Avoiding Waiting

- Solutions seen so far teach us:
  - How to ensure exclusive access
  - How to avoid deadlocks

- But, in all cases, if P0 is in CS, and P1 needs CS, then P1 is busy waiting, checking repeatedly for some condition to hold. Waste of system resources!

- Suppose we have following system calls for synchronization
  - Sleep: puts the calling thread/process in a blocked/waiting state
  - Wakeup(pid): puts the argument thread/process in ready state

### The Producer/Consumer Problem

- from time to time, the producer places an item in the buffer
- the consumer removes an item from the buffer
- careful synchronization required
- the consumer must wait if the buffer empty
- the producer must wait if the buffer full
- typical solution would involve a shared variable `count`
- also known as the Bounded Buffer problem

### Sleep/wakeup Solution to Producer-Consumer Problem

- bounded buffer (of size `N`)
- producer writes to it, consumer reads from it
- Solution using sleep/wakeup synchronization

```c
int count = 0; /* number of items in buffer */
Producer code:
while (TRUE) {
    /* produce */
    if (count == N)
        sleep;
    /* add to buffer */
    count = count + 1;
    if (count == 1)
        wakeup(Consumer);
}
Consumer code:
while (TRUE) {
    if (count == 0)
        sleep;
    /* remove from buffer*/
    count = count -1;
    if (count == N-1)
        wakeup(Producer);
    /* consume */
}
```

### Problematic Scenario

- Count is initially 0
- Consumer reads the count, gets swapped out
- Producer produces an item, inserts it, and increments count to 1
- Producer executes wakeup, but there is no waiting consumer (at this point)
- Consumer continues its execution and goes to sleep
- Consumer stays blocked forever unnecessarily
- Main problem: wakeup was lost

Solution: Semaphores keeping counts

### Dijkstra's Semaphores

- A semaphore `s` has a non-negative integer value
- It supports two operations
  - `up(s)` or `V(s)`: simply increments the value of `s`
  - `down(s)` or `P(s)`: decrements the value of `s` if `s` is positive, else makes the calling process wait
- When `s` is 0, `down(s)` does not cause busy waiting, rather blocks process
- Internally, there is a queue of sleeping processes
- When `s` is 0, `up(s)` also wakes up one sleeping process (if there are any)
- `up` and `down` calls are executed as atomic actions
Mutual Exclusion using Semaphores

Shared variable: a single semaphore s = 1
Solution for any process

while (TRUE) {
    down(s); /* wait for s to be 1 */
    CS();
    up(s); /* unblock a waiting process */
    Non_CS();
}
• No busy waiting
• Works for an arbitrary number of processes, i ranges over 0..n

The Producer-Consumer Problem

• bounded buffer (of size n)
• one set of processes (producers) write to it
• one set of processes (consumers) read from it

semaphore: full = 0; /* number of full slots */
empty = n; /* number of empty slots */
mutex = 1; /* binary semaphore for CS */

Producer code:
while (TRUE) {
    /* produce */
    down(&empty);
    down(&mutex);
    /*add to buffer */
    up(&mutex);
    up(&full);
}

Consumer code:
while (TRUE) {
    down(&full);
    down(&mutex);
    /*remove from buffer*/
    up(&mutex);
    up(&empty);
    /* consume */
}

Mutex and Thread Synchronization

• Mutex is a simplified semaphore – binary semaphore
• Can only be locked or unlocked
• int pthread_mutex_init(pthread_mutex_t *mp, const
    pthread_mutexattr_t *attr): initialize the mutex referenced by
    mp with attributes specified by attr. If attr is NULL, the default attributes
    are used;
• int pthread_mutex_destroy(pthread_mutex_t *mp): destroys
    any state related to the mutex pointed to by mp.
• int pthread_mutex_lock(pthread_mutex_t *mp): locks the
    calling thread until the mutex pointed to by mp unlocked, and then it
    atomically locks the mutex.
• int pthread_mutex_unlock(pthread_mutex_t *mp): atomically
    unlocks the mutex pointed to by mp.
• int pthread_mutex_trylock(pthread_mutex_t *mp): atomically
    locks the mutex pointed to by mp if currently unlocked and
    returns 0; otherwise, it returns non-zero value.
• int pthread_mutex_timeline(pthread_mutex_t *mp): atomically
    unlocks the mutex pointed to by mp. If there are any threads
    blocked on the mutex, one will be unblocked.

