Synchronizing Two Cooperating Processes

Process P_0

```
while (TRUE) {

   /* ENTRY SECTION */

   critical section

   /* EXIT SECTION */

   remainder section
}
```

Process P_1

```
while (TRUE) {

   /* ENTRY SECTION */

   critical section

   /* EXIT SECTION

   remainder section
}
```

- In the critical sections, P_0 and P_1 access some shared data structure.
- What entry/exit code will properly synchronize these two processes?
- Three properties are needed: mutual exclusion, progress, and bounded waiting.
- No assumption about the speed or number of CPUs should be made.
- Interesting generalization: $n \geq 2$ cooperating processes

Idea #1: Taking Turns

Introduce a shared variable: `int turn = 0;`

Process P_0

```
while (TRUE) {

   /* ENTRY SECTION */
   while (turn != 0) {
      /* Spin: not my turn */
   }

   critical section

   /* EXIT SECTION */
   turn = 1;   /* Let the other process take a turn */

   remainder section
}
```

Process P_1

```
while (TRUE) {

   /* ENTRY SECTION */
   while (turn != 1) {
      /* Spin: not my turn */
   }

   critical section

   /* EXIT SECTION */
   turn = 0;   /* Let the other process take a turn */

   remainder section
}
```

Idea #2: State Your Interest

Introduce a shared array variable: `bool interested[2] = {FALSE, FALSE};`

Process P_0

```
while (TRUE) {

   /* ENTRY SECTION */
   interested[0] = TRUE;  /* I want access */
   while (interested[1]) {
      /* Spin: the other process wants access */
   }

   critical section

   /* EXIT SECTION */
   interested[0] = FALSE;   /* I don't want access */

   remainder section
}
```

Process P_1

```
while (TRUE) {

   /* ENTRY SECTION */
   interested[1] = TRUE;  /* I want access */
   while (interested[0]) {
      /* Spin: the other process wants access */
   }

   critical section

   /* EXIT SECTION */
   interested[1] = FALSE;   /* I don't want access */

   remainder section
}
```

Idea #3: Best of Both Worlds (Peterson's algorithm)

Introduce shared variables:

```
int turn = 0;
bool interested[2] = {FALSE, FALSE};
```

Process P_0

```
while (TRUE) {

   /* ENTRY SECTION */
   interested[0] = TRUE;  /* I want access */
   turn = 1;              /* but you go ahead */
   while (interested[1] && turn == 1) {
      /* Spin... */
   }

   critical section

   /* EXIT SECTION */
   interested[0] = FALSE;   /* I don't want access */

   remainder section
}
```

Process P_1

```
while (TRUE) {

   /* ENTRY SECTION */
   interested[1] = TRUE;  /* I want access */
   turn = 0;              /* but you go ahead */
   while (interested[0] && turn == 0) {
      /* Spin... */
   }

   critical section

   /* EXIT SECTION */
   interested[1] = FALSE;   /* I don't want access */

   remainder section
}
```

The "Classical" Semaphore

- Invented and popularized by Dijkstra $\approx$ 1965
- Software solution
- Uses busy waiting
- A semaphore is a shared integer variable : can take on values $0, 1, 2, \ldots, k$
- Apart from initialization, we are restricted to these **atomic** operations
  - `wait` (originally named `P`)
  - `signal` (originally named `V`)
- Two variations:
  - **counting semaphore** : take on values $0, 1, 2, \ldots, k$
  - **binary semaphore** : take on values 0 and 1 only
  - Counting semaphores "feel" more capable, but both types have the same power

```
wait(S) {
   while (S <= 0) {
      /* spin */
   }
   S--;
}
```

```
signal(S) {
   S++;
}
```

A Common Use of Semaphores

Introduce a semaphore:

```
semaphore mutex = 1;
```

Process P_0

```
while (TRUE) {

   /* ENTRY SECTION */
   wait(mutex)

   critical section

   /* EXIT SECTION */
   signal(mutex)

   remainder section
}
```

Process P_1

```
while (TRUE) {

   /* ENTRY SECTION */
   wait(mutex)

   critical section

   /* EXIT SECTION */
   signal(mutex)

   remainder section
}
```

A Semaphore Without Busy Waiting

- A semaphore is a shared integer variable : can take on values $0, 1, 2, \ldots, k$
- Semaphore operations continue to be **atomic**
- To avoid busy-waiting, OS support is required: `block()` and `wakeup(P)`
- The semantics described here are used by the BACI system

```
wait(S) {
   if (S > 0)
      S--;
   else
      block();
}
```

```
signal(S) {
   if ((S == 0) && (processes are blocked on S)) {
      choose a process P blocked on S;
      wakeup(P);
      }
   else
      S++;
}
```

List-based Semaphores (Silberschatz textbook)

- Does not use busy waiting
- Requires OS support to block/wakeup processes
- As before, atomicity required
- Each semaphore has two components:
  - an integer `value`
  - a `list` of blocked processes
- Semaphores can now take on negative, zero, or positive values

```
wait(S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

```
signal(S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Solving the Bounded Buffer Problem

Initially,

```
sem mutex = 1;   /* allow at most one process access to the shared buffer */
sem full = 0;    /* the number of occupied slots in the buffer */
sem empty = N;   /* the number of empty slots in the buffer */
```

Producer:

```
while (TRUE) {

   // ... produce an item in nextp ...

   wait(empty);

   wait(mutex);
   // ... add nextp to buffer ...
   signal(mutex);

   signal(full);
}
```

Consumer:

```
while (TRUE) {
   wait(full);

   wait(mutex);
   // ... remove an item from buffer to nextc ...
   signal(mutex);

   signal(empty);

   // ... consume the item in nextc ...

}
```

Solving the Readers/Writers Problem (First Version)

Initially,

```
sem mutex = 1;       /* allows controlled access to update readcount */
sem wrt = 1;         /* allow at most one writer access at a time */
int readcount = 0;   /* the number of current readers */
```

Writer:

```
while (TRUE) {

   wait(wrt);
   // ... update the database ...
   signal(wrt);

}
```

Reader:

```
while (TRUE) {

   wait(mutex);
   readcount++;
   if (readcount == 1) {
      /* first reader to gain access */
      wait(wrt);
      }
   signal(mutex);

   // ... read the database ...

   wait(mutex);
   readcount--;
   if (readcount == 0) {
      /* last reader to relinquish database */
      signal(wrt);
      }
   signal(mutex);
}
```