Synchronization

Kai Shen

Bounded Buffer

- **Shared data**
  ```c
  typedef struct { ... } item;
  item buffer[BUFFER_SIZE];
  int in = 0, out = 0;
  int counter = 0;
  ```

- **Producer task**
  ```c
  item nextProduced;
  while (1) {
    while (counter==BUFFER_SIZE)
      /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
  }
  ```

- **Consumer task**
  ```c
  item nextConsumed;
  while (1) {
    while (counter==0)
      /* do nothing */
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter--;
  }
  ```

Bounded Buffer

- The statement "counter++" may be compiled into the following instruction sequence:
  ```c
  register1 = counter;
  register1 = register1 + 1;
  counter   = register1;
  ```

- The statement "counter--" may be compiled into:
  ```c
  register2 = counter;
  register2 = register2 - 1;
  counter   = register2;
  ```

- The following statements must be performed atomically:
  ```c
  counter++;
  counter--;
  ```

- Atomic operation means an operation that completes in its entirety without interruption.

Race Condition

- **Race condition:**
  - The situation where several tasks access and manipulate shared data concurrently.
  - The final value of the shared data and/or effects on the participating tasks depends upon the order of task execution – nondeterminism.

- To prevent race conditions, concurrent tasks must be synchronized.
Synchronization Principles

- Background
  - Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating tasks.
- The Critical-Section Problem
  - Pure software solution
  - With help from the hardware
  - Synchronization that coordinates with process/thread scheduler (when waiting, we can yield the CPU instead of busy noop loop)
    - Semaphore
    - Mutex lock
    - Condition variables

The Critical-Section Problem

- Problem context:
  - n tasks all competing to use some shared data
  - Each task has a code segment, called critical section, in which the shared data is accessed.
- Find a solution that satisfies the following:
  1. Mutual Exclusion. No two tasks simultaneously in the critical section.
  2. Progress. No task running outside its critical section may block other tasks.
  3. Bounded Waiting/Fairness. Given the set of concurrent tasks, a bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

Critical Section for Two Tasks

- Only 2 tasks, \( P_0 \) and \( P_1 \)
- General structure of task \( P_i \) (other task \( P_j \))

\[
\begin{align*}
\text{do} \{ \\
\quad &\text{entry section} \ 
\} \text{ while } (1), \\
\quad &\text{critical section} \ 
\} \text{ while } (1), \\
\quad &\text{exit section} \ 
\} \text{ while } (1), \\
\quad &\text{remainder section} \ 
\}
\end{align*}
\]

- Tasks may share some common variables to synchronize their actions.
- Assumption: instructions are atomic and no re-ordering of instructions.

Algorithm 1

- Shared variables:
  - int turn; initially \( \text{turn} = 0; \)
  - \( \text{turn}==i \Rightarrow P_i \) can enter its critical section

Task \( P_i \)

\[
\begin{align*}
\text{do} \{ \\
\quad &\text{while } (\text{turn} != i) ; \\
\quad &\text{critical section} \\
\quad &\text{turn} = j; \\
\quad &\text{remainder section} \\
\} \text{ while } (1); \\
\}
\end{align*}
\]

- Satisfies mutual exclusion, but not progress
Algorithm 2

- Shared variables:
  - boolean flag[2];
  - initially flag[0] = flag[1] = false;
  - flag[i] == true \(\Rightarrow\) \(P_i\) ready to enter its critical section
- Task \(P_i\):
  - do {
    - flag[i] = true;
    - while (flag[j]) ;
    - critical section
    - flag[i] = false;
    - remainder section
  } while (1);
- Satisfies mutual exclusion, but may lead to deadlock.

Algorithm 3

- Combine shared variables of algorithms 1 and 2.
- Task \(P_i\):
  - do {
    - flag[i] = true;
    - turn = j;
    - while (flag[j] && turn==j) ;
    - critical section
    - flag[i] = false;
    - remainder section
  } while (1);
- Meets all three requirements; solves the critical-section problem for two tasks. \(\Rightarrow\) called **Peterson's algorithm**.

Synchronization Using Special Instruction: **TSL** (test-and-set)

- entry_section:
  - TSL R1, LOCK | copy lock to R1 and set lock to 1
  - CMP R1, #0 | was lock zero?
  - JNE entry_section | if it wasn’t zero, lock was set, so loop
  - RET | return; critical section entered

- exit_section:
  - MOV LOCK, #0 | store 0 into lock
  - RET | return; out of critical section

- Solve the synchronization problem
- Work for multiple (>2) tasks
- Instruction atomicity and ordering only necessary on TSL
- What if you have special instruction **SWP** (swap the value of a register and a memory word)?

