Inter-Process Communications

These slides are created by Dr. Huang of George Mason University. Students registered in Dr. Huang’s courses at GMU can make a single machine readable copy and print a single copy of each slide for their own reference as long as the slide contains the copyright statement, and the GMU facilities are not used to produce the paper copies. Permission for any other use, either in machine-readable or printed form, must be obtained form the author in writing.

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
**Producer-Consumer Problem**

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.

---

**Shared Data**

```c
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```
**Bounded-Buffer – Producer Process**

item nextProduced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}

**Bounded-Buffer – Consumer Process**

item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
Discussions

- Solution is correct, but can only use BUFFER_SIZE-1 elements.
- A solution, where all $N$ buffers are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable `counter`, initialized to 0, and incremented each time a new item is added to the buffer.

Shared Data

```c
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```
Modified Producer

item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

Modified Consumer

item nextConsumed;

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

Bounded Buffer

- The statements
  
  ```
  counter++;  
  counter--;  
  ```

  must be performed *atomically*.

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

- The statement “**count++**” may be implemented in machine language as:
  
  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- The statement “**count--**” may be implemented as:
  
  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```
Problem of Concurrency

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

One Possible Execution

- Assume `counter` is initially 5. One interleaving of statements is:

  producer: `register1 = counter` \((register1 = 5)\)
  producer: `register1 = register1 + 1` \((register1 = 6)\)
  consumer: `register2 = counter` \((register2 = 5)\)
  consumer: `register2 = register2 – 1` \((register2 = 4)\)
  producer: `counter = register1` \((counter = 6)\)
  consumer: `counter = register2` \((counter = 4)\)

- The value of `count` may be either 4 or 6, where the correct result should be 5.
Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section Problem

- $n$ processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
Shared Data of Project 2

Front end

Back end

Shared Area

N:
Command:
Result:

Critical Sections of Project 2

Foreground Process

... 
while (1) {
    Ask for cmd and n;
    Update cmd and n.
    Check results
}
...

Background Process

... 
while (1) {
    Check cmd and n
    Compute results
}
...

Solution Criteria to the Critical-Section Problem

1. **Mutual Exclusion.** If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting.** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the $n$ processes.

---

**Initial Attempts to Solve Problem**

- 2 processes, $P_0$ and $P_1$
- General structure of process $P_i$ (other process $P_j$)
  ```
  do {
    entry section
    critical section
    exit section
    reminder section
  } while (1);
  ```
- Processes may share some common variables to synchronize their actions.
Algorithm 1

- Shared variables:
  - `int turn=0`;
  - `turn = i` ⇒ $P_i$ can enter its critical section
- Process $P_i$
  - do {
    - while (`turn != i`) ;
    - critical section
    - `turn = j`;
    - reminder section
  } while (1);
- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - boolean flag[2];
    - initially `flag [0] = flag [1] = false`.
  - `flag [i] = true` ⇒ $P_i$ ready to enter critical section
- Process $P_i$
  - do {
    - `flag[i] := true;`
    - while (`flag[j]`) ;
    - critical section
    - `flag [i] = false;`
    - remainder section
  } while (1);
- Satisfies mutual exclusion, but not progress.
Revision

- Called Algorithm 3 in the textbook.

- Shared variables
  - `int turn=0;`
    - `turn = i ⇒ P_i` can enter its critical section
  - `boolean flag[2];`
    - initially `flag [0] = flag [1] = false`.
    - `flag [i] = true ⇒ P_i` ready to enter critical section

- Process `P_i`
  ```
  do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) ;
    critical section
    flag [i] = false;
    remainder section
  } while (1);
  ```
Seeing Both Sides

Process $P_i$
\[
\text{do } \{
\begin{align*}
\text{flag}[i] & := \text{true}; \\
\text{turn} & = j; \\
\text{while (flag}[j] \text{ and turn} = j) & \text{;}
\end{align*}
\]
\[\text{critical section}\]
\[
\text{flag}[i] = \text{false};
\]
\[\text{remainder section}\]
\[\text{while (1)};\]

Process $P_j$
\[
\text{do } \{
\begin{align*}
\text{flag}[j] & := \text{true}; \\
\text{turn} & = i; \\
\text{while (flag}[i] \text{ and turn} = i) & \text{;}
\end{align*}
\]
\[\text{critical section}\]
\[
\text{flag}[i] = \text{false};
\]
\[\text{remainder section}\]
\[\text{while (1)};\]

Correctness

- Mutual exclusion ?
- Progress ?
- Bounded waiting ?
Bakery Algorithm

Critical section for $n$ processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

- If processes $P_i$ and $P_j$ receive the same number, if $i < j$, then $P_i$ is served first; else $P_j$ is served first.

