

Thread Synchronization

- POSIX.*THR* supports
 - *mutexes* – for short-term locking
 - *condition variables* – for waiting on events of unbounded duration
- Use of semaphores to synchronize threads is also possible
- Signal handling in threaded programs presents complications which can be reduced if signal handlers are replaced with dedicated threads

Thread Address Space

- Threads are created within the address space of a process and share resources such as static variables and open file descriptors
- When threads use shared resources, they must synchronize their activities to produce consistent results.

Producer-Consumer

- Examples:
 - Producer threads manufacture messages and deposits them in a FIFO queue – Consumer threads remove data items from queue
 - Producer threads generate print requests – Consumer threads are the printers
 - Scheduling queues in a multiprocessor system
 - Network message buffers used when routing messages through intermediate nodes of a WAN
- *Unbounded buffer* – queue size is unlimited
- *Bounded buffer* – queue is of unbounded size
- *Zero-sized buffer* – there is no queue (rendezvous)

Mutex Thread Operations

SYNOPSIS

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

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutexattr_t *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);
```

- Program must declare variable of type `pthread_mutex_t`
- Typically, mutex variables are static available to all threads in the process

pthread_mutex_init

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

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

```
#include <string.h>
```

```
#include <errno.h>
```

```
pthread_mutex_t my_lock;
```

```
if(!pthread_mutex_init(&my_lock, NULL);
```

```
    perror("Could not initialize my_lock");
```

Static Initializer

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

```
pthread_mutex_t
```

```
my_lock=PTHREAD_MUTEX_INITIALIZER
```

- Static Initializer has following advantages over `pthread_mutex_init`
 - Usually more efficient
 - Guaranteed to be executed exactly once before any thread begins execution

Protecting CS with Mutex

```
#include <pthread.h>
pthread_mutex_t my_lock=PTHREAD_MUTEX_INITIALIZER
pthread_mutex_lock(&my_lock);
    /* critical section */
pthread_mutex_unlock
```

- Mutex must be released by the same thread that acquired it
- Holds mutex only for a short period of time
- Threads waiting for events of unpredictable duration, should use semaphores or condition variables

get_item/put_item – Version 1

```
void get_item (int *itemp)
{ pthread_mutex_lock(&buffer_lock);
  Get *itemp from buffer;
  pthread_mutex_unlock(&buffer_lock);
  return; }
```

```
void put_item (int item)
{ pthread_mutex_lock(&buffer_lock);
  Put item in buffer;
  pthread_mutex_unlock(&buffer_lock);
  return; }
```

- Items will be placed in buffer even when the buffer is full
- Items will be retrieved from buffer even when buffer is empty

producer-consumer/main – Version 1

```
void * producer(void *arg1)
{ int i;
  for(i <= SUMSIZE; i++) put_item(i*i);
  return NULL; }
void *consumer(void *arg2)
{ int i, myitem;
  for (i = 1; i<= SUMSIZE; i++) get_item(&myitem);
  sum += myitem;
  return NULL;
}
void main(void)
{ pthread_t prodtid;
  pthread_t constid;
  pthread_create(&constid,NULL,consumer,NULL);
  pthread_create(&prodtid,NULL,producer,NULL);
  pthread_join(constid,NULL);
  pthread_join(prodtid,NULL);
  print out sum = sum of the squares; }
```

sched_yield

```
#include <sched.h>
```

```
int sched_yield(void);
```

- Returns 0 on success or -1 and sets errno on failure
- Causes calling thread to lose the processor

sched_yield – Version 2

```
void *producer(void *arg1)
{ int i;
  for (i = 1; i <= SUMSIZE; i++) put_item(i*i);
  sched_yield();
  return NULL; }

void *consumer(void *arg2)
{ int i, myitem;
  for (i = 1; i <= SUMSIZE; i++) { get_item(&myitem);
    sum += myitem;
    sched_yield();
  }
  return NULL; }
```

Version 2 causes the producer and consumer to take turns in strict alternation

Semaphore – Version 3

