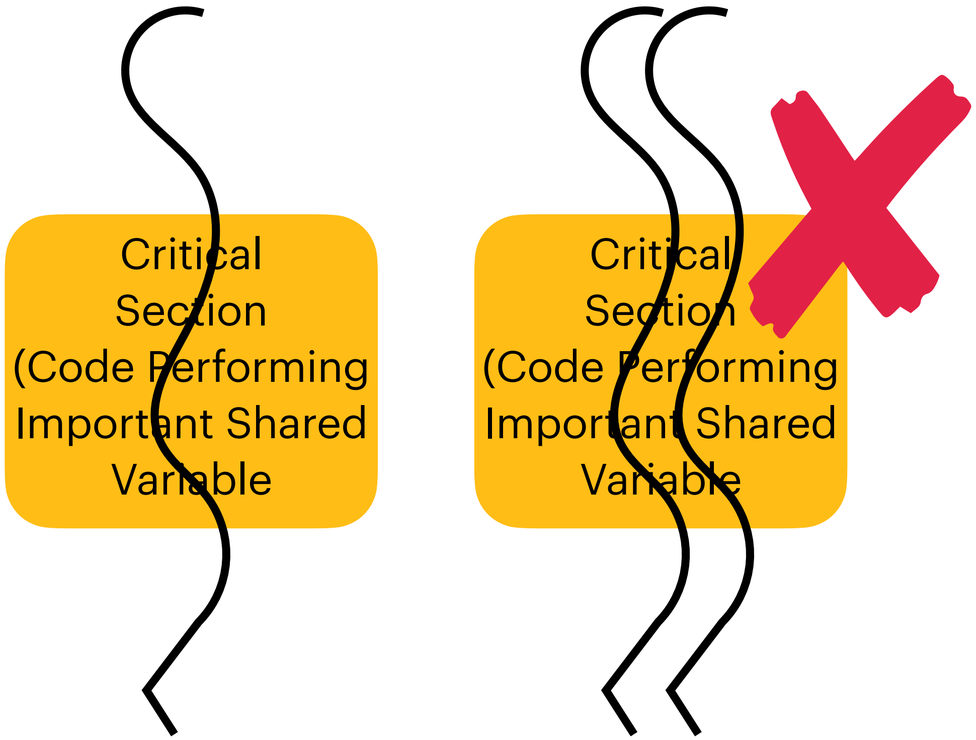
Synchronization Techniques

Cooperating Processing with Processes or Threads

Our Objectives

- Describe the critical section problem and illustrate a race condition
- Illustrate a software solution to critical section problem
- Illustrate hardware solutions to critical sections, using memory barriers, compare-and-swap, atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve critical section problems
- Evaluate tools that solve critical section problems in different scenarios



Concurrent Processes

We can have two types of concurrent processes

- Independent concurrent processes - they are competing for resources - but don't affect each other
  - Easy to deal with - isolate one processes from another
  - Carefully allocate resources so all processes are making progress fairly
- Cooperating concurrent processes - they are using some IPC to work with each other
  - Dealing with cooperating processes is much harder - correctness of the program needs to be enforced
  - Data/resource sharing can lead to inconsistent execution

Cooperating Processes

- Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other
- Example - transaction processes in an airline reservation system
- Properties
  - Share a common object or exchange messages
  - Non-deterministic, may be irreproducible
  - Subject to race conditions
- Cooperation clearly presents challenges - why do we want it?
- We may want to share resources - sharing a global task queue
- We may want to do things faster - read next block while processing the current one, divide a job into pieces and execute concurrently
- We may want to solve problems modularly - UNIX example:

```
cat infile | tr ‘’ ‘\012’ | tr
‘[A-Z]’ ‘[a-z]’ | sort | uniq -c
```

Problem with Concurrency

- Hard to understand or visualize concurrently executing programs
- Instructions of the programs are interleaved arbitrarily
- Certain instruction combinations must be avoided for correct execution

Process A	Process B	concurrent access
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

