child2
message 3 by parent
isscgy@ubuntu:/mnt/os-2020$ 
```

```
static void *runner2(void *param)
{
    struct msg_stru *ptr = (struct msg_stru *)param;
    sprintf(ptr->msg2, "message 2 changed by child2");
    pthread_exit(0);
}
```

Thread Libraries

- Pthreads - phread_attr_setstack testing
  - alg.13-3-pthread-stack.c (1)

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <malloc.h>
#define STACK_SIZE (524288-10000)*4096 /* nearly 2G: 2^19 = 524288 */

int i = 0;
static void *test(void *arg)
{
    char buffer[1024]; /* 1KiB saved to the thread stack */
    if(i > 5 && i < 1965030)
       printf("\b\b\b\b\b\b\b\b%8d", i);
    else
       printf("\niteration = %8d", i);
    i++;
    test(arg); /* recursive calling until segmentation fault */
}
```

Thread Libraries

- Pthreads - phread_attr_setstack testing
  - alg.13-3-pthread-stack.c (1)

```
int main(void)
{
    int ret;
    pthread_t ptid;
    pthread_attr_t tattr = {0};
    char *stackptr = malloc(STACK_SIZE);
    if(!stackptr) {
        perror("malloc()");
        return 1;
    }
    pthread_attr_init(&tattr);
    pthread_attr_setstack(&tattr, stackptr, STACK_SIZE);
    ret = pthread_create(&ptid, &tattr, &test, NULL);
    if(ret) {
        perror("pthread_create()");
        return 1;
    }
    sleep(1);
    pthread_join(ptid, NULL);

    return 0;
}
```

Thread Libraries

- Pthreads - phread_attr_setstack testing
  - alg.13-3-pthread-stack.c (1)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-3-pthread-stack.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out

iteration =        0
iteration =        1
iteration =        2
iteration =        3
iteration =        4
iteration =  1965029
iteration =  1965030
iteration =  1965031
iteration =  1965032
Segmentation fault (core dumped)
isscgy@ubuntu:/mnt/os-2020$
```

514288*4096-1965032*1024 = 94330880
94330880/1965032 = 48 (bytes), for system overhead of each iteration

```
    pthread_join(tid, NULL);

    return 0;
}
```

- **Implicit Threading**
  - *Implicit Threading* is a strategy to give better supports for the design of multithreaded applications. The creation and management of threading are transferred from application developers to compilers and run-time libraries.
  - There are three alternative approaches for designing multithreaded programs that can take advantage of multicore processors through implicit threading.
    - Threads Pools
    - OpenMP
    - Grand Central Dispatch (Apple iOS)

Implicit Threading

- Thread Pools
  - Question:
    - Unlimited threads could exhaust system resources, such as CPU time or memory.
  - Solution.
    - Create a number of threads at process startup and place them into a *thread pool*, where they sit and wait for work.
    - When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request for service.
    - Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.
  - Example.
    - The max number of threads per process in IA-32.
      - The number is about 300 with *10M* default stack-size for each thread in 3G address space.
    - A pool of less than 255 threads may be created.

Implicit Threading

- Thread Pools
  - Benefits of thread pools
    - Usually *slightly faster* to service a request with an existing thread than wait to create a new thread.
    - A thread pool *limits the number of threads* that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
    - Separating the task to be performed from the mechanics of creating the task allows us to use different *strategies* for running the task.
  - Size of a thread pool
    - The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.
    - More sophisticated thread-pool architectures such as Apple's Grand Central Dispatch can dynamically adjust the number of threads in the pool according to usage patterns.

Implicit Threading

- OpenMP
  - OpenMP is a set of *compiler directives* as well as an *API* for programs written in C, C++, or FORTRAN that provides support for parallel programming in *shared-memory* environments.
  - OpenMP identifies *parallel regions* as blocks of code that may run in parallel.
    - Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.
  - The following C program illustrates a compiler directive above the parallel region containing the printf() statement.
    - When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads as there are processing cores in the system (e.g., two threads per core for Intel CPU). All the threads simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

Implicit Threading

OpenMP

alg.13-4-openmp-demo.c

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
```

```
/* compliling: gcc -fopenmp */
#define __NR_gettid 186 /* 186 for x86-64; 224 for i386-32 */
#define gettid() syscall(__NR_gettid)

int main()
{
    #pragma omp parallel
    {
        printf("region 1. pid = %d, tid = %ld\n", getpid(), gettid());
    }
    #pragma omp parallel num_threads(2)
    {
        printf("region 2, num_thread(2). pid = %d, tid = %ld\n", getpid(), gettid());
    }
    #pragma omp parallel num_threads(4)
    {
        printf("region 3, num_thread(4). pid = %d, tid = %ld\n", getpid(), gettid());
    }
    #pragma omp parallel num_threads(6)
    {
        printf("region 4, num_thread(6). pid = %d, tid = %ld\n", getpid(), gettid());
    }

    return 0;
}
```