```
void *producer(void *arg1)
{ int i;
  for (i = 1; i <= SUMSIZE; i++) {
    sem_wait(&slots);
    put_item(i*i);
    sem_post(&items);
  }
  return NULL; }

void *consumer(void *arg2)
int i, myitem;
{ for (i =1;i <= SUMSIZE; i++){
  sem_wait(&items);
  get_item(&myitem);
  sem_post(&slots);
  sum += myitem;
}
  return NULL; }
```

Version 3 - Problems

- Version 3 may not work correctly
- Consider 1 producer and 2 consumer threads
- The producer terminates after producing SUMSIZE items
- Both consumers try to process SUMSIZE items
- One or both consumers will block on empty buffer

producer – Version 4

```
void *producer(void *arg1)
{ int i;
  for (i = 1; i <= SUMSIZE; i++) {
    sem_wait(&slots);
    put_item(i*i);
    sem_post(&items);
  }
  pthread_mutex_lock(&my_lock);
  producer_done = 1;
  pthread_mutex_unlock(&my_lock);
  return NULL; }
```

```
void *consumer(void *arg2)
```

```
{ int myitem;
```

```
  for(;;) {
```

```
    pthread_mutex_lock(&mylock);
```

```
    if (producer_done) {
```

```
      pthread_mutex_unlock(&my_lock);
```

```
      if(sem_trywait(&items)) break; }
```

```
      else {
```

```
        pthread_mutex_unlock(&my_lock);
```

```
        sem_wait(&items); }
```

```
      get_item(&myitem);
```

```
      sem_post(&slots);
```

```
      sum += myitem;
```

```
    }
```

```
    return NULL; }
```

consumer – Version 4

Version 4 – Problems

- After all items in the buffer have been consumed, one or both of the consumers can block on `sem_wait(&items)` before the producer sets `producer_done` to 1.
- When the producer finishes, if a consumer is waiting on the items semaphore queue, there is no way to unblock it without a `sem_post(&items)`
- The producer cannot issue `sem_post` without making the consumer think there is an item in the buffer to consume
- The producer could issue `sem_destroy`, but implementations would respond differently

Producer – Version 5

```
#define MAXCONSUMERS 2
#define SUMSIZE 100
...
void *producer(void *arg1)
{ int i;
  for (i = 1; i <= SUMSIZE; i++) {
    sem_wait(&slots);
    put_item(i*i);
    sem_post(&items);
  }
  pthread_mutex_lock(&my_lock);
  producer_done = 1;
  for (i = 0; i < MAXCONSUMERS; i++)
    sem_post(&items);
  pthread_mutex_unlock(&my_lock);
  return NULL; }
```

```
void *consumer(void *arg2)
```

```
{ int myitem;
```

```
  for(;;) {
```

```
    sem_wait(&items);
```

```
    pthread_mutex_lock(&mylock);
```

```
    if(!producer_done) {
```

```
      pthread_mutex_unlock(&my_lock);
```

```
      get_item(&myitem);
```

```
      sem_post(&slots);
```

```
      sum += myitem; }
```

```
    else {
```

```
      pthread_mutex_unlock(&my_lock);
```

```
      break; }
```

```
  }
```

```
  return NULL; }
```

consumer – Version 5

Version 5 – Problem

Early termination of producer – If the producer sets `producer_done` to 1 before the buffer is empty, the consumer breaks from the loop without consuming the remaining buffer items

Condition Variables

- *cond_wait* and *cond_signal* are analogous to *sem_wait* and *sem_post*, but not identical
- Semaphores test the predicate $S > 0$ as a part of *sem_wait* and blocks only if the predicate is false
- Condition variables:
 - a) Acquire the mutex
 - b) Test the predicate
 - c) If the predicate is true, do some work and release mutex
 - d) If the predicate is false, call *cond_wait* and go to b) when it returns

cond_wait and cond_signal

