

CS241 System Programming

Process Synchronization (1)

Klara Nahrstedt

Lecture 6

2/1/2006



Content

- Data Races
 - A simple game
- Critical region and mutual exclusion
- Mutual exclusion using busy waiting
 - Disabling Interrupts
 - Lock Variables
 - Strict Alternation
 - Peterson's solution
- Summary

Administrative

- Read : T:2.2 on Process Synchronization
- Read: R&R 13 and 14
- MP1 posted – deadline February 13

Review

- Process: a program in execution
- Threads: a light weight process

Inter-Process Communication (IPC)

- Communication
 - Pass information to each other
- Mutual exclusion & Synchronization
 - Proper sequencing
- The last one also applies to threads

A simple game

- Two volunteers
 - Producer: produce 1 card per iteration
 - Step1: increment the counter
 - Step2: put the card on the table
 - Consumer:
 - Step1: check the counter to see if it is zero
 - Step2a: if the counter is zero, go back to step1
 - Step2b: if the counter is nonzero, take a card from the table
 - Step3: decrement counter
- I am the OS
 - I decide who should go, who should stop

First Round

- Stop Producer before step2 and let Consumer go.
- What happens?
- Two volunteers
 - Producer: produce 1 card per iteration
 - Step1: increment the counter
 - Step2: put the card on the table
 - Consumer:
 - Step1: check the counter to see if it is zero
 - Step2a: if the counter is zero, go back to step1
 - Step2b: if the counter is nonzero, take a card from the table
 - Step3: decrement the counter

switch



Second Round

- Stop Producer before step2 and let Consumer go.
- What happens?
- Two volunteers
 - Producer: produce 1 card per iteration
 - Step1: put the card on the table
 - Step2: **increment the counter**
 - Consumer:
 - Step1: check the counter to see if it is zero
 - Step2a: if the counter is zero, go back to step1
 - Step2b: if the counter is nonzero, take a card from the table
 - Step3: **decrement the counter**

Data Races

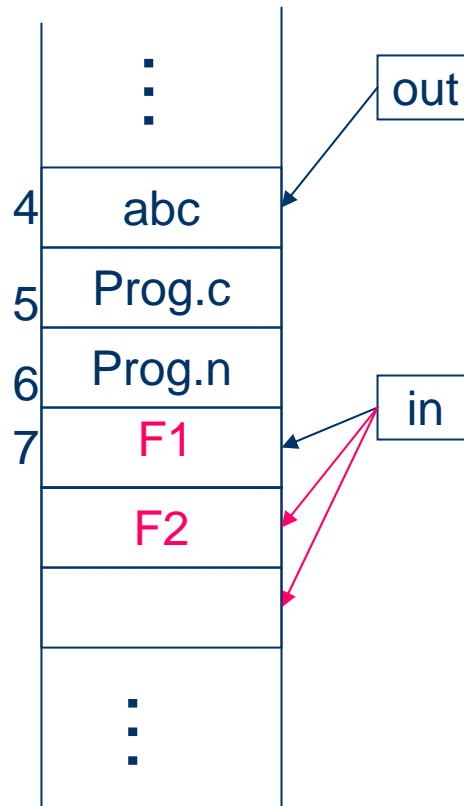
- Reason: data sharing
- Previous game: producer and consumer
 - Share the counter
 - Share the cards

Spooling Example: Correct

Process 1
int next_free;

- 1 next_free = in;
- 2 Stores F1 into next_free;
- 3 in=next_free+1

Shared memory



Process 2
int next_free;

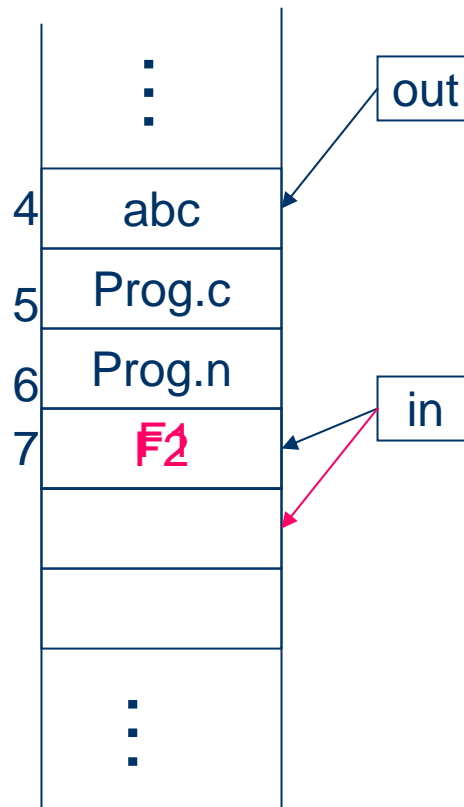
- 4 next_free = in
- 5 Stores F2 into next_free;
- 6 in=next_free+1

