

# Semaphores

# Synchronization Primitives

## xCOUNTING SEMAPHORES

- xPermit a limited number of threads to execute a section of the code

## xBINARY SEMAPHORES - MUTEXES

- xPermit only one thread to execute a section of the code

## xCONDITION VARIABLES

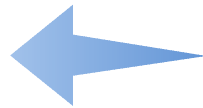
- xCommunicate information about the state of shared data

# POSIX Semaphores

## xNAMED SEMAPHORES (IMPLEMENTED AS “FILES”)

- xProvides synchronization between unrelated process and related process as well as between threads
- xKernel persistence
- xSystem-wide and limited in number
- xUses `sem_open`

## xUNNAMED SEMAPHORES



- xProvides synchronization between threads and between related processes
- xThread-shared or process-shared
- xUses `sem_init`

# POSIX Semaphores

## ✕DATA TYPE

✕Semaphore is a variable of type `sem_t`

```
#include <semaphore.h>
```

## ✕ATOMIC OPERATIONS

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

```
int sem_destroy(sem_t *sem);
```

```
int sem_post(sem_t *sem);
```

```
int sem_trywait(sem_t *sem);
```

```
int sem_wait(sem_t *sem);
```

# Unnamed Semaphores

```
#include <semaphore.h>
```

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

**x**INITIALIZE AN UNNAMED SEMAPHORE

**x**RETURNS

**x**0 on success

**x**-1 on failure, sets errno

**x**PARAMETERS

**x**sem:

**x**Target semaphore

**x**pshared:

**x**0: only threads of the creating process can use the semaphore

**x**Non-0: other processes can use the semaphore

**x**value:

**x**Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!

# Sharing Semaphores

- ✗ SHARING SEMAPHORES BETWEEN THREADS WITHIN A PROCESS IS EASY, USE `PSHARED=0`
- ✗ A NON-ZERO `PSHARED` ALLOWS ANY PROCESS THAT CAN ACCESS THE SEMAPHORE TO USE IT
  - ✗ Places the semaphore in the global (OS) environment
  - ✗ Forking a process creates copies of any semaphore it has
    - ✗ Note: unnamed semaphores are not shared across unrelated processes

# sem\_init can fail

✗ ON FAILURE

✗ sem\_init returns -1 and sets errno

errno	cause
<b>EINVAL</b>	<b>Value &gt; sem_value_max</b>
<b>ENOSPC</b>	Resources exhausted
<b>EPERM</b>	Insufficient privileges

```
sem_t semA;  
if (sem_init(&semA, 0, 1) == -1)  
    perror("Failed to initialize semaphore semA");
```

# Semaphore Operations

```
#include <semaphore.h>
```

```
int sem_destroy(sem_t *sem);
```

- ✕DESTROY A SEMAPHORE

- ✕RETURNS

  - ✕0 on success

  - ✕-1 on failure, sets errno

- ✕PARAMETERS

  - ✕sem:

    - ✕Target semaphore

- ✕Notes

  - ✕Can destroy a sem\_t only once

  - ✕Destroying a destroyed semaphore gives undefined results

  - ✕Destroying a semaphore on which a thread is blocked gives undefined results



# Semaphore Operations: signal

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

- ✗ UNLOCK A SEMAPHORE - SAME AS SIGNAL

- ✗ RETURNS

  - ✗ 0 on success

  - ✗ -1 on failure, sets errno (== EINVAL if semaphore doesn't exist)

- ✗ PARAMETERS

  - ✗ sem:

    - ✗ Target semaphore

    - ✗ sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented

    - ✗ sem = 0: one blocked thread will be allowed to run

# Semaphore Operations (wait)

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

- ✗ LOCK A SEMAPHORE

- ✗ Blocks if semaphore value is zero

- ✗ RETURNS

- ✗ 0 on success

- ✗ -1 on failure, sets errno (== EINTR if interrupted by a signal)

- ✗ PARAMETERS

- ✗ sem:

- ✗ Target semaphore

- ✗ sem > 0: thread acquires lock

- ✗ sem = 0: thread blocks

# Semaphore Operations

```
#include <semaphore.h>
```

```
int sem_trywait(sem_t *sem);
```

- ✕TEST A SEMAPHORE'S CURRENT CONDITION

  - ✕Does not block

- ✕RETURNS

  - ✕0 on success

  - ✕-1 on failure, sets errno (== AGAIN if semaphore already locked)