```
a: | lock_mutex(&m);  
b: |   while (x != y)  
c: |     cond_wait(&v, &m);  
d: |   /* do stuff related to x and y */  
e: | unlock_mutex(&m);  
  
f: | lock_mutex(&m);  
g: |   x++;  
h: |   cond_signal(&v);  
i: | unlock_mutex(&m);
```

Interleavings

If initially $x = 0$ and $y = 2$, are the following execution orders possible?

– $a_1 b_1 c_1 f_2 g_2 h_2 i_2$

Yes, followed by: $b_1 c_1 f_2 g_2 h_2 i_2 b_1 c_1 d_1 e_1$

– $a_1 b_1 c_1 f_2 g_2 h_2 b_1 c_1 i_2$

No, b_1 cannot follow h_2 since thread2 currently has mutex and thread 1 must reacquire mutex before proceeding

– $a_1 b_1 f_2 g_2 c_1 h_2$

No, f_2 cannot follow b_1 , since thread 1 does not release mutex until c_1 is executed and f_2 must reacquire the mutex

Condition Variable Operations

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, const
    pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);

int pthread_cond_timewait(pthread_cond_t *cond
    pthread_mutex_t *mutex, const struct timespec
    *abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

                                POSIX.1THR
```

Decs – Condition Version 6

```
/* Program 10.6 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define SUMSIZE 100
#define BUFSIZE 8
int sum = 0;
pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
pthread_cond_t items = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slot_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t item_lock = PTHREAD_MUTEX_INITIALIZER;
int nslots = BUFSIZE;
int producer_done = 0;
int nitems = 0;
void get_item(int *itemp);
void put_item(int item);
```

producer – Condition Version 6

```
void *producer(void *arg1)
{ int i;
  for (i = 1; i <= SUMSIZE; i++) {
    pthread_mutex_lock(&slot_lock); /* acquire right to a slot */
    while (nslots <= 0)
      pthread_cond_wait (&nslots, &slot_lock);
    nslots--;
    pthread_mutex_unlock(&slot_lock);
    put_item(i*i);
    pthread_mutex_lock(&item_lock); /* release right to an item */
    nitems++;
    pthread_cond_signal(&nitems);
    pthread_mutex_unlock(&item_lock);
  }
  pthread_mutex_lock(&item_lock);
  producer_done = 1;
  pthread_cond_broadcast(&nitems);
  pthread_mutex_unlock(&item_lock);
  return NULL; }
```

consumer – Condition Version 6

```
void *consumer(void *arg2)
{ int myitem;
  for ( ; ; ) {
    pthread_mutex_lock(&item_lock); /* acquire right to an item */
    while ((nitems <=0) && !producer_done)
      pthread_cond_wait(&items, &item_lock);
    if ((nitems <= 0) && producer_done) {
      pthread_mutex_unlock(&item_lock);
      break;
    }
    nitems--;
    pthread_mutex_unlock(&item_lock);
    get_item(&myitem);
    sum += myitem;
    pthread_mutex_lock(&slot_lock); /* release right to a slot */
    nslots++;
    pthread_cond_signal(&slots);
    pthread_mutex_unlock(&slot_lock);
  }
  return NULL; }
```

main – Condition Version 6

```
void main(void)
{ pthread_t prodtid;
  pthread_t constid;
  int i, total;

                                     /* check value */
  total = 0;
  for (i = 1; i <= SUMSIZE; i++)
    total += i*i;
  printf("The checksum is %d\n", total);
                                     /* create threads */
  pthread_create(&prodtid, NULL, producer, NULL);
  pthread_create(&constid, NULL, consumer, NULL);
                                     /* wait for the threads to finish */
  pthread_join(prodtid, NULL);
  pthread_join(constid, NULL);
  printf("The threads produced the sum %d\n", sum);
  exit(0);
}
```