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., $1, 2, 3, 3, 3, 3, 4, 5...$

Bakery Algorithm

- Notation $\equiv$ lexicographical order (ticket #, process id #)
  - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$

- Shared data
  
  ```
  boolean choosing[n];
  int number[n];
  ```
  initialized to $\text{false}$ and $0$ respectively
Bakery Algorithm

do {
    choosing[i] = true;
    number[i] = max (number[0], number[1], ..., number[n – 1]) + 1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while (((number[j] ≠ 0) && (number[j],i) < (number[i],i)) ;
        }
    }  
    critical section
    number[i] = 0;
    remainder section
} while (1);

Correctness

❑ Mutual exclusion ?

❑ Progress ?

❑ Bounded waiting ?
Synchronization Hardware

- Test and modify the content of a memory word atomically

```java
boolean TestAndSet (boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:
  ```java
  boolean lock = false;
  ```
- Process $P_i$
  ```java
  do {
      while (TestAndSet(lock)) ;
      critical section
      lock = false;
      remainder section
  }
  ```
Synchronization Hardware

- Atomically swap two variables.

```c
void Swap (boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Mutual Exclusion with Swap

- Shared data: `boolean lock = false;`
- Process $P_i$
  
  ```c
  do {
    key = true;
    while (key == true) swap (lock, key);
      critical section
    lock = false;
      remainder section
  }
  ```
Bounded Waiting?

- Previous two algorithms do not satisfy the bounded waiting requirement.
- Can you see why?

---

Bounded Waiting with TestAndSet

Do {
    waiting [me] = true;
    key = true;
    while (waiting[me] && key)
        key = TestAndSet (lock);
    waiting [me] = false;
    … critical section …
    if (waiting [other]) waiting [other] = false;
    else lock = false;
    … remainder section …
}
Semaphores

- Synchronization without busy waiting.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

  \[
  \text{wait}(S):
  \]
  \[
  \begin{align*}
  &\text{wait until } S > 0; \\
  &S--; \\
  \end{align*}
  \]

  \[
  \text{signal}(S):
  \]
  \[
  S++; \\
  \]

Critical Section of $n$ Processes

- Shared data:

  \[
  \text{semaphore mutex}; \quad //\text{initially } mutex = 1
  \]

- Process $P_i$:

  \[
  \begin{align*}
  \text{do} & \{ \\
  &\text{wait}(mutex); \\
  &\text{critical section} \\
  &\text{signal}(mutex); \\
  &\text{remainder section} \\
  \} & \text{while (1)};
  \end{align*}
  \]
Semaphore Implementation

- Define a semaphore as a record
  
  ```
  typedef struct {
    int value;
    struct process *L;
  } semaphore;
  ```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(P)** resumes the execution of a blocked process **P**.

Implementation

**wait(S):**

- `S.value--;`
- `if (S.value < 0) {`
  - `add this process to S.L;`
  - `block;`
- `}`

**signal(S):**

- `S.value++;`
- `if (S.value <= 0) {`
  - `remove a process P from S.L;`
  - `wakeup(P);`
- `}`
Semaphore as a General Synchronization Tool

- Execute \( B \) in \( P_j \) only after \( A \) executed in \( P_i \)
- Use semaphore flag initialized to 0
- Code:
  
  \[
  \begin{align*}
  & P_i & & P_j \\
  & \vdots & & \vdots \\
  & A & \text{wait}(\text{flag}) & \\
  & \text{signal}(\text{flag}) & & B
  \end{align*}
  \]

Deadlock and Starvation

- **Deadlock** – two or more processes waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let \( S \) and \( Q \) be two semaphores initialized to 1
  
  \[
  \begin{align*}
  & P_0 & & P_1 \\
  & \text{wait}(S); & & \text{wait}(Q); \\
  & \text{wait}(Q); & & \text{wait}(S); \\
  & \vdots & & \vdots \\
  & \text{signal}(S); & & \text{signal}(Q); \\
  & \text{signal}(Q) & & \text{signal}(S);
  \end{align*}
  \]
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Two Types of Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain.
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore $S$ as a binary semaphore.

