Operating System Concepts
Ch. 6: Process Synchronization

Silberschatz, Galvin & Gagne

Universiteit Leiden
The Netherlands

Motivation

- When processes share data, writes to this data must be coordinated.
- In particular when pre-emptive scheduling is used.
  - A process can be interrupted at any time.
  - Also when it is in the midst of manipulating a shared data structure, the data structure may be left in an inconsistent state and may be accessed by the other process.
- This is also the case for kernel data structures that are used to implement system calls.
  - E.g. system-wide open file table.

Example Race Condition

- Consider the following (classical) example. The variables `buffer`, `in`, `out` and `counter` are shared.

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

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

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

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

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

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

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Example Race Condition (2)

- We must realize ourselves that counter increments typically consist of multiple machine instructions:

```
load r1,$counter
r1 = r1 + 1
store r1,$counter
```

- These are not guaranteed to be executed one after the other, or in a single go. It is not an *atomic* sequence of instructions.
- When does this become a problem?

Example Race Condition (3)

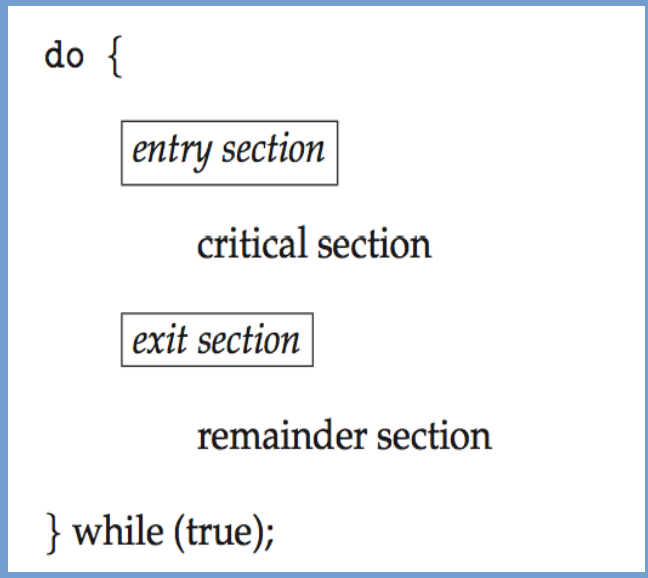
- We now execute the producer (P) and consumer (C) processes. Recall that they may be pre-empted!
- The value of `counter` is initially 5.
- Example sequence of instructions:

P: load r1,$counter	(value of r1 in P becomes 5)
P: r1 = r1 + 1	(value of r1 in P becomes 6)
C: load r1,$counter	(value of r1 in C becomes 5)
C: r1 = r1 – 1	(value of r1 in C becomes 4)
P: store r1,$counter	(value of counter becomes 6)
C: store r1,$counter	(value of counter becomes 4)

- When first P is fully executed, followed by C, then the value of counter would be 5!!!!
- *Race Condition*: both processes are “racing”, the last value written remains in memory. This is thus dependent on instruction order and time.

Critical Sections

- These problems can be solved using *critical sections*.
  - Each process defines a critical section.
  - Only ***one process*** may be in its critical section at any time.
  - Manipulate shared resources (memory, opened files) while within the critical section.
- The entry section contains code to decide if/when a process may enter its critical section.
  - Permission has to be asked, or turn awaited.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Critical Sections (2)

- Classical solution of the problem: Peterson's solution.
  - A solution for two processes and assumes load/store machine instructions are atomic.
  - `int turn` and `boolean flag[2]` are shared variables.
- Processes change turn.
- `flag` is used to indicate process wants to enter its critical section

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
          critical section
    flag[i] = false;
          remainder section
} while (true);
```

Source: Silberschatz et al., Operating System Concepts, 9th Edition

Critical Sections (3)

- Solutions of the critical sections problem must adhere to the following properties:
  - *Mutual Exclusion*: “if a process is executing its critical section, none of the other processes may be in their critical section.”
  - *Progress*: “if no process is executing its critical section and there exist some processes that have indicated they want to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.”

    Loosely: we must regularly select a process that may enter the critical section next.
  - *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.”

    Loosely: others must get a turn in due time.
- Refer to the textbook for more exact and detailed definitions

Modern Solutions

- Peterson's solution cannot always be implemented on modern hardware.
  - Think about pipelining, speculation, caching, etc., etc.
- To be able to implement synchronization, we need some guarantees from the hardware.
- The hardware therefore commonly implements *atomic instructions* that can be used to implement synchronization primitives.
  - Atomic instructions are guaranteed to be executed as a whole and cannot be interrupted.
  - Hardware also guarantees that sequences of atomic instructions emitted by different cores will be serialized.
- For old single processor systems it suffices to turn off interrupts, which will disable pre-emption.
  - Not always wanted and does not scale to multiprocessor systems.

Modern Solutions (2)

- Modern solutions that are implemented using hardware support are always centered around the concept of ***locking***.
  - Entering a critical section: *acquire* the lock.
  - Leaving critical section: *release* the lock.
- Locking is implemented in different ways, depending on the support provided by the hardware.

