Other Classical Synchronization Problems

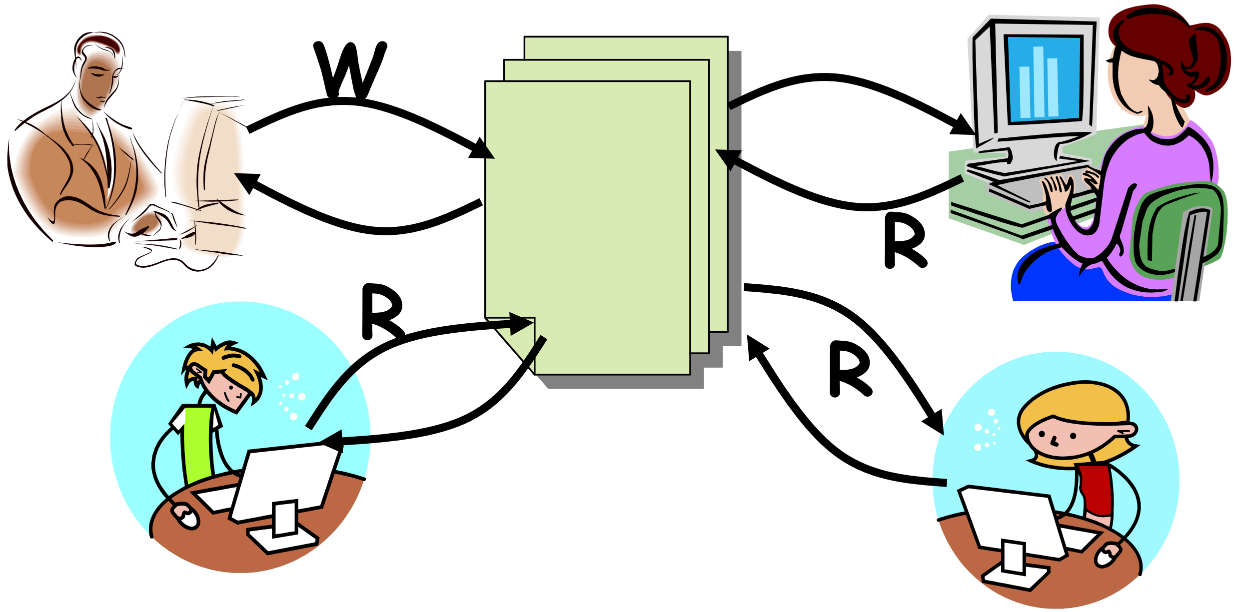
Outline

- Classic Synchronization problems
  - Producer-Consumer with bounded buffer (already covered)
  - Reader-Writer
  - Dining Philosophers
- Solutions to classic problems via semaphores
- Synchronization in Windows/Linux/Pthreads

Readers and Writers

- In this model, threads share data that
  - some threads “read” and other threads “write”.
- Instead of CSEnter and CSExit we want
  - StartRead…EndRead; StartWrite…EndWrite
- Goal: allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers and other writers wait for it to finish

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

Readers-Writers Problem

- Courtois et al 1971
- Models access to a database
  - A <u>reader</u> is a thread that needs to look at the database but won't change it.
  - A <u>writer</u> is a thread that modifies the database
- Example: booking movie tickets on BookMyShow
  - When you browse to look at movie schedules and seat availability, the website is acting as a reader on your behalf
  - When you reserve a ticket, the website has to write into the database to make the reservation

Readers-Writers Problem

- Many threads share an object in memory
  - Some write to it, some only read it
  - Only one writer can be active (enters CS) at a time
  - Any number of readers can be active simultaneously
- Key insight: generalizes the critical section concept
- One issue we need to settle, to clarify problem statement.
  - Suppose that a writer is active and a mixture of readers and writers now shows up. Who should get in next?
  - Or suppose that a writer is waiting and an endless of stream of readers keeps showing up. Is it fair for them to become active?
- Ideally we would like a kind of back-and-forth form of fairness:
  - Once a reader is waiting, readers will get in next.
  - If a writer is waiting, one writer will get in next.