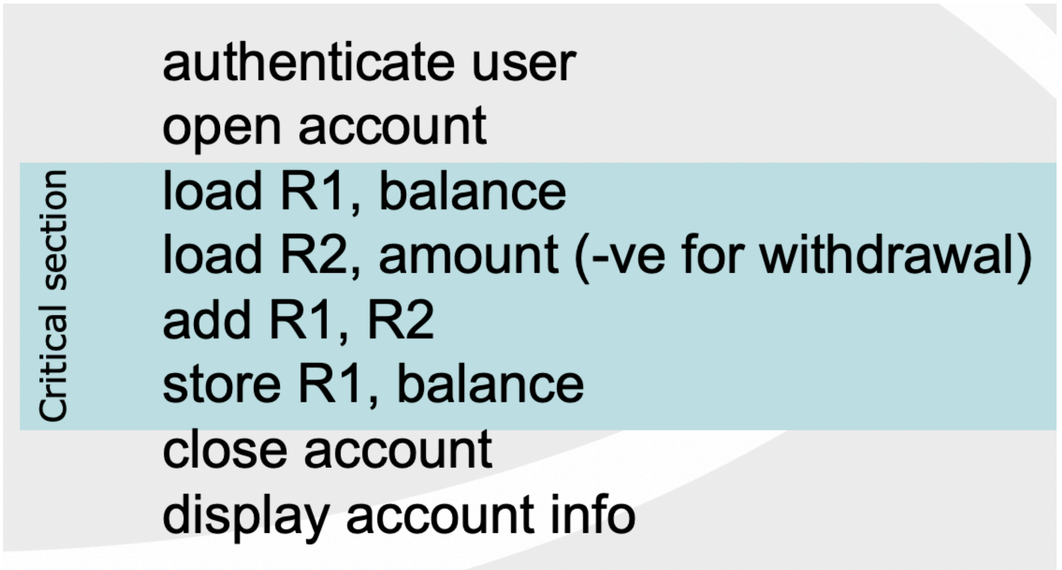
Race Conditions

- When two or more processes are reading or writing shared data, final result depends on who runs precisely when is a race condition
- How to avoid race condition?
  - Prohibit more than one process reading and writing shared data at the same time - need mutual exclusion

An Example

- Suppose two processes A & B are updating a shared account. A is withdrawing $50 and B is depositing $50, initial balance is $100
- What are the possible final balances?

Process A	**Process B**
load R1, balance	load R3, balance
load R2, 50	load R4, 50
sub R1, R2	add R3, R4
store R1, balance	store R3, balance



Critical Section

- Critical section (CS) is the part of the program that accesses shared data
- If we arrange runs of a program such that no two runs (processes) are in their CS at the same time, race conditions can be avoided
- CS execution should atomic - runs all or none
- Should not be interrupted in the middle
- Efficiency -> minimize the length of the CS
- Most inefficient scenario -> whole program is a CS -> no multiprogramming

Requirements of a Critical Section

- For multiple programs to cooperate correctly and efficiently
  - No two processes may be ***simultaneously in*** their critical sections
  - No assumptions be made about ***speeds*** or number of CPUs
  - No process running ***outside*** its critical section may block other processes
  - No process should have to ***wait forever*** to enter its critical section

Implementing Critical Sections

Software only approach without any hardware support

Simplest solution

- Disable all interrupts
- This should work in a single processor situation
- Because interrupts cause out-of-order execution due to interrupt servicing
- With interrupts disabled, a process will run until it yields the resources voluntarily

- Not practical
  - OS won't all a user process to disable all interrupts
  - Does not work in multiprocessors

Another Idea

- Use "lock" variables to prevent two processes entering the CS at the same time - we are considering a single CS
- Use variable `lock`
- If `lock == 0` set `lock = 1` and `enter_CS`
- If `lock == 1` wait until lock becomes o
- Does **not work**, why??

Strict Alternation

- Two processes take turns in entering the critical section
- Global variable turn set either to 0 or 1

```
Process 0                       Process 1

while (TRUE) {              while (TRUE) {
    while (turn !=0);           while (turn !=1);
    critical_section();         critical_section();
    turn = 1;                   turn = 0;
    non_critical();             non_critical();
}                           }
```

- What is the problem with strict alternation?
- We can have starvation? Why?
- What are the other drawbacks?
  - Continuously testing a variable until some value appears is called **busy waiting**
  - Busy waiting wastes CPU time - should be used only when expected wait is short
  - A lock that uses busy waiting is called **spin lock**

Example - Busy Waiting

- Kernel level multi-threading is used for concurrent processing in a data parallel application
- The application has a critical section
- Single thread execution: CS takes 2 sec and other part takes 6 sec
- We have a large dataset - we use 2 threads, how long would it take?
- CS is implemented using a busy waiting approach

Have a single processor

Developing a Critical Section

Many attempts and their problems

- Simplest is taking turns as considered earlier - not working

```
/* process 0 */
.
.
while (turn != 0);
/* critical section */
turn = 1;
.
.
```

```
/* process 1 */
.
.
while (turn != 1);
/* critical section */
turn = 0;
.
.
```

