Synchronization

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

Process synchronization
- When one process has to wait for another
- Events in processes that occur “at the same time”

Uses of synchronization
- Prevent race conditions
- Wait for resources to become available

Credit/Debit Problem: Race Condition

Process $P_0$
- Credit (int $a$) {
  - int $b$;
  - $b =$ ReadBalance ($b$);
  - $b =$ $b + a$;
  - WriteBalance ($b$);
  - PrintReceipt ($b$);
  - Write $900$
  - Credit by $100$
  - Write $1100$
}

Process $P_1$
- Debit (int $a$) {
  - int $b$;
  - $b =$ ReadBalance ($b$);
  - $b =$ $b - a$;
  - WriteBalance ($b$);
  - PrintReceipt ($b$);
  - Debit by $900$
}

Say $P_1$ runs first
- read $1000$ into $b$
- Switch to $P_0$
- read $1000$ into $b$
- Debit by $900$
- Write $900$

Switch to $P_0$
- Credit $100$
- Write $1100$

Bank just lost $100$!
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 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 > < critical section > < exit code >

Entry code should act as a barrier
• If another process is in 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

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

Alternate which process can enter critical section

```
shared int turn = P0;
P0
while ( turn != P0 ) ;
< critical section >
turn = P1;
P1
while ( turn != P1 ) ;
< critical section >
turn = P0;
```

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

**Possible Solution: State Intention?**

Each process states it wants to enter critical section

```
shared boolean flag[2] = {FALSE, FALSE};
P0
flag[P0] = TRUE;
< critical section >
flag[P0] = FALSE;
P1
flag[P1] = TRUE;
< critical section >
flag[P1] = FALSE;
```

Race condition: prevents entry forever (breaks Rule 3)

**Peterson’s Solution**

If there is competition, take turns; otherwise, enter

```
shared int turn;
shared boolean flag[2] = {FALSE, FALSE};
P0
flag[P0] = TRUE;
turn = P1;
while ( flag[P1] && turn == P1 ) ;
< critical section >
flag[P0] = FALSE;
P1
flag[P1] = TRUE;
turn = P0;
while ( flag[P0] && turn == P0 ) ;
< critical section >
flag[P1] = FALSE;
```

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

**What About Disabling Interrupts?**

By disabling interrupts, context-switching can’t occur

```
P0
DisableInterrupts ();
< critical section >
EnableInterrupts ();
P1
DisableInterrupts ();
< critical section >
EnableInterrupts ();
```

Too restrictive (breaks Rule 2)

- locks out all processes
- even those not in critical section

Doesn’t work on a multiprocessor (breaks Rule 4)
Hardware Solution: TSET Instruction

Hardware instruction: TSET mem
- simultaneously test mem and set it to 1
- \{ reg ← mem, mem ← 1, set cond code \} atomically

C function: tset (lock) sets lock to 1, returns orig value

```c
shared int lock = 0;
P0while ( tset (lock) == 1 );< critical section >
lock = 0;
P1while ( tset (lock) == 1 );< critical section >
lock = 0;
```

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

Semaphores

Semaphore: synchronization variable
- Has integer value
- List of waiting processes

Works like a gate
- If sem > 0, gate is open
  - value indicates number of processes that can enter
- Else, gate is closed
  - possibly with waiting processes

Semaphore Operations

```c
sem s = n;        /* declare and initialize */
wait (sem s)      // Decrement s
  • If sem < 0, block process (and associate with sem)
signal (sem s)    // Increment s
  • If blocked processes, unblock (any) one of them

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

An Incorrect Definition

```c
sem s = n;        /* declare and initialize */
wait (sem s)      // Decrement s
  • If s == 0, block process (and associate with sem)
  • Decrement s (note: occurs after process unblocks)
signal (sem s)    // Increment s
  • If blocked processes, unblock (any) one of them

WHERE IS THE RACE CONDITION?
```
Mutual Exclusion with Semaphores

Use a "mutex" semaphore initialized to 1

\[
\text{sem mutex} = 1
\]

\[
P_0 \quad \text{wait (mutex);} \\
\quad \text{< critical section >} \\
\quad \text{signal (mutex);} \\
\]

\[
P_1 \quad \text{wait (mutex);} \\
\quad \text{< critical section >} \\
\quad \text{signal (mutex);} \\
\]

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

\[
\text{sem synch} = 0
\]

\[
P_0 \quad \text{signal (synch);} \quad \text{< to be done before } P_1 \text{ >} \\
\quad \text{wait (synch);} \\
\quad \text{< to be done after } P_0 \text{ >} \\
\quad \text{signal (synch);} \\
\]

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 (s) and signal (s) are atomic operations

- Their bodies are critical sections

Mutual exclusion achieved using a lower-level mechanism

- Test-and-set locks
- Disabling interrupts (on a uniprocessor)

Therefore, problems such as busy-waiting still exist

- But at a "lower-level"
- For brief (and known) periods of time