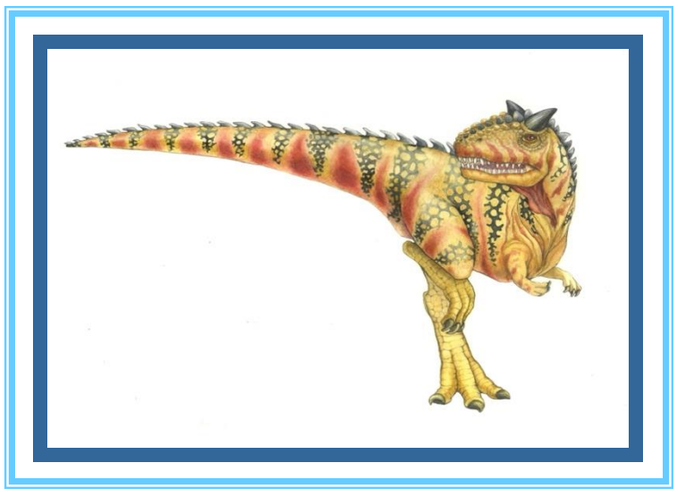
Chapter 6: Synchronization Tools



Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Synchronization Hardware → function provided by hardware!
- ① Mutex Locks → lock
- ② Semaphores → Mutex Locks
- ③ Condition variable → if else

Objectives

learn

? definition → just memorize!

- Describe the **critical-section problem** and illustrate the **race condition**
- Describe hardware solutions to the critical-section problem using **compare-and-swap operations**, and **atomic variables**
- Demonstrate how **mutex locks, semaphores**, and **condition variables** can be used to solve the critical section problem

← in Ch7!

? software to help u to do sth.

Solution need to meet 3 requirements!

- ? define. ? to judge

Background

process
thread
Scheduling

- Processes execute concurrently
  - Processes may be interrupted at any time, partially completing execution, due to a variety of reasons.

may be interrupt, preemptive!
Order of execution will be changed

Do things together!

- Concurrent access to any shared data may result in data inconsistency

corrupt in extreme case!

- Maintaining data consistency requires OS mechanisms to ensure the orderly execution of cooperating processes

↳ cause unexpected result! >.<
Order 不一致!

Illustration of the Problem

IPC problem → use ring buff!

- Think about the Producer-Consumer problem
  - ↳ consume data from the buff!
- An integer **`counter`** is used to keep track of the number of buffers occupied.
  - ↳ check whether full or not!
  - Initially, **`counter`** is set to 0
  - It is incremented each time by the producer after it produces an item and places in the buffer
  - It is decremented each time by the consumer after it consumes an item in the buffer.

corrected version

Producer-Consumer Problem

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

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Producer

// high level language, C code

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

Consumer

Everything is fine
but mind!!! This is high-level language
不是 instruction in machine code