Implicit Threading

- OpenMP
  - alg.13-4-openmp-demo.c

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-4-openmp-demo.c -fopenmp
isscgy@ubuntu:/mnt/os-2020$ ./a.out
parallel region 1. pid = 19375, tid = 19375
parallel region 1. pid = 19375, tid = 19378
parallel region 1. pid = 19375, tid = 19377
parallel region 1. pid = 19375, tid = 19376
parallel region 2 with num_thread(2). pid = 19375, tid = 19376
parallel region 2 with num_thread(2). pid = 19375, tid = 19375
parallel region 3 with num_thread(4). pid = 19375, tid = 19375
parallel region 3 with num_thread(4). pid = 19375, tid = 19379
parallel region 3 with num_thread(4). pid = 19375, tid = 19376
parallel region 3 with num_thread(4). pid = 19375, tid = 19380
parallel region 4 with num_thread(6). pid = 19375, tid = 19379
parallel region 4 with num_thread(6). pid = 19375, tid = 19380
parallel region 4 with num_thread(6). pid = 19375, tid = 19376
parallel region 4 with num_thread(6). pid = 19375, tid = 19375
parallel region 4 with num_thread(6).                        19382
parallel region 4 with num_thread(6). pid = 19375, tid = 19381
isscgy@ubuntu:/mnt/os-2020$
```

tid 19376 is reused.

Implicit Threading

- OpenMP
  - OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops.
    - Assume we have two arrays a and b of size N. We wish to sum their contents and place the results in array c. We can have this task run in parallel by using the following code segment, which contains the compiler directive for parallelizing for loops:

```
#pragma omp parallel for
/* divides the work contained in the for loop among
the threads it has created */
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

Implicit Threading

OpenMP

alg.13-5-openmp-matrixadd.c (1)

```
/* compliling: gcc -fopenmp */
/* omp works for matrix addition: when od = 12200 with 4 threads, system utility may
have 30% improvment */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <ctype.h>
#include <omp.h>
#define MAX_N 12288

int a[MAX_N][MAX_N], b[MAX_N][MAX_N];
int ans[MAX_N][MAX_N];
int od = 10;

void matrixadd(void)
{
    for(int i = 0; i < od; i++){
        for(int j = 0; j < od; j++){
            ans[i][j] = a[i][j] + b[j][j];
        }
    }

    return;
}
```

Implicit Threading

OpenMP

alg.13-5-openmp-matrixadd.c (2)

```
int main(int argc, void *argv[])
{
    int i, iteration;

    if(argc > 1)
        od = atoi(argv[1]);
    if (od > MAX_N || od < 0)
        return 1;
    i = MAX_N;
    printf("MAX_N = %d, od = %d\n", i, od);
    for(int i = 0; i < od; i++)
        for(int j = 0; j < od; j++)
            a[i][j] = 20;
    for(int i = 0; i < od; i++)
        for(int j = 0; j < od; j++)
            b[i][j] = 30;

      /* with no omp  */
    long start_us, end_us;
    struct timeval t;
    gettimeofday(&t, 0);
    start_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
    matrixadd();
    gettimeofday(&t, 0);
    end_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
    printf("Overhead time usec = %ld, with no omp\n", end_us-start_us);
```

