Last lectures: scheduling

- Preemptive and non-preemptive
- Non-preemptive works because of typical IO/CPU burst sizes
- Constraints in scheduling are reflecting application domain, e.g.,
  + Throughput
  + Average waiting time
  + Predictability, worst-case execution time
  + Load-sharing in multiprocessor systems
- Based on what we learned you should be able to design a different scheduling algorithm that fits your constraints, application domain.

Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Example: producer-consumer type of problems, bounded-buffer scheme.

Bounded-Buffer

- Shared data

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

Bounded-Buffer

- Producer process

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

Bounded-Buffer

- Consumer process

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

- Note that variable “counter” is accessed from both consumer and producer
- The statements
  
  ```
  counter++;
  counter--;
  ```

  must therefore be performed atomically.

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

Bounded Buffer

- The statement “count++” may be implemented in machine language as:
  ```
  register1 = counter           [lw $1, (counter)] // MIPS like ISA
  register1 = register1 + 1     [addi, $1,$1,1]
  counter = register1          [sw $1, (counter)]
  ```

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

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

There is no guarantee that the two C statements are executed atomically

Interleaving depends upon how the producer and consumer processes are scheduled. Not something you control…

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

- The problem called critical-section
- 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.
**Solution to 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**

- Let us assume first only 2 processes, \( P_2 \) and \( P_1 \)
- General structure of process \( P_i \) (other process \( P_j \))

```plaintext
algorithm 1

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

Process \( P_i \)

\[
\text{do } \begin{cases} 
  \text{while (turn} \neq i) ; \\
  \text{critical section} \\
  \text{turn} = j \\
  \text{reminder section} \\
\end{cases} \text{ while (1)}; 
\]

- Satisfies mutual exclusion, but not progress
- Process 2 cannot enter until Process 1 enters
```

---

**Algorithm 2**

- Shared variables
  - \( \text{boolean flag}[2]\) - retain state about each thread/process...
  - \( \text{flag}[i] = \text{true} \Rightarrow P_i \) ready to enter its critical section

Process \( P_j \)

\[
\text{do } \begin{cases} 
  \text{flag}[i] : = \text{true}; \\
  \text{while (flag}[i] \text{ and turn} = i) ; \\
  \text{critical section} \\
  \text{flag}[i] = \text{false}; \\
  \text{remainder section} \\
\end{cases} \text{ while (1)}; 
\]

- Satisfies mutual exclusion, but not progress requirement.
- Both flag[i] could be set concurrently, both will loop forever

---

**Algorithm 3**

- Combined shared variables of algorithms 1 and 2.
- Process \( P_i \)

\[
\text{do } \begin{cases} 
  \text{flag}[i] : = \text{true}; \\
  \text{turn} = i \\
  \text{while (flag}[i] \text{ and turn} = i) ; \\
  \text{critical section} \\
  \text{flag}[i] = \text{false}; \\
  \text{remainder section} \\
\end{cases} \text{ while (1)}; 
\]

- Meets all three requirements; solves the critical section problem for two processes.
- Let us analyze this

---

**Algorithm 4**

**Bakery Algorithm**

Critical section algorithm for \( n \) processes developed by Lamport 1974
- Before entering its critical section, process receives a number (ticket). 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., \( 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 5, \ldots \)
Bakery Algorithm

- Notation
  - lexigraphical order (ticket #, process id #)
  - \((a,b) < (c,d)\) if \(a < c\) or \(a = c\) and \(b < d\)
  - \(\max(a_0, ..., a_n)\) is a number, \(k\), such that \(k \geq a_i\) for some \(i\)
- Shared data
  - boolean choosing[n];
  - int number[n];

Data structures are initialized to false and 0 respectively.

Bakery Algorithm

```java
do {
  choosing[i] = true;  // signal choosing a number
  number[i] = max(number[0], number[1], ..., number[n-1])+1;
  choosing[i] = false;  // done with choosing a number
  for (j = 0; j < n; j++) {
    while (choosing[j]) ; // wait until Pj has chosen a nr
    while ((number[j] != 0) && (number[i] < number[j])) ;
  }
  critical section
  number[i] = 0;  // signal out of critical section
  remainder section
} while (1);
```

Someone is in, as it has a smaller number?

Synchronization Hardware

- Disabling interrupts around statements to provide atomicity—difficult to implement in multiprocessor environments
- Adding ISA support is another option
- E.g., Test and modify the content of a word atomically
  ```java
  boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
  }
  ```

Mutual Exclusion with Test-and-Set

- Can be used to implement a simple lock
- Shared data:
  ```java
  boolean lock = false;
  ```
- Process \(P_i\)
  ```java
  do {
    key = true;  // key is local
    while (key == true) {
      Swap(lock,key);
    }
    critical section
    lock = false;
    remainder section
  }
  ```

Wait here-test until Lock is TRUE, if it is not, set it and continue!