CPU do scheduling

Race Condition

- `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}

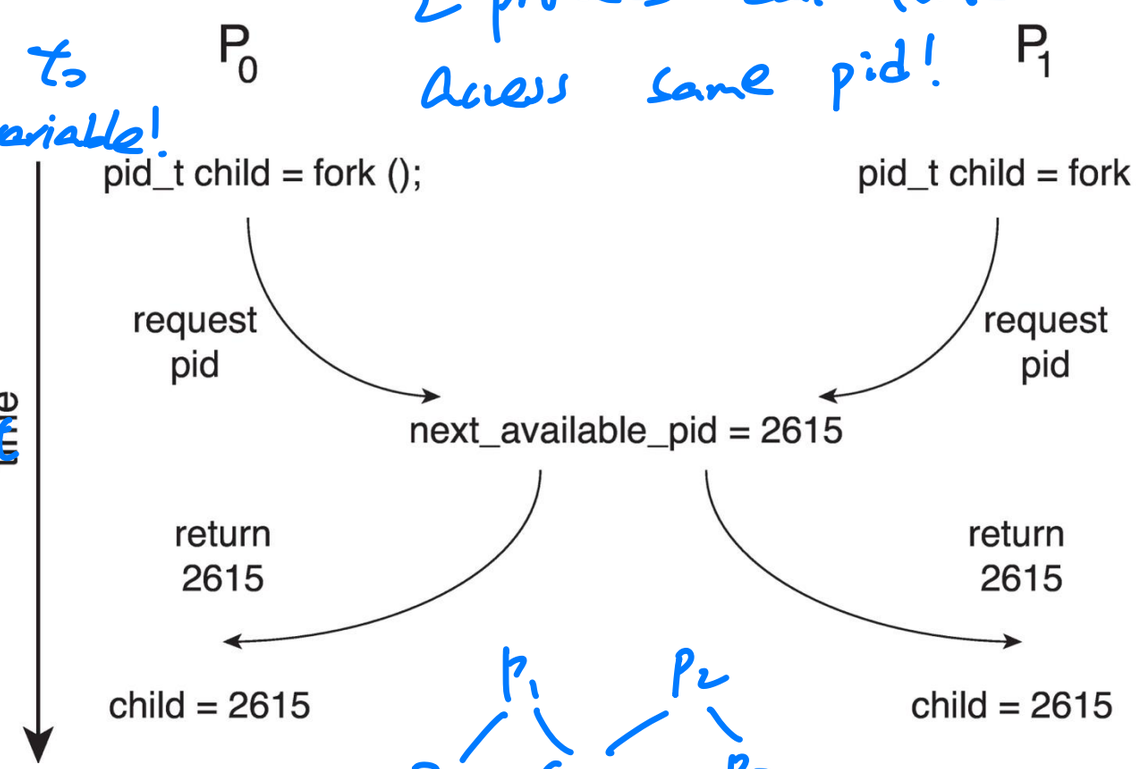
Race Condition

- Processes P_0 and P_1 are creating child processes using the **`fork()`** system call
- Race condition on kernel variable **`next_available_pid`** which represents the next available process identifier (**`pid`**)

→ Assign new pid

Definition:
– ≥ 2 process want to access the same variable!

2 process call fork - 要C!
access same pid!



Solution:

1. Disable interrupt
⇒ OS become low efficient

2. Make things mutually exclusive!

before pid++!!!
Cause serve problem!!!

same child!!!!! Race condition

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

Partially solve this problem!

Only one access in one time!
Only allow one thread to access one variable at one time!

⇒ May cause other problem

Critical Section Problem

Example: Counter!

- A **Race Condition** is an undesirable situation where several processes access or/and manipulate a shared data concurrently and the outcome of the executions depends on the particular order in which the accesses or executions take place - The results depend on the timing or the order of program execution. With some bad luck (i.e., context switches that occur at untimely points during execution), the result become **non-deterministic**

- Consider a system with *n* processes {P_0, P_1, … P_{n-1}}
- A process has a **Critical Section** segment of code (can be short or long), during which
  - A process or thread may be changing shared variables, updating a table, writing a file, etc.
  - We need to ensure when one process is in Critical Section, no other can be in its critical section
  - In a way, **mutual exclusion** and **critical section** imply the same thing
- Critical section problem is to design a protocol to solve this
  - Specifically, each process must request/ask permissions before entering a critical section in entry section, may follow critical section with exit section, then remainder section

Critical Section

- The general structure of process p_i is

```
do {
    entry section
        critical section
    exit section

    remainder section

} while (true);
```

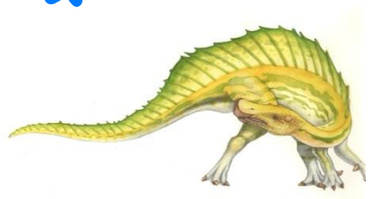
(locked room)

counter++ (其他不可access this)

can be implemented by other tools!

(key)

done can give the key to others.



Solution to Critical-Section Problem

Meet 3 requirements at the same time

1. **Mutual exclusion** - If process P_i is executing in its critical section, no other processes can be executing in their critical sections

Most easy requirement!

↳ only one process! Room with SINGLE key!