Implicit Threading

OpenMP

alg.13-5-openmp-matrixadd.c (3)

```
    /* with 2 threads */
  gettimeofday(&t, 0);
  start_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
  #pragma omp parallel num_threads(2)
  {
      matrixadd();
  }
  gettimeofday(&t, 0);
  end_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
  printf("Overhead time usec = %ld, omp thread = 2\n", end_us-start_us);

    /* with 4 threads */
  gettimeofday(&t, 0);
  start_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
  #pragma omp parallel num_threads(4)
  {
      matrixadd();
  }
  gettimeofday(&t, 0);
  end_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
  printf("Overhead time usec = %ld, omp thread = 4\n", end_us-start_us);

  return 0;
}
```

Implicit Threading

- OpenMP
  - alg.13-5-openmp-matrixadd.c (3)

```
/* with 2 threads */
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-5-openmp-matrixadd.c -fopenmp
isscgy@ubuntu:/mnt/os-2020$ ./a.out 10
MAX_N = 12288, od = 10
Overhead time usec = 17, with no omp
Overhead time usec = 96, omp thread = 2
Overhead time usec = 2779, omp thread = 4
isscgy@ubuntu:/mnt/os-2020$ ./a.out 1000
MAX_N = 12288, od = 1000
Overhead time usec = 15797, with no omp
Overhead time usec = 14968, omp thread = 2
Overhead time usec = 13119, omp thread = 4
isscgy@ubuntu:/mnt/os-2020$ ./a.out 12200
MAX_N = 12288, od = 12200
Overhead time usec = 6487102, with no omp
Overhead time usec = 4929145, omp thread = 2
Overhead time usec = 3538193, omp thread = 4
isscgy@ubuntu:/mnt/os-2020$
```

Threading Issues

- Some of the issues to consider in designing multithreaded programs.
  - Semantics of fork() and exec() system calls
  - Signal handling
  - Thread cancellation
    - Asynchronous or deferred.
  - Thread-local storage
  - Scheduler activations.

fork() and exec()

- The fork() system call is used to create a separate, duplicate process. But the semantics of the fork() and exec() system calls change in a multithreaded program.
  - If one thread in a program calls fork(), the new process may
    - duplicate all threads, or
    - only the thread that invoked the fork() system call (in Ubuntu).
      - It may be in high risk.
  - Some UNIX systems have two versions of fork().
- The exec() system call typically works in the way that if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the calling process including all of its threads.
  - If exec() is called immediately after forking, the forked process needs only duplicating the calling thread.
    - duplicating all threads is unnecessary as the program specified in the parameters to exec() will replace the calling process.
  - Otherwise, the forked process does not call exec() after forking, it should duplicate all threads of the calling process.
- Suggestion: avoid using fork() in multithreading.

fork() and exec()

fork() and Multithreading

alg.13-6-fork-pthread-demo1.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>

int i = 0;

static void *thread_worker(void *args)
{
    while (1) {
        printf("%d\n", i);
          /* will print ‘0’ only, by thread_worker of parent main() */
        sleep(1);
    }

    pthread_exit(0);
}
```

fork() and exec()

fork() and Multithreading

alg.13-6-fork-pthread-demo1.c (2)

```
int main(void)
{
    pthread_t ptid;
    pthread_create(&ptid, NULL, &thread_worker, NULL);

    pid_t pid = fork(); /* child duplicates parent's main thread only,
without thread_worker() */
    if(pid == 0){ /* child process */
        i = 1;
        printf("In child\n");
        system("ps -l -T");
        while (1) ;
        exit (0);
    }

    wait(&pid);
    printf("In parent\n");
    system("ps -l -T");
    while (1) ;

