Synchronization

CSD2180 Operating Systems
BSc in Computer Science (IMGD / RTIS)
Singapore Institute of Technology / DigiPen Institute of Technology
September 2021

SingaporeTech.edu.sg

Adapted from Operating Systems Concepts - 10th Edition; Silberschatz, Galvin, and Gagne. ©2018

Background

Concurrency

In a system supporting concurrency

- **>1 execution flow** exist at the same time
- These execution flows often **share resources**
- In some systems, **share the same processor** (interleaving with each other)
- In other systems they **run on multiple processors**, (i.e., run in parallel)

Concurrency allows more efficient use of resources (e.g., time, processor, memory, input, output devices, etc.)

All modern operating systems are concurrent systems, as the kernel and multiple processes exist at the same time

Problem?

Problems with Concurrency

Race condition
Multiple processes reading and writing shared data; result depends on relative timing of processes

Deadlock
Multiple processes wait for each other and none can proceed any further

Starvation
A process is stuck because it cannot obtain the resource(s) it needs to continue

Livelock
Multiple processes continuously change their states in response to changes in other processes without doing any useful work

Let's Talk About Data Consistency

- Processes can execute concurrently; may be interrupted at any time, partially completing execution
- Concurrent access to **shared data** may result in **data inconsistency**
- Maintaining **data consistency** requires mechanisms to ensure the **orderly execution of cooperating processes**

Introducing the producer-consumer problem 🏃

The Producer-Consumer Problem

Producer produces an item and **places it in a buffer** for the consumer

Consumer will **remove an item from the buffer** and consume it

Simple, right?

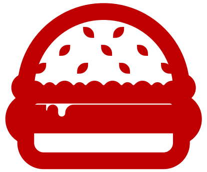
```
/* le producer */
while (true) {
    /* produce an item in next_produced */
    while (counter == BUFFER_SIZE); /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
/* le consumer */
while (true) {
    while (counter == 0); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next consumed */
}
```

What can possibly go wrong? 

The Producer-Consumer Problem

Suppose at one point, `counter` == 5

Concurrently

- Producer produces one item
- Consumer consumes one item

We should now have `counter` == 5

But we may end up with `counter` == 4, 5, or 6! (why?)

```
/* le producer */
while (true) {
    /* produce an item in next_produced */
    while (counter == BUFFER_SIZE); /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
/* le consumer */
while (true) {
    while (counter == 0); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next consumed */
}
```

Race Condition

`counter++` could be implemented as a 3-step set of instructions in machine language

- `r1 = counter`
- `r1 = r1 + 1`
- `counter = r1`

`counter--` could be implemented as another 3-step set of instructions in machine language

- `r2 = counter`
- `r2 = r2 - 1`
- `counter = r2`

Okay, but still, what can possibly go wrong?

Race Condition

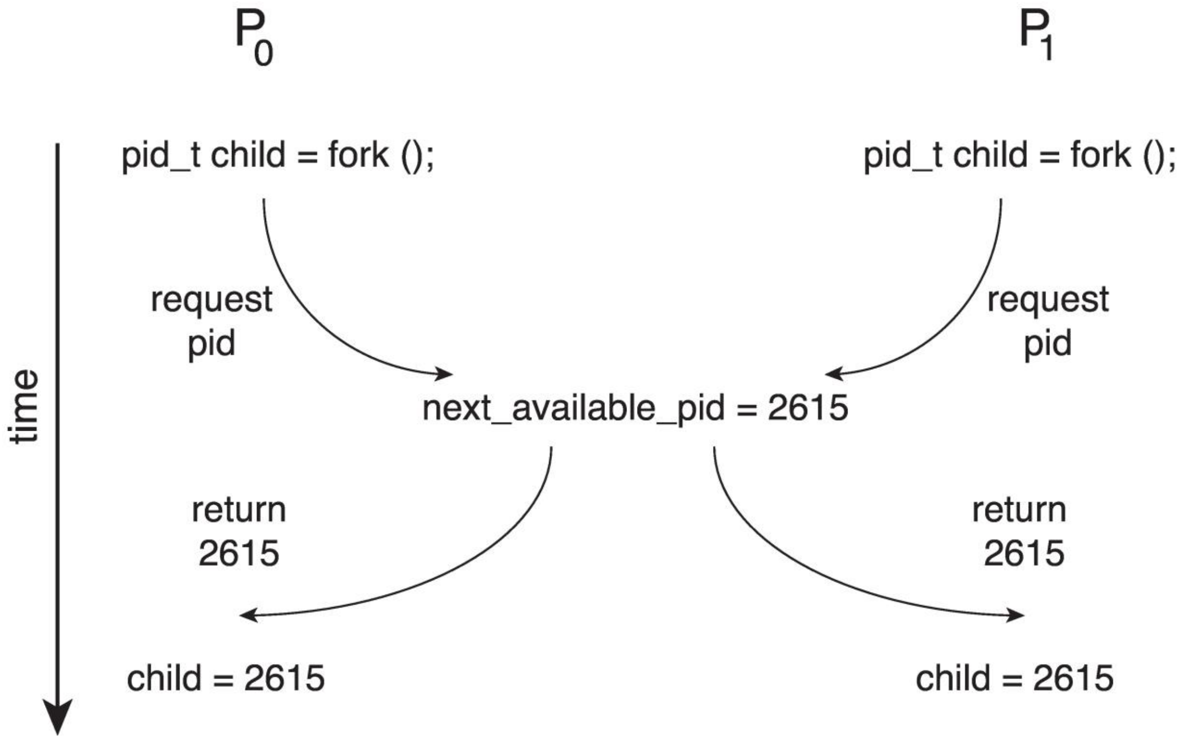
T+0: `counter == 5`	`{counter == 5; r1 == 0; r2 == 0}`
T+1: producer executes `r1 = counter`	`{counter == 5; r1 == 5; r2 == 0}`
T+2: producer executes `r1 = r1 + 1`	`{counter == 5; r1 == 6; r2 == 0}`
T+3: consumer executes `r2 = counter`	`{counter == 5; r1 == 6; r2 == 5}`
T+4: consumer executes `r2 = r2 - 1`	`{counter == 5; r1 == 6; r2 == 4}`
T+5: producer executes `counter = r1`	`{`**`counter == 6`**`; r1 == 6; r2 == 4}`
T+6: consumer executes `counter = r2`	`{`**`counter == 4`**`; r1 == 6; r2 == 4}`

