OPERATING SYSTEMS

Monitors & Deadlocks

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

Linux

- Linux versions
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections (non-preemptive).
  - Version 2.6 and later, fully preemptive.
- Linux provides several different mechanisms for synchronization in the kernel:
  - __sync_fetch_ type
  - spinlocks
  - mutex lock
  - semaphores
  - reader-writer versions of spinlocks and semaphores.
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption.

Linux

- Linux gcc __sync_ atomic family

```
type __sync_fetch_and_add(type *ptr, type value);
    /* oldval = *ptr;
       *ptr += value;
       return oldval; */
type __sync_fetch_and_sub(type *ptr, type value);
type __sync_fetch_and_or(type *ptr, type value);
type __sync_fetch_and_and(type *ptr, type value);
type __sync_fetch_and_xor(type *ptr, type value);
type __sync_fetch_and_nand(type *ptr, type value);
type __sync_add_and_fetch(type *ptr, type value);
    /* *ptr += value;
       return *ptr; */
type __sync_sub_and_fetch(type *ptr, type value);
type __sync_or_and_fetch(type *ptr, type value);
type __sync_and_and_fetch(type *ptr, type value);
type __sync_xor_and_fetch(type *ptr, type value);
type __sync_nand_and_fetch(type *ptr, type value);
```

type can be: `uint8_t`, `unt16_t` `uint32_t`, `unt64_t`.

Linux

- Linux gcc __sync_ atomic family

```
bool __sync_bool_compare_and_swap(type *ptr, type oldval, typ
e newval, ...);
    /* if *ptr is equal to oldval, then set *ptr to newval
and return TRUE */
type __sync_val_compare_and_swap(type *ptr, type oldval, type
newval, ...);
    /* if *ptr is equal to oldval, then set *ptr to newval
and return the oldval */
__sync_synchronize(...)
    /* set a full memory barrier */
type __sync_lock_test_and_set(type *ptr, type value, ...)
    /* set *ptr to value and return the old value */
void __sync_lock_release (type *ptr, ...)
    /* set *ptr to 0, with no return value */
```

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-1-syn-fetch-1.c

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

int main(void)
{
    int i;

    i = 10;
    printf("ret = %d, i = %d\n", __sync_fetch_and_add(&i, 20), i);
    printf("i = %d\n", i);

    i = 10;
    printf("ret = %d, i = %d\n", __sync_add_and_fetch(&i, 20), i);
    printf("i = %d\n", i);

    return 0;
}
```

两个值不同

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-1-syn-fetch-1.c

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

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-1-syn-fetch-1.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out
ret = 10, i = 10
i = 30
ret = 30, i = 10
i = 30
isscgy@ubuntu:/mnt/os-2020$
```

```
);
    printf("i = %d\n", i);

    i = 10;
    printf("ret = %d, i = %d\n", __sync_add_and_fetch(&i, 20), i);
    printf("i = %d\n", i);

    return 0;
}
```

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-1-syn-fetch-2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_N 40
static int count = 0;

void *test_func(void *arg)
{
    for (int i = 0; i < 20000; ++i)
        __sync_fetch_and_add(&count, 1);
        /* count++; gave a wrong result */
    return NULL;
}

int main(void)
{
    pthread_t ptid[MAX_N];
    int i;

    for (i = 0; i < MAX_N; ++i)
         pthread_create(&ptid[i], NULL, &test_func, NULL);
    for (i = 0; i < MAX_N; ++i)
        pthread_join(ptid[i], NULL);
    printf("result conut = %d\n", count);
    return 0;
}
```

同步加

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-1-syn-fetch-2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_N 40
static int count = 0;
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-1-syn-fetch-2.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
result conut = 800000
isscgy@ubuntu:/mnt/os-2020$
```

```
    return NULL;
}

int main(void)
{
    pthread_t ptid[MAX_N];
    int i;

    for (i = 0; i < MAX_N; ++i)
         pthread_create(&ptid[i], NULL, &test_func, NULL);
    for (i = 0; i < MAX_N; ++i)
        pthread_join(ptid[i], NULL);
    printf("result conut = %d\n", count);
    return 0;
}
```

