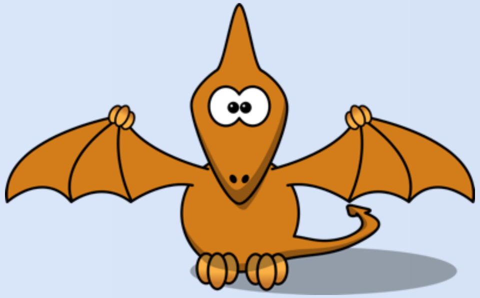
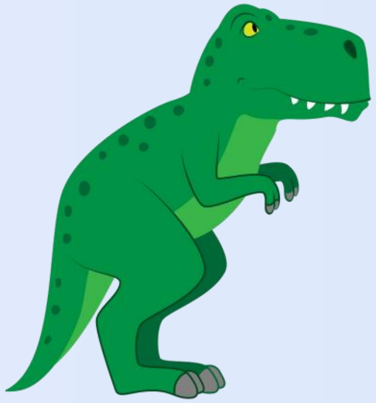


Chapter 6.

Synchronization Tools



Operating System Concepts (10th Ed.)

6.1 Background

- Cooperating processes
  - can either *affect* or *be affected by* each other.
  - can *share a logical address space* or be allowed to *share data*.

  - However, *concurrent access* to *shared data*
    - may result in ***data inconsistency***.

  - Hence, we need to ensure
    - the *orderly execution* of cooperating processes
    - that share a logical address space to *maintain data consistency*.

6.1 Background

- The ***integrity of data*** shared by several processes (or threads)
  - *Concurrent* execution
    - a process may be *interrupted at any point* in its instruction stream.
    - the processing core may be assigned to another process.
  - *Parallel* execution
    - two or more instruction streams (representing different processes)
    - *execute simultaneously* on *separate* processing *cores*.

6.1 Background

- Consider an example of how this is happen:
  - Let us revisit the **producer-consumer problem**,
    - where two processes *share data* and are *running asynchronously*.
  - To count items in the buffer, add an integer variable `count`:
    - initialized to 0,
    - *incremented* every time we *add a new item* to the buffer,
    - *decremented* every time we *remove one item* from the buffer.

6.1 Background

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

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

```
while (true) {
     while (count == 0)
        ; /* do nothing */

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

     /* consume the item in next_consumed */
}
```

6.1 Background

- Data inconsistency:
  - Although two processes are correct *separately*,
    - they *may not function correctly* when executed *concurrently*.
  - Suppose that the value of `count` is currently 5,
    - the producer and consumer concurrently execute
    - two statements: `count++;` and `count–;`
  - Then, the value of the variable `count` may be 4, 5, or 6!
    - is it posssible? why?

6.1 Background

- 다음 프로그램의 출력값은?

```
#include <stdio.h>
#include <pthread.h>

int sum;

void *run(void *param)
{
    int i;
    for (i = 0; i < 10000; i++)
        sum++;
    pthread_exit(0);
}
```

```
int main()
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, run, NULL);
    pthread_create(&tid2, NULL, run, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d\n", sum);
}
```

6.1 Background

- 다음 프로그램의 출력값은?

```c
int sum;

void *run1(void *param)
{
    int i;
    for (i = 0; i < 10000; i++)
        sum++;
    pthread_exit(0);
}

void *run2(void *param)
{
    int i;
    for (i = 0; i < 10000; i++)
        sum--;
    pthread_exit(0);
}
```

```c
int main()
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, run1, NULL);
    pthread_create(&tid2, NULL, run2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d\n", sum);
}
```

6.1 Background

- How these results can happen?
  - Note that two statements "`count++`" and "`count--`"
    - may be implemented in *machine language* as follows:

$$\begin{aligned} &register_1 = \texttt{count} \\ &register_1 = register_1 + 1 \\ &\texttt{count} = register_1 \end{aligned} \qquad\qquad \begin{aligned} &register_2 = \texttt{count} \\ &register_2 = register_2 - 1 \\ &\texttt{count} = register_2 \end{aligned}$$

  - Even though $register_1$ and $register_2$ may be the same physical register,
    - the contents of these registers will be
    - ***saved*** and ***restored*** by the *interrupt handler* (or *scheduler*).

