Lecture Topics

- Today: Concurrency
  (Stallings, chapter 5.1-5.4, 5.7)
- Next: Exam #1

Announcements

- Self-Study Exercise #5
- Project #3 (due 9/28)
- Project #4 (due 10/12)
Exam #1

- Tuesday, 10/3 during lecture
- 80 minutes, 18% of course grade
- Topics:
  - computer systems overview
  - operating system overview
  - processes and threads
  - concurrency
- Study suggestions on course website

Problems with Concurrent Execution

- Concurrent tasks (processes or threads) often need to share data in memory or files
- Actions performed by concurrent tasks depend on the order in which their execution is interleaved, which typically is not deterministic
- Must control access to shared data (otherwise data may become corrupted)
Review: Critical Sections

- When a process executes code that manipulates shared data, we say that the process is in a critical section (CS) for that shared data (aka critical region).

- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple CPUs).

Review: Critical Section Problem

- The critical section problem: design a protocol for tasks so that their actions will not depend on the order in which their execution is interleaved (possibly on many processors).

- Each task requests permission to enter its critical section (CS): entry section.

- Each task signals that it has left its critical section (CS): exit section.
Framework for Analysis of Solutions

Generalized structure of every process:

```plaintext
repeat
    preceding section
    entry section
    critical section
    exit section
    following section
forever
```

Framework (cont)

- More than one CPU may be present
- Hardware "serializes" accesses to memory
- Each process operates at non-zero speed, but no other assumptions about speed
- No assumptions about the order of interleaved execution
## Valid Solution: Classic Requirements

1. **Mutual Exclusion**: at any moment, at most one process can be in its critical section (CS)

2. **Progress**: if no process is executing in its CS, a process that requests entry should be allowed to enter its CS immediately

3. **Bounded Waiting**: upper bound on how long any process is forced to wait for entry to CS

## Valid Solution: Stallings

1. **Mutual Exclusion must be enforced**: at any moment, at most one process can be in its CS

2. **A process which halts outside its CS must not interfere with other processes**

3. **No indefinite delay**: deadlock and starvation cannot occur
Valid Solution: Stallings (cont)

4. When no process is executing in its CS, a process that requests entry should be allowed to enter its CS immediately

5. No assumptions about number (or relative speed) of processors

6. A process remains in its CS for a finite amount of time

Software Solutions

- Code for entry section and exit section uses loops and shared variables, does not depend on special hardware instructions or OS support

- Example: Peterson’s Algorithm (1981)

- Drawback: uses busy waiting
Hardware Solutions

- Code for entry section and exit section uses special hardware instructions which execute atomically

- Examples: SPARC SWAP, Intel XCHG

- Drawbacks: uses busy waiting, more complex entry section and exit section needed to ensure bounded waiting

Summary:

- Software solutions to Critical Section Problem using normal instructions are possible, but use busy waiting and are somewhat complex when extended to more than two processes

- Hardware solutions using special "atomic" instructions are possible, but use busy waiting and are somewhat complex when extended to ensure bounded waiting
Review: Semaphores

- Synchronization mechanism that blocks (and unblocks) processes to avoid busy waiting

- Three basic operations on semaphore S:
  - `init(S, value)`
  - `wait(S)` – block process, if necessary
  - `signal(S)` – unblock process, if any waiting

- Operations must be executed without interruption (semaphore is shared resource)

Semaphore Operations (Atomic)

Semaphore implemented as a record with two fields:
  - `count` -- integer
  - `queue` -- list of blocked processes

```c
struct semaphore {
    int count;
    queueType queue;
};
```
void wait( semaphore s ) {
    s.count--;
    if (s.count < 0)
        // put process in s.queue, block it
};

void signal( semaphore s ) {
    s.count++;
    if (s.count <= 0)
        // get some process from s.queue
};

Semaphores for Critical Sections

Initialize S to 1

repeat
    preceding section
    wait(S)
    critical section
    signal(S)
    following section
forever
Semaphores for Synchronization

- We have two processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- Initialize semaphore Synch to 0

```
P1: S1
    signal(Synch)

P2: wait(Synch)
    S2
```

The Producer/Consumer Problem

- A producer process produces information that is consumed by a consumer process
  - Example: a program produces characters that are consumed by a printer
- We need a buffer to hold items that are produced and eventually consumed
- A common paradigm for cooperating processes
Bounded Buffer (circular buffer of size k)

- Finite number of slots in the buffer (circular)
- Producer(s) and consumer(s) update the buffer (since in and out pointers are modified)
- Can consume **only** when there is at least one item (number N of items is at least one)
- Can produce **only** when there is at least one empty slot (number E of empty spaces is at least one)
### General Pattern

