Part 4: Process Synchronization

- **Race Condition & Critical-Section**
- User-level Solutions
- OS-level Solutions
  - Synchronization Hardware
  - Semaphores
- Classical Problems of Synchronization

* Important but difficult ☺

Background

- Access to shared data from concurrent processes may result in data inconsistency
  - Inconsistent Data: Data value depends on the order of instruction executions from concurrent processes
  - That is, data value depends on when context switches occur between the concurrent processes
- Maintaining data consistency requires mechanisms to ensure the orderly execution of concurrent processes
  - Causal ordering (sequencing) of reads and writes to the shared data from concurrent processes

Example: Producer-Consumer (with bounded buffer)

- #define BUFFER_SIZE 10
- typedef struct {. . .} item;
- int in=0; //the next-to-fill empty slot (producer variable)
- int out=0; //the next-to-process item (consumer variable)

- Shared Data
  - item buffer[BUFFER_SIZE];
  - int counter = 0; // Number of items in buffer

Example: Producer-Consumer (with bounded buffer)

- Producer process

```
item nextProduced;
while (1) {

      while (counter == BUFFER_SIZE); // wait for space

      buffer[in] = nextProduced; // produce item

      in = (in + 1) % BUFFER_SIZE; // prepare for next item

      counter++;

}
```

Example: Producer-Consumer (with bounded buffer)

- Consumer process

```
item nextConsumed;
while (1) {
    while (counter == 0); // wait for buffer to receive an item
    nextConsumed = buffer[out]; // Consume the item
    out = (out + 1) % BUFFER_SIZE; // Prepare for next item
    counter--;
}
```

Example: Producer-Consumer (with bounded buffer)

- Lets take a close look at these instructions
  - *Producer process:* **counter++;**
  - *Consumer process:* **counter--;**

- They may create inconsistent value for variable counter due to *race condition*
  - The actual value depends on when the producer or consumer process context switches

Race Condition

In Producer counter++ implemented as:

register1 = counter
register1 = register1 + 1
counter = register1

In Consumer counter-- implemented as:

register2 = counter
register2 = register2 - 1
counter = register2

With "counter = 5" initially, suppose we execute producer and consumer once using the following interleaving

1. Producer executes register1 = counter {register1 = 5}
2. Producer executes register1 = register1 + 1 {register1 = 6}
3. **Context switch occurs from Producer to Consumer**
4. Consumer executes register2 = counter {register2 = 5}
5. Consumer executes register2 = register2 – 1 {register2 = 4}
6. **Context switch occurs from Consumer to Producer**
7. Producer executes counter = register1 {counter = 6 }
8. **Context switch occurs from Producer to Consumer**
9. Consumer executes counter = register2 {counter = 4}

Counter value is 4 instead of 5!

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
  - Process may be changing shared variables, updating a shared table, writing a shared file, etc.
  - At least one process modifies (writes to) the shared data
- Problem: Design protocol to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

Generic Solution Structure for the Critical-Section Problem

1. Entry section: Ask permission to enter critical section
2. Exit section: Exit critical section (notify other processes)

Generic Process Structure

What should the logic inside here be?

```
while (1) {
    entry section
        critical section
    exit section
        remainder section
}
```

Non critical section executions

Desired Properties of Solutions to the Critical-Section Problem

- Assumptions of critical and remainder sections
  - Each process is guaranteed to make progress over time in these sections
  - No assumption about relative execution speed of different processes in these sections (may execute at different speeds)

- Desired Properties of entry and exit sections

  1. Mutual Exclusion: If a process is executing in its critical section, then no other process can be executing in its critical section at the same time

Desired Properties of Solutions to the Critical-Section Problem (Cont.)