Synchronization Hardware

- Another ISA support example: the swap instruction
- Atomically swap two variables
- Implementation shown below in pseudo-code (for illustration only) is actually what happens when the instruction is executed.
  ```java
  void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
  }
  ```

Mutual Exclusion with Swap

- Shared data (initialized to false):
  ```java
  boolean lock;  // lock is global/shared
  ```
- Process \(P_i\)
  ```java
  do {
    key = true;  // key is local
    while (key == true) {
      Swap(lock,key);
    }
    critical section
    lock = false;
    remainder section
  }
  ```

Wait here until Lock is TRUE (key will be TRUE), if it is not, set it by swapping it!
Semaphores

- The solutions in Algorithm 1-3 plus Bakery's are not easy to work with!
- Semaphores are Synchronization tool that does not require busy waiting.
- Semaphore S – integer variable
- can only be accessed via two atomic operations
- semantics (will see later how it can be implemented to avoid busy-waiting)

```plaintext
wait (S);  // also called spinlock due to busy-
// waiting
  while (S <= 0) do no-op;
  S--;  
signal (S);  
S++;  
```

Semaphores as a General Synchronization Tool

- Execute B in Pj only after A executed in Pi
- Use semaphore flag initialized to 0
- Code:

```plaintext
P_i  P_j  M  M  A  
  wait(flag)  wait(flag)
  signal(flag)  
```

Usage of semaphores: Critical Section of n Processes

- Let us solve the n Process problem with semaphores
- Shared data:
  - semaphore mutex;  // initially mutex = 1
- Process Pi:
  - do {
    - wait(mutex);  
    - critical section  
    - signal(mutex);  
    - remainder section  
  } while (1);

Semaphore Implementation

- Avoid busy-waiting, a disadvantage as it would use the processor unnecessarily
- Define a semaphore as a record
  ```plaintext
typedef struct {
  int value;  
  struct process *L;  // queue for processes  
} semaphore;
```
- Assume two simple operations:
  - block suspends the process that invokes it.
  - wakeup(P) resumes the execution of a blocked process P.

Implementation

- Semaphore operations now defined as

```plaintext
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 Pj only after A executed in Pi
- Use semaphore flag initialized to 0
- Code:

```plaintext
P_i  P_j  M  M  A  
  wait(flag)  wait(flag)
  signal(flag)  
```

Deadlock and Starvation

- 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

```plaintext
P_i  P_j  M  M  A  
  wait(S);  
  wait(Q);  
  M  
  signal(Q);  
  signal(Q);  
```
- 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.
  - It is used when we want to control access to a given resource consisting of only a finite number
  - The semaphore is initialized to the number of resources available
  - When the count goes down to 0 all resources are used
- Binary semaphore – integer value can range only between 0 and 1, can be simpler to implement.
  - If value is set to 1 it is like a lock (blocks for the second accessor)
  - If value is set to 0 can be used to guarantee order
- We can implement a counting semaphore \( S \) with binary semaphores.

Implementing \( S \) with Binary Semaphores

- Data structures:
  - binary-semaphore \( S_1, S_2 \);
  - int \( C \);
- Initialization:
  - \( S_1 = 1 \)
  - \( S_2 = 0 \)
  - \( C = \) initial value of semaphore "\( S \)"
  (this is where you should initialize to the finite number of resources you want to control) */

Implementing \( S \)

- wait operation
  - \( \text{wait}(S_1); \) // note that only one thread
  - // can enter \( \text{wait}(S) \) and
  - // \( \text{signal}(S) \) due to \( S_1 \),
  - // \( C \) access is mutexed
  - \( C--; \)
  - if (\( C < 0 \)) {
    - \( \text{signal}(S_1); \) // enable \( \text{signal}(S) \)
    - \( \text{wait}(S_2); \) // block here
  - }
  - \( \text{signal}(S_1); \)

- signal operation
  - \( \text{wait}(S_1); \)
  - \( C++; \)
  - if (\( C == 0 \))
  - \( \text{signal}(S_2); \) // wake-up one blocked in \( S_2; \)
  - else
  - \( \text{signal}(S_1); \) // enable access to \( C \)

Classical Problems of Synchronization

- Bounded-Buffer Problem
  - Accessing a finite buffer from producer and consumer threads

- Readers and Writers Problem
  - Accessing a database for example

- Dining-Philosophers Problem
  - A classical famous problem of five philosophers spending their lives thinking and eating

Bounded-Buffer Problem

- Shared data
  - semaphore full, empty, mutex;
  - Initially:
    - full = 0, empty = \( n \), mutex = 1
- Use a counting and a binary semaphore
  - The mutex assures one thread at a time in CS
  - The counter makes sure buffer length is taken into account

Bounded-Buffer Problem Producer Process

\[
\begin{align*}
\text{do} & \{ \\
& \quad \ldots \text{produce an item in } \text{nextp} \\
& \quad \text{wait(empty);} \\
& \quad \text{wait(mutex);} \\
& \quad \text{\ldots add } \text{nextp} \text{ to buffer} \\
& \quad \text{signal(mutex);} \\
& \quad \text{signal(full);} \\
& \quad \text{while (1);} \\
\end{align*}
\]
### Bounded-Buffer Problem Consumer Process