Thread Signal Delivery (1)

Type	Delivery Action
Asynchronous	Delivered to some thread that has it blocked
Synchronous	Delivered to the thread that caused it
Directed	Delivered to the identified thread (pthread_kill)

Thread Signal Delivery (2)

- Signals such as SIGFPE (floating point exception) are synchronous to the thread that caused it
- Synchronous signals are sometimes called traps (traps are handled by threads that caused them)
- Asynchronous signals
 - Not generated at a predictable time
 - Not associated with a particular thread
 - If several threads have an asynchronous signal unblocked, one of them will be selected to handle it

pthread_kill

SYNOPSIS

```
#include <signal.h>
```

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

```
int pthread_kill(pthread_t thread, int sig);
```

POSIX.THR

Signals can be directed to a particular thread with `pthread_kill`

pthread_sigmask

SYNOPSIS

```
#include <signal.h>
```

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

```
int pthread_sigmask(int how, const sigset_t *set,  
    sigset_t *oset);
```

POSIX.1*THR*

- A thread can examine or set its signal mask with `pthread_sigmask`
- `pthread_sigmask` is similar to `sigprocmask`
- `how` parameter can be `SIG_BLOCK`, `SIG_SETMASK`, or `SIG_UNBLOCK`
- Signal handlers are process wide – thread specific signal handlers are important

catch_siguser1, Version 7

```
void catch_siguser1(int&slot_lock)
{ pthread_mutex_lock(&slot_lock);
  producer_shutdown = 1;
  pthread_mutex_unlock(&slot_lock); }
```

producer (top) – Version 7

```
void *producer(void * arg1)
{ int i;
  sigset_t intmask;
  sigemptyset(&intmask);
  sigaddset(&intmask, SIGUSR1);
  for (i = 1; ; i++){
    pthread_sigmask(SIG_BLOCK, &intmask, NULL);
    pthread_mutex_lock(&slot_lock); /* acquire right to a slot */
    while ((nslots <= 0) && (!producer_shutdown))
      pthread_cond_wait (&slots, &slot_lock);
    if (producer_shutdown) {
      pthread_mutex_unlock(&slot_lock);
      break;
    }
    nslots--;
    pthread_mutex_unlock(&slot_lock);
    pthread_sigmask(SIG_UNBLOCK, &intmask, NULL);
    put_item(i*i);
    pthread_mutex_lock(&item_lock); /* release right to an item */
    nitems++;
    totalproduced++;
    pthread_cond_signal(&items);
    pthread_mutex_unlock(&item_lock);
  }
}
```

producer (bottom) – Version 7

```
pthread_mutex_lock(&item_lock);
    producer_done = 1;
    pthread_cond_broadcast(&items);
pthread_mutex_unlock(&item_lock);
return NULL;
}
```

```
void *consumer(void *arg2)
{ int myitem;
  for ( ; ; ) {
    pthread_mutex_lock(&item_lock); /* acquire right to an item */
    while ((nitems <= 0) && !producer_done)
      pthread_cond_wait(&items, &item_lock);
    if ((nitems <= 0) && producer_done) {
      pthread_mutex_unlock(&item_lock);
      break;
    }
    nitems--;
    pthread_mutex_unlock(&item_lock);
    get_item(&myitem);
    sum += myitem;
    pthread_mutex_lock(&slot_lock); /* release right to a slot */
    nslots++;
    pthread_cond_signal(&slots);
    pthread_mutex_unlock(&slot_lock);
  }
  return NULL;
}
```