2. Progress: If no process is executing in its critical section and there exist processes that wish to enter their critical section, then the selection of the next process to enter the critical section cannot be postponed indefinitely

3. Bounded Waiting: After a process has requested to enter its critical section and before that request is granted, other processes are allowed to enter their critical section only a bounded number of times

These two properties together ensure that a process is not stuck in the entry section forever

Part 4: Process Synchronization

- Race Condition & Critical-Section
- **User-level Solutions**
- OS-level Solutions
  - Synchronization Hardware
  - Semaphores
- Classical Problems of Synchronization

Initial Attempts to Solve the Problem

- Generic structure of process

```
while (1){
    entry section
        critical sections
    exit section
    remainder section
}
```

- We start with only 2 processes P_0 and P_1
  - Although P_0 and P_1 have the same generic structure, they may have different implementations for the four sections
- We look at user-level software approaches w/o OS support

Algorithm 1

- Variable shared between processes:
  int *turn*; // initially turn=0
  - *turn* = *i* ⇒ P_i can enter critical section
- Process P_i : while (1){

Entry Section → while (*turn* != *i*);

critical section

Exit Section → *turn* = *k*;

remainder section

}

Process P_0

```
while (1){
    while (turn != 0);
        critical section
    turn = 1;
        remainder section
}
```

Process P_1

```
while (1){
    while (turn != 1) ;
        critical section
    turn = 0;
        remainder section
}
```

Algorithm 1: Property Analysis

1. Mutual Exclusion? ☺

- Suppose **turn = 0** and P_0 enters critical section
- **turn** is only updated in exit section, when P_0 exits critical section
- Process P_1 therefore cannot enter its critical section

2. Progress? ☹

- **turn = 0** and P_0 is in a long remainder section (think while(1) loop or blocked I/O), and P_1 wants to enter critical section

3. Bounded Waiting? ☺

- Assuming **turn** is updated in a fair (e.g., round-robin) manner among the two concurrent processes

Algorithm 2

- Variables shared: **boolean *flag*[2];**
  - Initially *flag* [0] = *flag* [1] = *false*
  - *flag* [*i*] = *true* $\Rightarrow$ P_i ready to enter its critical section

- Process P_i : while (1) {

Entry Section → *flag* [*i*] = *true*;
while (*flag* [*k*]);

critical section

Exit Section → *flag* [*i*] = *false*;

remainder section

}

Algorithm 1 had an issue that it did not know if a process wants to enter its critical section

- **Issue solved in Algorithm 2 by using one flag for each process to express interest**

Algorithm 2: Property Analysis

1. Mutual Exclusion? ☺

– Suppose P_0 enters critical section, then **flag [1] = false** and **flag [0] = true**
– **flag [0]** is only updated to false in the exit section of P_0
– Process P_1 therefore cannot enter its critical section

2. Progress? ☹ Consider the following interleaving:

1. P_0 executes **flag [0] = true;**
2. **Context switch to P_1**
3. P_1 executes **flag [1] = true;**

Both P_0 and P_1 are stuck in an infinite while loop now!

3. Bounded Waiting? ☺

– Once P_0 sets **flag [0] = true**, process P_1 cannot enter its critical section

Algorithm 3

- Variables are a combination of Algorithms 1 and 2
  - *flag* [*i*] = *true* $\Rightarrow$ P_i ready to enter its critical section
  - *turn* = *i* $\Rightarrow$ P_i can enter critical section

- Proc. P_i: while(1){

Entry Section →

flag [*i*] = *true*;
turn = *k*;
while (*flag* [*k*] and *turn* = *k*);

critical section

Exit Section →

flag [*i*] = *false*;

remainder section

}

Algorithm 2 could not choose between two processes that wanted to enter their critical section at the same time

- **Issue solved in Algorithm 3 by using turn variable to denote whose turn it is**

Algorithm 3: Property Analysis

1. Mutual Exclusion? ☺
   - If P_0 enters critical section, then **flag [0] = true**
   - If P_1 now wants to enter critical section, then it will first set **turn = 0**
   - P_1 then cannot enter because **turn = 0** and **flag [0] = true**

2. Progress? ☺ Suppose P_0 wants to enter critical section
   - If P_1 is in remainder section then **flag [1] = false** and P_0 can enter
   - If P_1 is in entry section, then access is granted either to P_0 or P_1 depending on the latest value of **turn**

3. Bounded Waiting? ☺ Assuming **turn** is updated fairly

How to extend Algorithm 3 for say three processes?