    return 0;
}
```

- **fork() and exec()**

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-6-fork-pthread-demo1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
in child
0
F S   UID     PID    SPID    PPID  C PRI  NI ADDR SZ WCHAN   TTY          TIME CMD
0 S  1000    1856    1856    1846  0  80   0 -  6124 wait    pts/0    00:00:04 bash
0 S  1000   19526   19526    1856  0  80   0 - 20107 wait    pts/0    00:00:00 a.out
1 S  1000   19526   19527    1856  0  80   0 - 20107 hrtime  pts/0    00:00:00 a.out
1 S  1000   19528   19528   19526  0  80   0 -  3723 wait    pts/0    00:00:00 a.out
0 S  1000   19529   19529   19528  0  80   0 -  1158 wait    pts/0    00:00:00 sh
0 R  1000   19530   19530   19529  0  80   0 -  7667 -       pts/0    00:00:00 ps
in parent
F S   UID     PID    SPID    PPID  C PRI  NI ADDR SZ WCHAN   TTY          TIME CMD
0 S  1000    1856    1856    1846  0  80   0 -  6124 wait    pts/0    00:00:04 bash
0 S  1000   19526   19526    1856  0  80   0 - 20107 wait    pts/0    00:00:00 a.out
1 S  1000   19526   19527    1856  0  80   0 - 20107 hrtime  pts/0    00:00:00 a.out
0 S  1000   19531   19531   19526  0  80   0 -  1158 wait    pts/0    00:00:00 sh
0 R  1000   19532   19532   19531  0  80   0 -  7667 -       pts/0    00:00:00 ps
0
0
0
0
^C
isscgy@ubuntu:/mnt/os-2020$
```

printing 0s only, by thread_worker of parent main() pro

spid=19526, pid=19526: parent main() thread
spid=19527, pid=19526: thread_worker
spid=19528, pid=19526: forked child pro

}

fork() and exec()

fork() and Multithreading

alg.13-7-fork-pthread-demo2.c (1)

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

int i = 0;
static void *thread_worker(void *args)
{
    pid_t pid = fork(); /* the forked child takes thread_worker as main thread. This 
may cause unexpect behaviours in synchronization or signal handling */
    if(pid < 0 )
        return (void *)EXIT_FAILURE;
    if(pid == 0) { /* child pro */
        i = 1;
        printf("In thread_worker's forked child\n");
        system("ps -l -T");
    }

    sleep(2);
    while (1) {
        printf("%d\n", i);
        /* will print ‘0’ by thread_worker of parent main(); ‘1’ by forked child pro */
        sleep(2);
    }
    pthread_exit(0);
}
```

fork() and exec()

- fork() and Multithreading
  - alg.13-7-fork-pthread-demo2.c (2)

```
int main(void)
{
    pthread_t tid;
    pthread_create(&tid, NULL, &thread_worker, NULL);

    sleep(2) ;
    printf("In start main()\n");
    system("ps -l -T");

    while (1) ;

    pthread_join(tid, NULL);

    return EXIT_SUCCESS;
}
```

fork() and exec()

- fork() and Multithreading
  - alg.13-7-fork-pthread-demo2.c (2)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-7-fork-pthread-demo2.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
in thread_worker's forked child
0 S  1000  19605  19605   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19605  19606   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19607  19607  19605  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
in start main()
0
1
0 S  1000  19605  19605   1856  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
1 S  1000  19605  19606   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19607  19607  19605  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
0
1
0
1
0
1
^C
isscgy@ubuntu:/mnt/os-2020$
```

printing 0s by thread_worker of start main(); 1s by forked child

spid=19605, pid=19605: start main() pro
spid=19606, pid=19605: thread_worker
spid=19607, pid=19607: forked child pro

fork() and exec()

fork() and Multithreading

alg.13-7-fork-pthread-demo2.c (2)

```
int main(void)
{
    pthread_t ptid;
    pthread_create(&ptid, NULL, &thread_worker, NULL);

    sleep(2) ;
    printf("in start main()\n");
    system("ps -l -T");

    return 1; /* what will happen? you may have to pkill the forked child process */

    while (1) ;

    pthread_join(tid, NULL);

    return EXIT_SUCCESS;
}
```

fork() and exec()

