Western

UNIVERSITY · CANADA

Chapter 6 & 7 – Process Synchronization

Spring 2023

Western

Overview

- Race Conditions
- The Critical-Section Problem
- Peterson's Solution
- Hardware Solutions
- Mutex Locks
- Semaphores

Race Conditions

- Processes can run concurrently
- Processes can be interrupted at any time
  - Hardware interrupts
  - Software interrupts (traps)
    - Including timer interrupts for exceeding time quantum
- Two or more processes may read data that is in an inconsistent state

Race Conditions

- Example
  - We have seen that a producer and consumer can share a pointer to the next slot
  - `count++` could be implemented as
    - ```
      register1 = count
      register1 = register1 + 1
      count = register1
      ```
  - `count--` could be implemented as
    - ```
      register2 = count
      register2 = register2 - 1
      count = register2
      ```

Race Conditions

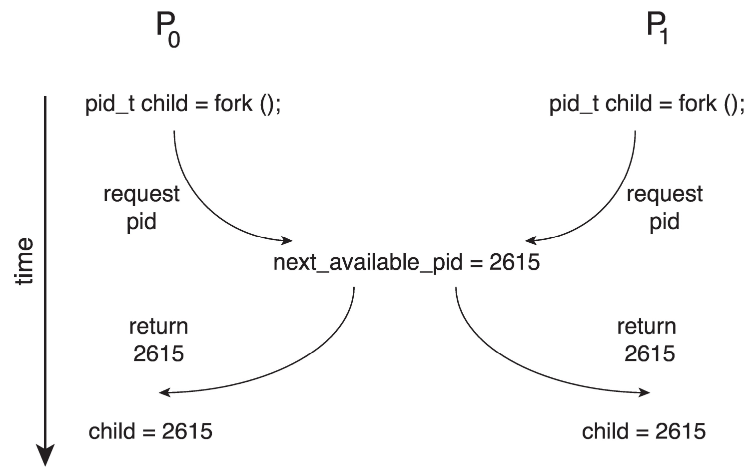
- Example
  - Consider the following order of operations when `count == 5`

Time	Who	Instruction	Value
T_0	Producer	register1 = count	register1 = 5
T_1	Producer	register1 = register1 + 1	register1 = 6
T_2	Consumer	register2 = count	register2 = 5
T_3	Consumer	register2 = register2 – 1	register2 = 4
T_4	Producer	count = register1	count = 6
T_5	Consumer	count = register2	count = 4

  - What is the value of `count`?

Race Conditions

- Example
  - When `fork()` is called, how does the operating system pick the next `pid`?



  - Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Race Conditions

- Example
  - Suppose a bank account has $1000
  - Suppose a request for authorization to withdraw $600 comes from two ATMs

ATM 1	ATM 2
Read balance: $1000	
Authorize withdrawal of $600	
Context Switch	Read balance: $1000
	Authorize withdrawal of $600
Update balance to -$600 (Balance is now $400)	*Context Switch*
Context Switch	Update balance to -$600 (Balance is now -$200)
…	…

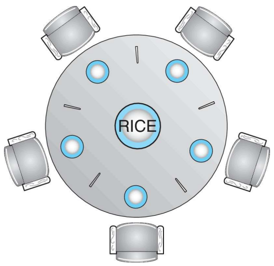
Race Conditions

- Example
    - What if the value of the balance was read and then used later in the update?

ATM 1	ATM 2
Read balance: $1000	
Authorize withdrawal of $600	
Context Switch	Read balance: $1000
	Authorize withdrawal of $600
Update balance to $1000-$600 (Balance is now $400)	*Context Switch*
Context Switch	Update balance to $1000-$600 (Balance is now $400)
…	…

Race Conditions

- Example: Multiple processes require multiple resources
  - The "Dining Philosopher's Problem"
  - Five philosophers sit at a table with 5 chopsticks
  - A philosopher will either think or eat
    - If the philosopher thinks, the philosopher leaves two chopsticks available
    - If the philosopher eats, the philosopher requires two chopsticks
  - High potential for deadlock and starvation (literally!)