Part 4: Process Synchronization

- Race Condition & Critical-Section
- User-level Solutions
- **OS-level Solutions**
  - **Synchronization Hardware**
  - **Semaphores**
- Classical Problems of Synchronization

OS Support for Synchronization

- Previous software approaches are difficult to implement for more than two processes
  - Solution: Operating system support
- OS has the following three kinds of support for the critical section problem (low level to high level)
  - Synchronization hardware
  - Semaphore
  - Monitor

Synchronization Hardware

- Modern processors provide special atomic hardware instructions
  - Atomic = non-interruptible (no context switches)

- TestAndSet: Test and modify the content of a main memory word atomically

boolean *TestAndSet* (boolean *target) {

Get current value → boolean *rv* = **target*;

Store true → **target* = *true*;

Return old value → return *rv*;

}

Interrupt is disabled

No context switches allowed during this execution

Interrupt is enabled

Critical-Section with Test-and-Set

- Variable shared: **boolean *lock;***
  - Initially lock = false
  - If TestAndSet on lock returns false, then process can enter critical section (because it set lock to true)

- Process P_i : while (1) {

Entry Section → while(TestAndSet(&*lock*)); — Acquire lock

critical section

Exit Section → *lock* = *false*; — Release lock

remainder section

}

TestAndSet: Property Analysis

1. Mutual Exclusion? ☺
   - A process enters critical section → **lock = false** immediately before it executes TestAndSet(&lock) and **lock = true** thereafter
   - **lock** is reset to false **only** in the exit section of the process
   - No process can thereafter enter critical section since **lock = true**

2. Progress? ☺
   - If **lock = false**, then the first process that executes TestAndSet(&lock) will immediately enter critical section

3. Bounded Waiting? ☹ See next slide
   - More complex solution with TestAndSet that satisfies bounded waiting is in the textbook – not examinable

Bounded Waiting fails with TestAndSet

- Process P_0

1. Initialization code
2. while (1) {
3. while(TestAndSet(&*lock*));
4. critical section
5. *lock = false*;
6. }

- Process P_1

1. while (1) {
2. while(TestAndSet(&*lock*));
3. critical section
4. *lock = false*;
5. }

Context switch points

- Suppose for P_0 lines 1-3 take 10ms and 2-6 also take 10ms
- Suppose we use round-robin scheduling with quantum 10ms
- Whenever P_0 context switches out, it holds the lock (is in critical section)

P_0 (1,2,3)	P_1 (1,2)	P_0 (4,5,6,2,3)	P_1 (2)	P_0 (4,5,6,2,3)	. . .

Semaphore

- Semaphore ***S***: Integer shared variable
- **Only** accessible by two **atomic** system calls: *Wait* and *Signal*
  - *Wait (S):* **if S>0, S--; else, wait();**
  - *Signal (S):* **S++;**

Enter waiting state or busy loop waiting

- Two types of semaphores
  - *Counting* semaphore: Integer value can range over an unrestricted domain
  - *Binary* semaphore: Integer value can never be greater than 1

Semaphore Solution for the Critical Section Problem

- Variable shared: ***semaphore mutex;*** // initially *mutex*=1
  - If mutex=1 and process executes Wait(mutex), then it can enter critical section
- Process P_i : while(1) {

Entry Section → *Wait(mutex);* Acquire Semaphore

critical section

Exit Section → *Signal(mutex);* Release Semaphore

remainder section

}

Classical Semaphore Implementation (Busy Waiting)

- *Wait* (*S*):

  while (*S* <= 0)
  ; // busy wait
  S--;

Assembly for Wait(S):

```
loop1: ldr ax, [S]
       cmp ax, 0
       jle loop1
       sub ax, 1
       str [S], ax
```

- *Signal* (*S*):

  ***S*++;**

Assembly for Signal(S):

```
ldr ax, [S]
add ax, 1
str [S], ax
```

- Pros: No context switch overhead (busy waiting)
- Cons: Inefficient if critical sections are long
  - Busy waiting wastes CPU cycles

Busy Waiting Semaphore (Property Analysis)