- fork() and Multithreading
  - alg.13-7-fork-pthread-demo2.c (2)

```
isscgy@ubuntu:/mnt/os-2020$ ./a.out
in thread_worker's forked child
0 S  1000  19647  19647   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19647  19648   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19649  19649  19647  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
0
in start main()
1
0 S  1000  19647  19647   1856  0  80   0 - 20107 wait   pts/0    00:00:00 a.out
1 S  1000  19647  19648   1856  0  80   0 - 20107 hrtime pts/0    00:00:00 a.out
1 S  1000  19649  19649  19647  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
isscgy@ubuntu:/mnt/os-2020$ 1
1
1
1
ps
   PID TTY          TIME CMD
  1856 pts/0    00:00:04 bash
 19649 pts/0    00:00:00 a.out
 19656 pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/os-2020$ 1
1
^C
isscgy@ubuntu:/mnt/os-2020$ 1
pk1
ill -f a1
.out
isscgy@ubuntu:/mnt/os-2020$
```

spid=19647, pid=19647: start main() pro
spid=19648, pid=19647: thread_worker
spid=19649, pid=19649: forked child pro

The forked child is still working, printing 1s. It cannot response to ^C and we have to pkill it.
$pkill -f a.out

Signal Handling

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
  - A signal may be received either synchronously or asynchronously
- All signals should follow the pattern:
  - A signal is generated by the occurrence of a particular event.
  - The signal is delivered to a process.
  - Once delivered, the signal must be handled.
- Signals are handled by one of these two signal handlers
  - Default handler run by kernel
  - User-defined handler that can override default handler.
- For single-threaded, one signal is delivered to a process.
- Where should a signal be delivered for multi-threaded?
  - 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.
  - Discuss later with CLONE_THREAD flag.

Signal Handling

- Example
  - Standard UNIX function for delivered a signal to a specified process

    ```
    int kill(pid_t pid, int signal)
    ```

  - POSIX Pthreads function pthread_kill() allows a signal to be delivered to a specified thread

    ```
    int pthread_kill(pthread_t ptid, int signal)
    ```

Signal Handling

- Linux

```
isscgy@ubuntu:/mnt/hgfs/os-2020$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

- 1-31号是常规信号（非实时信号），没有阻塞队列，多次产生时只有一次记录。
- 32-64号是实时信号，支持排队策略。

Signal Handling

- Linux – Data Structures

```
struct sigaction {
    union {
        __sighandler_t _sa_handler;
        void (*_sa_sigaction)(int, struct siginfo *, void *);
    } _u
    sigset_t sa_mask;
    unsigned long sa_flags;
}

struct sigaction {
    void (*sa_handler)(int); /* func name | SIG_IGN | SIG_DFL */
    void (*sa_sigaction)(int, siginfo_t *, void *); /* seldom
used */
    sigset_t sa_mask; /* 64bit mask, valid only for the action */
    int sa_flags; /* 0: mask itself */
    void (*sa_restorer)(void); /* abandoned */
};
```

- SIG_IGN and SIG_DFL are handles defined in the kernel which means ignoring signals or dealing with by default.

Signal Handling

- Linux – Signal Registering

```
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
/* signum: the registering signal, other than 9 SIGKILL and 19
SIGSTOP; *act: the new sigaction; *oldact: the old sigaction
saved, NULL if careless */
```

- Linux – Setting sigset_t sa_mask

```
int sigemptyset(sigset_t *set) /* clear all */
int sigfillset(sigset_t *set)  /* set all */
int sigaddset(sigset_t *set, int signum)   /* add */
int sigdelset(sigset_t *set, int signum)   /* delete */
int sigismember(sigset_t *set, int signum) /* query */
```

- Linux – Sending Signals

```
int sigqueue(pid_t pid, int signo, const union sigval value);
int kill(pid_t pid, int signal)
int tgkill(int tgid, int tid, int sigal);
int tkill(int tid, int sigal); /* obsolete predecessor to tgkill()
*/
```

Signal Handling

- alg.13-8-sigaction-demo.c (1)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void my_handler(int signo)  /* user signal handler */
{
    printf("\nhere is my_handler");
    printf("\nsignal catched: signo = %d", signo);
    printf("\nCtrl+\\ is masked");
    printf("\nsleeping for 10 seconds ... \n");
    sleep(10);
    printf("my_handler finished\n");
    printf("after returned to the main(), Ctrl+\\ is unmasked\n");
    return;
}
```