- ✕PARAMETERS

  - ✕sem:

    - ✕Target semaphore

    - ✕sem > 0: thread acquires lock

    - ✕sem = 0: thread returns

# Example: bank balance

✕ PROTECT SHARED VARIABLE BALANCE WITH A SEMAPHORE WHEN USED  
IN:

✕ decshared

✕ Decrements current value of balance

✕ incshared

✕ Increments the balance

# Example: bank balance

```
int decshared() {  
    while (sem_wait(&balance_sem) == -1)  
        if (errno != EINTR)  
            return -1;  
    balance--;  
    return sem_post(&balance_sem);  
}
```

```
int incshared() {  
    while (sem_wait(&balance_sem) == -1)  
        if (errno != EINTR)  
            return -1;  
    balance++;  
    return sem_post(&balance_sem);  
}
```

# Example: bank balance

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

```
static int balance = 0;
static sem_t bal_sem;
```

```
int initshared(int val) {
```

```
    if (sem_init(&bal_sem, 0, 1) == -1)
        return -1;
```

```
    balance = val;
    return 0;
```

```
}
```

pshared

value

# Example: bank balance

```
int decshared() {  
    while (sem_wait(&bal_sem)  
           == -1)  
        if (errno != EINTR)  
            return -1;  
    balance--;  
    return sem_post(&bal_sem);  
}
```

```
int incshared() {  
    while (sem_wait(&bal_sem)  
           == -1)  
        if (errno != EINTR)  
            return -1;  
    balance++;  
    return sem_post(&bal_sem);  
}
```

Which one is going first?

# Unix/Linux Advanced Semaphores

```
int semget(key_t key, int nsems, int semflg);
```

- ✗Get set of semaphores

```
int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

- ✗Atomically perform a user-defined array of semaphore operations on the set of semaphores



# Pthread Synchronization

## xTWO PRIMITIVES

- xMutex (binary semaphore)
  - xSemaphore with maximum value 1
- xCondition variable
  - xProvides a shared signal
  - xCombined with a mutex for synchronization

# Pthread Mutex

## xSTATES

### xLocked

- xSome thread holds the mutex

### xUnlocked

- xNo thread holds the mutex

## xWHEN SEVERAL THREADS COMPETE

### xOne wins

### xThe rest block

- xQueue of blocked threads

# Mutex Variables

## x A TYPICAL SEQUENCE IN THE USE OF A MUTEX

- 1) Create and initialize mutex
- 2) Several threads attempt to lock mutex
- 3) Only one succeeds and now owns mutex
- 4) The owner performs some set of actions
- 5) The owner unlocks mutex
- 6) Another thread acquires mutex and repeats the process
- 7) Finally mutex is destroyed

# Creating a mutex

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

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

- ✕ INITIALIZE A PTHREAD MUTEX: THE MUTEX IS INITIALLY UNLOCKED

- ✕ RETURNS

  - ✕ 0 on success

  - ✕ Error number on failure

    - ✕ EAGAIN: The system lacked the necessary resources; ENOMEM: Insufficient memory ; EPERM: Caller does not have privileges; EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

- ✕ PARAMETERS

  - ✕ mutex: Target mutex

  - ✕ attr:

    - ✕ NULL: the default mutex attributes are used

    - ✕ Non-NULL: initializes with specified attributes

# Creating a mutex

- ✗ DEFAULT ATTRIBUTES

- ✗ Use PTHREAD\_MUTEX\_INITIALIZER

- ```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- ✗ Statically allocated

- ✗ Equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter `attr` specified as `NULL`

- ✗ No error checks are performed

# Destroying a mutex

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

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- ✕ DESTROY A PTHREAD MUTEX

- ✕ RETURNS

  - ✕ 0 on success

  - ✕ Error number on failure

    - ✕ EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

- ✕ PARAMETERS

  - ✕ mutex: Target mutex

# Locking/unlocking a mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

## ✕RETURNS

- ✕0 on success

- ✕Error number on failure

- ✕EBUSY: already locked; EINVAL: Not an initialised mutex; EDEADLK: The current thread already owns the mutex; EPERM: The current thread does not own the mutex