POSIX Semaphore System Calls

• int sem_init(sem_t *sp, unsigned int count, int
    type): Initialize semaphore pointed to by sp to count. type can assign
    several different types of behaviors to a semaphore
• int sem_destroy(sem_t *sp): destroys any state related to the
    semaphore pointed to by sp.
• int sem_wait(sem_t *sp): blocks the calling thread until the
    semaphore count pointed to by sp is greater than zero, and then it
    atomically decrements the count.
• int sem_trywait(sem_t *sp): atomically decrements the
    semaphore count pointed to by sp if the count is greater than zero;
    otherwise, it returns -1 and sets errno.
• int sem_post(sem_t *sp): atomically increments the semaphore
    count pointed to by sp. If there are any threads blocked on the semaphore,
    one will be unblocked.

OS-level support (mutual exclusion and synchronization)
• Special variables: Semaphores, Mutexes
• Message passing primitives (send and receive)

Low-level (for mutual exclusion)
• Interrupt disabling
• Using read/write instructions
• Using powerful instructions [Test-and-set, Compare-and-Swap,...]

Roadmap

High-level Synchronization Primitives
• Monitors (Hoare, Brinch-Hansen)
• Synchronized method in Java

Idealized Problems
• Producer-Consumer
• Dining Philosophers
• Readers-Writers

How to prevent deadlock
• Philosophers eat/think
• Eating needs 2 forks
• Pick one fork at a time
• How to prevent deadlock

Dining Philosophers

• Philosophers eat/think
• Eating needs 2 forks
• Pick one fork at a time
• How to prevent deadlock
Dining Philosophers (2)

```c
#include <semaphore.h>

#define N 5

void philosopher(int i)
{
  /* philosopher number, from 0 to 4 */
  while (TRUE) {
    /* philosopher is thinking */
    think();
    /* take left fork */
    take_hex(i);
    /* take fork(i+1) % N; */
    eat();
    /* put fork(i) back on the table */
    put_hex(i);
    /* put left fork back on the table */
    put_hex((i+1)%N);
  }
}

A nonsolution to the dining philosophers problem
```

The Dining Philosopher Problem

- One simple solution is to represent each fork by a semaphore.
- Down before picking it up & up after using it.

```c
#define fork array[0..4] of semaphores=1

philosopher i
{
  repeat
    down(fork[i]);
    down(fork[(i+1)%5]);
    eat;
    up(fork[i]);
    up(fork[(i+1)%5]);
    think
    forever
}
```

- Is deadlock possible?

Number of possible states

- 5 philosophers
- Local state (LC) for each philosopher
  - thinking, waiting, eating
- Global state = (LC1, LC2, ..., LC5)
  - E.g., (thinking, waiting, waiting, eating, thinking)
  - E.g., (waiting, eating, waiting, eating, waiting)
- So, the number of global states are 3^5 = 243
- Actually, it is a lot more than this since waiting can be
  - Waiting for the first fork
  - Waiting for the second fork

Readers and Writers Problem

- Goal: Design critical section access so that it has
  - Either a single writer
  - Or one or more readers (a reader does not block another reader)
- First step: Let’s use a semaphore, wrt, that protects the critical section
  - Initially wrt is 1
  - wrt should be zero whenever a reader or writer is inside it
- Code for writer:
  down(wrt); write(); up(wrt);
- How to design a reader?
  - Only the first reader should test the semaphore (i.e., execute down(wrt))

Readers and Writers Problem

- More on Reader’s code
  - To find out if you the first one, maintain a counter, readcount, that keeps the number of readers
- First attempt for reader code:
  readcount++;
  if (readcount==1) down(wrt);
  read();
  readcount--;
- What are the problems with above code?
Readers and Writers Problem

• Corrected reader code:
  
down(mutex);
  reacount++;
  if (readcount==1) down(wrt);
  up(mutex);
  read();
  down(mutex);
  reacount--;
  if (readcount==0) up(wrt);
  up(mutex);

The Sleeping Barber Problem

Other Solutions

• PL-based solutions
  – Monitor
  – Synchronized methods
• Message-based IPC

Summary of IPC

• Two key issues:
  – Mutual exclusion while accessing shared data
  – Synchronization (sleep/wake-up) to avoid busy waiting
• Solutions at many levels
  – Low-level (Peterson’s, test-and-set)
  – System calls (semaphores, message passing)
  – Programming language level (monitors)
• Solutions to classical problems
  – Correct operation in worst-case
  – As much concurrency as possible
  – Avoid busy-waiting
  – Avoid deadlocks