Consumer – Version 7

main – Signal Version 7

```
void main (void) {  
    ...  
    act.sa_handler = catch_sigusr1;  
    sigemptyset(&act.sa_mask);  
    act.sa_flags = 0;  
    sigaction(SIGUSR1, &act, NULL);  
    sigemptyset(&set);  
    sigaddset(&set, SIGUSR1);  
    sigprocmask(SIG_BLOCK, &set, NULL);  
                                /* create threads */  
    pthread_create(&prodtid, NULL, producer, NULL);  
    pthread_create(&constid, NULL, consumer, NULL);  
    ...  
}
```

sigwait

SYNOPSIS

```
#include <signal.h>
```

```
int sigwait (sigset_t *sigmask, int *signo);
```

POSIX.1*THR*

- sigwait blocks until the thread receives any of the signals specified in *sigmask
- sigwait returns 0 if the call is successful and -1 otherwise and sets errno
- compare sigwait and sigsuspend
 - in both the first parameter is a signal mask
 - In sigsuspend, signals in the mask are the ones that can cause sigsuspend to return
 - In sigwait, signals in the mask are the ones that cause sigwait to return

sigusr1_thread – Version 8

```
void *sigusr1_thread(void *arg)
{ sigset_t intmask;
  struct sched_param param;
  int policy;
  int sig;
  sigemptyset(&intmask);
  sigaddset(&intmask, SIGUSR1);
  pthread_getschedparam(pthread_self(), &policy, &param);
  fprintf(stderr,
    "sigusr1_thread entered with policy %d and priority %d\n",
    policy, param.sched_priority);
  sigwait(&intmask,&sig);
  fprintf(stderr, "sigusr1_thread returned from sigwait\n");
  pthread_mutex_lock(&slot_lock);
  producer_shutdown = 1;
  pthread_cond_broadcast(&slots);
  pthread_mutex_unlock(&slot_lock);
  return NULL; }
```

producer (top) – Version 8

```
void *producer(void *arg1)
{ int i;
  for (i = 1; ; i++) {
    pthread_mutex_lock(&slot_lock); /* acquire right to a slot */
    while ((nslots <= 0) && (!producer_shutdown))
      pthread_cond_wait (&slots, &slot_lock);
    if (producer_shutdown) {
      pthread_mutex_unlock(&slot_lock);
      break;
    }
    nslots--;
    pthread_mutex_unlock(&slot_lock);
    put_item(i*i);
    pthread_mutex_lock(&item_lock); /* release right to an item */
    nitems++;
    pthread_cond_signal(&items);
    pthread_mutex_unlock(&item_lock);
    totalproduced = i;
  }
}
```

producer (bottom) – Version 8

```
pthread_mutex_lock(&item_lock);
    producer_done = 1;
    pthread_cond_broadcast(&items);
pthread_mutex_unlock(&item_lock);
return NULL;
}
```

consumer – Version 8

```
void *consumer(void *arg2)
{ int myitem;
  for ( ; ; ) {
    pthread_mutex_lock(&item_lock); /* acquire right to an item */
    while ((nitems <=0) && !producer_done)
      pthread_cond_wait(&items, &item_lock);
    if ((nitems <= 0) && producer_done) {
      pthread_mutex_unlock(&item_lock);
      break;
    }
    nitems--;
    pthread_mutex_unlock(&item_lock);
    get_item(&myitem);
    sum += myitem;
    pthread_mutex_lock(&slot_lock); /* release right to a slot */
    nslots++;
    pthread_cond_signal(&slots);
    pthread_mutex_unlock(&slot_lock);
  } return NULL; }
```

main – Version 8

```
void main(void) {
    ...
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    sigprocmask(SIG_BLOCK, &set, NULL);
    fprintf(stderr, "Signal blocked\n");
                                /* create threads */
    pthread_attr_init(&high_prio_attr);
    pthread_attr_getschedparam(&high_prio_attr, &param);
    param.sched_priority++;
    pthread_attr_setschedparam(&high_prio_attr, &param);
    pthread_create(&sighandid, &high_prio_attr, sigusr1_thread,
        NULL);
    pthread_create(&prodtid, NULL, producer, NULL);
    pthread_create(&constid, NULL, consumer, NULL);
    ...
}
```