Synchronization

Synchronize: to (arrange events to) happen at same time

Process synchronization
  • when one process has to wait for another
  • points in each process occur “at the same time”

Uses of synchronization
  • avoid race conditions
  • wait for resources to become available

Example of a Race Condition

The Credit/Debit Problem

Process $P_0$

Credit (a)
  int a;
  { int b;
    b = ReadBalance ();
    b = b + a;
    WriteBalance (b);
    PrintReceipt (b);
  }

Process $P_1$

Debit (a)
  int a;
  { int b;
    b = ReadBalance ();
    b = b - a;
    WriteBalance (b);
    PrintReceipt (b);
  }

Critical sections

To Avoid Race Conditions

Identify critical sections
• sections of code executed by multiple processes
• must run atomically with respect to each other

Enforce mutual exclusion
• only one process active in a critical section

Four Rules for Mutual Exclusion
1. No two processes inside critical sections at same time
2. No process outside a critical section may block others
3. No process may wait forever to enter critical section
4. No assumptions about speeds or number of CPU’s

How to Achieve Mutual Exclusion?
Surround critical section with entry/exit code

Entry code should act as a barrier
• if another process is in their critical section, block
• otherwise, allow process to proceed

Exit code should act to release other entry barriers

Possible Solution: Software Lock?
Lock indicates whether any process is in critical section

Race condition in while loop (breaks Rule 1)
**Possible Solution: Take Turns?**

Alternate which process can enter critical section

\[
\text{shared int turn} = P_0; \\
\text{P}_0 \quad \text{while ( turn} \neq P_0); \text{> critical section > turn} = P_1; \\
\text{P}_1 \quad \text{while ( turn} \neq P_1); \text{> critical section > turn} = P_0;
\]

Prevents entry even if OK to enter (breaks Rule 2)

**Possible Solution: State Intention?**

Each process states it wants to enter critical section

\[
\text{shared boolean flag[2] = \{FALSE, FALSE\};} \\
\text{P}_0 \quad \text{flag[P}_0\text{]} = \text{TRUE}; \text{> critical section > flag[P}_0\text{]} = \text{FALSE}; \\
\text{P}_1 \quad \text{flag[P}_1\text{]} = \text{TRUE}; \text{> critical section > flag[P}_1\text{]} = \text{FALSE};
\]

Race condition: prevents entry forever (breaks Rule 3)

**Peterson’s Solution**

If there is competition, take turns; otherwise, enter

\[
\text{int turn;} \\
\text{shared boolean flag[2] = \{FALSE, FALSE\};} \\
\text{P}_0 \quad \text{flag[P}_0\text{]} = \text{TRUE;} \text{ turn} = P_1; \text{> critical section > flag[P}_1\text{]} = \text{FALSE;} \\
\text{P}_1 \quad \text{flag[P}_1\text{]} = \text{TRUE;} \text{ turn} = P_0; \text{> critical section > flag[P}_0\text{]} = \text{FALSE;}
\]

Works! Extends to \( n \) processes. Problem: busy-waiting

**What about Turning Off Interrupts?**

By turning off interrupts, context-switching can’t occur

\[
\text{P}_0 \quad \text{InterruptsOff ();} \text{> critical section > InterruptsOn ();} \\
\text{P}_1 \quad \text{InterruptsOff ();} \text{> critical section > InterruptsOn ();}
\]

Too restrictive: locks out all processes, even those not in critical section (breaks Rule 2)

Doesn’t work on a multiprocessor (breaks Rule 4)
**Hardware Solution: Test&Set Instr.**

TSET mem: simultaneously test mem and set it to 1

- \( \{ \text{reg} \leftarrow \text{mem}, \text{mem} \leftarrow 1, \text{set cond code} \} \) atomically
- C func: tset (lock) sets lock to 1, returns orig value

\[
\begin{align*}
\text{shared int lock} & = 0: \\
P_0 & \quad \text{while (tset (lock) == 1):} \\
& \quad < \text{critical section} > \\
& \quad \text{lock} = 0: \\
P_1 & \quad \text{while (tset (lock) == 1):} \\
& \quad < \text{critical section} > \\
& \quad \text{lock} = 0:
\end{align*}
\]

Simple, works for n processes (but still busy-waits!)

---

**Semaphores**

Semaphore: integer variable used for synchronization

- has integer value
- list of waiting processes

Works like a gate

- If value > 0, gate is open, and value indicates number of processes that can enter
- Otherwise, gate is closed, possibly with waiting processes

**Semaphore Operations**

Semaphore \( s = n \) /* declare and initialize */

Wait (s)

- if \( s \) is zero, block process (and associate with \( s \))
- decrement \( s \) (note: occurs after process unblocks)

Signal (s)

- increment \( s \)
- if any blocked processes, unblock one of them

No other operations permitted (e.g., can’t test value)

**Mutual Exclusion with Semaphores**

Use a “mutex” semaphore initialized to 1

\[
\begin{align*}
\text{sem mutex} & = 1 \\
P_0 & \quad \text{Wait (mutex):} \\
& \quad < \text{critical section} > \\
& \quad \text{Signal (mutex):} \\
P_1 & \quad \text{Wait (mutex):} \\
& \quad < \text{critical section} > \\
& \quad \text{Signal (mutex):}
\end{align*}
\]

Allows only one process to enter critical section

Simple, works for n processes, no busy-waiting (really?)
Synchronization with Semaphores

Use a “synch” semaphore initialized to 0

\[
\text{sem synch} = 0; \\
P_0 \text{ } \begin{array}{l} \text{P}_0 < \text{ to be done before P}_1 > \\ \text{Signal (synch)}: \end{array} \text{Wait (synch);} \begin{array}{l} \text{P}_1 \text{ < to be done after P}_0 > \\ \end{array}
\]

Allows a process to wait for another before proceeding

Semaphores provide pure synchronization
- no way for a process to tell it blocked
- i.e., no information transfer

Atomicity of Semaphore Operations

Wait (sem) and Signal (sem) are atomic operations
- their bodies are critical sections

Mutual exclusion achieved using a lower-level mechanism
- test-and-set locks
- turning off interrupts (on a uniprocessor)

Therefore, problems such as busy-waiting still exist
- but at a “lower-level”
- for brief (and known) periods of time