CS330 - Operating Systems

Synchronization

11-09-2025

Interprocess Communication (IPC)

- Cooperating processes
  - Want to communicate with each other
  - Why?

Interprocess Communication (IPC)

- Cooperating processes
  - Want to communicate with each other
  - Allow information sharing, better performance, modularity
- But what about the restriction from the OS?
  - Prevent one process from violating other's space
  - Need some other mechanism

Producer-Consumer Problem

- Two distinct processes
  - Producer (writer) produces some data that the consumer (reader) has to consume
- How to allow this communication?

Producer-Consumer Problem

- Two distinct processes
  - Producer (writer) produces some data that the consumer (reader) has to consume

- How to allow this communication?
  - Shared memory
  - Message-passing

Producer-Consumer Problem

- Two distinct processes
  - Producer (writer) produces some data that the consumer (reader) has to consume
- Can we use a file?
  - Parent and child share file-descriptors
  - Use persistent storages like files to exchange data
  - Expensive if only transient communication is required
- Use memory-buffers
  - Finite (cannot assume unbounded buffers)
  - If producer tries to write when buffer full, it blocks
  - If consumer tries to read when buffer empty, it blocks

Threads with Shared Data

- May not give desired result
- May not give consistent result

```
static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

Uncontrolled Scheduling

- counter = counter + 1
  - May be translated as:

```
mov    0xc     %eax
add    $0x1    %eax
mov    %eax    0xc
```

Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
			100	0	50
	`mov 0xc, %eax`		105	**50**	50
	`add $0x1, %eax`		108	**51**	50
interrupt *save T1* *restore T2*			100	0	50
		`mov 0xc, %eax`	105	**50**	50
		`add $0x1, %eax`	108	**51**	50
		`mov %eax, 0xc`	113	51	**51**
interrupt *save T2* *restore T1*			108	51	51
	`mov %eax, 0xc`		113	51	**51**

Race Condition

Critical Section

Question: can this *panic*?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while (!pInitialized) ;

q = someFunction(p);
if (q != someFunction(p))
   panic
```

Why *Synchronize*?

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not ***atomic***

Atomicity

- All or none execution of an instruction
  - Execute as a unit
  - Guarantee that no other instruction will be run in between

```
mov 0xc, %eax
add $0x1, %eax
mov %eax, 0xc
```

- Synchronization primitives
  - Introduce some order

Definitions

- **Race condition**: output of a concurrent program depends on the order of operations between threads
- **Critical section**: piece of code that only one thread can execute at once
- **Mutual exclusion**: only one thread does a particular thing at a time
- **Synchronization**: using atomic operations to ensure cooperation between threads

Example : Buy milk

- **Ideal**:
  Someone buys, if needed **and** at most one person buys

Progress Safety

Example : Buy milk

- **Ideal**:
  Someone buys, if needed **and** at most one person buys

Progress | Safety

Time	Person A	Person B
T1	Out of milk	
T2	Leave for store	
T3	Arrive at store; buy milk	Out of milk
T4	Arrive home with milk	Leave for store
T5		Arrive at store; buy milk
T6		Arrive home with milk
		(*Too much milk*)

Example : Buy milk

- Leave a note

```
If no note
    If no milk
        Leave note
        Buy milk (go to store; buy; arrive at home)
        Remove note
```

Example : Buy milk

- Multiple threads

```
Leave noteA
If no noteB
    If no milk
        Buy milk
Remove noteA
```

```
Leave noteB
If no noteA
    If no milk
        Buy milk
Remove noteB
```

Example : Buy milk

- Multiple threads

```
Leave noteA
while noteB
    do nothing
If no milk
    Buy milk
Remove noteA
```

```
Leave noteB
If no noteA
    If no milk
        Buy milk

Remove noteB
```

At **while** and **if**, can guarantee that:
Safe for me to buy **or** other will buy

Locks

Image from codegym.cc

Locks

- Add locks (mutex) to code
    - Around critical sections
    - Guarantee atomic execution of critical section
- Acquire
    - wait until lock is free, then take it
- Release
    - release lock, waking up anyone waiting for it
- Correctness of Locks
    - At most one lock holder at a time (safety)
    - If no one holding, acquire gets lock (progress)
    - If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Example : Buy milk

- Using locks

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
// lock.acquire()
Pthread_mutex_lock(&lock); // wrapper; exits on failure
If no milk
    Buy milk
// lock.release()
Pthread_mutex_unlock(&lock);
```

Allow concurrent code to be much simpler

Locks for allocating/freeing memory

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate n memory
    heaplock.release();
    return p;
}
```

```
void free (char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

Rules for using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Goals when Building Locks

- Mutual exclusion
  - Prevent multiple threads from entering a critical section
- Fairness
  - Each thread has fair shot of acquiring lock
- Performance
  - Time overheads

Implementing Synchronization

- Interrupt-controls
  - Disable interrupts when in critical section

Implementing Synchronization

- Interrupt-controls
  - Disable interrupts when in critical section
  - Doesn't work on multiprocessors, requires privileged op., inefficient

Implementing Synchronization

- Interrupt-controls
- Peterson’s solution
  - Classic solution (works on early architectures) for two threads
  - Uses `load` and `store` (which were atomic in early arch.)

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

void init() {
      flag[0] = flag[1] = false;
      // whose turn is it? (thread 0 or 1)
      turn = 0;
}
```

```
void lock() {
      flag[i] = true;
      turn = j; // make it other thread’s turn
      while (flag[j] && (turn == j));
      // wait until it’s i’s turn
}

void unlock() {
    flag[i] = false;
}
```