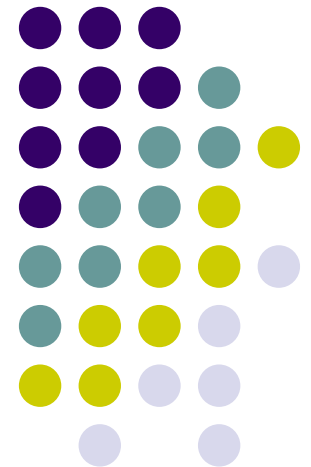
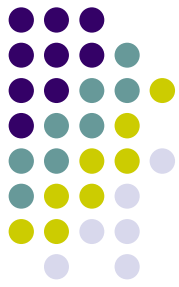


CS241

System Programming

Discussion Section 4
Feb 13 – Feb 16





Outline

- Synchronization
 - Condition Variables
 - Producer-Consumer Problem
 - Reader-Writer Problem
- CPU Scheduling
 - Metrics
 - Examples

Review



- What is wrong with the following code fragment in multi-threaded environment? How would you modify it?

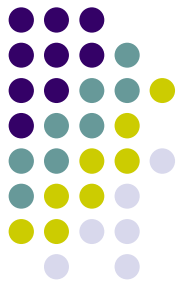
```
static int count;

void increment(void) {
    count++;
}

void decrement(void) {
    count--;
}

int getcount() {
    return count;
}
```

Review

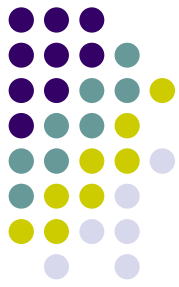


- Thread-Safe version (using mutex)

```
static int count = 0;
static pthread_mutex_t countlock = PTHREAD_MUTEX_INITIALIZER;

int increment(void) { /* decrement() similar */
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    count++;
    return pthread_mutex_unlock(&countlock);
}

int getcount(int *countp) {
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    *countp = count;
    return pthread_mutex_unlock(&countlock);
}
```

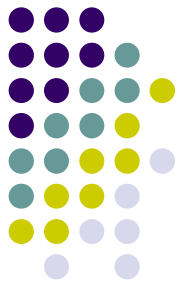


Condition Variables

- Allows explicit event notification
- Implements a monitor along with a mutex

- `#include <pthread.h>`
- **Type:** `pthread_cond_t`
- **Two main operations**
 - **Wait:** `pthread_cond_wait`
 - **Signal:** `pthread_cond_signal`

Example

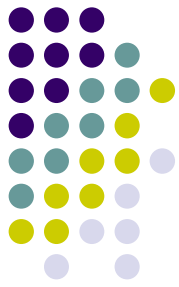


- Waiting for $x==y$ condition

```
pthread_mutex_lock(&m);  
while (x != y)  
    pthread_cond_wait(&v, &m);  
    /* modify x or y if necessary */  
pthread_mutex_unlock(&m);
```

- Notifying the waiting thread that it has incremented x

```
pthread_mutex_lock(&m);  
x++;  
pthread_cond_signal(&v);  
pthread_mutex_unlock(&m);
```



Creating / Destroying CVs

- Creating a condition variable

- Standard Initializer

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

- Static Initializer

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Destroying a condition variable

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

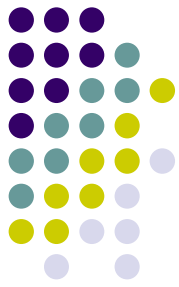
- Returns 0 if successful, nonzero error code if unsuccessful



Waiting on CVs

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

- Called with a mutex lock held
- Internals
 - Causes the thread to release the mutex
 - Sleeps until signaled
 - Reacquires the lock when waken up
- Variation: `pthread_cond_timedwait`



Signaling on CVs

- Signal

- Wakes up one waiting thread

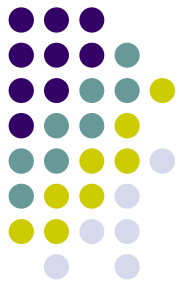
```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Broadcast

- Wakes up all waiting threads

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

Example



- Thread-safe Barrier (Program 13.13)

```
int waitbarrier(void) {      /* wait at the barrier until all n threads arrive */
    int berror = 0;
    int error;

    if (error = pthread_mutex_lock(&bmutex))          /* couldn't lock, give up */
        return error;
    if (limit <= 0) {      /* make sure barrier initialized (limit = #threads) */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    count++;
    while ((count < limit) && !berror)
        berror = pthread_cond_wait(&bcond, &bmutex);
    if (!berror)
        berror = pthread_cond_broadcast(&bcond);      /* wake up everyone */
    error = pthread_mutex_unlock(&bmutex);
    if (berror)
        return berror;
    return error;
}
```

Producer-Consumer Problem



- Given variables