Both processes shared a single key to the critical section

- Each process with its own key to the critical section

Does not provide a CS (no mutual exclusion)

```
/* process 0 */
.
.
while (flag[1]);
flag[0] = true;
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
while (flag[0]);
flag[1] = true;
/* critical section */
flag[1] = false;
.
.
```

- Provides mutual exclusion (prevents both processes inside the CS at the same time)

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1]);
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0]);
/* critical section */
flag[1] = false;
.
.
```

Has a deadlock

Towards Peterson's Algorithm

Working solution for two processes without hardware support

- Works - but has livelock, why?

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1])
{
        flag[0] = false;
        /* random delay */
        flag[0] = true;
}
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0])
{
        flag[1] = false;
        /* random delay */
        flag[1] = true;
}
/* critical section */
flag[1] = false;
.
.
```

- Working Peterson's Algorithm
- Only for two processes

```
/* process 0 */
…
flag[0] = true;
turn = 1;
while (flag[1] &&
        turn == 1);
/ * critical section */
flag[0] = false;
/* remainder */
…
```

```
/* process 1 */
…
flag[1] = true;
turn = 0;
while (flag[0] &&
        turn == 0);
/* critical section */
flag[1] = false;
/* remainder */
…
```

Race Condition in a Producer-Consumer

If producer-consumer is not carefully implemented, you can lose, duplicate data

counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

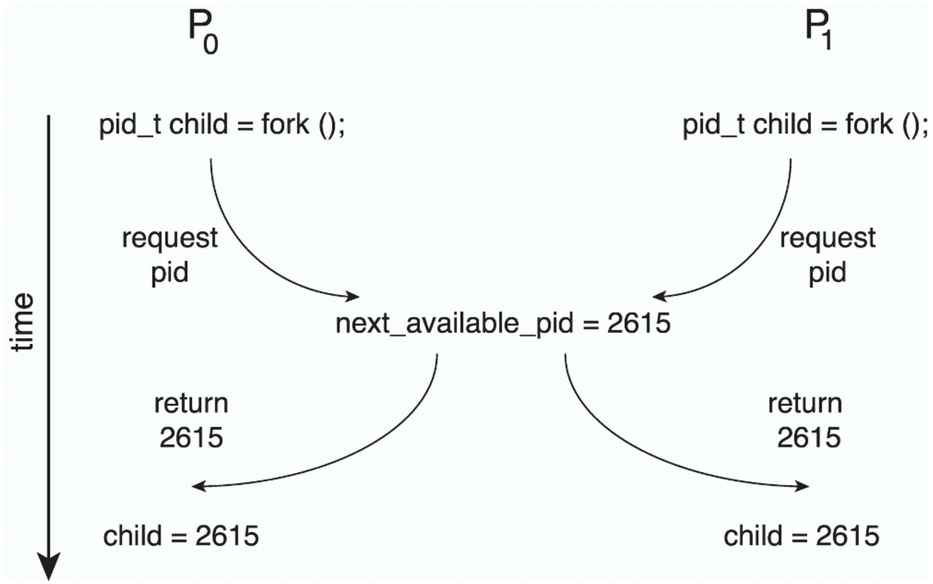
```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition In Process Creation

This *does not happen* - just an illustration of what could be a problem!

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

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



Unless there is mutual exclusion, the same pid could be assigned to two different processes!

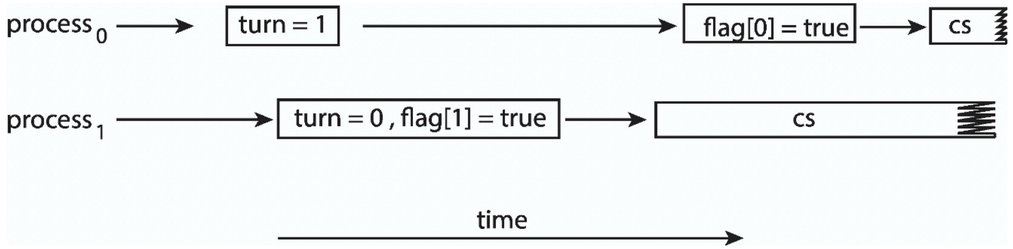
Peterson's Algorithm, Limitations

Why Peterson's Algorithm might fail in modern computer systems?

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

These two statements can be reordered by the compiler because they are independent

The effects of instruction reordering in Peterson's Solution



This allows both processes to be in their critical section at the same time!

Reordering the statements breaks Peterson's Algorithm