# Simple Example

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

```
static pthread_mutex_t my_lock =
    PTHREAD_MUTEX_INITIALIZER;
```

```
void *mythread(void *ptr) {
    long int i,j;
    while (1) {
```

```
        pthread_mutex_lock (&my_lock);
```

```
        for (i=0; i<10; i++) {
            printf ("Thread %d\n", int) ptr);
            for (j=0; j<500000000; j++);
        }
```

```
        pthread_mutex_unlock (&my_lock);
        for (j=0; j<500000000; j++);
```

```
    }
}
```

```
int main (int argc, char *argv[]) {
    pthread_t thread[2];
```

```
    pthread_create(&thread[0], NULL, mythread, (void
        *)0);
```

```
    pthread_create(&thread[1], NULL, mythread, (void
        *)1);
```

```
    getchar();
}
```



# Condition Variables

- ✗ USED TO COMMUNICATE INFORMATION ABOUT THE STATE OF SHARED DATA
  - ✗ Execution of code depends on the state of
    - ✗ A data structure or
    - ✗ Another running thread
- ✗ ALLOWS THREADS TO SYNCHRONIZE BASED UPON THE ACTUAL VALUE OF DATA
- ✗ WITHOUT CONDITION VARIABLES
  - ✗ Threads continually poll to check if the condition is met

# Condition Variables

- ✗ SIGNALING, NOT MUTUAL EXCLUSION
  - ✗ A mutex is needed to synchronize access to the shared data
- ✗ EACH CONDITION VARIABLE IS ASSOCIATED WITH A SINGLE MUTEX
  - ✗ Wait atomically unlocks the mutex and blocks the thread
  - ✗ Signal awakens a blocked thread

# Creating a Condition Variable

**x**SIMILAR TO PTHREAD MUTEXES

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

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

# Using a Condition Variable

## x\WAITING

- xBlock on a condition variable.
- xCalled with mutex locked by the calling thread
- xAtomically release mutex and cause the calling thread to block on the condition variable
- xOn return, mutex is locked again

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t  
*mutex, const struct timespec *abstime);
```

# Using a Condition Variable

## ✗SIGNALING

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- ✗unblocks at least one of the blocked threads

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- ✗unblocks all of the blocked threads

## ✗SIGNALS ARE NOT SAVED

- ✗Must have a thread waiting for the signal or it will be lost

# Condition Variable: Why do we need the mutex?

```
pthread_mutex_lock(&mutex);          /* lock mutex */
while (!predicate) {                 /* check predicate */

    pthread_cond_wait(&condvar, &mutex);

    /* go to sleep - recheck
       pred on awakening */
}
pthread_mutex_unlock(&mutex);        /* unlock mutex */
```

---

```
pthread_mutex_lock(&mutex);          /* lock the mutex */
predicate=1;                         /* set the predicate */
pthread_cond_broadcast(&condvar);    /* wake everyone up */
pthread_mutex_unlock(&mutex);        /* unlock the mutex */
```

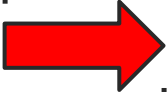
# Condition Variable: No mutex!

```
pthread_mutex_lock(&mutex);           /* lock mutex */
while (!predicate) {                  /* check predicate */
    pthread_mutex_unlock(&mutex);      /* unlock mutex */
    pthread_cond_wait(&condvar);       /* go to sleep - recheck
   pred on awakening */
    pthread_mutex_lock(&mutex);        /* lock mutex */
}
pthread_mutex_unlock(&mutex);         /* unlock mutex */
```

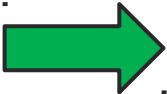
What can happen here?

```
pthread_mutex_lock(&mutex);           /* lock the mutex */
predicate=1;                          /* set the predicate */
pthread_cond_broadcast(&condvar);     /* wake everyone up */
pthread_mutex_unlock(&mutex);         /* unlock the mutex */
```

# Condition Variable: No mutex!