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, the selection of a process that will enter the critical section next *cannot* be postponed *indefinitely* – selection of one process entering

↳ waiting in entry part → cannot delay it! Waiting no one executing → let that process come in!

Very difficult to design!

3. **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 – any waiting process

- Assume that each process executes at a nonzero speed, and there is no assumption concerning relative speed of each individual process

Assume speed are the same!

Process try to enter the session → cannot let it try so many times!

process one

↓

random assign the "key" to the early session!

↓

waiting trial is bounded → for fairness!

very unlucky → cannot enter the room

↓

unfairness issue! Starvation!

Equal opportunity

Critical-Section Problem in Kernel

- Kernel code - the code the operating system is running, is also subject to several possible race conditions
  - A kernel data structure that maintains a list of all open files can be updated by multiple kernel processes, i.e., two processes were to open files simultaneously
  - Other kernel data structures such as the one maintaining memory allocation, process lists or process table (keep track of processes in a system), interrupt handling etc.
- Two general approaches are used to handle critical sections in operating system, depending on whether the kernel is preemptive or non-preemptive
  - Preemptive – allows preemption of process when running in the kernel mode, not free from the *race condition*, and becomes increasingly more difficult in SMP architectures.
  - Non-preemptive – runs until exiting the kernel mode, blocks, or voluntarily yields CPU. This is essentially free of race conditions in the kernel mode, possibly used in single-processor systems

Synchronization Tools

processor / CPUs can provide the solutions

- Many systems provide hardware support for implementing the critical section code. On uniprocessor systems – it could simply disable interrupts, currently running code would execute without being preempted or interrupted. But this is generally inefficient on multiprocessor or SMP systems
- Operating systems provide hardware and high level API support for critical section code

high level

Programs	Share Programs
Hardware	Load/Store, Disable Interrupts, Test&Set, Compare&Swap
High level APIs	Locks, Semaphores

x use

in lectures

include atomic levels!

and variables

Synchronization Hardware

- Modern OS provides special atomic hardware instructions
  - **Atomic** = non-interruptible
  - This ensures the execution of atomic instruction can not be interrupted, thus, no race condition can occur
  - These also serve as building blocks for more sophisticated synchronization mechanisms

- There are two commonly used atomic hardware instructions, which can be used to construct more sophisticated synchronization tools
  - Test a memory word and set a value – **`Test_and_Set()`**
  - Swap contents of two memory words – **`Compare_and_Swap()`**

test_and_set Instruction

return original value

set the original to be true!

- Definition:

```
boolean test_and_set (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
}
```

set target to be true!

return bool (original value)

- 不可 interrupt!

不用 be interrupted

Solution using test_and_set()

?

- Shared Boolean variable **`lock`**, initialized to **`FALSE`**
- Solution:

insured by hardware

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

   /* critical section */

   lock = false;

   /* remainder section */

} while (true);
```

- Access mutually exclusive easily!

not satisfied the boundary!

most difficult

compare_and_swap Instruction

hardware!

- Definition:

called CAS

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

- same as expected true ⇒ *value = new value

Cannot be interrupted!

Solution using compare_and_swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

- Solution:

change test and set to this

```
do {
   while (compare and swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
   lock = 0;
   /* remainder section */
} while (true);
```

X ensure bounded condition.

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
     lock = false; /* no one is waiting, so release the lock */
   else
      waiting[j] = false; /* Unblock process j */
   /* remainder section */
} while (true);
```


Sketch Proof

- **Mutual-exclusion**: P_i enters its critical section only if either `waiting[i]==false` or `key==false`. The value of `key` can become `false` only if `test and set()` is executed. Only the first process to execute `test and set()` will find `key==false;` all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, thus maintaining the mutual-exclusion requirement.

ensure only one!

- **Progress**: since a process existing its critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

- **Bounded-waiting**: when a process leaves its critical section, it scans the array `waiting` in cyclic order {`i+1, i+2, …,n-1,0,1,…i-1`}. It designates the first process in this ordering that is in the entry section (`waiting[j]==true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within `n-1` turns.

one chance within n-1

Satisfied ALL conditions!!!