```
do {
        acquire lock

                critical section

        release lock

                remainder section

} while (TRUE);
```

Source: Silberschatz et al., Operating System Concepts, 9th Edition

Locking with test-and-set

- test-and-set instruction sets value behind given pointer to TRUE and returns the original value.
  - The pseudocode given is implemented and executed as a single, atomic instruction.
- To implement locking, the test-and-set instruction is used to manipulate a shared boolean variable.
- Example: BTS (bit test and set) instruction on x86, prefixed with lock.

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

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

  /* critical section */
  lock = false;
  /* remainder section */
} while (true);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Locking with compare-and-swap

- Returns original value behind pointer target, if this equals `expected`, value is overwritten with `new_value`.
- Slightly different from test-and-set.
- Not difficult to adapt our locking function.

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

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

  /* critical section */
  lock = 0;
  /* remainder section */
} while (true);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Bounded Waiting

- We can also construct more complicated primitives using hardware support.
- Such as bounded waiting mutual exclusion shown on the right.
- Note that this code gives every process a turn, so complies with the bounded waiting criterion.

```
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;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

User-space Locking

- There are a number of problems with the solutions that we have discussed:
  - They are platform-dependent, as they rely on specific machine instructions.
  - They are sometimes complicated and tricky to get right (depends on platform).
  - The machine instructions may not always be accessible by user-space processes.
- To solve this OS kernels often implement locks that can be used by application programmers e.g. through system calls.
  - Well-known is the mutex lock (MUTual EXclusion).
  - And its further generalization, the semaphore.

Mutex Locks

- Mutex locks consist of two calls: `acquire()` and `release()`.
- The locks are implemented within the OS kernel, usually like the locking functions we have just seen.
  - Note that this involves continuously running the loop until the original value is what we expected.
  - This is called *busy waiting*, the program is not making progress (waiting), but is keeping the CPU busy spinning the loop.
  - Mutex locks that are implemented this way are also often called *spin locks*.

Mutex Locks (2)

- Implementation of calls (compare with e.g. test-and-set example):

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

```
release()
{
  available = true;
}
```

- Example usage:

```
do {
   acquire();
   /* critical section */
   release();

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

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Semaphores

- Synchronization primitive devised by Edsger Dijkstra.
- We have two atomic operations that may operate on an integer variable S:
  - wait() and signal()
  - Dijkstra originally wrote P and V, probably from “proberen” and “verhogen”.

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

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

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Semaphores (2)

- Important consideration for the implementation: only one process may execute wait() or signal() at the same time.
  - These implementations must be placed in critical sections.
  - Problem: the busy wait loop in wait() will be made part of a critical section and we don't want that.
  - Time spent in critical sections must be as short as possible such that other processes also get a chance.
- This problem is solved by putting a process to sleep instead of busy waiting. Other processes can now make progress.

Semaphores (3)

- When a process is put to sleep, it is put on a waiting queue (cf. waiting queues for I/O).
- Waking up means process is transferred from this waiting queue to the ready queue to await being scheduled again.

```
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);
   }
}
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Semaphores (4)

- Two “kinds” of semaphores are distinguished:
  - *Counting semaphore*: the integer variable may hold any value of a range of integers.
  - *Binary semaphore*: the integer variable is either one or zero (cf. mutex).
- Example: two processes A and B.
  - S1 in A must happen before S2 in B.
  - Initialize a semaphore S to zero.

```
A:
    S1;
    signal(S);

B:
    wait(S);
    S2;
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Deadlock and Starvation

- Getting concurrent code right is hard.
- Often occurring problem when you are not careful: deadlock.
  - *Deadlock*: two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes.
- Example: given two semaphores P and Q, initialized to 1.

```
A                B
wait(S);         wait(Q);
wait(Q);         wait(S);
...              ...
signal(S);       signal(Q);
signal(Q);       signal(S);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Deadlock and Starvation (2)

- In case of *starvation*, a process may never be removed from the waiting queue of a semaphore in which it is suspended.
  - Compare scheduling: process may never be removed from ready queue, because higher priority processes are always picked first.
- Systems which implement priority scheduling can also be struck by the *priority inversion problem*.
  - A high-priority process needs a lock held by a low-priority process.
  - This can, for example, be solved using a *priority-inheritance protocol*.

    Low-priority process may temporarily inherit priority of higher priority process in order to quickly complete the work and release the lock.

Classic Synchronization Problems

- Several classic (textbook) synchronization problems exist that act as good illustrations of how to use synchronization primitives.
- Study these yourself using pen and paper: only way to get your head around it.

- Bounded buffer problem
  - We have a buffer that can hold a total of *n* items maximum.
  - We declare three semaphores: `mutex` (initialized 1), `full` (initialized 0), `empty` (initialized n).