6.1 Background

- How these results can happen?
  - The concurrent execution of "`count++`" and "`count--`"
    - is equivalent to a sequential execution
    - in which the lower-level statements presented previously
    - are *interleaved* in some *arbitrary order*.

$$
\begin{array}{llll}
T_0: & \textit{producer} & \text{execute} & \textit{register}_1 = \texttt{count} & \{\textit{register}_1 = 5\} \\
T_1: & \textit{producer} & \text{execute} & \textit{register}_1 = \textit{register}_1 + 1 & \{\textit{register}_1 = 6\} \\
T_2: & \textit{consumer} & \text{execute} & \textit{register}_2 = \texttt{count} & \{\textit{register}_2 = 5\} \\
T_3: & \textit{consumer} & \text{execute} & \textit{register}_2 = \textit{register}_2 - 1 & \{\textit{register}_2 = 4\} \\
T_4: & \textit{producer} & \text{execute} & \texttt{count} = \textit{register}_1 & \{\textit{count} = 6\} \\
T_5: & \textit{consumer} & \text{execute} & \texttt{count} = \textit{register}_2 & \{\textit{count} = 4\}
\end{array}
$$

- **Race Condition**:
  - A situation
    - where *several processes* (or *threads*)
    - access and manipulate the *same* (or *shared*) *data concurrently*
    - and the outcome of the execution
    - *depends on* the *particular order* in which the access takes place.

6.1 Background

- To guard against the *race condition*,
  - We need to ensure that
    - *only one process at a time* can manipulate the shared data (e.g. the variable `count`).
  - To make such a guarantee,
    - we require that the processes are *synchronized* in some way.
    - to say, *process* (or *thread*) *synchronization*.

6.1 Background

- Race Condition in Java Threads:

```
public class RaceCondition1 {

    public static void main(String[] args) throws Exception {
        RunnableOne run1 = new RunnableOne();
        RunnableOne run2 = new RunnableOne();
        Thread t1 = new Thread(run1);
        Thread t2 = new Thread(run2);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println("Result: " + run1.count + ", " + run2.count);
    }
}
```

```java
class RunnableOne implements Runnable {

    int count = 0;

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++)
            count++;
    }
}
```

6.1 Background

```java
class RunnableTwo implements Runnable {

    static int count = 0;

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++)
            count++;
    }
}
```

```java
public class RaceCondition2 {

    public static void main(String[] args) throws Exception {
        RunnableTwo run1 = new RunnableTwo();
        RunnableTwo run2 = new RunnableTwo();
        Thread t1 = new Thread(run1);
        Thread t2 = new Thread(run2);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println("Result: " + RunnableTwo.count);
    }
}
```

Exercises:

6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Exercises:

6.7 The pseudocode of Figure 6.15 illustrates the basic `push()` and `pop()` operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

```
push(item) {
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
}
```

```
pop() {
   if (!is_empty()) {
      top--;
      return stack[top];
   }
   else
      ERROR
}
```

```
is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

6.2 The Critical Section Problem

The Critical Section Problem:

- Consider a system consisting of n processes $\{P_0, P_1, \cdots, P_{n-1}\}$.
  - Each process has *a segment of code*, called a ***critical section***.
  - in which the process may be *accessing* – and *updating* – *data*
  - that is *shared with* at least one *other process*.

- The important feature of the system is that,
  - when *one process* is *executing* in its *critical section*,
  - *no other process* is *allowed to execute* in its critical section.

6.2 The Critical Section Problem

- The **critical-section problem**:
  - No two processes are executing in their critical sections at the same time.
  - To design a protocol that
    - the processes can use to ***synchronize*** their activity
    - so as to *cooperatively share data*.

6.2 The Critical Section Problem

- Sections of codes:
  - The ***entry-section***: the section of code
    - to *request permission* to *enter* its critical section.
  - The *critical-section* follows the entry-section.
  - The ***exit-section*** follows the critical-section.
  - The *remainder-section* is the section of remaining code.

6.2 The Critical Section Problem