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-1-syn-fetch-3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_N 40
static int count = 0;

void *test_func(void *arg)
{
    for (int i = 0; i < 20000; ++i)
        /* __sync_fetch_and_add(&count, 1); */
        count++; /* gave a wrong result */
    return NULL;
}

int main(void)
{
    pthread_t ptid[MAX_N];
    int i;

    for (i = 0; i < MAX_N; ++i)
         pthread_create(&ptid[i], NULL, &test_func, NULL);
    for (i = 0; i < MAX_N; ++i)
        pthread_join(ptid[i], NULL);
    printf("result conut = %d\n", count);
    return 0;
}
```

产生冲突冒险

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-1-syn-fetch-3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_N 40
static int count = 0;
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-1-syn-fetch-3.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
result conut = 327786
isscgy@ubuntu:/mnt/os-2020$
```

```
    return NULL;
}

int main(void)
{
    pthread_t ptid[MAX_N];
    int i;

    for (i = 0; i < MAX_N; ++i)
         pthread_create(&ptid[i], NULL, &test_func, NULL);
    for (i = 0; i < MAX_N; ++i)
        pthread_join(ptid[i], NULL);
    printf("result conut = %d\n", count);
    return 0;
}
```

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-2-syn-compare-test.c (1)

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

int main(void)
{
    int value, oldval, newval, ret;

    value = 200000; oldval = 123456; newval = 654321;
    printf("value = %d, oldval = %d, newval = %d\n", value, oldval, newval);
    printf("ret = __sync_bool_compare_and_swap(&value, oldval, newval)\n");
    ret = __sync_bool_compare_and_swap(&value, oldval, newval);
    printf("ret = %d, value = %d, oldval = %d, newval = %d\n\n", ret, value, oldval,
newval);

    value = 200000; oldval = 200000; newval = 654321;
    printf("value = %d, oldval = %d, newval = %d\n", value, oldval, newval);
    printf("ret = __sync_bool_compare_and_swap(&value, oldval, newval)\n");
    ret = __sync_bool_compare_and_swap(&value, oldval, newval);
    printf("ret = %d, value = %d, oldval = %d, newval = %d\n\n", ret, value, oldval,
newval);
    value = 200000; oldval = 123456; newval = 654321;
    printf("value = %d, oldval = %d, newval = %d\n", value, oldval, newval);
    printf("ret = ___sync_val_compare_and_swap(&value, oldval, newval)\n");
    ret = __sync_val_compare_and_swap(&value, oldval, newval);
    printf("ret = %d, value = %d, oldval = %d, newval = %d\n\n", ret, value, oldval,
newval);
```

Linux

- Linux gcc __sync_ atomic family
  - algorithm 18-2-syn-compare-test.c (2)

```
    value = 200000; oldval = 200000; newval = 654321;
    printf("value = %d, oldval = %d, newval = %d\n", value, oldval, newval);
    printf("ret = ___sync_val_compare_and_swap(&value, oldval, newval)\n");
    ret = __sync_val_compare_and_swap(&value, oldval, newval);
    printf("ret = %d, value = %d, oldval = %d, newval = %d\n\n", ret, value, oldval,
newval);

    value = 200000; newval = 654321;
    printf("value = %d, newval = %d\n", value,  newval);
    printf("ret = __sync_lock_test_and_set(&value, newval)\n");
    ret = __sync_lock_test_and_set(&value, newval);
    printf("ret = %d, value = %d, newval = %d\n\n", ret, value, newval);

    value = 200000;
    printf("value = %d\n", value);
    printf("ret = __sync_lock_release(&value)\n");
    __sync_lock_release(&value); /* no return value */
    printf("value = %d\n",value);