```
static buffer_t buffer[BUFSIZE];
static pthread_mutex_t bufferlock = PTHREAD_MUTEX_INITIALIZER;
static int bufin = 0;
static int bufout = 0;
static pthread_cond_t items = PTHREAD_COND_INITIALIZER;
static pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
static int totalitems = 0;
```

- Implement the following functions using CVs

- `int getitem(buffer_t *itemp)`
 - removes item from butter and put in *itemp
- `int putitem(buffer_t item)`
 - Inserts item in the buffer

Implementation using CVs



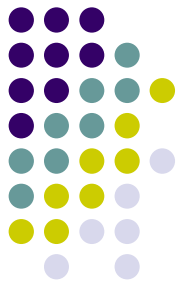
```
int getitem(buffer_t *itemp) {
    int error;
    if (error = pthread_mutex_lock(&bufferlock))
        return error;
    while ((totalitems <= 0) && !error)
        error = pthread_cond_wait (&items, &bufferlock);
    if (error) {
        pthread_mutex_unlock(&bufferlock);
        return error;
    }
    *itemp = buffer[bufout];
    bufout = (bufout + 1) % BUFSIZE;
    totalitems--;
    if (error = pthread_cond_signal(&slots)) {
        pthread_mutex_unlock(&bufferlock);
        return error;
    }
    return pthread_mutex_unlock(&bufferlock);
}
```

Implementation using CVs



```
int putitem(buffer_t item) {
    int error;
    if (error = pthread_mutex_lock(&bufferlock))
        return error;
    while ((totalitems >= BUFSIZE) && !error)
        error = pthread_cond_wait (&slots, &bufferlock);
    if (error) {
        pthread_mutex_unlock(&bufferlock);
        return error;
    }
    buffer[bufin] = item;
    bufin = (bufin + 1) % BUFSIZE;
    totalitems++;
    if (error = pthread_cond_signal(&items)) {
        pthread_mutex_unlock(&bufferlock);
        return error;
    }
    return pthread_mutex_unlock(&bufferlock);
}
```

Producer-Consumer Problem



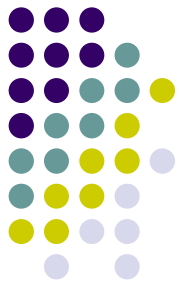
- **Given variables**

```
static buffer_t buffer[BUFSIZE];
static pthread_mutex_t  bufferlock = PTHREAD_MUTEX_INITIALIZER;
static int bufin = 0;
static int bufout = 0;
static sem_t semitems;
static sem_t semslots;
```

- **Implement the following functions using semaphores**

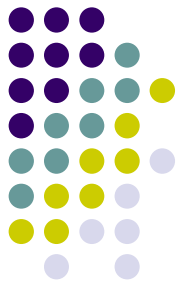
- `int getitem(buffer_t *itemp)`
 - removes item from butter and put in *itemp
- `int putitem(buffer_t item)`
 - Inserts item in the buffer

Implementation using Semaphores



```
int getitem(buffer_t *itemp) {
    int error;
    while (((error = sem_wait(&semitems)) == -1) && (errno == EINTR)) ;
    if (error)
        return errno;
    if (error = pthread_mutex_lock(&bufferlock))
        return error;
    *itemp = buffer[bufout];
    bufout = (bufout + 1) % BUFSIZE;
    if (error = pthread_mutex_unlock(&bufferlock))
        return error;
    if (sem_post(&semslots) == -1)
        return errno;
    return 0;
}
```

Implementation using Semaphores