```
while (true) {

        entry section

                critical section

        exit section

                remainder section

}
```

Figure 6.1 *General structure of a typical process.*

6.2 The Critical Section Problem

- Three requirements for the solution:
  - ***Mutual Exclusion***:
    - If process P_i is executing in its critical section,
    - then no other processes can be executing in their critical section.
  - ***Progress***: (avoid ***deadlock***)
    - If no process is executing in its critical section and some processes wish to enter their critical section,
    - then the selection of next process will enter its critical section next *cannot be postponed indefinitely*.
  - ***Bounded Waiting***: (avoid ***starvation***)
    - A *bound* (or *limit*) 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.

6.2 The Critical Section Problem

- Example of race condition:

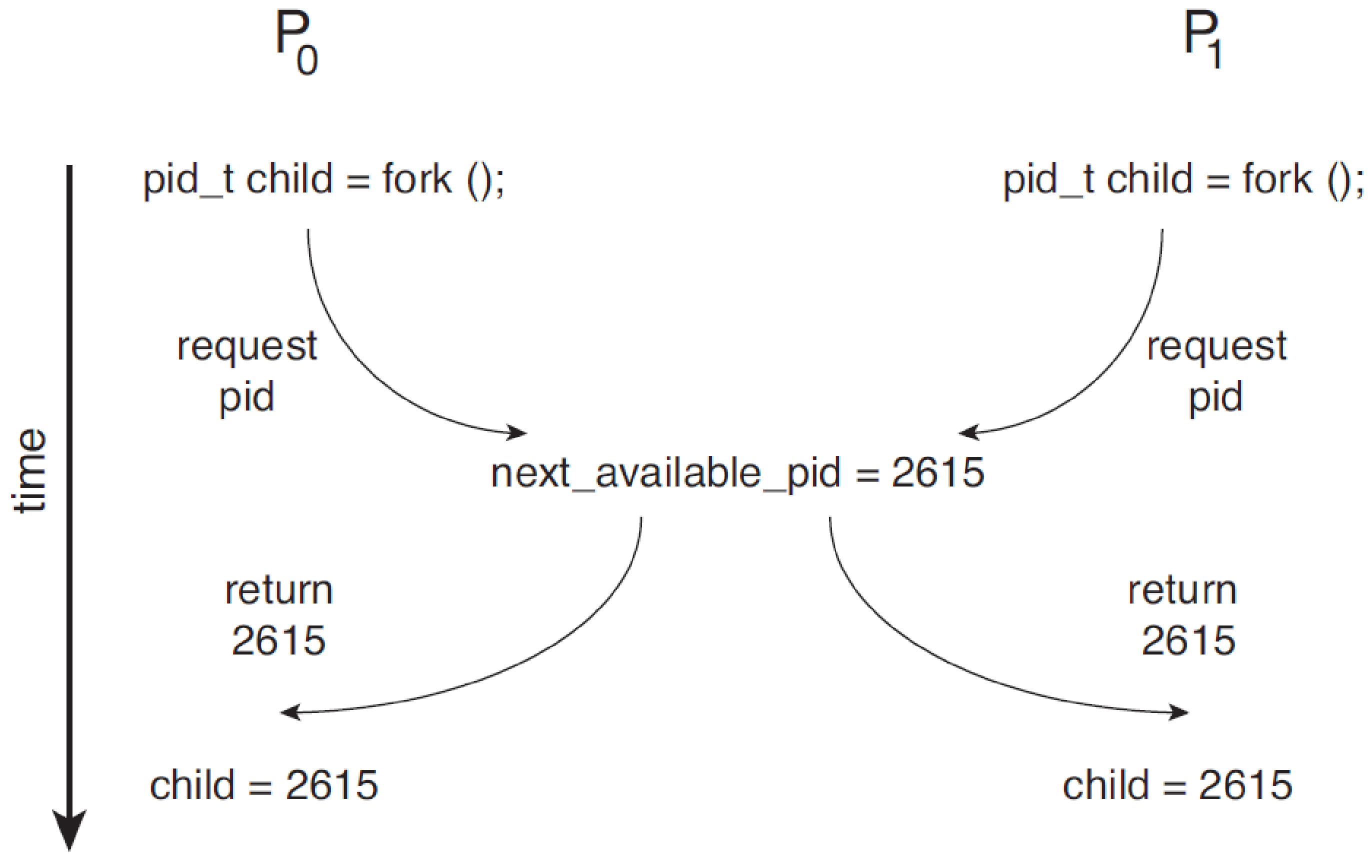


Figure 6.2 *Race condition when assigning a pid.*

6.2 The Critical Section Problem

- A simple solution in a single-core environment:
  - *Prevent interrupts* from occurring
    - while a shared variable was being modified.
  - We could be sure that
    - the current sequence of instructions
    - would be allowed to *execute in order without preemption*.
  - No other instructions would be run,
    - so no unexpected modifications could be made to the shared data.