    return 0;
}
```

Linux

- Linux gcc sync atomic family

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-2-syn-compare-test.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out
value = 200000, oldval = 123456, newval = 654321
ret = __sync_bool_compare_and_swap(&value, oldval, newval)
ret = 0, value = 200000, oldval = 123456, newval = 654321

value = 200000, oldval = 200000, newval = 654321
ret = __sync_bool_compare_and_swap(&value, oldval, newval)
ret = 1, value = 654321, oldval = 200000, newval = 654321

value = 200000, oldval = 123456, newval = 654321
ret = ___sync_val_compare_and_swap(&value, oldval, newval)
ret = 200000, value = 200000, oldval = 123456, newval = 654321

value = 200000, oldval = 200000, newval = 654321
ret = ___sync_val_compare_and_swap(&value, oldval, newval)
ret = 200000, value = 654321, oldval = 200000, newval = 654321

value = 200000, newval = 654321
ret = __sync_lock_test_and_set(&value, newval)
ret = 200000, value = 654321, newval = 654321

value = 200000
ret = __sync_lock_release(&value)
value = 0
isscgy@ubuntu:/mnt/os-2020$
```

al,
ne
}

- **Linux**
  - Linux Spinlocks and Kernel Preemption
    - Linux provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking *in the kernel*.
      - On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for *short* durations.
      - On single-processor machines (such as many embedded systems), spinlocks are inappropriate for use. They are replaced by enabling and disabling kernel preemption.

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

Linux

- Linux Spinlocks and Kernel Preemption
  - Linux uses preempt_disable() and preempt_enable() system calls for disabling and enabling kernel preemption.
  - The kernel is not preemptible if a task running in the kernel is holding a lock.
    - Each task in the system has a thread-info structure containing a preempt_count, to indicate the number of locks being held by the task. preempt_count is incremented when a lock is acquired and decremented when a lock is released. If the value of preempt_count is greater than 0, it is not safe to preempt the corresponding task from running in the kernel.
  - Spinlocks—along with enabling and disabling kernel preemption—are *used in the kernel only* when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores or mutex locks are appropriate for use.

Linux

- Linux Mutex Lock
  - The usage of atomic integers is limited. In situations where there are several variables contributing to a possible race condition, more sophisticated locking tools must be used.
  - Mutex locks are available in Linux for protecting critical sections *within the kernel*. Here, a task must invoke the mutex_lock() function prior to entering a critical section and the mutex_unlock() function after exiting the critical section. If the mutex lock is unavailable, a task calling mutex_lock() is put into a sleep state and is awakened when the lock's owner invokes mutex_unlock().

POSIX Synchronization

- POSIX API/Pthreads
  - POSIX API is available for programmers *at the user level* and is OS-independent.
    - The synchronization methods we have discussed pertain to synchronization within the kernel and are therefore available only to *kernel developers*.
  - POSIX protocol provides
    - mutex locks
    - semaphores
    - condition variables
    - Non-portable extensions include:
      - read-write locks
      - spinlocks
  - These APIs are ultimately implemented using tools provided by the host operating system. They are widely used for thread creation and synchronization by developers on UNIX, Linux, and macOS systems.

POSIX Synchronization

- POSIX Mutex Locks
  - A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section.
  - Pthreads uses the pthread_mutex_t data type for mutex locks. A mutex is created with the pthread_mutex_init() function.

```
#include <pthread.h>

pthread_mutex_t mutex;

  /* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- The first parameter is a pointer to the mutex. By passing NULL as a second parameter, we initialize the mutex to its default attributes.

POSIX Synchronization

- POSIX Mutex Locks
  - The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions. If the mutex lock is unavailable when pthread_mutex_lock() is invoked, the calling thread is blocked in the waiting queue of mutex until the owner invokes pthread_mutex_unlock() to release mutex.
  - The following code illustrates protecting a critical section with mutex locks:

```
  /* acquire the mutex lock */
pthread_mutex_lock(&mutex);
critical section
  /* release the mutex lock */
pthread_mutex_unlock(&mutex);
remainder section
```

  - All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

POSIX Synchronization

- POSIX Mutex Locks
  - Some other pthread mutex lock functions

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
const struct timespec *restrict abs_timeout);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

POSIX Synchronization

POSIX Mutex Locks

algorithm 18-3-syn-pthread-mutex.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#define MAX_N 40
```

```
  /* gcc -pthread */
static int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
  /*  pthread_mutex_t mutex; */
void *test_func_syn(void *arg)
{
    for (int i = 0; i < 20000; ++i) {
        pthread_mutex_lock(&mutex);
        count++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

void *test_func_asy(void *arg)
{
    for (int i = 0; i < 20000; ++i) {
        count++;
    }

    pthread_exit(NULL);
}
```

互斥锁

产生冲突冒险

POSIX Synchronization

POSIX Mutex Locks

algorithm 18-3-syn-pthread-mutex.c (2)

```
int main(int argc, const char *argv[])
{
    int i = 0;
    pthread_t ptid[MAX_N];
    /*  pthread_mutex_init (&mutex, NULL); */

    if(argc > 1 && !strncmp(argv[1], "syn", 3))
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_syn, NULL);
    else
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_asy, NULL);

    for (i = 0; i < MAX_N; ++i) {
        pthread_join(ptid[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    printf("result count = %d\n", count);
    return 0;
}
```

POSIX Synchronization

- POSIX Mutex Locks
  - algorithm 18-3-syn-pthread-mutex.c (2)

```
int main(int argc, const char *argv[])
{
    int i = 0;
    pthread_t ptid[MAX_N];
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-3-syn-pthread-mutex.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
result count = 446465
isscgy@ubuntu:/mnt/os-2020$ ./a.out syn
result count = 800000
isscgy@ubuntu:/mnt/os-2020$
```

```
        for (i = 0; i < MAX_N; ++i)
            pthread_create(&ptid[i], NULL, &test_func_asy, NULL);

    for (i = 0; i < MAX_N; ++i) {
        pthread_join(ptid[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    printf("result count = %d\n", count);
    return 0;
}
```

POSIX Synchronization

POSIX Semaphore 信号量

与管道相似

- Two types of semaphores—*named* and *unnamed* are specifies by the POSIX SEM extension. Beginning with Version 2.6 of the kernel, Linux systems provide support for both types.
- POSIX Named Semaphores
  - The function sem_open() is used to create a new or open an existing semapahore:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t
mode, unsigned int value);
```

  - For example:

```
sem_t *sem;
sem = sem_open("MYSEM", O_CREAT, 0666, 1);
```

    - The named semaphore MYSEM is created and initialized to 1. It has read and write access for other processes.

POSIX Synchronization

- POSIX Semaphore
  - POSIX Named Semaphores
    - Multiple unrelated processes can easily use a common named semaphore as a synchronization mechanism by simply referring to the semaphore's name.
    - In the example above, once the semaphore MYSEM has been created, subsequent calls to sem_open() with the same parameters by other processes return a descriptor sem to the existing semaphore. POSIX declares these operations sem_wait(sem) and sem_post(sem), respectively.
    - The following illustrates protecting a critical section using the named semaphore created above:

```
sem_wait(sem); /* acquire the semaphore */
critical section
sem_post(sem); /* release the semaphore */
...
sem_close(sem);
```

POSIX Synchronization

POSIX Semaphore

POSIX Unnamed Semaphores

- An unnamed semaphore is created and initialized using the sem_init() function, which is passed three parameters:
  (1) A pointer to the semaphore
  (2) A flag indicating the level of sharing
  (3) The semaphore's initial value

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

- For example:

```
#include <semaphore.h>

sem_t sem;
sem_init(&sem, 0, 1); /* create the semaphore and
initialize it to 1 */
```

- pshared = 0 indicates that this semaphore can be shared only by threads *belonging to the same process* that created the semaphore.
- The semaphore is set to the value 1.

POSIX Synchronization

POSIX Semaphore

POSIX Unnamed Semaphores

- POSIX unnamed semaphores also use the same sem_wait(sem) and sem_post(sem) operations on the descriptor sem as named semaphores.
- The following illustrates protecting a critical section using the unnamed semaphore created above:

```
sem_wait(&sem); /* acquire the semaphore */
critical section
sem_post(&sem); /* release the semaphore */
...
sem_destroy(&sem);
```

- Usually a named semaphore is used in inter-process synchronization, and an unnamed semaphore is for inter-thread communication.

POSIX Synchronization

POSIX Semaphore

algorithm 18-4-syn-phread-sem-unnamed.c (1)

```
sem_t unnamed_sem; /* not a pointer */
static int count = 0;

void *test_func_syn(void *arg)
{
    for (int i = 0; i < 20000; ++i) {
        sem_wait(&unnamed_sem);
        count++;
        sem_post(&unnamed_sem);
    }

    pthread_exit(NULL);
}

void *test_func_asy(void *arg)
{
    for (int i = 0; i < 20000; ++i) {
        count++;
    }

    pthread_exit(NULL);
}
```

无名信号量实现互斥

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#define MAX_N 40
```

POSIX Synchronization

POSIX Semaphore

algorithm 18-4-syn-phread-sem-unnamed.c (2)

```
int main(int argc, const char *argv[])
{
    pthread_t ptid[MAX_N];
    int i, ret;
    ret = sem_init(&unnamed_sem, 0, 1);
    if(ret == -1) {
        perror("sem_init()");
        return EXIT_FAILURE;
    }
    if(argc > 1 && !strncmp(argv[1], "syn", 3))
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_syn, NULL);
    else
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_asy, NULL);
    for (i = 0; i < MAX_N; ++i) {
        pthread_join(ptid[i],NULL);
    }
    printf("result count = %d\n", count);
    sem_destroy(&unnamed_sem);
    return 0;
}
```

POSIX Synchronization

POSIX Semaphore

- algorithm 18-4-syn-phread-sem-unnamed.c (2)

```
int main(int argc, const char *argv[])
{
    pthread_t ptid[MAX_N];
    int i, ret;
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-4-syn-pthread-sem-unnamed.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out syn
result count = 800000
isscgy@ubuntu:/mnt/os-2020$ ./a.out
result count = 308105
isscgy@ubuntu:/mnt/os-2020$ 
```

```
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_syn, NULL);
    else
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_asy, NULL);
    for (i = 0; i < MAX_N; ++i) {
        pthread_join(ptid[i],NULL);
    }
    printf("result count = %d\n", count);
    sem_destroy(&unnamed_sem);
    return 0;
}
```

POSIX Synchronization

- POSIX Semaphore
  - algorithm 18-5-syn-phread-sem-named.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <fcntl.h>
#define MAX_N 40

sem_t *named_sem; /* a pointer */
```

命名信号量是指针

```
static int count = 0;

void *test_func_syn(void *arg)
{
    for (int i = 0; i < 20000; ++i) {
        sem_wait(named_sem);
        count++;
        sem_post(named_sem);
```

命名信号量实现互斥

```
    }

    pthread_exit(NULL);
}
```

POSIX Synchronization

- POSIX Semaphore
  - algorithm 18-5-syn-phread-sem-named.c (2)