```
pthread_mutex_lock(&mutex);           /* lock mutex */
while (!predicate) {                  /* check predicate */
    pthread_mutex_unlock(&mutex);      /* unlock mutex */
    pthread_cond_wait(&condvar);       /* go to sleep - recheck
   pred on awakening */
    pthread_mutex_lock(&mutex);        /* lock mutex */
}
pthread_mutex_unlock(&mutex);         /* unlock mutex */
```




---

```
pthread_mutex_lock(&mutex);           /* lock the mutex */
predicate=1;                          /* set the predicate */
pthread_cond_broadcast(&condvar);      /* wake everyone up */
pthread_mutex_unlock(&mutex);         /* unlock the mutex */
```





# Condition Variable: No mutex!

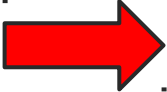
The problem is here

```
pthread_mutex_lock(&mutex);           /* lock mutex */
while (!predicate) {                  /* check predicate */
    pthread_mutex_unlock(&mutex);      /* unlock mutex */
    pthread_cond_wait(&condvar);       /* go to sleep - recheck
   pred on awakening */
    pthread_mutex_lock(&mutex);        /* lock mutex */
}
pthread_mutex_unlock(&mutex);         /* unlock mutex */
```

Another thread might acquire the mutex, set the predicate, and issue the broadcast before **pthread\_cond\_wait()** gets called

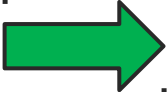
```
pthread_mutex_lock(&mutex);           /* lock the mutex */
the predicate                          /*
/* wake everyone up */
/* unlock the mutex */
```

# Condition Variable: We need the mutex!



```
pthread_mutex_lock(&mutex);           /* lock mutex */
while (!predicate) {                  /* check predicate */
    pthread_cond_wait(&condvar, &mutex);
                                     /* go to sleep - recheck
                                     pred on awakening */
}
pthread_mutex_unlock(&mutex);         /* unlock mutex */
```

---



```
pthread_mutex_lock(&mutex);           /* lock the mutex */
predicate=1;                          /* set the predicate */
pthread_cond_broadcast(&condvar);      /* wake everyone up */
pthread_mutex_unlock(&mutex);         /* unlock the mutex */
```



# Condition Variable: Why do we need the mutex?

## ✗SEPARATING THE CONDITION VARIABLE FROM THE MUTEX

- ✗Thread goes to sleep when it shouldn't

- ✗Problem

  - ✗`pthread_mutex_unlock()` and `pthread_cond_wait()` are not guaranteed to be atomic

## ✗JOINING CONDITION VARIABLE AND MUTEX

- ✗Call to `pthread_cond_wait()` unlocks the mutex

- ✗UNIX kernel can guarantee that the calling thread will not miss the broadcast

# Using a Condition Variable: Challenges

- ✗ CALL `PTHREAD_COND_SIGNAL()` BEFORE CALLING `PTHREAD_COND_WAIT()`
  - ✗ Logical error – waiting thread will not catch the signal
- ✗ FAIL TO LOCK THE MUTEX BEFORE CALLING `PTHREAD_COND_WAIT()`
  - ✗ May cause it NOT to block
- ✗ FAIL TO UNLOCK THE MUTEX AFTER CALLING `PTHREAD_COND_SIGNAL()`
  - ✗ May not allow a matching `pthread_cond_wait()` routine to complete (it will remain blocked).

# Example without Condition Variables

```
int data_avail = 0;
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *) {
    pthread_mutex_lock(&data_mutex);

    <Produce data>
    <Insert data into queue;>
    data_avail=1;

    pthread_mutex_unlock(&data_mutex);
}
```

# Example without Condition Variables

```
void *consumer(void *) {  
    while( !data_avail ); /* do nothing */  
  
    pthread_mutex_lock(&data_mutex);  
  
    <Extract data from queue;>  
    if (queue is empty)  
        data_avail = 0;  
  
    pthread_mutex_unlock(&data_mutex);  
    <Consume Data>  
}
```

Busy Waiting!

# Example with Condition Variables

```
int data_avail = 0;
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t data_cond = PTHREAD_COND_INITIALIZER;

void *producer(void *) {
    pthread_mutex_lock(&data_mutex);
    <Produce data>
    <Insert data into queue>
    data_avail = 1;

    pthread_cond_signal(&data_cond);
    pthread_mutex_unlock(&data_mutex);
}
```

# Example with Condition Variables

Mutex solution

