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

\[
\begin{align*}
\text{counter} & \text{++}; \\
\text{counter} & \text{--};
\end{align*}
\]

must therefore be performed \textit{atomically}.

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

Bounded Buffer

The statement “\texttt{count++}” may be implemented in machine language as:

\[
\begin{align*}
\text{register1} & = \text{counter} \quad \text{(lw $1$, (counter))} \quad \text{MIPS like ISA} \\
\text{register1} & = \text{register1} + 1 \quad \text{(addi, $1$, $1$)} \\
\text{counter} & = \text{register1} \quad \text{(sw $1$, (counter))}
\end{align*}
\]

The statement “\texttt{count--}” may be implemented as:

\[
\begin{align*}
\text{register2} & = \text{counter} \\
\text{register2} & = \text{register2} - 1 \\
\text{counter} & = \text{register2}
\end{align*}
\]
Bounded Buffer

- 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…

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

- *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, \( P_0 \) and \( P_1 \)
- General structure of process \( P_i \) (other process \( P \))

```plaintext
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;`
    - initially `turn = 0`
  - `turn - i ⇒ Pi` can enter its critical section
- Process `Pi`
  
  ```
  do {
    while (turn != i);
    critical section
    turn = j;
    reminder section
  } while (1);
  ```

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

Algorithm 2

- Shared variables
  - `boolean flag[2];` //retain state about each thread/proc…
    - initially `flag[0] = flag[1] = false.`
  - `flag[i] = true ⇒ Pi` ready to enter its critical section
- Process `Pi`
  
  ```
  do {
    flag[i] := true;
    while (flag[i]);
    critical section
    flag[i] = false;
    reminder section
  } 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 $P_i$
  
  ```
  do {
    flag[i] := true;
    turn = i;
    while (flag[j] and turn = j) ;
    critical section
    flag[i] = false;
    remainder section
  } while (1);
  ```

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

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., 1,2,3,3,3,3,4,5...
Bakery Algorithm

- Notation $\leq$ lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - $\max(a_0,\ldots,a_{n-1})$ is a number, $k$, such that $k \geq a_i$ for $i$ a number in $0,\ldots,n-1$

- Shared data
  
  ```
  boolean choosing[n];
  int number[n];
  ```

  Data structures are initialized to false and 0 respectively

```c
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[j] < number[i])) {
        }
    }
    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 {
      while (TestAndSet(lock)) ;
      critical section
      lock = false;
      remainder section
  }
  ```

  Wait here/test until/if 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.

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

Mutual Exclusion with Swap

- Shared data (initialized to false):
  ```c
  boolean lock; // lock is global/shared
  ```

- Process \( P_i \)
  ```c
  do {
      key = true; // key is local
      while (key == true)
          Swap(lock,key);
      lock = false;
  }
  ```
  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)

$$\text{wait (S):}$$

\[
\text{while } S \leq 0 \text{ do no-op;}
\]

\[
S--; \\
\]

$$\text{signal (S):}$$

\[
S++; \\
\]

Usage of semaphores: Critical Section of $n$ Processes

- Let us solve the $n$ Process problem with semaphores
- Shared data:
  - semaphore mutex; //initially $\text{mutex} = 1$
- Process $P_i$:

$$\text{do}$$

\[
\text{wait(mutex);}
\]

\[
\text{critical section}
\]

\[
\text{signal(mutex);}
\]

\[
\text{remainder section}
\]

$$\text{while (1);}$$
Semaphore Implementation

- Avoid busy-waiting, a disadvantage as it would use the processor unnecessarily
- Define a semaphore as a record
  
  ```
  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

  ```
  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:

\[
\begin{array}{c|c}
P_i & P_j \\ 
M & M \\ 
A & \text{wait(flag)} \\
\text{signal(flag)} & B \\
\end{array}
\]

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

\[
\begin{array}{c|c}
P_0 & P_1 \\
\text{wait(S)}; & \text{wait(Q)}; \\
\text{wait(Q)}; & \text{wait(S)}; \\
M & M \\
\text{signal(S)}; & \text{signal(Q)}; \\
\text{signal(Q)} & \text{signal(S)}; \\
\end{array}
\]

- **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:

```c
full = 0, empty = n, mutex = 1
```

Bounded-Buffer Problem Producer Process

```c
do {
    ...
    produce an item in `nextp`
    ...
    wait(empty);
    wait(mutex);
    ...
    add `nextp` to buffer
    ...
    signal(mutex);
    signal(full);
} while (1);
```
Bounded-Buffer Problem 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
Readers-Writers Problem Writer Process

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```

Readers-Writers Problem Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
  wait(rt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
  signal(wrt);
signal(mutex);
```
Dining-Philosophers Problem

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

Dining-Philosophers Problem

- Philosopher $i$:
  ```
  do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think
    ...
  } while (1);
  ```
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
  $$\text{region } v \text{ when } B \text{ do } S$$
  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 $v$. 
Example – Bounded Buffer

- Shared data:

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

Bounded Buffer Producer Process

- Producer process inserts `nextp` into the shared buffer

```c
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 `nextc`
  ```c
  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.

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

Schematic View of a Monitor
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;
    }
}
```
Dining Philosophers

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);
}

Dining Philosophers

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:

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

- 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.

- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - 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.
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstile* 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.
- Also provides *dispatcher objects* which may act as wither mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.