```
void *test_func_asy(void *arg)
{
    for (int i = 0; i < 20000; ++i) {
        count++;
    }

    pthread_exit(NULL);
}

int main(int argc, const char *argv[])
{
    pthread_t ptid[MAX_N];
    int i, ret;

    named_sem = sem_open("MYSEM", O_CREAT, 0666, 1);
    /* a file named "sem.MYSEM" is created in /dev/shm/ to be shared by
processes who know the file name */
    if (named_sem == SEM_FAILED) {
        perror("sem_open()");
        return EXIT_FAILURE;
    }
```

POSIX Synchronization

POSIX Semaphore

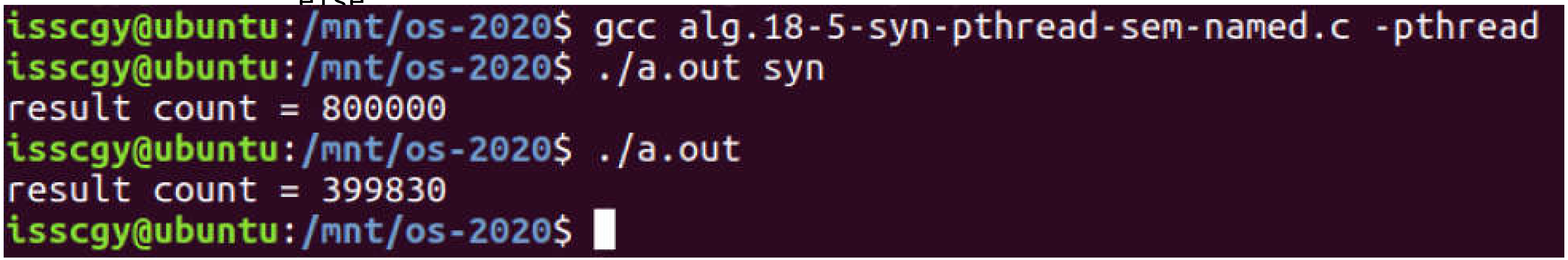
algorithm 18-5-syn-phread-sem-named.c (3)

```
    if (argc > 1 && !strncmp(argv[1], "syn", 3))
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_syn, NULL);
    else
        for (i = 0; i < MAX_N; ++i)
             pthread_create(&ptid[i], NULL, &test_func_asy, NULL);

    for (i = 0; i < MAX_N; ++i) {
        pthread_join(ptid[i],NULL);
    }

    printf("result count = %d\n", count);

    sem_close(named_sem);

    sem_unlink("MYSEM"); /* remove sem.MYSEM from /dev/shm/ when its 
references is 0 */

    return 0;
}
```

POSIX Synchronization

- POSIX Semaphore
  - algorithm 18-5-syn-phread-sem-named.c (3)

```
if (argc > 1 && !strncmp(argv[1], "syn", 3))
    for (i = 0; i < MAX_N; ++i)
        pthread_create(&ptid[i], NULL, &test_func_syn, NULL);
else
```



```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-5-syn-pthread-sem-named.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out syn
result count = 800000
isscgy@ubuntu:/mnt/os-2020$ ./a.out
result count = 399830
isscgy@ubuntu:/mnt/os-2020$
```

```
    printf("result count = %d\n", count);

    sem_close(named_sem);

    sem_unlink("MYSEM"); /* remove sem.MYSEM from /dev/shm/ when its
references is 0 */

    return 0;
}
```

POSIX Synchronization

- Multi-producer-Multi-consumer Problem
  - alg.8-6-syn-pc-con-6.h

```
#define BASE_ADDR 10
/* the first 10 units of the shared memory are reserved for ctln_pc_st, data
   start from the unit indexed 10
/* circular data queue is indicated by
   (enqueue | dequeue) % buffer_size + BASE_ADDR */
struct ctln_pc_st
{
    int BUFFER_SIZE;  // unit number for data in the shared memory
    int MAX_ITEM_NUM; // number of items to be produced
    int THREAD_PRO;   // number of producers
    int THREAD_CONS;  // number of consumers
    sem_t sem_mutex;  // semophore for mutex, type of long int */
    sem_t stock;      // semophore for number of stocks in BUFFER
    sem_t emptyslot;  // semophore for number of empty units in BUFFER
    int item_num;     // total number of items having produced
    int consume_num;  // total number of items having consumed
    int enqueue;      // current position of PRO in buffer
    int dequeue;      // current positions of CONS in buffer
    int END_FLAG;     // producers met MAX_ITEM_NUM, finished their works
}; /* 60 bytes */
```

POSIX Synchronization

- Multi-producer-Multi-consumer Problem
  - alg.18-6-syn-pc-con-6.h