Signal Handling

alg.13-8-sigaction-demo.c (2)

```
int main(void)
{
    int ret;
    struct sigaction newact;

    newact.sa_handler = my_handler; /* set the user-defined handler */
    sigemptyset(&newact.sa_mask); /* clear the mask */
    sigaddset(&newact.sa_mask, SIGQUIT); /* sa_mask, set signo=3 
(SIGQUIT:Ctrl+\) */
    newact.sa_flags = 0; /* default */

    printf("now start catching Ctrl+c\n");

    ret = sigaction(SIGINT, &newact, NULL); /* register signo=2 
(SIGINT:Ctrl+C) */
    if (ret == -1) {
        perror("sigaction()");
        exit(1);
    }

    while (1);
    return 0;
}
```

Signal Handling

- alg.13-8-sigaction-demo.c (2)

```
int main(void)
{
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-8-sigaction-demo.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out
now start catching Ctrl+c
^C
here is my_handler
signal catched: signo = 2
Ctrl+\ is masked
sleeping for 10 seconds ...
^\my_handler finished
after returned to the main(), Ctrl+\ is unmasked
Quit (core dumped)
isscgy@ubuntu:/mnt/os-2020$
```

```
        perror("sigaction()");
        exit(1);
    }

    while (1);
    return 0;
}
```

signo 2 (Ctrl+\) is masked and pending until my_handler finished, causing core dumped.

Thread Cancellation

- *Thread Cancellation* involves terminating a thread (here called the target thread) before it has completed.
- Cancellation of a target in two approaches:
  - Asynchronous cancellation
    - terminates the target thread immediately.
  - Deferred cancellation
    - allows the target thread to periodically check if it should be cancelled.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.

Thread Cancellation

- Example
  - In POSIX Pthreads, the pthread_cancel() function cancels a specified thread. The identifier of the target thread is passed as a parameter to the function.
  - The following code illustrates creating—and then canceling—a thread.

```
pthread_t ptid;
  /* create the thread */
pthread_create(&ptid, 0, &thread_worker, NULL);
    . . .
  /* cancel the thread */
pthread_cancel(ptid);
```

Thread Cancellation

- Pthreads Cancellation Modes
  - Pthreads supports three cancellation modes. Each mode is defined as a state and a type.
    - Off Mode
      - State = PTHREAD_CANCEL_DISABLE;
      - no Type defined
    - Deferred Mode
      - State = PTHREAD_CANCEL_ENABLE;
      - Type = PTHREAD_CANCEL_DEFERRED
    - Asynchronous Mode
      - State = PTHREAD_CANCEL_ENABLE;
      - Type = PTHREAD_CANCEL_ASYNCHRONOUS
  - The default cancellation type is deferred cancellation. Cancellation request is pending until the thread reaches a *cancellation-point.*

Thread Cancellation

- Pthreads Cancellation Points
  - *Cancelation-point*
    - By POSIX, the functions of pthread_join(), pthread_testcancel(), pthread_cond_wait(), pthread_cond_timedwait(), sem_wait(), sigwait() and the system calls read(), write() that may cause system blocking are Cancelation-point.
  - The function

    ```
    void pthread_testcancel(void)
    ```

    can be used for setting a cancellation point, or cancelling the thread whose cancellation request is deferred.
  - Example:

    ```
    pthread_testcancel();
    retcode = read(fd, buffer, length);
       /* be set as a cancellation point */
    pthread_testcancel();
    ```

Thread Cancellation

- Pthreads functions with respect to thread cancellation
  - `int pthread_create(pthread_t *restrict ptid, const pthread_attr_t *restrict_attr, void*(*start_rtn)(void*), void *restrict arg);`
  - `int pthread_cancel(pthread_t ptid);`
  - `void pthread_testcancel(void);`
  - `int pthread_setcancelstate(int state, int *oldstate);`
  - `int pthread_setcanceltype(int type, int *oldtype);`
  - `void pthread_cleanup_push(void (*routine) (void *), void *arg);`
  - `void pthread_cleanup_pop(int execute);`