```
int putitem(buffer_t item) {
    int error;
    while (((error = sem_wait(&semslots)) == -1) && (errno == EINTR)) ;
    if (error)
        return errno;
    if (error = pthread_mutex_lock(&bufferlock))
        return error;
    buffer[bufin] = item;
    bufin = (bufin + 1) % BUFSIZE;
    if (error = pthread_mutex_unlock(&bufferlock))
        return error;
    if (sem_post(&semitems) == -1)
        return errno;
    return 0;
}
```



Reader-Writer Problem

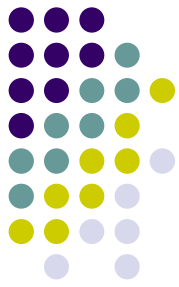
- Two types of access
 - Read: may be shared
 - Write: must be exclusive
- Reader-Writer Synchronization
 - Strong Reader Synchronization
 - Preference to readers (e.g. a library database)
 - Strong Writer Synchronization
 - Preference to writers (e.g. an airline reservation system)
- POSIX provides read-write locks



Read-Write Locks

- Allows multiple readers to acquire a lock
 - When a writer does not hold the lock
- `#include <pthread.h>`
- **Type:** `pthread_rwlock_t`
- **Three main operations**
 - Read Lock: `pthread_rwlock_rdlock`
 - Write Lock: `pthread_rwlock_wrlock`
 - Unlock: `pthread_rwlock_unlock`

Creating / Destroying rwlock



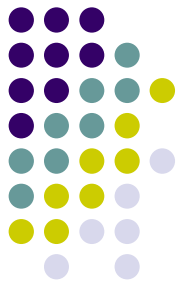
- Creating a read-write lock

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,  
                        const pthread_rwlockattr_t *restrict attr);
```

- Destroying a read-write lock

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

- Returns 0 if successful, nonzero error code if unsuccessful



Acquiring / Releasing rwlock

- Acquiring a read-write lock

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

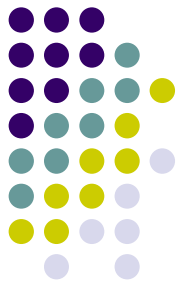
- Releasing a read-write lock

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

- Returns 0 if successful, nonzero error code if unsuccessful

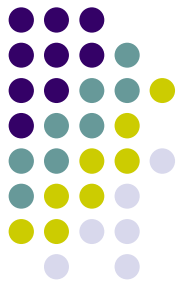
- Implementations may favor writers over readers to avoid writer starvation.

CPU Scheduling

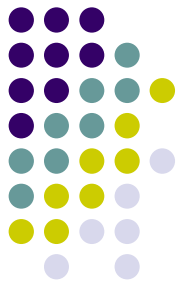


- Decides which thread should be in the running state
- Algorithms
 - First-Come First-Serve (FCFS)
 - Shortest Job First (SJF)
 - Round Robin (RR)
 - Priority

CPU Scheduling



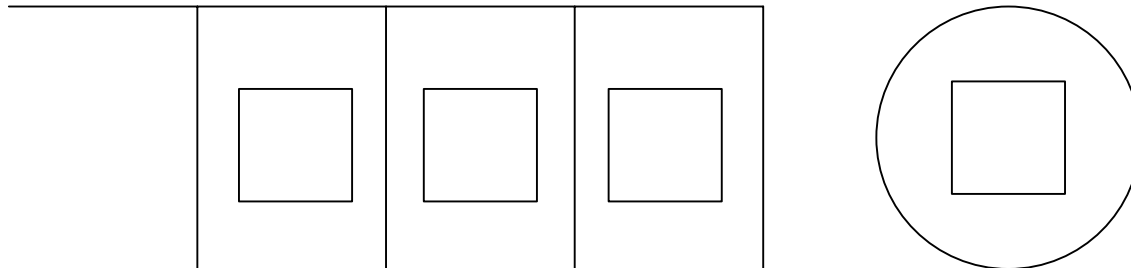
- Preemptive vs Non-preemptive
 - Non-Preemptive: The running thread keeps CPU until it voluntarily gives it up
 - Preemptive: The running thread can be forced to give up CPU by another thread
- Metrics
 - Waiting Time
 - Total amount time that a thread waits
 - Turnaround Time
 - (Thread finish time – thread entry time)

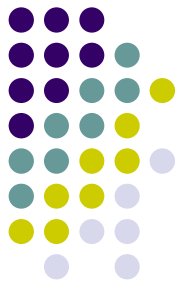


A Simple Example (Priority)

Process	Duration	Priority #	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

Initial Condition

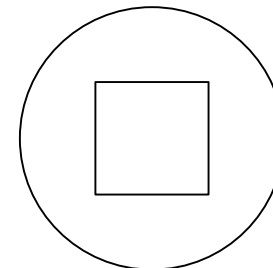
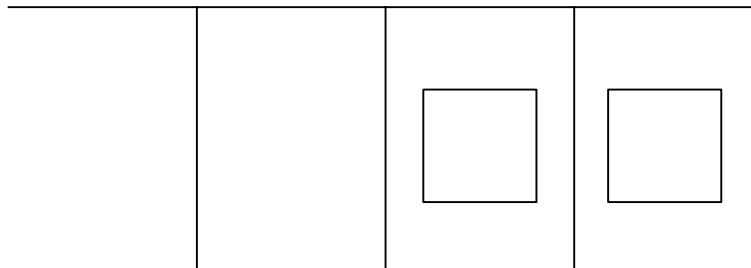