Bounded Buffer Solution

```
/* Producer process */
do {
   ...
   /* produce item in next_produced */
   ...
   wait(empty);
   wait(mutex);
   ...
   /* add next produced to buffer */
   ...
   signal(mutex);
   signal(full);
} while (true);
```

```
/* Consumer process */
do {
   wait(full);
   wait(mutex);
   ...
   /* remove item from buffer
      to next_consumed */
   ...
   signal(mutex);
   signal(empty);
   ...
   /* consume item in next consumed */
   ...
} while (true);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Readers-Writers Problem

- Another often recurring problem is that of controlling access to shared data where:
  - you want to allow readers to access the data concurrently,
  - you want to grant writers exclusive access (so no reader and writer may access shared data at the same time).
  - Where is this a problem? Consider for example database systems.
- The solution involves:
  - Binary semaphore `mutex` (initialized 1).
  - Binary semaphore `rw_mutex` (initialized 1).
  - Counting semaphore `read_count` (initialized 0).

Readers-Writers Problem (2)

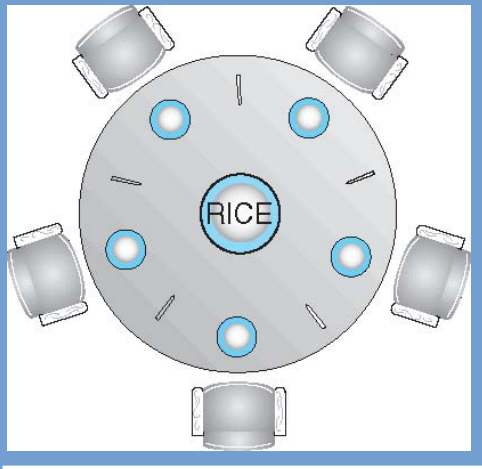
```
/* Writer process */
do {
  wait(rw_mutex);
  ...
  /* Exclusive access:
     writing is performed */
  ...
  signal(rw_mutex);
} while (true);
```

- Open question: give priority to readers or writers?
  - Different variations exist.
  - How to avoid starvation?
- Some kernels provide reader/writer locks.
  - Linux RCU: Read-Copy Update

```
/* Reader process */
do {
wait(mutex);
read_count++;
if (read_count == 1)
  wait(rw_mutex);
signal(mutex);
...
/* Shared access:
   reading is performed */
...
wait(mutex);
read_count--;
if (read_count == 0)
  signal(rw_mutex);
signal(mutex);
} while (true);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

Dining Philosophers



Source: Silberschatz et al., Operating System Concepts, 9th Edition

- Philosophers think or eat.
- When a philosopher wants to eat: need to pick up 2 chopsticks. Release when done.
  - May acquire one chopstick at a time, want to avoid deadlocks!!
- We have 5 philosophers and 5 chopsticks.
- Model using semaphores, bowl of rice is the shared data, we have an array of semaphores `chopstick` all initialized to 1.
- (There's also a variation that considers pasta and forks).

Dining Philosophers (2)

```
// Code for philosopher "i"
do {
  wait (chopstick[i] );
  wait (chopStick[ (i + 1) % 5] );
  //  eat
  signal (chopstick[i] );
  signal (chopstick[ (i + 1) % 5] );

  //  think

} while (true);
```

Example taken from: Silberschatz et al., Operating System Concepts, 9th Edition

- This solution may deadlock.
- Other solutions:
  - Allow maximum of 4 philosophers.
  - Pick up both sticks within critical section.

High-level solutions

- Mistakes with semaphores are quickly made:
  - signal/wait in wrong order,
  - wrong pairing of signal/wait,
  - accidental double wait,
  - etc.
- We can ease the life of programmers by providing high-level solutions.
  - The high-level solutions use the lower-level primitives in their implementations.
  - Or perhaps a compiler generates lower-level code that uses these primitives (like OpenMP does for threads).

High-level solutions (2)

- Monitors
  - Imagine a class with internal variables.
  - The internal variables may only be accessed from methods within that class.
  - Only one process may be inside the monitor at any time.
  - May add condition variables, which have wait/signal methods. wait suspends the calling process, signal wakes up a blocked process.
  - For example Java and C# provide implementations of monitors.

Alternative Approaches

- Also OpenMP includes support for critical sections.
  - You can mark a block of code as a critical section using a pragma.
  - The compiler generates the necessary mutexes/semaphores to correctly implement the critical section.
- Another approach is transactional memory.
  - Allows you to write blocks of code containing memory transactions that will be executed atomically.
  - Compare with database transactions (!).
  - Hardware & software implementations possible.
  - Modern Intel CPUs actually have special instructions to help implement this (TSX: Transactional Synchronization Extensions).

End of Chapter 6.

What about Chapter 7?

Chapter 7

- In the introduction of Chapter 7 in the textbook is written:

  "... operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs."

- So we will not discuss deadlock-prevention facilities in this class.