  - Unfortunately, *not feasible* in a *multiprocessor* environment.

6.2 The Critical Section Problem

- Two general approaches:
  - ***preemptive*** kernels and ***non-preemptive*** kernels.

  - *Non-preemptive kernel*
    - a kernel-mode process will *run*
    - until it *exits* kernel mode, blocks, or voluntarily yields the CPU.
    - essentially *free from race conditions* on kernel data structures.
  - *Preemptive kernel*
    - *allows* a process to be *preempted* when it is running in kernel mode.
    - essentially *difficult* to design,
    - but *favorable* since it may be more *responsive*.

6.3 Peterson's Solution

- Software Solutions to the Critical-Section Problem:
  - *Dekker's* Algorithm:
    - for *two* processes (refer to Exercise 6.13)
  - *Eisenberg and McGuire's* Algorithm:
    - for n processes with a lower bound on *waiting* of $n - 1$ turns (refer to Exercise 6.14)
  - ***Peterson's*** Algorithm:
    - a classic software solution to the critical-section problem.
    - no guarantees that Peterson's solution will work correctly,
    - since modern computers perform basic machine-language instructions
    - such as ***load*** and ***store***.

6.3 Peterson's Solution

- Peterson's solution
  - restricted to two processes that alternate execution
    - between their *critical* sections and *remainder* sections.

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

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