Race Condition in an Operating System

- Processes P_0 and P_1 are creating child processes using the `fork()` system call

- Race condition may happen on kernel variable `next_available_pid` which represents the next available process identifier (`pid`)

- Unless there is a mechanism to control, in an orderly manner, P_0's and P_1's access to the variable `next_available_pid`, the same `pid` could be assigned to two different processes!



Critical-Section Problem

The Critical-Section Problem

Consider system of n processes $\{p_0, p_1, \ldots, p_{n-1}\}$

Each process has **critical section** segment of code

- Process may be changing **common** variables, updating table, writing file, etc.
- When one process in their critical section, **no other processes** may be in its critical section

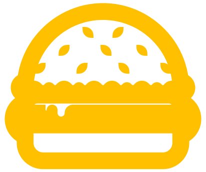
Such a model presents the critical-section problem

The aim is to design a protocol so that these processes can cooperatively share data

Each process must ask permission to enter its critical section (entry section), and the critical section may be followed by an exit section

```
while (true) {
    /* entry section */
    . . .
    /* critical section */
    . . .
    /* exit section */
    . . .

    /* remainder section */
}
```

How do we solve this problem? 

Solving the Critical-Section Problem

Mutual exclusion

If process P_i is executing its critical section with respect to a particular resource, then no other processes can be executing in their critical sections with respect to that resource

Progress

If no process is executing its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely

Bounded waiting

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Let's see...

Software Solution for Two Processes

Solution for two processes P_0 and P_1

```
while (true) {

    while (turn == (1 - i)); /* entry section */
    . . .
    /* critical section */
    . . .
    turn = (1 - i); /* exit section */
    . . .

    /* remainder section */
}
```

The two processes share one variable `int turn`

- `turn` indicates which process can enter its critical section

Let `i` denote the process number, either 0 for P_0 or 1 for P_1

Mutual exclusion is preserved!

What about bounded-wait? What about progress?

Software Solution for Two Processes

```
/* at P0 */
while (true) {

    /* entry section */
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

```
/* at P1 */
while (true) {

    /* entry section */
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* okay, i'm gonna chill */
    /* proceeds to run indefinitely... */
}
```

Software Solution for Two Processes

```
/* at P0 */
while (true) {

    /* entry secti
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

Dude, can I enter my critical section now? 🤔

```
/* at P1 */
while (true) {

    /* entry section */
    while (turn == (1 - i));
    . . .
    /* critical section */
    . . .
    /* exit section */
    turn = (1 - i);
    . . .

    /* okay, i'm
    /* proceeds to run indefinitely... */
}
```

Hahaha... No ☺

Peterson's Solution

Solution for two processes P_0 and P_1

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* remainder section */
}
```

The two processes share *two* variables `int turn` and `boolean flag[2]`

- `turn` indicates **which process** can enter its critical section
- `flag` array indicates if a process **is ready** to enter the critical section; `flag[i] = true` implies that process P_i is ready

Let `i` denote the process number, either 0 for P_0 or 1 for P_1

Again, mutual exclusion is preserved!

What about bounded-wait? What about progress? 🍔

Peterson's Solution Example

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* okay, i'm gonna chill */
    /* proceeds to run indefinitely... */
}
```

Peterson's Solution Example

```
while (true) {

    /* entry sectio
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* remainder section done quickly */
    /* back to start of while(true) */
}
```

Dude, can I enter my critical section now? 🤔

```
while (true) {

    /* entry section */
    flag[i] = true;
    turn = (1 - i);
    while (turn == (1 - i) && flag[1 - i]);

    . . .
    /* critical section */
    . . .

    /* exit section */
    flag[i] = false;

    . . .

    /* okay, i'm g
    /* proceeds to run indefinitely... */
}
```

Okay, sure! ☺

Okay, so Peterson's Solution meets the three requirements set out for the problem 🍔

But...

Although useful for demonstration, Peterson's Solution is not guaranteed to work on modern architectures...

Questions?

Thank You!

Weihan.Goh {at} Singaporetech.edu.sg

https://www.singaporetech.edu.sg/directory/faculty/weihan-goh

https://sg.linkedin.com/in/weihan-goh