The Critical-Section Problem

- A critical section could be any section of code that uses shared data
  - Process may be
    - changing common variables
    - updating tables
    - writing files
    - etc.
  - Read-only data can be safely shared (e.g. a lookup table, opening a file for reading only)

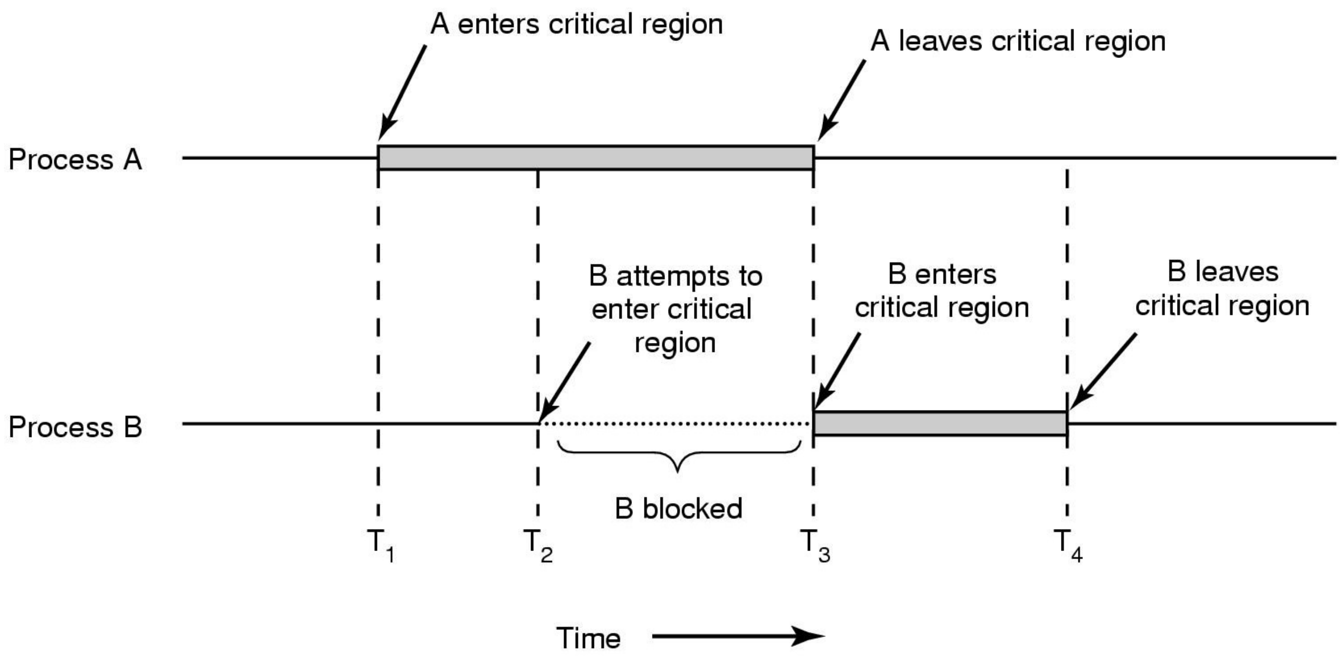
The Critical-Section Problem

- General structure of a process with a critical section:
  - Entry section – Request permission to enter the critical section
  - Critical section ← modify part
  - Exit section – Relinquish control of the critical section
  - Remainder section – (non-critical section parts)

```
while (true) {
      entry section
          critical section
      exit section
          remainder section
}
```

The Critical-Section Problem

- Suppose Process A and Process B both request access to their critical section



The Critical-Section Problem

- "Solutions" that don't work
  - Disallow interrupts (nonpreemptive). This is impractical.
  - Temporarily disable interrupts. Re-enable when you are done.
    - What if a process spends too much in the critical section? Processes could starve!
    - Too much control given to the developers/users
    - How do you deal with multiple CPUs?