Spooling Example: Races

Process 1
int next_free;

- 1 next_free = in;
- 3 Stores F1 into next_free;
- 4 in=next_free+1

Shared memory



Process 2
int next_free;

- 2 next_free = in
/* value: 7 */
- 5 Stores F2 into next_free;
- 6 in=next_free+1

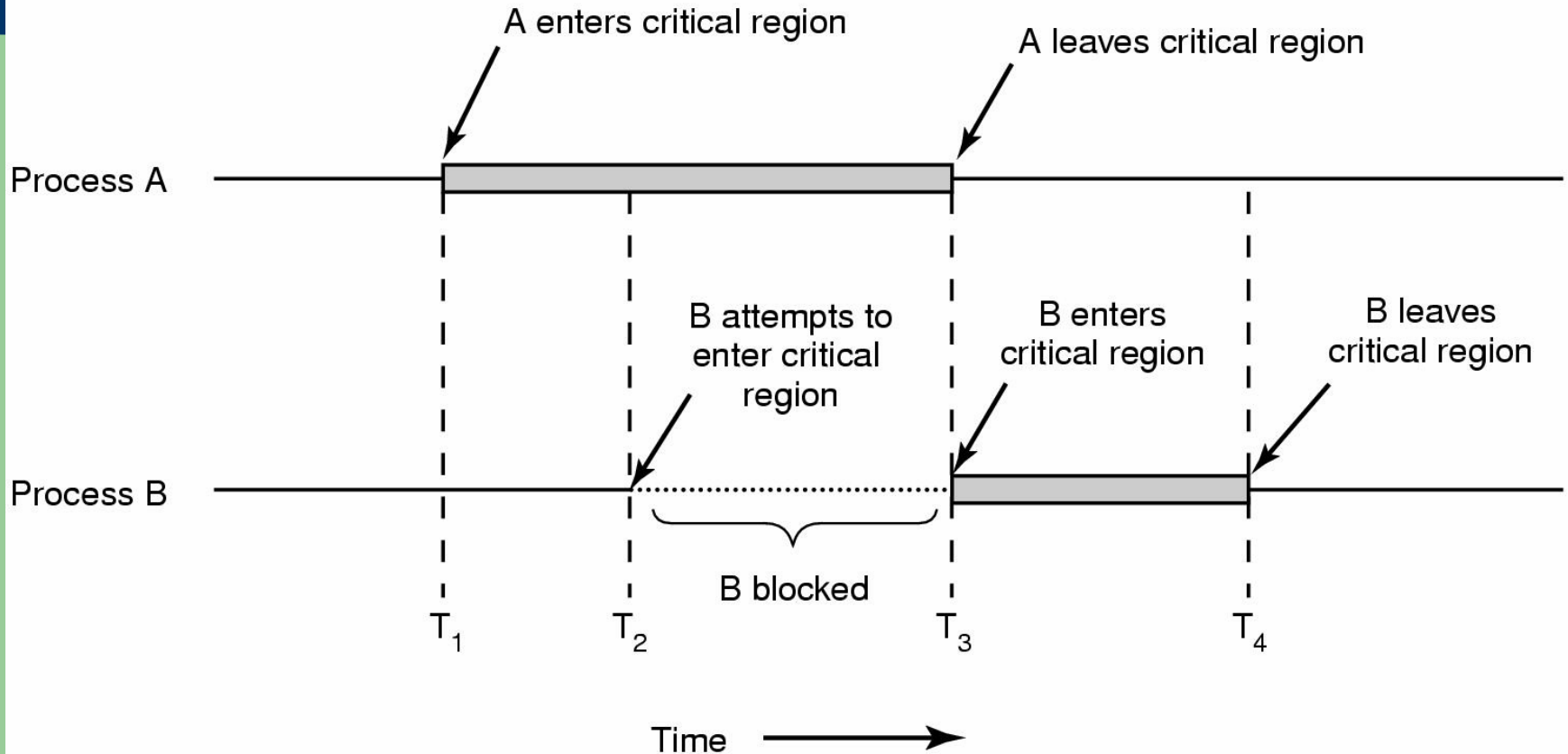
Critical Region (Critical Section)

```
Process {  
    while (true) {  
        ENTER CRITICAL SECTION  
        Access shared variables; // Critical Section;  
        LEAVE CRITICAL SECTION  
        Do other work  
    }  
}
```

Critical Region Requirement

- **Mutual Exclusion:** No other process must execute within the critical section while a process is in it.
- **Progress:** If no process is waiting in its critical section and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely.
- **Bounded Wait:** A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section.
- **Speed and Number of CPUs:** No assumption may be made about speeds or number of CPUs.