A Simple Example (Priority)

Process	Duration	Priority #	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

After 8 seconds

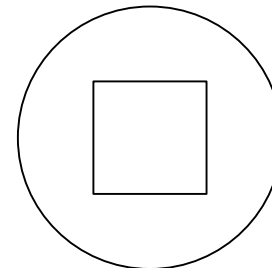
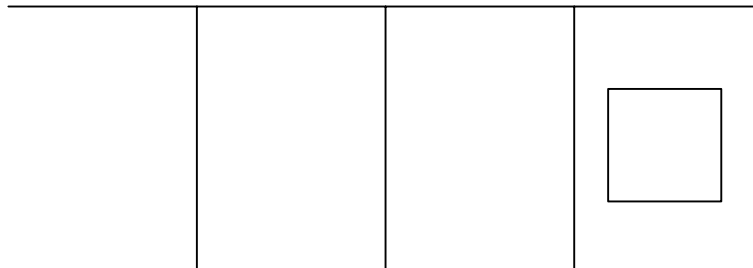


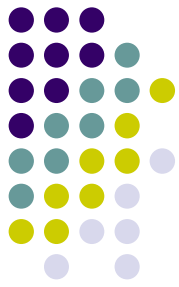


A Simple Example (Priority)

Process	Duration	Priority #	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

After 11 seconds

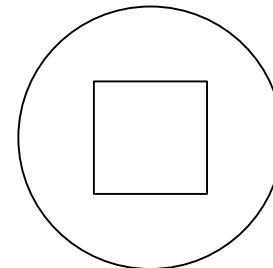
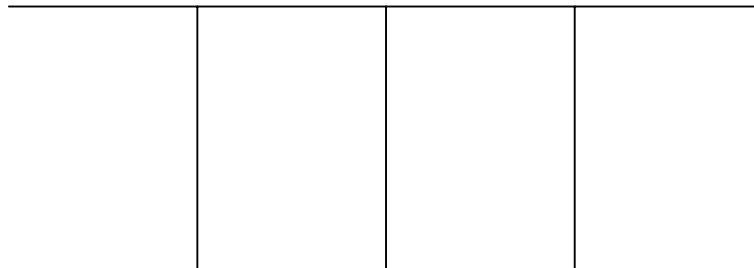


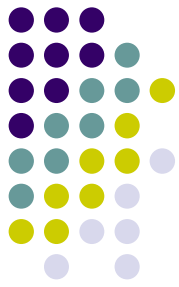


A Simple Example (Priority)

Process	Duration	Priority #	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

After 18 seconds

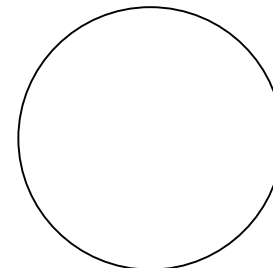
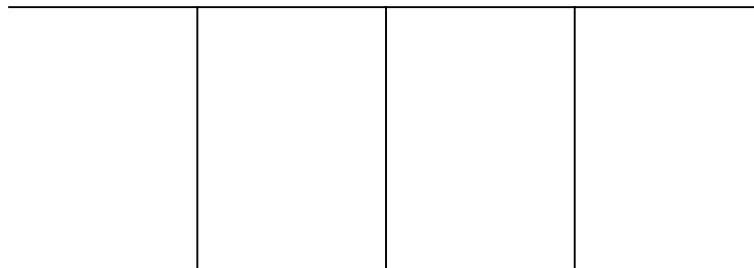




A Simple Example (Priority)

Process	Duration	Priority #	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

After 24 seconds





POSIX Scheduling

- `#include <sched.h>`
- Defines `sched_param` structure
- Defines 4 scheduling policies
 - `SCHED_FIFO`
 - `SCHED_RR`
 - `SCHED_SPORADIC`
 - `SCHED_OTHER`
- Defines several scheduling functions. For example,
 - `int sched_yield(void);`



Summary

- Synchronization
 - Condition Variables: `pthread_cond_*`
 - Producer-Consumer Problem
 - Using condition variables
 - Using semaphores
 - Reader-Writer Problem: `pthread_rwlock_*`
- CPU Scheduling
 - Metrics
 - Waiting Time, Turnaround Time
 - Examples