Readers-Writers

```
Shared variables: Semaphore mutex, rwl;
                  integer rcount;

Init: mutex = 1, rwl = 1, rcount = 0;



Writer
do {

    Wait(rwl);
    . . .
    /*writing is performed*/
    . . .
    Signal(rwl);

}while(TRUE);
```

```
Reader
do {
    Wait(mutex);
    rcount++;
    if (rcount == 1)
        Wait(rwl);

    Signal(mutex);
    . . .
    /*reading is performed*/
    . . .
    Wait(mutex);
    rcount--;
    if (rcount == 0)
        Signal(rwl);

    Signal(mutex);
}while(TRUE);
```

CS parts??

Readers-Writers Notes

- If there is a writer active
  - First reader blocks on **rwl**
  - Other readers block on **mutex**
- Once a reader is active, all readers get to go through
  - Any problem?
- Once a writer exits, which one gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers waiting on **rwl**, and writer exits
  - Who gets to go in first?
  - Reader (which Reader gets in first?) or Writer?
- Why doesn't a writer need to use **mutex**?

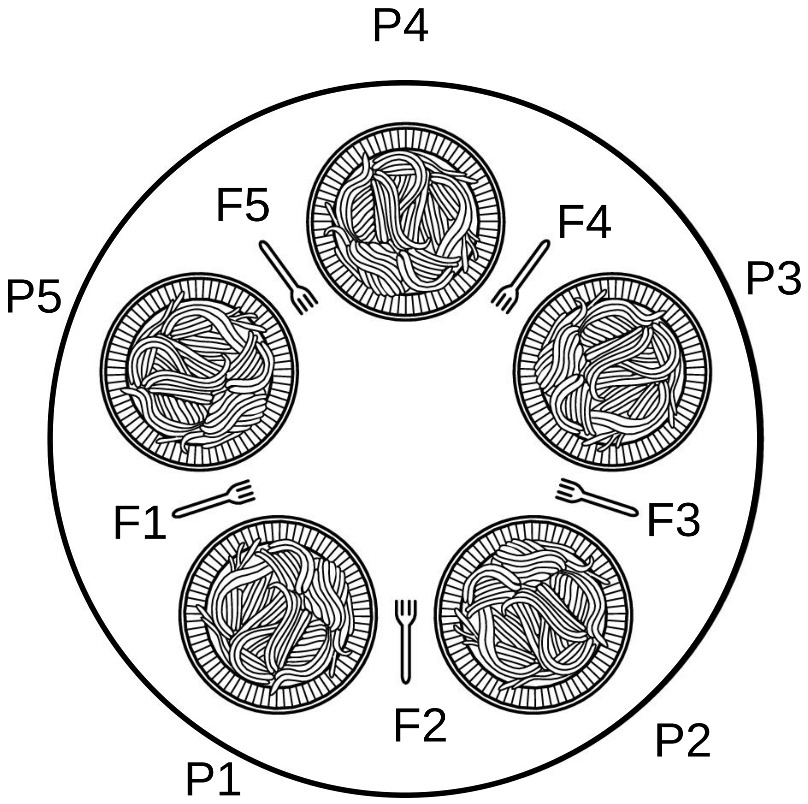
Does this work as we hoped?

- If readers are active, no writer can enter
  - The writers wait doing a Wait(**rwl**)
- While writer is active, nobody can enter
  - Any other reader or writer will wait
- But back-and-forth switching is buggy:
  - Any number of readers can enter in a row
  - Readers can “starve” writers
- With semaphores, building a solution that has the desired back-and-forth behavior is really, really tricky!
  - We recommend that you try, but not too hard…

Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- Fork → Binary Semaphore
  - take_fork(i): Wait(i)
  - put_fork(i): Signal(i)

- Any problem?
- How to prevent deadlock?



Dining Philosophers (2)

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```

A nonsolution to the dining philosophers problem

Dining Philosophers (3)

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N−1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];   = {0}         /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
```

Philosopher → Binary Semaphore

Solution to dining philosophers problem (part 1)

Dining Philosophers (4)

```
void take_forks(int i)                 /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                    /* enter critical region */
      state[i] = HUNGRY;               /* record fact that philosopher i is hungry */
      test(i);                         /* try to acquire 2 forks */
      up(&mutex);                      /* exit critical region */
      down(&s[i]);                     /* block if forks were not acquired */
}

void put_forks(i)                      /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                    /* enter critical region */
      state[i] = THINKING;             /* philosopher has finished eating */
      test(LEFT);                      /* see if left neighbor can now eat */
      test(RIGHT);                     /* see if right neighbor can now eat */
      up(&mutex);                      /* exit critical region */
}

void test(i)                           /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Solution to dining philosophers problem (part 2)