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other more sophisticated synchronization tools.
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and Booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption - **`increment(&sequence);`**
  - The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
   int temp;

   do {
         temp = *v;
   }
   while (temp !=
(compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks

- OS builds a number of software tools to solve the Critical Section problem
- The simplest tool that most OSes use is **mutex lock**
- To access the critical regions with it by first **`acquire()`** a lock then **`release()`** it afterwards – both atomic operations
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic** (non-interruptible)
  - Usually implemented via hardware atomic instructions

- But this solution requires **busy waiting**. This lock therefore called a **spinlock**
  - **Spinlock** wastes CPU cycles due to busy waiting, but it has one distinct advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlock is useful
  - Spinlocks are often used in multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true;
}

do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Solutions based on the idea of **lock** to protect critical section

- Operations are **atomic** (non-interruptible) – at most one thread acquires a lock at a time
- Lock before entering critical section for accessing share data
- Unlock upon departure from critical section after accessing shared data
- Wait if locked - all synchronization involves busy waiting, should "sleep" or "block" if waiting for a long time

Semaphore

- **Semaphore** ***S*** – non-negative integer variable, can be considered as a generalized lock
  - First defined by Dijkstra in late 1960s. It can behave similarly as mutex lock, but it has more sophisticated usage - the main synchronization primitive used in original UNIX
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**, where **`P()`** stands for "**`proberen`**" (to test) and **`V()`** stands for "**`verhogen`**" (to increment) in Dutch
- It is essential that semaphore operations are executed atomically, which guarantees that no more than one process can execute **`wait()`** and **`signal()`** operations on the same semaphore at the same time – serialization
- The semaphore can only be accessed via these two atomic operations except initialization

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
signal (S) {
    S++;
}
```

Semaphore Usage

useful tools!

- **Counting semaphore** – An integer value can range over an unrestricted domain
  - Counting semaphore can be used to control access to a given set of resources consisting of a finite number of instances; semaphore value is initialized to the number of resources available
- **Binary semaphore** – integer value can range only between `0` and `1`
  - This can behave like mutex locks, can also be used in different ways
- This can also be used to solve various synchronization problems
- Consider P_1 and P_2 that shares a common semaphore **`synch`**, initialized to `0`; it ensures that P_1 process executes S_1 before P_2 process executes S_2

=> still race condition!

only initidize to be 1!!!

help to solve the critical solution whether can successfully depends on scheduling!

synch

to ensure critical session!

have no flexibility!

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

in parallel!

init. to 0

Always S_1 before S_2!

Semaphore + mutplex!!!

Do prove for 3 theatre!!!

Semaphore Implementation with no Busy waiting

- Each semaphore is associated with a waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record on the queue
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it on the ready queue
- Semaphore values may become negative, whereas this value can never be negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes currently waiting on the semaphore.

- Only running state can consume CPU cycle!

-1: only one thread in the waiting queue.

-x: x threads in the waiting queue

Semaphore Implementation with no Busy waiting (Cont.)

less be used in modern OS.

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

block

wake up one by one!

Noticing that usually we in classic!

- Increment and decrement are done before checking the semaphore value, unlike the busy waiting implementation
- The **`block()`** operation suspends the process that invokes it.
- The **`wakeup(P)`** operation resumes the execution of a suspended process P.

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes (to be examined in Chapter 8)
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
…	…
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- Consider if P_0 executes **`wait(S)`** and P_1 executes **`wait(Q)`**. When P_0 executes **`wait(Q)`**, it must wait until P_1 executes **`signal(Q)`**, However, P_1 is waiting until P_0 executes **`signal(S)`**. Since these **`signal()`** operations will never be executed, P_0 and P_1 are **deadlocked**. This is extremely difficult to debug
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue, in which it is suspended. For instance, if we remove processes from the queue associated with a semaphore using LIFO (last-in, first-out) order or based on certain priorities.

End of Chapter 6