Implementing $S$ as a Binary Semaphore

- Data structures:
  
  ```
  binary-semaphore S1, S2;
  int C:
  ```

- Initialization:
  
  ```
  S1 = 1
  S2 = 0
  C = initial value of semaphore S
  ```
Implementing $S$

- **wait operation**
  
  ```
  wait(S1);
  C--;
  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  signal(S1);
  ```

- **signal operation**
  
  ```
  wait(S1);
  C++;
  if (C <= 0)
    signal(S2);
  else
    signal(S1);
  ```

---

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

- Shared data

```c
semaphore full, empty, mutex;
Initially:
full = 0, empty = n, mutex = 1
```

Producer Process

```c
do {
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
} while (1);
```
Consumer Process

do {
    wait(full)
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while (1);

Readers-Writers Problem

- Shared data

semaphore mutex, wrt;

Initially

mutex = 1, wrt = 1, readcount = 0
Reader Writer Problem

wait(wrt);

... writing

... signal(wrt);

wait(mutex);
readcount++;
if (readcount == 1) wait(rt);
signal(mutex);
...
reading
...
wait(mutex);
readcount--;
if (readcount == 0) signal(wrt);
signal(mutex):

Dining-Philosophers Problem

☐ Shared data: semaphore chopstick[5];
Initially all values are 1
Philosopher $i$

do {
    wait(chopstick[i])
    wait(chopstick[(i+1) \mod 5])
    …
    eat
    …
    signal(chopstick[i]);
    signal(chopstick[(i+1) \mod 5]);
    …
    think
    …
} while (1);

Critical Regions

- High-level synchronization construct
- A shared variable $v$ of type $T$, is declared as:
  
  $v$: shared $T$

- Variable $v$ accessed only inside statement

  region $v$ when $B$ do $S$
  
  where $B$ is a boolean expression.

- While statement $S$ is being executed, no other process can access variable $v$. 
Critical Regions

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression $B$ is evaluated. If $B$ is true, statement $S$ is executed. If it is false, the process is delayed until $B$ becomes true and no other process is in the region associated with $\nu$.

Example – Bounded Buffer

- Shared data:

```c
struct buffer {
    int pool[n];
    int count, in, out;
}
```
Produce

- Producer process inserts `nextp` into the shared buffer

```java
region buffer when (count < n) {
    pool[in] = nextp;
    in := (in + 1) % n;
    count++;
}
```

Consumer

- Consumer process removes an item from the shared buffer and puts it in `nextc`

```java
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out + 1) % n;
    count--;
}
```
Implementation region $x$
when $B$ do $S$

- Associate with the shared variable $x$, the following variables:
  
  semaphore mutex, first-delay, second-delay;
  int first-count, second-count;

- Mutually exclusive access to the critical section is provided by $\text{mutex}$.

- If a process cannot enter the critical section because the Boolean expression $B$ is false, it initially waits on the $\text{first-delay}$ semaphore; moved to the $\text{second-delay}$ semaphore before it is allowed to reevaluate $B$.

- Keep track of the number of processes waiting on $\text{first-delay}$ and $\text{second-delay}$, with $\text{first-count}$ and $\text{second-count}$ respectively.

- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```java
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    } procedure body P2 (...) {
        ...
    } procedure body Pn (...) {
        ...
        {
            initialization code
        }
    }
}
```

To allow a process to wait within the monitor, a `condition` variable must be declared, as

```
condition x, y;
```

- Condition variable can only be used with the operations `wait` and `signal`. 
The operation

```cpp
def x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```cpp
def x.signal();
```

The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.
Monitor With Condition Variables

Dining Philosophers Example

```c
monitor dp
{
enum {thinking, hungry, eating} state[5];
condition self[5];
void pickup(int i) // following slides
void putdown(int i) // following slides
void test(int i) // following slides
void init() {
    for (int i = 0; i < 5; i++)
        state[i] = thinking;
}
}
```
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating) self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
}

void test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}
Monitor Implementation Using Semaphores

- Variables
  
  ```
  semaphore mutex;  // (initially = 1)
  semaphore next;   // (initially = 0)
  int next-count = 0;
  ```

- Each external procedure $F$ will be replaced by
  ```
  wait(mutex);
  ...
  body of $F$;
  ...
  if (next-count > 0)
    signal(next)
  else
    signal(mutex);
  ```

- Mutual exclusion within a monitor is ensured.
Monitor Implementation

- For each condition variable $x$, we have:
  
  ```
  semaphore x-sem; // (initially = 0)
  int x-count = 0;
  ```

- The operation $x$.wait can be implemented as:
  
  ```
  x-count++;
  if (next-count > 0)
    signal(next);
  else
    signal(mutex);
  wait(x-sem);
  x-count--; 
  ```

Monitor Implementation

- The operation $x$.signal can be implemented as:

  ```
  if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
  }
  ```
Monitor Implementation

- Conditional-wait construct: `x.wait(c);`
  - `c` – integer expression evaluated when the `wait` operation is executed.
  - value of `c` (a priority number) stored with the name of the process that is suspended.
  - when `x.signal` is executed, process with smallest associated priority number is resumed next.