```
int turn;
boolean flag[2];
```

Figure 6.3 *The structure of process P_i in Peterson's solution.*

6.3 Peterson's Solution

- A simple implementation of Peterson's solution:

```c
#include <stdio.h>
#include <pthread.h>

#define true 1
#define false 0

int sum = 0;

int turn;
int flag[2];
```

```c
int main()
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, producer, NULL);
    pthread_create(&tid2, NULL, consumer, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("sum = %d\n", sum);
}
```

6.3 Peterson's Solution

```
void *producer(void *param)
{
    int k;
    for (k = 0; k < 10000; k++) {
        /* entry section */
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1)
            ;

        /* critical section */
        sum++;

        /* exit section */
        flag[0] = false;

        /* remainder section */
    }
    pthread_exit(0);
}
```

```
void *consumer(void *param)
{
    int k;
    for (k = 0; k < 10000; k++) {
        /* entry section */
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn == 0)
            ;

        /* critical section */
        sum--;

        /* exit section */
        flag[1] = false;

        /* remainder section */
    }
    pthread_exit(0);
}
```

6.3 Peterson's Solution

- What happen?
  - There are *no guarantees* that
    - Peterson's solution *will work correctly*,
    - if the architecture perform basic machine-language instructions,
    - such as **load** and **store**.
  - However, Peterson's solution provides
    - a good algorithmic description of solving the CSP.
    - illustrates some of the complexities involved in
    - the requirements of *mutual exclusion*, *progress*, and *bounded waiting*.

6.3 Peterson's Solution

- Peterson's solution is *provably correct*.
  - **Mutual exclusion** is preserved.
    - Note that each P_i enters its critical section,
    - only if either `flag[j]==false` or `turn==i`.
  - The *progress* requirement is satisfied. (**No deadlock**)
  - The *bounded-waiting* requirement is met. (**No starvation**)

6.4 Hardware Support for Synchronization

- Hardware-based Solutions
  - Hardware instructions that provide
    - support for solving the critical-section problem.
    - can be used *directly* as *synchronization tools*,
    - can be used to form the foundation of *more abstract mechanisms*.
  - Three primitive operations
    - ***memory barriers or fences***
    - ***hardware instructions***
    - ***atomic variables***

6.4 Hardware Support for Synchronization

- **Atomicity**:
  - An *atomic operation* is *one uninterruptible unit* of operation.
  - Modern computer systems provide *special hardware instructions*
    - i.e., *atomic instructions*
    - that allow us either to *test and modify* the content of a word
    - or to *test and swap* the contents of two words

- Two types of conceptual atomic instructions:
  - `test_and_set()` and `compare_and_swap()`

6.4 Hardware Support for Synchronization

- The test_and_set() instruction:

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 *The definition of the atomic* `test_and_set()` *instruction.*

6.4 Hardware Support for Synchronization

- A global Boolean variable `lock`
  - is declared and initialized to `false`.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 6.6 *Mutual-exclusion implementation with* `test_and_set()`.

6.4 Hardware Support for Synchronization

- The `compare_and_swap()` instruction:

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7 *The definition of the atomic* `compare_and_swap()` *instruction.*

6.4 Hardware Support for Synchronization

- A global Boolean variable `lock`
  - is declared and initialized to `0`.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Figure 6.8 *Mutual-exclusion with the* `compare_and_swap()` *instruction.*

6.4 Hardware Support for Synchronization

Atomic Variable

- Typically, the compare_and_swap() instruction
  - is used for construction other tools such as an *atomic variable*.
- An ***atomic variable*** provides
  - *atomic operations* on *basic data types* such as integers and Booleans.
  - can be used to ensure *mutual exclusion* in situations
  - where there may be a *single variable* with *race condition*.

6.4 Hardware Support for Synchronization

- Java implementation of Peterson's solution:

```
public class Peterson1 {

    static int count = 0;

    static int turn = 0;
    static boolean[] flag = new boolean[2];

    public static void main(String[] args) throws Exception {
        Thread t1 = new Thread(new Producer());
        Thread t2 = new Thread(new Consumer());
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(Peterson1.count);
    }
}
```

```
static class Producer implements Runnable {
    @Override
    public void run() {
        for (int k = 0; k < 10000; k++) {
            /* entry section */
            flag[0] = true;
            turn = 1;
            while (flag[1] && turn == 1)
                ;

            /* critical section */
            count++;

            /* exit section */
            flag[0] = false;

            /* remainder section */
        }
    }
}
```

6.4 Hardware Support for Synchronization

```java
static class Consumer implements Runnable {
    @Override
    public void run() {
        for (int k = 0; k < 10000; k++) {
            /* entry section */
            flag[1] = true;
            turn = 0;
            while (flag[0] && turn == 0)
                ;

            /* critical section */
            count--;

            /* exit section */
            flag[1] = false;

            /* remainder section */
        }
    }
}
```

6.4 Hardware Support for Synchronization

```
import java.util.concurrent.atomic.AtomicBoolean;

public class Peterson2 {

    static int count = 0;

    static int turn = 0;
    static AtomicBoolean[] flag;
    static {
        flag = new AtomicBoolean[2];
        for (int i = 0; i < flag.length; i++)
            flag[i] = new AtomicBoolean();
    }


}
```

6.4 Hardware Support for Synchronization

```
static class Producer implements Runnable {
        @Override
        public void run() {
            for (int k = 0; k < 100000; k++) {
                /* entry section */
                flag[0].set(true);
                turn = 1;
                while (flag[1].get() && turn == 1)
                    ;

                /* critical section */
                count++;

                /* exit section */
                flag[0].set(false);

                /* remainder section */
            }
        }
    }
```

6.4 Hardware Support for Synchronization

```
static class Consumer implements Runnable {
        @Override
        public void run() {
            for (int k = 0; k < 100000; k++) {
                /* entry section */
                flag[1].set(true);
                turn = 0;
                while (flag[0].get() && turn == 0)
                    ;

                /* critical section */
                count--;

                /* exit section */
                flag[1].set(false);

                /* remainder section */
            }
        }
    }
```

Any Questions?