1. To avoid race condition, S++ and S-- must be atomic
2. Wait(S) must be atomic to ensure mutual exclusion
   – Consider the following with S>0 initially
     1. Process P_0 executes **while (S <= 0);** and exits busy wait
     2. **Context switch from P_0 to P_1**
     3. Process P_1 executes **while (S <= 0);** and **S--;**
     4. **Context switch from P_1 to P_0**
     5. Process P_0 executes **S--;**
     6. *Mutual Exclusion?* ☹

Assembly for Wait(S):

```
loop1: ldr ax, [S]   ┐
       cmp ax, 0     ├ while (S <= 0);
       jle loop1     ┘
       sub ax, 1     ┐
       str [S], ax   ┘ S--;
```

Assembly for Signal(S):

```
       ldr ax, [S]   ┐
       add ax, 1     ├ S++;
       str [S], ax   ┘
```

Current Semaphore Implementation (Blocking)

- Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

L is a queue that stores the processes waiting on this semaphore (in the form of PCB list)

- Need two simple operations
  - *block*( ): blocks the current process
  - *wakeup*( ): resumes the execution of a blocked process in list L

Blocking Semaphore Implementation block() and wakeup()

block( ):

1. Dequeue current process from ready queue
2. Enqueue the process to list L
3. Change state of the process to **waiting**

Will lead to a context switch

wakeup( ):

1. Dequeue a process from list L
2. Enqueue the process to the ready queue
3. Change state of the process to **ready**

Blocking Semaphore Implementation Wait(S) and Signal(S)

Wait(*S*):

```
S.value--;
if (S.value < 0)
{
   block( );
}
```

Signal(*S*):

```
S.value++;
if (S.value <= 0)
{
   wakeup(P);
}
```

If S.value = -2, how many processes are waiting in list L? **2**

Blocking Semaphore Implementation Wait and Signal must still be atomic?

Wait(*S*): **S.value--;**

if (*S.value* < 0)

{ *block*( ); }

Signal(*S*): **S.value++;**

if (*S.value* <= 0)

{ *wakeup*(*P*); }

1. Initially, **S.value = 1**
2. Process P_0 executes **S.value--;**
3. **Context switch from P_0 to P_1**
4. Process P_1 executes **S.value--;** and blocks since **S.value = -1**
5. **Context switch from P_1 to P_0**
6. Process P_0 also blocks
7. *Progress for P_0 and P_1?* ☹

1. Initially, **S.value = -1** (P_2 blocked for **S**)
2. Process P_0 executes **S.value++;**
3. **Context switch from P_0 to P_1**
4. Process P_1 executes **S.value++;** and exits. No ***wakeup(P);*** since **S.value = 1**
5. **Context switch from P_1 to P_0**
6. Process P_0 also exits without ***wakeup(P);***
7. *Progress for P_2?* ☹

Also lead to mutual exclusion violation due to -- and ++

All updates and checks for S must be atomic

Blocking Semaphore Implementation S.value-- at the end of Wait(S)?

Wait(*S*):

```
if (S.value <= 0) {
   block( );
}
S.value--;
```

Signal(*S*):

```
S.value++;

if (S.value <= 1) {
   wakeup(P);
}
```

Problems:

1. This would require a context switch between ***block*( );** and **S.value--;** if the process blocks
2. If this context switch is allowed, mutual exclusion will be violated
   - Say **S.value = 0** and P_0 executes ***block*( );**
   - P_1 uses Signal(S) to increment S.value to **1** and executes ***wakeup*(P_0);**
   - P_2 executes Wait(S), locks S, enters critical section
   - P_0 executes **S.value--;** and enters critical section

Part 4: Process Synchronization

- The Critical-Section Problem
- User-level Solutions
- OS-level Solutions
  - Synchronization Hardware
  - Semaphores
- **Classical Problems of Synchronization**

Best Practices for Semaphores

- Step 1: Understand the application scenario and identify the following:
  - Shared variables/data
  - Shared resources
- Step 2: Protect the shared variables
  - Identify the critical section
  - Use binary semaphore, and add entry section (Wait) and exit section (Signal)
- Step 3: Protect the shared resources
  - Identity each kind of resource; one semaphore for one kind
  - Initial value: number of resource instances
  - When the resource is requested/consumed: Wait
  - When the resource is released: Signal

