IOWA STATE UNIVERSITY

Department of Electrical and Computer Engineering

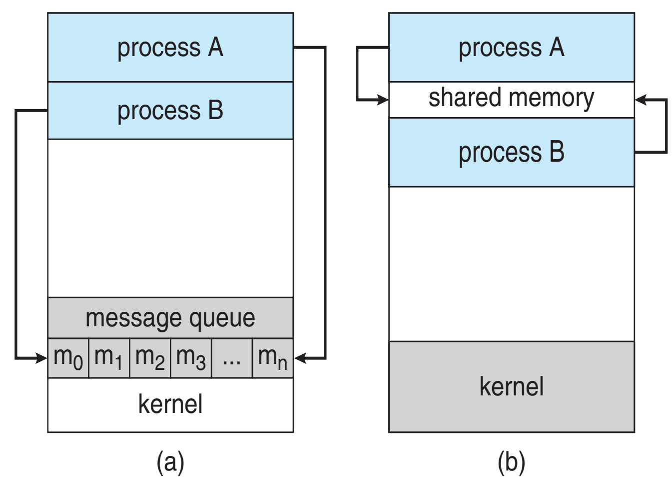
Lecture 12:
Inter-Process Communication (IPC) II

Agenda

- Recap
- Inter-Process Communication II
  - Solutions of Mutual Exclusion (cont.)

Recap

- IPC Concepts
  - Cooperating processes need **inter-process communication (IPC)** mechanism:
    - Communicate with each other
    - Ensure
      - (1) do not get in each other's way
      - (2) proper ordering when dependencies are present
  - Two basic methods of IPC
    - Message passing
    - Shared memory



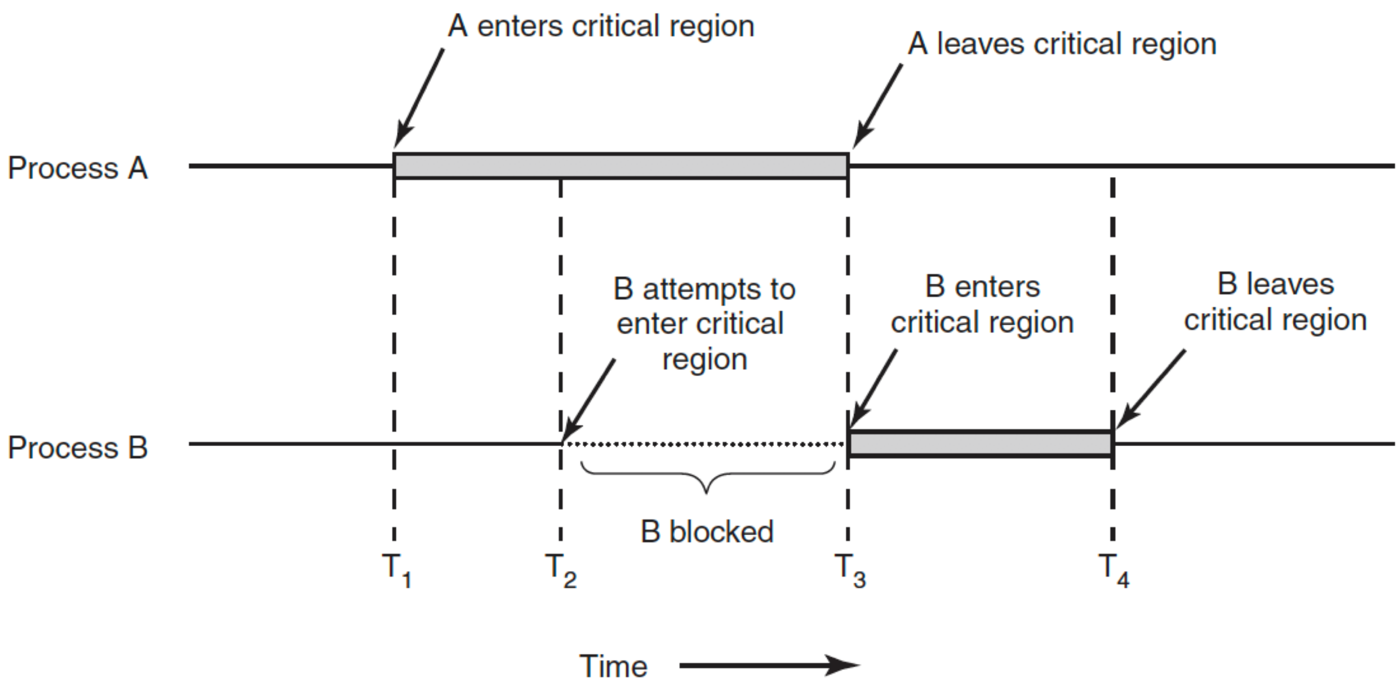
Recap

- Race Condition
  - E.g., two threads perform "counter = counter +1"
    - "counter" is a shared variable

OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
			100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
`interrupt`					
`save T1's state`					
`restore T2's state`			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
`interrupt`					
`save T2's state`					
`restore T1's state`			108	51	50
	`mov %eax, 0x8049a1c`		113	51	$\boxed{\textbf{51}}$

Recap

- Critical Region (Critical Section)
  - A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread
  - Need to support **mutual exclusion**



Recap

- Solutions of Mutual Exclusion
  - Disabling Interrupts
    - Disable interrupts when a process is in critical region
    - Cons
      - Not safe
      - Not applicable for multiple processors
  - Strict Alternation
    - Use an variable to keep track of whose turn it is to enter the critical region
    - Cons
      - **Busy waiting** (continuously testing a variable until some value appears) wastes CPU cycles
      - requires two processes strictly alternate in entering their critical regions

Agenda

- ~~Recap~~
- Inter-Process Communication II
  - Solutions of Mutual Exclusion (cont.)

Solutions of Mutual Exclusion

- Peterson’s Solution (1981)
  - General Structure

```
enter_region();
critical_region();
leave_region();
```

Solutions of Mutual Exclusion

- Peterson's Solution (1981)

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
     int other;                           /* number of the other process */

     other = 1 – process;                 /* the opposite of process */
     interested[process] = TRUE;          /* show that you are interested */
     turn = process;                      /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
     interested[process] = FALSE;         /* indicate departure from critical region */
}
```

Solutions of Mutual Exclusion

- Hardware Support
  - special instruction: TSL (Test and Set Lock)
    - TSL R, Lock
      - TSL = “Test and Set Lock”
      - R   = register,
      - Lock = memory location
    - reads the contents of the memory word Lock into register R, and then stores a nonzero value at the memory address Lock.
    - Atomic (“all or nothing”/indivisible)
      - reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished
      - The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done
        - different from disabling interrupt

Solutions of Mutual Exclusion

- Hardware Support
  - special instruction: TSL (Test and Set Lock)
    - Example

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was not zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Solutions of Mutual Exclusion

- Hardware Support
  - another instruction: XCHG (exchange)
    - exchanges the contents of a register and a memory location atomically

```
enter_region:
        MOVE REGISTER,#1              | put a 1 in the register
        XCHG REGISTER,LOCK            | swap the contents of the register and lock variable
        CMP REGISTER,#0               | was lock zero?
        JNE enter_region              | if it was non zero, lock was set, so loop
        RET                           | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                  | store a 0 in lock
        RET                           | return to caller
```

Summary of Solutions So Far

- Software solution
  - Disabling interrupts
    - Single processor only
    - Use for kernel
  - Strict alternation
    - Strict ordering
    - Busy waiting
  - Peterson's solution
    - Busy waiting
- Hardware solution
  - TSL/XCHG
    - Work on multiprocessors
    - Busy waiting

Better solution?

Better solution?

- Sleep and Wakeup
  - Yield to other thread/process if unable to lock
    - If cannot enter critical region, calls *sleep()* to give up CPU
    - wakes up another thread using *wakeup()*

Pthread Locks

- Provide mutual exclusion to a critical section
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Pthread Locks

- All locks must be **properly initialized.**
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Pthread Locks

- These two calls are also used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - `trylock`: return failure if the lock is already held
  - `timelock`: return after a timeout or after acquiring the lock

Agenda

- ~~Recap~~
- ~~Inter-Process Communication II~~
    - ~~Solutions of Mutual Exclusion (cont.)~~

Questions?

*acknowledgement: slides include content from “Modern Operating Systems” by A. Tanenbaum, “Operating Systems Concepts” by A. Silberschatz etc., “Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau etc., and anonymous pictures from internet.