```
#define BASE_ADDR 10
/* the first 10 units of the shared memory are reserved for ctln_pc_st, data
   start from the unit indexed 10
/* circular data queue is indicated by
   (enqueue | dequeue) % buffer_size + BASE_ADDR */
struct data_pc_st
{
    int item_no;       // the item's serial number when it is made
    int pro_no;        // reserved
    long int pro_tid; // tid of the producer who made the item
}; /* 16 bytes */
```

POSIX Synchronization

- Multi-producer-Multi-consumer Problem
  - alg.18-6-syn-pc-con-6.h
  - alg.18-6-syn-pc-con-6.c
  - alg.18-7-syn-pc-producer-6.c
  - alg.18-8-syn-pc-consumer-6.c

POSIX Synchronization

- Multi-producer-Multi-consumer Problem

```
isscgy@ubuntu:/mnt/os-2020$ ./a.out /home/myshm
Pls input the buffer size(1-100, 0 quit): 4
Pls input the max number of items to be produced(1-10000, 0 quit): 8
Pls input the number of producers(1-500, 0 quit): 2
Pls input the number of consumers(1-500, 0 quit): 3

syn-pc-con console pid = 123683
producer pid = 123684, shmid = 44
consumer pid = 123685, shmid = 44
producer tid 123687 prepared item no 1, now enqueue = 1
producer tid 123686 prepared item no 2, now enqueue = 2
                                consumer tid 123688 taken item no 1 by pro 123687, now dequeue = 1
                                consumer tid 123690 taken item no 2 by pro 123686, now dequeue = 2
producer tid 123686 prepared item no 3, now enqueue = 3
producer tid 123687 prepared item no 4, now enqueue = 0
                                consumer tid 123689 taken item no 3 by pro 123686, now dequeue = 3
                                consumer tid 123689 taken item no 4 by pro 123687, now dequeue = 0
producer tid 123686 prepared item no 5, now enqueue = 1
producer tid 123687 prepared item no 6, now enqueue = 2
                                consumer tid 123690 taken item no 5 by pro 123686, now dequeue = 1
                                consumer tid 123689 taken item no 6 by pro 123687, now dequeue = 2
producer tid 123686 prepared item no 7, now enqueue = 3
                                consumer tid 123688 taken item no 7 by pro 123686, now dequeue = 3
producer tid 123687 prepared item no 8, now enqueue = 0
                                consumer tid 123690 taken item no 8 by pro 123687, now dequeue = 0
waiting pro_pid 123684 success.
waiting cons_pid 123685 success.
isscgy@ubuntu:/mnt/os-2020$
```

POSIX Synchronization

- POSIX Condition Variables
  - Condition variables in Pthreads behave similarly to those used within the context of a monitor, which provides a locking mechanism to ensure data integrity.
  - Pthreads is typically used in C programs. Since C-language does not have a monitor, a mutex lock is associated with a condition variable to accomplish locking.
  - Condition variables in Pthreads use the pthread_cond_t data type and are initialized by pthread_cond_init(). The following code creates and initializes a condition variable as well as its associated mutex lock:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```

POSIX Synchronization

- POSIX Condition Variables
  - Example:
    - A thread can wait for the condition clause (a == b) to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);
critical section
pthread_mutex_unlock(&mutex);
```

    - mutex associated with con_var must be locked before the pthread_cond_wait() function is called, since it is used to protect the data in the conditional clause from a possible race condition.
    - The pthread_cond_wait() function is used for waiting on a condition variable.
    - Once this lock is acquired, the thread check the condition and invoking pthread_cond_wait(), passing mutex and con_var as parameters when (a != b), the condition is not true.