<table>
<thead>
<tr>
<th>Producer:</th>
<th>Consumer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
<td>repeat</td>
</tr>
<tr>
<td>produce item</td>
<td>item = buffer[out]</td>
</tr>
<tr>
<td>buffer[in] = item</td>
<td>out = (out+1) mod K</td>
</tr>
<tr>
<td>in = (in+1) mod K</td>
<td>consume item</td>
</tr>
<tr>
<td>forever</td>
<td>forever</td>
</tr>
</tbody>
</table>

### Solution using Semaphores

- Semaphore S to enforce mutual exclusion when accessing the buffer
- Semaphore N to synchronize producer and consumer (number of consumable items)
- Semaphore E to synchronize producer and consumer (number of empty slots)
Solution using Semaphores

Initialization:

\[
\begin{align*}
    \text{in} &= 0; \quad \text{// Next location to put item} \\
    \text{out} &= 0; \quad \text{// Next location to get item} \\
    \text{S.count} &= 1; \quad \text{// CS is free initially} \\
    \text{N.count} &= 0; \quad \text{// 0 slots in use} \\
    \text{E.count} &= K; \quad \text{// K slots empty}
\end{align*}
\]

Producer:

\[
\begin{align*}
\text{repeat} & \\
    \text{produce item} & \\
    \text{wait}(E) & \\
    \text{wait}(S) & \\
    \text{buffer}[\text{in}] &= \text{item} & \\
    \text{in} &= (\text{in}+1) \mod K & \\
    \text{signal}(S) & \\
    \text{signal}(N) & \\
    \text{forever} &
\end{align*}
\]

Consumer:

\[
\begin{align*}
\text{repeat} & \\
\text{wait}(N) & \\
\text{wait}(S) & \\
\text{item} &= \text{buffer}[\text{out}] & \\
\text{out} &= (\text{out}+1) \mod K & \\
\text{signal}(S) & \\
\text{signal}(E) & \\
\text{consume item} & \\
\text{forever} &
\end{align*}
\]
Problems with Semaphores

- Semaphores: powerful tool for enforcing mutual exclusion and coordinating processes.
- Uses of wait(S) and signal(S) are scattered among several processes: difficult to understand their effects.
- Usage must be correct in all the processes: one bad (or malicious) process can cause the entire collection of processes to fail.

Readers/Writers Problem

- A data area is shared among processes: some processes only read the data area (readers) and some only write to the data area (writers).
- Conditions that must be satisfied:
  - any number of readers may simultaneously read from the data
  - only one writer at a time may write to the data area
  - if a writer is writing to the data area, no reader may read from the data area.
Readers/Writers Problem

- Two possible strategies:
  - readers have priority over writers
  - writers have priority over readers

R/W: readers have priority

- Semaphore wsem used to enforce mutual exclusion on the critical sections in both readers and writers
- Integer rc used to track the number of readers who are currently reading the data
- Updating rc is therefore a critical section; semaphore x used to enforce mutual exclusion on the critical sections in readers
Reader:  
repeat  
  wait(x)  
  rc++  
  if (rc==1) wait(wsem)  
  signal(x)  
  READUNIT()  
  wait(x)  
  rc--  
  if (rc==0) signal(wsem)  
  signal(x)  
forever

Writer:  
repeat  
  wait(wsem)  
  WRITEUNIT()  
  forever

R/W: writers have priority  
- More complex to give writers priority  
- Integers readcount and writecount  
- Semaphores wsem, rsem, x, y and z
The Dining Philosophers Problem

- Five philosophers who only eat and think
- Each needs to use two forks to eat
- There are only 5 forks
- Illustrates the difficulty of allocating resources among processes without deadlock and starvation
The Dining Philosophers Problem

- One process for each philosopher
- Array of semaphores (one per fork); each initialized to 1
- Problem? Starvation if each philosopher picks up left fork

**Process Pi:**

```
repeat
  think
  wait(fork[i])
  wait(fork[i+1 mod 5])
  eat
  signal(fork[i+1 mod 5])
  signal(fork[i])
forever
```

The Dining Philosophers Problem

- One solution: only allow four philosophers to sit at the table
- Guarantees that one will be able to eat (even if other three are holding left forks)
- Use semaphore T to control access to table (initialize to 4)

**Process Pi:**

```
repeat
  think
  wait(T)
  wait(fork[i])
  wait(fork[i+1 mod 5])
  eat
  signal(fork[i+1 mod 5])
  signal(fork[i])
  signal(T)
forever
```