The Critical-Section Problem

- "Solutions" that don't work
  - Use a shared flag to indicate which process can enter their critical section
  - E.g. Consider Process i and Process j. The only valid values for `turn` is `i` or `j`
  - The code for Process i:
    - ```
      while (true){
        while (turn == j);
        /* critical section */
        turn = j;
        /* remainder section */
      }
      ```
      ```

The Critical-Section Problem

- "Solutions" that don't work
  - Why a shared flag does not work
    - What if Process i is ready to enter the critical section again, but Process j hasn't entered its critical section yet?
    - `turn` is still set to `j` so Process i must block unnecessarily
    - The critical section is available to be used but Process i can't make progress
    - What if Process j never enters its critical section?
    - This solution allows entering critical sections by alternating only

The Critical-Section Problem

- A solution must satisfy three conditions
  - **Mutual exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - **Progress** – If no process is executing in 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. (The "no deadlock" rule)

    推迟 不确定的
  - **Bounded waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections (the "no starvation" rule)

    to avoid the case that a process may take too long time.

Peterson's Solution

- Developed in 1981 but no longer applicable on modern architectures
- Restricted to two, single-threaded processes on a single-core architecture
- Some modern architectures re-order instructions for efficiency
- However, it is still an illustrative solution

Peterson's Solution

- The two processes share two variables:
  - ```
    int turn;
    boolean flag[2];
    ```
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section.
  - `flag[i] = true` implies that process P_i is ready
    ```

Peterson's Solution

- The code for P_i

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

- If both processes were in their critical sections, then `flag[i] == flag[j] == true`

  - 互 斥
    Mutual Exclusion Achieved!
    ```

Peterson's Solution

- If P_i is blocking while P_j is in its critical section, as soon as P_j finishes, P_i can immediately continue
  - Progress Achieved
- P_i must wait only as long as P_j is in its critical section. The next process to enter its critical section is P_i
  - Bounded Waiting Achieved

Hardware Solutions

- Modern computer architectures provide tools to protect data
  - **Memory barrier instructions** – Instructions to force memory updates to all processors ensuring that all threads use the correct data
  - **Hardware instructions** – Instructions guaranteed to function atomically (uninterruptible)
  - **Atomic variables** – Variables with operations that use only atomic hardware instructions
- Applicable to kernel and assembly language programmers

互斥

Mutex Locks

- A "user-friendly" approach built by kernel developers for application developers to provide **mut**ual **ex**clusion
- **Mutex lock** – A Boolean variable indicating if a lock is available or not
  - Use **acquire()** to acquire the lock and enter the critical section
  - While the lock is held, no other process can hold the same lock
  - Use **release()** to release the lock and exit the critical section
- (Both acquire() and release() are implemented using atomic hardware instructions. So they can be trusted)

lol

Mutex Locks

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

- ```
  while (true){
    acquire()
    /* critical section */
    release()
    /* remainder section */
  }
  ```

Mutex Locks

- This solution requires **busy waiting** – Any process that wishes to enter its critical section must loop continuously while it waits. This wastes CPU cycles
- Therefore, this is known as a **spin-lock** (It spins while it waits for a lock to become available)
  - Spin-locks can be acceptable or even advantageous
    - A call to acquire() does not immediately force a context switch
    - If the spin-lock is short, a thread can spin on one core while another core completes the critical section

Semaphores

- A more sophisticated mutex lock

access to mutex

- **Binary semaphore** – integer value can range only between 0 and 1 (Same as a mutex lock)

non-negative value.

- **Counting semaphore** – integer value can range over an unrestricted domain

- Can be used to control access to a given resource consisting of a finite number of instances (not just one). For example, initialize the semaphore to N when N processes can be in the critical section simultaneously

i.e. process for N times.

Semaphores

- Set S to the number of resources available
  - Use **wait(S)** to obtain a resource if one is available
  - Use **signal(S)** to release a resource
- If the value of **S** reaches 0, then no resources are available, and the call **wait(S)** will block until one becomes available
- (Again, both wait() and signal() are implemented using atomic hardware instructions. So they can be trusted)

Semaphores

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

S=1: available S=0: currently used

wait(S) checks the availability of the semaphore

by testing S ≤ 0 or not

S>0: available, keep looping
  ```

