Lecture Topics

- Today: Concurrency: Mutual Exclusion
  (Stallings, chapter 5.1-5.4, 5.7)
- Next: Concurrency: Deadlock and Starvation
  (Stallings, chapter 6.1, 6.6-6.8)

Announcements

- Self-Study Exercise #5
- Project #3 (due 9/27)
- Project #4 (due 10/11)
Exam #1

- Thursday, 10/4 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)
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)

The 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:

\[
\text{repeat}
\quad \text{preceding section}
\quad \text{entry section}
\quad \text{critical section}
\quad \text{exit section}
\quad \text{following section}
\quad \text{forever}
\]

Framework (continued)

- 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 (continued)

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

Types of Solutions

- Software solutions: algorithms which do not rely on any other assumptions beyond framework described earlier
- Hardware solutions: rely on special machine instructions
- OS solutions: provide system calls and data structures in the kernel for use by user programs
Software Solutions

- Peterson's Algorithm works for two processes
- Can be generalized to more processes (Lamport's Bakery Algorithm)
- Notation:
  - Two processes: P0 and P1
  - When discussing process Pi, Pj always denotes the other process (i ≠ j)

Peterson’s Algorithm

repeat
  preceding section
  flag[i] = true;
  turn = j;
  do {} while (flag[j] and (turn==j));

  critical section
  flag[i] = false;

  following section
forever
**Drawbacks of Software Solutions**

- Processes that are requesting to enter their CS are **busy waiting** (consuming CPU cycles needlessly)

- If critical sections are long, it would be more efficient to block those processes that are waiting

- Note: busy waiting is acceptable if a critical section is short (such as in the kernel)

---

**Hardware Solutions: Special Instructions**

- Hardware already serializes memory access (if two accesses are attempted at the same time, the hardware does them one at a time)

- Create machine instructions that perform two actions **atomically (indivisibly)** on the same memory location (ex: reading and writing)

- Enforces mutual exclusion (more complex logic to meet requirements of CS problem)
SPARC SWAP instruction

- Assembly language notation:

  \[ \text{swap [address], register} \]

- Definition: the SWAP instruction atomically exchanges the contents of the register (4 bytes) and the contents of the word (4 bytes) at the specified memory location. In a multiprocessor system, two or more processors addressing the same memory location simultaneously are guaranteed to access it in an undefined, but serial, order.

Mutual Exclusion Using Swap Instruction

- Shared variable Bolt initialized to 0

- The first \( P_i \) which finds Bolt equal to 0 enters CS

\[
\text{Process } i: \\
\text{repeat} \\
\quad \text{preceding section} \\
\quad \text{Key} = 1 \\
\quad \text{do Swap( &Key, &Bolt ) while Key } \neq 0; \\
\quad \text{critical section} \\
\quad \text{Bolt} = 0; \\
\quad \text{following section} \\
\text{forever}
\]
Drawbacks of Hardware Solutions

- Processes that are attempting to enter their CS are **busy waiting** (consuming CPU cycles needlessly)

- Does not meet all criteria for valid solution to Critical Section problem: starvation is possible since choice of next process to enter CS is arbitrary

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
Semaphores

- A semaphore is a special variable that, apart from initialization, can only be accessed via **atomic and mutually exclusive** operations:
  - wait(S) -- sometimes P(S)
  - signal(S) -- sometimes V(S) or post(S)

- When a process has to wait, it will be put in a queue of processes which are blocked on that same semaphore

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: Observations

- When S.count >= 0: the number of processes that can execute wait(S) without being blocked equals S.count
- When S.count < 0: the number of processes waiting on S equals |S.count|
Semaphores: Observations

- Atomicity and mutual exclusion: must ensure that only one process is executing code in wait(S) or signal(S) for a given semaphore S

- Hence, the blocks of code defining wait(S) and signal(S) are critical sections and must be guarded

Semaphores: Observations

- The critical sections defined by wait(S) and signal(S) are very short (~10 instructions)

- Use one of the software or hardware schemes discussed previously

- Amount of busy waiting should be very small (and thus acceptable)
Semaphores for Critical Sections

Initialize S to 1

repeat
  preceding section
  wait(S)
  critical section
  signal(S)
  following section
forever

Normal execution can proceed in parallel, but execution in critical sections is serialized:
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

<table>
<thead>
<tr>
<th></th>
<th>P1: S1</th>
<th>P2: wait(Synch)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>signal(Synch)</td>
<td>S2</td>
</tr>
</tbody>
</table>

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

**Producer:**

```
repeat
  produce item
  buffer[in] = item
  in = (in+1) mod K
forever
```

**Consumer:**

```
repeat
  item = buffer[out]
  out = (out+1) mod K
  consume item
forever
```

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:

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

Producer:

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

Consumer:

\[
\begin{align*}
\text{repeat} \\
\text{wait}(N) \\
\text{wait}(S) \\
\text{item} &= \text{buffer}[out] \\
out &= (out+1) \ mod \ K \\
\text{signal}(S) \\
\text{signal}(E) \\
\text{consume item} \\
\text{forever}
\end{align*}
\]
Summary: 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
R/W: writers have priority

- More complex to give writers priority
- Integers \textit{readcount} and \textit{writecount}
- Semaphores \textit{wsem}, \textit{rsem}, \textit{x}, \textit{y} and \textit{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
```