Semaphore as General Synchronization Tool

- We wish to execute code segment *B* in process P_k only after code segment *A* in process P_i
  - Use a semaphore *flag* initialized to **0**
  - Code: P_i P_k

P_i	P_k
⋮	⋮
A	***Wait*(*flag*)**
***Signal*(*flag*)**	*B*

Deadlocks (Incorrect Semaphore Use)

- Deadlock: Two or more processes are 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

P_0	P_1
Wait(S);	***Wait(Q);***
Wait(Q);	***Wait(S);***
⋮	⋮
Signal(S);	***Signal(Q);***
Signal(Q);	***Signal(S);***

Consider the following interleaving:

1. P_0 executes **Wait(S);**
2. **Context switch from P_0 to P_1**
3. P_1 executes **Wait(Q);** and **Wait(S);**
4. **Context switch from P_1 to P_0**
5. P_0 executes **Wait(Q);**

Starvation (Incorrect Semaphore Use)

- Starvation: Indefinite postponement (no progress)
  - A process may never have a chance to be removed from the semaphore queue (list L) due to queue discipline
  - Examples of queue disciplines that can cause starvation
    - Priority-based: Low priority process may starve if higher priority processes keep joining the queue
    - Last-In-First-Out policy

Classical Problems of Synchronization

- Bounded-Buffer (Producer-Consumer) Problem
- Dining-Philosophers Problem
- Readers and Writers Problem

Producer-Consumer with Bounded-Buffer

- Consider multiple producers/consumers
  - We used **counter, in** and **out** variables for a producer-consumer pair
  - **in**/**out** are also shared variables now (**in** - producers, **out** - consumers)

- Shared data and resources (Step 1)
  - **item buffer[n];**
  - buffer is a shared data structure
  - Slots in buffer are shared resources

Binary semaphore for mutual exclusion to update buffer

- **in** replaced with **empty**
- **out** replaced with **full**
- **counter** not used (why?)
- **mutex** introduced to control access to buffer

- Semaphores to be used

  semaphore *full, empty, mutex*;

  ***full* =0; *empty* = *n*; *mutex* =1;**

Counting semaphore for consumers to identify full buffer to consume

Counting semaphore for producers to identify empty buffer to fill

Bounded-Buffer Problem (Cont.)

Producer process (Step 2)

Without synchronization:

```
item nextProduced;
while (1) {
      …
   produce nextProduced;
      …
   add nextProduced to buffer;
      …
}
```

Protect Shared Data:

```
item nextProduced;
while (1) {
      …
   produce nextProduced;
      …
   Wait(mutex);
      …
   add nextProduced to buffer;
      …
   Signal(mutex);
}
```

Bounded-Buffer Problem (Cont.)

Producer process (Step 3)

Protect Shared Data:

```
item nextProduced;
while (1) {
    …
  produce nextProduced;
    …
  wait(mutex);
    …
  add nextProduced to buffer;
    …
  signal(mutex);

}
```

Manage Shared Resources:

```
item nextProduced;
while (1) {
    …
  produce nextProduced;
    …
  Wait(empty);
  Wait(mutex);
    …
  add nextProduced to buffer;
    …
  Signal(mutex);
  Signal(full);
}
```

Consume one empty slot

Signal consumers

Bounded-Buffer Problem (Cont.)

Consumer process

Protect Shared Data and Manage Shared Resources:

```
item nextConsumed;
while (1) {
   Wait(full);
   Wait(mutex);
      …
   nextConsumed = item from buffer;
      …
   Signal(mutex);
   Signal(empty);
      …
   consume the item nextConsumed;
}
```