Semaphores

- Solution to the critical section problem: Create a semaphore initialized to 1 (a Mutex)
  - ```
    wait(mutex);
    /* critical section */
    signal(mutex);
    ```
- Force synchronization: Suppose S_1 in P_1 must execute before S_2 in P_2. Initialize `synch` to 0
  - ```
    /* S1 */
    signal(synch);
    ```
    release resource here for S_2
  - ```
    wait(synch);
    /* S2 */
    ```
    ← waiting for the resource from S_1.
    ```

Semaphores

- Semaphores without busy waiting
  - In addition to an integer value, also maintain a waiting queue of processes
  - Instead of using a spin-lock, place the process into a waiting queue and put it to **sleep()**
  - When a resource is released, place a process in the waiting queue in the ready queue and use **wakeup()**

spin lock for short expected wait time

sleep/wake up for long waiting time

Semaphores

-
```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

Semaphores

- ```
  wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
         add this process to S->list;
         sleep();
     }
  }
  ```

- ```
  signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
          remove a process P from S->list;
          wakeup(P);
      }
  }
  ```

Semaphores

- Note this solution allows `value` to be negative
- The negative value can indicate the number of processes waiting on that semaphore

Semaphores

- Watch out for common mistakes with semaphores
  - Mixing up signal() and wait() (e.g. Using signal() before a critical section and wait() after a critical section)
  - Calling wait() twice (e.g. Using wait() before a critical section and wait() after a critical section)
  - Forgetting to call either signal() or wait()

Examples

- Binary semaphores for threads in Linux
  - `pthread_mutex_t lock`
  - `pthread_mutex_init(&lock)`
  - `pthread_mutex_lock(&lock)`
  - `pthread_mutex_unlock(&lock)`
  - `pthread_mutex_destroy(&lock);`

Examples

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

pthread_mutex_t lock;

void *thread_prints_msg(void *msg) {
 pthread_mutex_lock(&lock);  // ENTRY
 printf(" ENTERING CRITICAL SECTION...\n");
 printf("From thread_prints_msg: %s\n", (char *) msg);
 printf(" LEAVING CRITICAL SECTION...\n");
 pthread_mutex_unlock(&lock); // EXIT
 return 0;
}
```

Examples

```
int main() {
  pthread_t thread_1;
  if (pthread_mutex_init(&lock, NULL) != 0) {
      printf("\n mutex init has failed\n");
      return 1;
  }
  printf("From main: Going to create Thread...\n\n");
  pthread_create(&thread_1, NULL, thread_prints_msg, "Hello
World");
  pthread_join(thread_1, NULL);
  printf("\nthread terminates...\n");
  pthread_mutex_destroy(&lock);
  return 0;
}
```

Examples

- Counting semaphores for Linux
  - `#include <semaphore.h>`
  - `sem_t S`
  - `sem_init(sem_t *sem, int pshared, unsigned int value);`
  - `sem_wait(sem_t *sem);`
  - `sem_post(sem_t *sem);`
  - `sem_destroy(sem_t *mutex);`

Examples

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

sem_t mutex;

void* thread(void* arg){
  sem_wait(&mutex); //wait
  printf("\nEntered..\n");
  sleep(4); //critical section
  printf("\nJust Exiting...\n");
  sem_post(&mutex); //signal
}
```

Examples

- ```
  int main(){
    sem_init(&mutex, 0, 1);

    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    sem_destroy(&mutex);
    return 0;
  }
  ```
  ```

Western

UNIVERSITY · CANADA