```
while( !data_avail ); /* do nothing */
```

```
void *consumer(void *) {  
    pthread_mutex_lock(&data_mutex);  
    while( !data_avail ) {  
        /* sleep on condition variable*/  
        pthread_cond_wait(&data_cond, &data_mutex);  
    }  
    /* woken up */  
    <Extract data from queue;>  
    if (queue is empty)  
        data_avail = 0;  
    pthread_mutex_unlock(&data_mutex);  
    <Consume Data>  
}
```

**No Busy Waiting!**



# More Complex Example

## xMASTER THREAD

- xSpawns a number of concurrent slaves
- xWaits until all of the slaves have finished to exit
- xTracks current number of slaves executing

xA MUTEX IS ASSOCIATED WITH COUNT AND A CONDITION VARIABLE WITH THE MUTEX

# Example

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

#define NO_OF_PROCS 4

typedef struct _SharedType {
    int count; /* number of active slaves */
    pthread_mutex_t lock; /* mutex for count */
    pthread_cond_t done; /* sig. by finished slave */
} SharedType, *SharedType_ptr;

SharedType_ptr shared_data;
```

# Example: main

```
main(int argc, char **argv) {  
    int res;  
  
    /* allocate shared data */  
    if ((sh_data = (SharedType *)  
        malloc(sizeof(SharedType))) == NULL) {  
        exit(1);  
    }  
    sh_data->count = 0;  
  
    /* allocate mutex */  
    if ((res = pthread_mutex_init(&sh_data->lock,  
        NULL)) != 0) {  
        exit(1);  
    }  
}
```

```
/* allocate condition var */  
if ((res = pthread_cond_init(&sh_data->done,  
    NULL)) != 0) {  
    exit(1);  
}  
  
/* generate number of slaves to create */  
srand(0);  
/* create up to 15 slaves */  
master((int) random()%16);  
}
```

# Example: main

```
main(int argc, char **argv) {  
    int res;  
  
    /* allocate shared data */  
    if ((sh_data = (SharedType *)  
        malloc(sizeof(SharedType))) == NULL) {  
        exit(1);  
    }
```

```
    sh_data->count = 0;
```

```
    /* allocate mutex */
```

```
    pthread_mutex_t data_mutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

```
    pthread_cond_t data_cond =  
    PTHREAD_COND_INITIALIZER;
```

```
    srand(0);
```

```
    /* create up to 15 slaves */
```

```
    master((int) random()%16);
```

```
}
```

# Example: Master

```
master(int nslaves) {  
    int i;  
    pthread_t id;  
    for (i = 1; i <= nslaves; i += 1) {  
        pthread_mutex_lock(&sh_data->lock);  
        /* start slave and detach */  
        shared_data->count += 1;  
        pthread_create(&id, NULL,  
            (void* (*)(void*))slave,  
            (void *)sh_data);  
        pthread_mutex_unlock(&sh_data->lock);  
    }  
}
```

```
pthread_mutex_lock(&sh_data->lock);
```

```
while (sh_data->count != 0)  
    pthread_cond_wait(&sh_data->done, &sh_data->lock);
```

```
pthread_mutex_unlock(&sh_data->lock);
```

```
printf("All %d slaves have finished.\n", nslaves);  
pthread_exit(0);  
}
```

# Example: Slave

```
void slave(void *shared) {  
    int i, n;  
    sh_data = shared;  
    printf("Slave.\n", n);  
    n = random() % 1000;
```

```
    for (i = 0; i < n; i += 1)  
        Sleep(10);
```

```
    /* mutex for shared data */  
    pthread_mutex_lock(&sh_data->lock);
```

```
    /* dec number of slaves */  
    sh_data->count -= 1;
```

```
    /* done running */  
    printf("Slave finished %d cycles.\n", n);
```

```
    /* signal that you are done working */  
    pthread_cond_signal(&sh_data->done);
```

```
    /* release mutex for shared data */  
    pthread_mutex_unlock(&sh_data->lock);
```

```
}
```

# Semaphores vs. CVs

## xSEMAPHORE

- xInteger value ( $\geq 0$ )
- xWait does not always block
- xSignal either releases thread or inc's counter
- xIf signal releases thread, both threads continue afterwards

## xCONDITION VARIABLES

- xNo integer value
- xWait always blocks
- xSignal either releases thread or is lost
- xIf signal releases thread, only one of them continue

# Dining Philosophers

- N PHILOSOPHERS AND N FORKS
- PHILOSOPHERS EAT/THINK
- EATING NEEDS 2 FORKS
- PICK ONE FORK AT A TIME