Consume one full slot

Signal producers

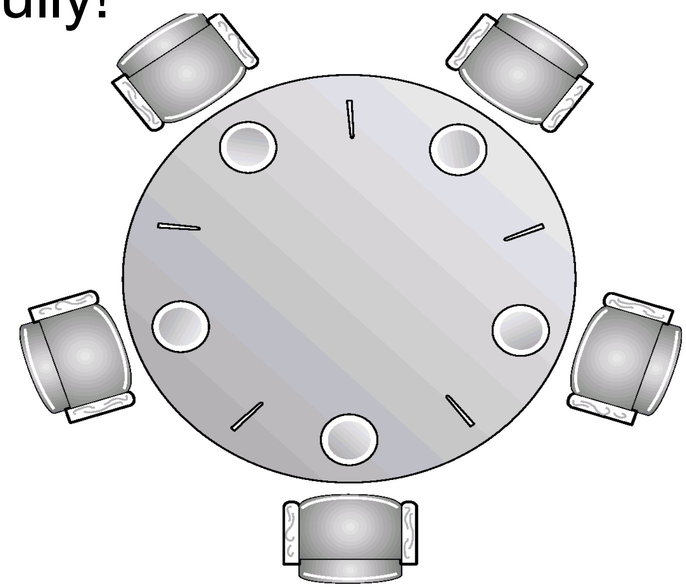
What if we swap as below?

Wait(mutex);
Wait(full);

DEADLOCK if buffer is empty!

Dining-Philosophers Problem

- Five philosophers seated in a round table alternating between eating and thinking
- Five plates (one for each philosopher) and only five chopsticks (one between each plate)
- Problem: Devise a strategy to enable the philosophers to eat peacefully!



Dining-Philosophers Problem (Cont.)

- Initial Solution
  - Each philosopher is a process
  - Each chopstick is a shared resource protected by a binary semaphore (**chopstick [5];**)
  - Initially, for all i, **chopstick [i] = 1;**

- Code (Philosopher i)

```
while (1) {
    Wait(chopstick [ i ]);
    Wait(chopstick [ (i+1)%5 ]);
    eat
    Signal(chopstick [ i ]);
    Signal(chopstick [ (i+1)%5 ]);
    think
}
```

Dining-Philosophers Problem (Cont.)

- Does the above solution have a problem?
- Consider the following interleaving (P_i=Philosopher i):
    1. P_0 executes **Wait(0);** and then **context switch** to P_1
    2. P_1 executes **Wait(1);** and then **context switch** to P_2
    3. P_2 executes **Wait(2);** and then **context switch** to P_3
    4. P_3 executes **Wait(3);** and then **context switch** to P_4
    5. P_4 executes **Wait(4);**

Each philosopher has 1 chopstick and needs another to eat

Deadlock!

Dining-Philosophers Problem (Cont.)

There are several possible remedies to avoid the above deadlock problem

- Allow at most four philosophers to be hungry simultaneously
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available
- Use an asymmetric solution
  - An odd philosopher picks up first his left chopstick and then his right chopstick
  - An even philosopher picks up first his right chopstick and then his left chopstick

Readers-Writers Problem

- Problem: Scenario in a database or file
  - A writer requires exclusive access
  - Multiple readers can concurrently access
    - The first reader: Before reading, it must **block the writers**
    - The last reader: After reading, it can **allow writers**
    - Readers given preference (writer preference also possible)

If a reader is in database, more readers can access it

- Shared data
  - **int *readcount* =0;** // Tracks #readers in the database
  - Database itself

- Semaphores to be used
  - **mutex=1;** Binary semaphore protects access to readcount
  - **wrt=1;** Binary semaphore for mutual exclusion to database

Readers-Writers Problem

Writer process

***Wait*(*wrt*);**

…

writing is performed

…

***Signal*(*wrt*);**

Readers-Writers Problem (Cont.)

Reader process

When a reader comes

***Wait*(*mutex*);**
readcount ++;
if (*readcount* == 1) ***Wait*(*wrt*);**
***Signal*(*mutex*);**

The first reader?
Request access to database

...
reading is performed
...

When a reader leaves

***Wait*(*mutex*);**
readcount --;
if (*readcount* == 0) ***Signal*(*wrt*);**
***Signal*(*mutex*);**

The last reader?
Release database

Monitor

- High-level synchronization construct that allows the sharing of variables among concurrent processes
- Monitor is a collection of procedures and data
- Processes may call procedures within monitor to access and update the data
- Only one process can be active in the monitor at any one time (i.e., only one process can be executing a monitor procedure at any one time)
- Data within monitor can only be accessed by procedures within it

<u>This slide is not examinable</u>