```java
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

- Several versions exist; readers can have different priorities
- Shared data
  - semaphore mutex, wrt;
  - Initially
    - mutex = 1, wrt = 1, readcount = 0
- Writers and readers
- Writers need exclusive access to data (e.g. a database), readers only need mutex to modify readcount.
- Readers need to synchronize with writers (do not allow writers while readers are accessing shared data)

#### Reader Process

```java
wait(mutex); // to ensure mutex access to readcount
readcount++;
if (readcount == 1)
    wait(wrt); // don't read if writer in CS
signal(mutex);
...
reading is performed
wait(mutex); // I am done reading, decr readcount
readcount--;
if (readcount == 0)
    signal(wrt); // no reader in CS so signal writer
signal(mutex);
```

#### Writer Process

```java
wait(mutex); // to ensure mutex access to readcount
readcount++;
if (readcount == 1)
    wait(wrt); // don't read if writer in CS
signal(mutex);
...
writing is performed
wait(mutex); // I am done writing, incr readcount
readcount--;
if (readcount == 0)
    signal(mutex);
```

### Dining-Philosophers Problem

- Shared data
  - semaphore chopstick[5]; // simple solution
  - Initially all values are 1
- Access to food is the CS, each Philosopher needs 2 chopsticks

#### Problem: deadlock is possible!
- Solutions: allow only 4 philosophers to eat; Allow a philosopher to pick up a chopstick only if both are available (e.g., in a critical section); Use asymmetric solutions: some pick the left chopstick some the right one.
Critical Regions

- High-level synchronization construct
- A shared variable \( v \) of type \( T \) is declared as:
  \( v: \text{shared} \ T \)
- 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 \( v \).

Example – Bounded Buffer

- Shared data:

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

Bounded Buffer Producer Process

- Producer process inserts \( \text{nextp} \) into the shared buffer

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

Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in \( \text{nextc} \)

  ```
  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 `mutex`.
- If a process cannot enter the critical section because the Boolean expression \( B \) is false, it initially waits on the `first-delay` semaphore, moved to the `second-delay` semaphore before it is allowed to reevaluate \( B \).
Implementation

- Keep track of the number of processes waiting on first-delay and second-delay, with first-count and second-count respectively.
- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
- For an arbitrary queuing discipline, a more complicated implementation is required.

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
}
```

![Schematic View of a Monitor](image)

Monitors

- 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 `x.wait();` means that the process invoking this operation is suspended until another process invokes `x.signal();`
  - The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect. (Note this is a key difference compared to the semaphore signal operation)

Monitor With Condition Variables

![Monitor With Condition Variables](image)

Dining Philosophers Example

```java
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
                for (int i = 0; i < 5; i++)
                    state[i] = thinking;
            }
        }
    }
}
```
Dining Philosophers

```c
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    test[i];
    test((i+1) % 5);
    test((i+4) % 5);
}
```

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

- The operation x.signal can be implemented as:
  ```c
  if (x-count > 0) {
      next-count++;  
      signal(x-sem);  
      wait(next);  
      next-count--;  
  }
  ```

Monitor Implementation

- Conditional-wait construct: x.wait(c);
  ```c
  if (c) {  
      x-sem.wait();  
      when x.signal is executed, process with smallest associated priority number is resumed next;  
  }
  ```
- Check two conditions to establish correctness of system:
  ```c
  if (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.
  ```
  Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.
Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses adaptive mutexes for efficiency when protecting data from short code segments. Goes from spinlock implementation to condition variable plus semaphore based depending on situation and number of processors.
- Uses condition variables and readers-writers locks when longer sections of code need access to data. This for data that is accessed frequently but mostly readers only. These locks are more efficient than semaphores for this purpose.
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Windows 2000 Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses spinlocks on multiprocessor systems similar to Solaris. Assumption is that the thread holding the lock will be done soon...
- Also provides dispatcher objects which may act as with mutexes and semaphores; for threads belonging to the same process.
- Dispatcher objects may also provide events. An event acts much like a condition variable; they may notify a thread when a desired condition occurs.
- Lesson: we need to use different synchronization constructs in different scenarios to provide good performance!