POSIX Synchronization

- POSIX Condition Variables
  - Example:
    - A thread can wait for the condition clause (a == b) to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);
critical section
pthread_mutex_unlock(&mutex);
```

    - pthread_cond_wait() will put the calling thread to the end of the CV-queue, release mutex to allow another thread to access the shared data and possibly update its value so that the condition clause (a == b) evaluates to true. When the calling thread is activated, it will lock mutex and rechecked the condition again.
      - This is important because when the condition clause is true, another thread prior to the calling thread in the CV-queue may scheduled.

POSIX Synchronization

- POSIX Condition Variables
  - Example:
    - A thread can invoke the pthread_cond_signal() function, thereby signaling one thread waiting on the condition variable.

```
pthread_mutex_lock(&mutex);
if (a == b)
    pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

  - It is important to note that:
    - pthread_cond_signal() does not release the mutex lock.
    - pthread_mutex_unlock() releases the mutex.
    - Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to pthread_cond_wait().

POSIX Synchronization

- POSIX Condition Variables:
  - algorithm 18-9-pthread-cond-wait.c (1)

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;

int count = 0;

void *decrement(void *arg)
{
    for (int i = 0; i < 4; i++) {
        pthread_mutex_lock(&mutex);
        while (count <= 0) /* wait until count > 0 */
            pthread_cond_wait(&cond, &mutex);
        count--;
        printf("\t\t\t\tcount = %d.\n", count);
        printf("\t\t\t\tUnlock decrement.\n");
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

POSIX Synchronization

- POSIX Condition Variables:
  - algorithm 18-9-pthread-cond-wait.c (2)

```
void *increment(void *arg) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 10000; j++); /* sleep for a while */
        pthread_mutex_lock(&mutex);
        count++;
        printf("count = %d.\n", count);
        if (count > 0)
            pthread_cond_signal(&cond);
        printf("Unlock increment.\n");
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t tid_in, tid_de;
    pthread_create(&tid_de, NULL, &decrement, NULL);
    pthread_create(&tid_in, NULL, &increment, NULL);
    pthread_join(tid_de, NULL);  pthread_join(tid_in, NULL);
    pthread_mutex_destroy(&mutex);  pthread_cond_destroy(&cond);
    return 0;
}
```

POSIX Synchronization

- POSIX Condition Variables:
  - algorithm 18-9-pthread-cond-wait.c (2)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.18-9-pthread-cond-wait.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
count = 1.
Unlock increment.
count = 2.
Unlock increment.
                                count = 1.
                                Unlock decrement.
                                count = 0.
                                Unlock decrement.
count = 1.
Unlock increment.
                                count = 0.
                                Unlock decrement.
count = 1.
Unlock increment.
                                count = 0.
                                Unlock decrement.
isscgy@ubuntu:/mnt/os-2020$
```

- **POSIX Synchronization**
  - POSIX Condition Variables
    - Exercise.
      - Solve producer-consumer problem with pthread_mutex and pthread_cond.

- **Solaris**
  - Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
  - Uses adaptive mutexes for efficiency when protecting data from short code segments:
    - Starts as a standard semaphore spinlock.
    - If lock held, and by a thread running on another CPU, spins.
    - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released.
  - Uses condition variables.
  - Uses readers-writers locks when longer sections of code need access to data.
  - Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock:
    - Turnstiles are per-lock-holding-thread, not per-object.
  - Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile.

- **Windows XP**
  - Uses interrupt masks to protect access to global resources on uniprocessor systems.
  - Uses spinlocks on multiprocessor systems:
    - Spinlocking-thread will never be preempted.
  - Also provides dispatcher objects user-land which may act mutexes, semaphores, events, and timers:
    - Events
      - An event acts much like a condition variable.
    - Timers notify one or more thread when time expired.
    - Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block).