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

The 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

\[
\text{while (TRUE) \{ \\
\quad \text{down}(s); /* wait for } s \text{ to be } 1 */ \\
\quad \text{CS();} \\
\quad \text{up}(s); /* unblock a waiting process */ \\
\quad \text{Non-CS();} \\
\text{\}}
\]

- 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

\[
\text{semaphore: full = 0; /* number of full slots */} \\
\text{empty = n; /* number of empty slots */} \\
\text{mutex = 1; /* binary semaphore for CS */}
\]

Producer code:

\[
\text{while (TRUE) \{ \\
\quad \text{down}(\text{empty}); \\
\quad \text{down}(\text{mutex}); \\
\quad \text{/* add to buffer */} \\
\quad \text{up}(\text{mutex}); \\
\quad \text{up}(\text{full}); \\
\text{\}}}
\]

Consumer code:

\[
\text{while (TRUE) \{ \\
\quad \text{down}(\text{full}); \\
\quad \text{down}(\text{mutex}); \\
\quad \text{/* remove from buffer */} \\
\quad \text{up}(\text{mutex}); \\
\quad \text{up}(\text{empty}); \\
\quad \text{/* consume */} \\
\text{\}}}
\]

POSIX Semaphore System Calls

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

Mutex and Thread Synchronization

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

Roadmap

- High-level Synchronization Primitives
  - Monitors (Hoare, Brinch-Hansen)
  - Synchronized method in Java
- Idealized Problems
  - Producer-Consumer
  - Dining Philosophers
  - Readers-Writers
- 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…)

Dining Philosophers

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

```c
#define N 5

void philosopher(int i) {
    /* philosopher is thinking */
    think();
    /* take left fork */
    take_fork(i); 
    /* take right fork */
    take_fork((i+1) % N);
    eat();
    put_fork(i);
    /* put left fork back on the table */
    put_fork((i+1) % N);
}
```

A non-solution 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.
- `var 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 is 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

The Readers and Writers Problem

Shared data to be accessed in two modes: reading and writing.
- Any number of processes permitted to read at one time
- writes must exclude all other operations.

```
Intuitively:
Reader:                    | Writer:
when(no_writers==0) {     | when(no_readers==0 and no_writers==0) {
    no_readers=no_readers+1 |    no_writers = 1
    <read>
}                           |    <write>
no_readers=no_readers-1 |    no_readers=no_readers+1 |    no_writers = 0
-                           |    -
-                           |    -
```

Readers and Writers Problem

- Goal: Design critical section access so that it has
  - Either a single writer
  - Or one or more readers (a reader should 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);
  readcount++;
  if (readcount==1) down(wrt);
  up(mutex);
  read();
  down(mutex);
  readcount--;
  if (readcount==0) up(wrt);
  up(mutex);
  ```

The Sleeping Barber Problem

Solution to sleeping barber problem.

Monitors

- Semaphores are powerful, but low-level, and can lead to many programming errors.
- Elegant, high-level, programming-language-based solution is monitors (proposed by Hoare and by Brinch Hansen).
- A monitor is a shared data object together with a set of operations to manipulate it (i.e., abstract data type).
- To enforce mutual exclusion, at most one process may execute a method for the monitor object at any given time.
- All uses of shared variables should be encapsulated by monitors.
- Data type "condition" for synchronization (can be waited or signaled within a monitor procedure).
- Two operations on "condition" variables:
  - wait: Forces the caller to be delayed, releases the exclusive access.
  - signal: One of the waiting processes is resumed.
- "synchronized" methods in Java are similar

Monitor Example

```java
monitor example
  integer : condition;
  procedure producer();
    ...
  end;
  procedure consumer();
    ...
  end
end monitor;
```

Message-based IPC

- Shared Memory
- Message Passing
  - Signals
  - send
  - receive
Design Attributes

- Naming
  - Process id, mailbox
- Buffering
  - Size: zero, bounded, unbounded
  - Place: kernel space, user space
- Send operation
  - Synchronous vs. asynchronous
- Receive operation
  - Blocking vs. non-blocking

Interprocess Communication

Message Passing

Many possible naming schemes. One is direct naming:

send(process_id, amessage)
receive(process_id, amessage)

Example

process P1:
declare x integer
send(P2, x)
end process

process P2:
declare y integer
receive(P1, y)
end process

Effect of this communication is

```
y := x
```

Summary of IPC

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