Critical Regions (2)



Mutual exclusion using critical regions

Mutual Exclusion With Busy Waiting

- Possible Solutions
 - Disabling Interrupts
 - Lock Variables
 - Strict Alternation
 - Peterson's solution
 - TSL

Disabling Interrupts

- How does it work?
 - Disable all interrupts just after entering a critical section and re-enable them just before leaving it.
- Why does it work?
 - With interrupts disabled, no clock interrupts can occur. (The CPU is only switched from one process to another as a result of clock or other interrupts, and with interrupts disabled, no switching can occur.)
- Problems:
 - What if the process forgets to enable the interrupts?
 - Multiprocessor? (disabling interrupts only affects one CPU)
- Only used inside OS

Lock Variables

```
While (lock);  
lock = 1;  
EnterCriticalSection;  
    access shared variable;  
LeaveCriticalSection;  
lock = 0;
```

Does the above code work?

Solution History

Approaches:

1. Turn Mutual Exclusion
2. Other Flag Mutual Exclusion
3. Two Flag Mutual Exclusion
4. Two Flag and Turn Mutual Exclusion

Turn Mutual Exclusion (Strict Alteration)

```
Process; /* For two processes */
{
  while (true)
  { while ( turn != my_process_id) { };
    Access shared variables; // Critical Section;
    turn = other_process_id;
    Do other work
  } }  What Properties does this satisfy?
```

mutual exclusion a) yes, b) no

Turn Mutual Exclusion (Strict Alteration)

```
Process; /* For two processes */
{
  while (true)
  { while ( turn != my_process_id) { };
    Access shared variables; // Critical Section;
    turn = other_process_id;
    Do other work
  } } What Properties does this satisfy?
```

Progress: a) yes, b) no

Other Flag Mutual Exclusion

```
int owner[2] = {false, false};  
Process Me;  
{ while (true)  
  { while (owner[other_process_id] ) { };  
    owner[my_process_id] = true;  
    Access shared variables; // Critical Section;  
    owner[my_process_id] = false;  
    Do other work  
  } } What Properties does this satisfy?  
mutual exclusion? a) yes, b) no
```

Other Flag Mutual Exclusion

```
int owner[2] = {false, false};  
Process Me;  
{ while (true)  
  { while (owner[other_process_id] ) { };  
    owner[my_process_id] = true;  
    Access shared variables; // Critical Section;  
    owner[my_process_id] = false;  
    Do other work  
  } } What Properties does this satisfy?  
Progress? a) yes, b) no
```

2 Flag Mutual Exclusion

```
int owner[2]= { false, false };
Process Me; {
  while (true)
  {  owner[my_process_id] = true;
    while (owner[other_process_id] ) { };
    Access shared variables; // Critical Section;
    owner[my_process_id] = false;
    Do other work
  } }  What Properties does this satisfy?
      Mutual exclusion a) yes, b) no
```

2 Flag Mutual Exclusion

```
int owner[2]= { false, false };  
Process Me; {  
    while (true)  
    {  
        owner[my_process_id] = true;  
        while (owner[other_process_id] ) { };  
        Access shared variables; // Critical Section;  
        owner[my_process_id] = false;  
        Do other work  
    } }  
    } }
```

What Properties does this satisfy?
Progress a) yes, b) no

2 Flag Mutual Exclusion

```
int owner[2]= {false, false};
Process Me; {
  while (true)
  {  owner[my_process_id] = true;
    while (owner[other_process_id] ) { };
    Access shared variables; // Critical Section;
    owner[my_process_id] = false;
    Do other work
  } }
```

What Properties does this satisfy?
Bounded Waiting a) yes, b no

2 Flag and Turn Mutual Exclusion (Peterson Solution)

```
int owner[2]={false, false};
```

```
int turn;
```

```
Process Me; { while (true)
```

```
{ owner[my_process_id] = true;
```

```
turn = other_process_id;
```

```
while (owner[other_process_id]
```

```
and turn == other_process_id ) { };
```

```
Access shared variables; // Critical Section;
```

```
owner[my_process_id] = false;
```

```
Do other work } }
```

Is this ok?

a) yes, b) no

2 Flag and Turn Mutual Exclusion (Peterson Solution)

```
int owner[2]={false, false};
```

```
int turn;
```

Yeah!!!

```
Process Me; { while (true)
```

```
{ owner[my_process_id] = true;
```

```
turn = other_process_id;
```

```
while (owner[other_process_id]
```

```
and turn == other_process_id ) { };
```

```
Access shared variables; // Critical Section;
```

```
owner[my_process_id] = false;
```

```
Do other work } }
```

Summary

- Concept of Critical Region is important
- Properties of Synchronization
- Synchronization via Busy Waiting in Software is possible!! –but not as effective
- Reminder:
 - Read Tanenbaum Section 2.2.1 – 2.2.3
 - Next lecture: Synchronization T: 2.2.4-2.2.8
 - MP1- due February 13