Summary: semaphores

- Semaphores are very “low-level” primitives
  - Users could easily make small errors
  - Similar to programming in assembly language
    - Small error brings system to grinding halt
  - Very difficult to debug

- Also, we seem to be using them in two ways
  - For mutual exclusion, the “real” abstraction is a critical section
  - But the bounded buffer example illustrates something different, where threads “communicate” using semaphores

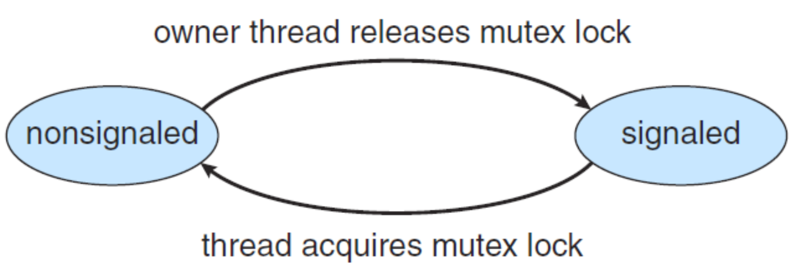
- Simplification: Provide concurrency support in compiler
  - Monitors (next class)

Synchronization Examples

- Windows
- Linux
- Pthreads

Windows Synchronization

- Multi-threaded kernel with so many global, shared resources
- Uses interrupt masks to protect access to global resources on single-CPU systems
- Uses **spinlocks** on multi-CPU systems
  - Spinlocked-thread will never be preempted by the kernel
- Outside kernel, it provides **dispatcher objects** which may act as mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more threads when time expired
  - Dispatcher objects in either **signaled-state** (object available) or **non-signaled state** (thread will block)



Mutex dispatcher object

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive and reentrant
- Linux provides following locks for sync in kernel:
  - Atomic integer (***atomic_t*** *counter*)
  - semaphores
  - spinlocks
  - Mutex locks (no spin, but blocking)
  - reader-writer versions of both
- SMP systems: fundamental locking mechanism is spinlocks
  - But kernel is coded such that these are held only for a shot while
- On single-CPU system, spinlocks are replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent and available to user/application programs
- It provides:
  - mutex locks (no spin, but blocking)
  - condition variables (no state associated unlike counting semaphores)
    - Refer Tanenbaum MOS: Fig 2.32 for sol. to P-C prob.
  - Read-write locks
  - Semaphores as part of POSIX SEM extension
    - Named → actual name in file system and shared among processes
    - Unnamed → among threads of same process
    - Wait: do not decrease its value below zero

Pthreads Synchronization Example

```
#include <pthread.h>
pthread_mutex_t mutex;
…
/* create the mutex lock */
pthread_mutex_init(&mutex,NULL); //dynamic
….
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
…
pthread_mutex_unlock(&mutex);
```

```
mutex=PTHREAD_MUTEX_INITIALIZER; //static
pthread_mutex_trylock(&mutex)
pthread_mutex_timedlock(&mutex,time)
pthread_mutex_destroy(&mutex);//for dynamic
```

Pthreads Synchronization Example

#include <semaphore.h>

sem_t sem;

/* Create the semaphore and initialize it to 1 */

sem_init(&sem, 0, 1); // 0: no sharing with other processes, but only among other threads

....

/* acquire the semaphore */

sem_wait(&sem);

/* critical section */

/* release the semaphore */

sem_post(&sem);

…

Other Approaches

- Multi-core systems
  - Make designing deadlock-free and starvation-free multithreading apps very difficult
- Transaction memory
  - From Databases
  - If all operations in a transaction (set of RW operations) are completed, it is committed
  - If not, roll back
  - It guarantees atomicity, so chances of deadlock and starvation bcz of developers mistakes!
- Compiler's helping hand: Example: OpenMP
  - #pragma omp critical

    Count++

Reading Assignment

- Chapter 5 from OSC by Galvin et al 9th Edition
- Chapter 2 from MOS by Tanenbaum e al
- http://man7.org/linux/man-pages/man7/sem_overview.7.html
- https://computing.llnl.gov/tutorials/openMP/
- https://computing.llnl.gov/tutorials/pthreads/

Quiz 1

- January 21st (TUE) at 2:30PM
- Syllabus: Lessons 1-4 on Synchronization