Solving Critical Section Problem with Busy Waiting

- In all our solutions, a task enters a loop until the entry is granted \(\Rightarrow\) busy waiting (or spin waiting).
- Problems with busy waiting:
  - Waste of CPU time
  - If a task is switched out of CPU during critical section
    - other tasks may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling
- Solution
  - Avoid busy wait as much as possible (yield the processor instead).
  - If you can’t avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the OS kernel)
Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore $S$ – integer variable which can only be accessed via two atomic operations
- Semantics (roughly) of the two operations:
  \[
  \begin{align*}
  \text{wait}(S) \text{ or } \text{P}(S): & \quad \text{wait until } S>0; \\
  \text{signal}(S) \text{ or } \text{V}(S): & \quad S++;
  \end{align*}
  \]
- Solving the critical section problem:
  \[
  \begin{align*}
  \text{Shared data:} & \quad \text{mutex}=1; \\
  \text{Task A:} & \quad \text{wait}(\text{mutex}); \\
  & \quad \text{critical section} \\
  & \quad \text{signal}(\text{mutex}); \\
  & \quad \text{remainder section}
  \end{align*}
  \]

Mutex Lock (Binary Semaphore)

- Mutex lock – a semaphore with only two state: locked/unlocked
- Semantics of the two (atomic) operations:
  \[
  \begin{align*}
  \text{lock(mutex):} & \quad \text{wait until } \text{mutex}==\text{unlocked}; \\
  & \quad \text{mutex}=\text{locked}; \\
  \text{unlock(mutex):} & \quad \text{mutex}=\text{unlocked};
  \end{align*}
  \]
- Can you implement mutex lock using semaphore?
- How about the opposite?

Implement Semaphore Using Mutex Lock

- Data structures:
  \[
  \text{mutex.lock L1, L2; int C;}
  \]
- Initialization:
  \[
  \begin{align*}
  & \text{L1 = unlocked;} \\
  & \text{L2 = locked;} \\
  & \text{C = initial value of semaphore;}
  \end{align*}
  \]
- wait operation:
  \[
  \begin{align*}
  & \text{lock(L1);} \\
  & \text{C --;} \\
  & \text{if (C < 0) \{ \\
  & \text{unlock(L1);} \\
  & \text{lock(L2);} \\
  & \text{\}} \\
  & \text{unlock(L1);} \\
  \end{align*}
  \]
- signal operation:
  \[
  \begin{align*}
  & \text{lock(L1);} \\
  & \text{C ++;} \\
  & \text{if (C <= 0) \{ \\
  & \text{unlock(L1);} \\
  & \text{unlock(L2);} \\
  & \text{\}}
  \end{align*}
  \]

Classic Synchronization Problem: Bounded Buffer Problem

- Shared data
  \[
  \text{buffer;}
  \]
- Producer task
  \[
  \begin{align*}
  \text{while (1) \{ \\
  & \text{... produce an item in nextp;} \\
  & \text{... add nextp to buffer;} \\
  & \text{...}
  \end{align*}
  \]
- Consumer task
  \[
  \begin{align*}
  \text{while (1) \{ \\
  & \text{... remove an item from buffer to nextc;} \\
  & \text{... consume nextc;} \\
  & \text{...}
  \end{align*}
  \]
- Protecting the critical section for safe concurrent execution.
- Synchronizing producer and consumer when buffer is empty/full.
Bounded Buffer Solution

- **Shared data**
  - buffer;
  - semaphore full=0;
  - semaphore empty=n;
  - semaphore mutex=1;

- **Producer task**
  ```
  while (1) {
    ...
    produce an item in nextp;
    ...
    wait(empty);
    ...
  }
  ```

- **Consumer task**
  ```
  while (1) {
    ...
    wait(full);
    wait(mutex);
    remove an item from buffer to nextc;
    signal(mutex);
    signal(empty);
    ...
    consume nextc;
    ...
  }
  ```

Dining-Philosophers Problem

- **Philosopher i (1 ≤ i ≤ 5):**
  ```
  while (1) {
    ...
    eat;
    ...
    think;
    ...
  }
  ```

  - Eating needs both chopsticks (the left and the right one).

Dining-Philosophers Solution

- **Shared data:**
  - semaphore chopstick[5];
  - Initially all values are 1;

- **Philosopher i:**
  ```
  while(1) {
    ...
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think;
    ...
  }
  ```

Thread Synchronization In Practice

- All threads share the same address space
- When only need to protect a short critical section (busy waiting is OK)
  - software/hardware spin locks
  - still has the risk of context switch in the middle of critical section
- For complex synchronization (busy waiting is not OK)
  - semaphore, mutex lock, condition variable, ...
  - cost is higher (may involve operating system)
Synchronization Primitives in Pthreads

- Mutex lock
  - `pthread_mutex_init`
  - `pthread_mutex_destroy`
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`

- Condition variable (used in conjunction with a mutex lock)
  - `pthread_cond_init`
  - `pthread_cond_destroy`
  - `pthread_cond_wait`
  - `pthread_cond_signal`
  - `pthread_cond_broadcast`

Condition Variables

- To allow a task to wait, a condition variable must be declared, as `condition x, y;`
- Condition variable can only be used with the operations wait and signal.
  - The operation `x.wait();` means that the task invoking this operation is suspended until another task invokes `x.signal();`
  - The `x.signal()` operation resumes exactly one suspended task. If no task is suspended, then the signal operation has no effect.
- Unlike semaphore, there is no counting in condition variables

Process Synchronization

- Processes naturally do not share the same address space
- Process synchronization:
  - semaphore
  - shared memory
  - messages