Interprocess Communication (IPC)

Computer Operating Systems
BLG 312E

2014-2015 Spring

Types of Interaction

- between concurrent processes
  - resource sharing
  - communication
  - synchronization

Levels of Interaction

- interaction between processes on three levels
  - processes not aware of each other (competing): using system resources (moderated by operating system)
  - processes indirectly aware of each other (sharing): resource sharing through mutual exclusion and synchronization
  - processes directly aware of each other (communicating)

Resource Sharing

- mutual exclusion
  - two types of resources
    - can be used by more than one process at a time (e.g. reading from a file)
    - can be used by only one process at a time
      - due to physical constraints (e.g. some I/O units)
      - if the actions of one process interferes with those of another (e.g. writing to a shared memory location)
- synchronization
  - a process needs to proceed after another process completes some actions

Example

2 processes: Observer and Reporter

`counter` shared variable

```
observer:
while TRUE {
      observe;
      counter ++;
}
```

```
reporter:
while TRUE {
      print_counter;
      counter=0;
}
```

Example – Possible Errors

observer	reporter
counter ← 6	
	print (6)
counter ← 7	
	counter ← 0

7. is lost!

Example – Possible Errors

```
counter++     LOAD  ACC, COUNTER
              INC   ACC
              SAVE  COUNTER, ACC
```

Race:

- when processes access a shared variable
- outcome depends on order and running speed of processes
- may be different for different runs

Example – Possible Errors

P1:

```
while TRUE
  k=k+1;
```

P2:

```
while TRUE
  k=k+1;
```

`k=0` (intial value)

what about the values of k depending on the order of **P1** and **P2** executions?

SOLUTION: mutual exclusion

Sharing

- two types of sharing:
  - READ (no need for mutual exclusion)
  - WRITE (mutual exclusion needed)
- for consistency
  - mutual exclusion
  - synchronization

Synchronization

- programs should not be dependent on running order of processes
- programs working together may need to be synchronized at some points
  - e.g. a program uses output calculated by another program

Mutual Exclusion

critical section (CS): Part of code in a process in which operations on shared resources are performed.

mutual exclusion: only one process can execute a CS for a resource at a time

Example

P1:

```
while TRUE {
  <non-CS>
  mx_begin
     <CS ops>
  mx_end
  <non-CS>
}
```

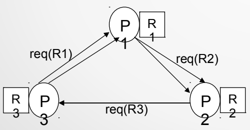
P2:

```
while TRUE {
  <non-CS>
  mx_begin
     <CS ops>
  mx_end
  <non-CS>
}
```

Mutual Exclusion – Possible Problems

- deadlock
  - more than one process requires the same resources
  - each process does not release the resource required by the other

Example: 3 processes and 3 resources



P1()	P2()	P3()
req(R1);	req(R2);	req(R3);
req(R2);	req(R3);	req(R1);

Mutual Exclusion

- `mx_begin`
  - is there a process in its CS which has not yet executed `mx_end` ?
  - if NOT
    - allow process to proceed into CS
    - leave mark for other processes
- `mx_end`
  - allow any process waiting to go into its CS to proceed
  - if there aren't any, then leave mark (empty)

Mutual Exclusion Implementation

- only one process may be in its CS
- if a process wants to enter its CS and if there are no others executing their CS, it shouldn't wait
- any process not executing its CS should not prevent another process from entering its own CS
- no assumptions should be made about the order and speed of execution of processes
- no process should stay in its CS indefinitely
- no process should wait to enter its own CS indefinitely

Mutual Exclusion Solutions

- software based solutions
- hardware based solutions
- software and hardware based solutions

A Software Based Solution

- use a shared flag that shows whether a process is in its CS or not: *busy*

  busy ← TRUE : process in CS

  busy ← FALSE : no process in CS
- **`mx_begin:`**
  ```
  while (busy);
  busy = TRUE;
  ```
  - wait until process in CS is finished
  - enter CS
- **`mx_end:`** `busy = FALSE;`

A Software Based Solution

- a possible error
  - *busy* is also a shared variable!
  - Example:
    - P1 checks and finds busy=FALSE
    - P1 interrupted
    - P2 checks and finds busy=FALSE
    - both P1 and P2 enter CS

Solutions Requiring *Busy Waiting*

```
        shared variable turn = 1;
Process 1:               Process 2:
local variables          local variables
my_turn=1;               my_turn=2;
others_turn=2;           others_turn=1;


mx_begin: while (turn != my_turn);
mx_end  : turn = others_turn;
```

Solutions Requiring *Busy Waiting*

- use up CPU time
- works properly but has limitations:
  - processes enter their CS in turn
  - depends on speed of process execution
  - depends on number of processes

Solutions Requiring *Busy Waiting*

- first correct solution: Dekker algorithm
- Peterson algorithm (1981)
  - similar approach
  - simpler

Peterson Algorithm

- shared variables:
  `req_1, req_2:` bool and initialized to FALSE
  `turn:` integer and initialized to "P1" or "P2"

```
P1:                               P2:
mx_begin:                         mx_begin:
req_1 = TRUE;                     req_2 = TRUE;
turn = P2;                        turn = P1;
while (req_2 && turn==P2);        while (req_1 && turn==P1);
 < CS >                            < CS >
mx_end: req_1 = FALSE;            mx_end: req_2 = FALSE;
```

Peterson Algorithm

- different scenarios:
  - P1 is active, P2 is passive
    req_1=TRUE and turn=P2
    req_2=FALSE so P1 proceeds after while loop
  - P1 in CS, P2 wants to enter CS
    req_2=TRUE and turn=P1;
    req_1=TRUE so P2 waits in while loop
      P2 continues after P1 executes max_end

Peterson Algorithm

- (*different scenarios cntd.):*
  - P1 and P2 want to enter CS at the same time

```
P1:              P2:
req_1=TRUE;      req_2=TRUE;
turn=P2;         turn=P1;
```

⇒ order depends on which process assigns value to the `turn` variable first.

Hardware Based Solutions

- with uninterruptable machine code instructions completed in one machine cycle
    - e.g.: `test_and_set`
  - busy waiting used
  - when a process exits CS, no mechanism to determine which other process enters next
    - indefinite waiting possible
- disabling interrupts
  - interferes with scheduling algorithm of operating system

Hardware Based Solutions

test_and_set(a): `cc ← a`
`a ← TRUE`

- with one machine instruction, contents of “a” copied into condition code register and “a” is assigned TRUE

```
mx_begin: test_and_set(busy);
          while (cc) {
            test_and_set(busy);
          }
mx_end:   busy=FALSE;
```

busy: shared variable
cc: local condition code

Semaphores

- hardware and software based solution
- no busy waiting
- does not waste CPU time
- **semaphore** is a special variable
  - only access through using two special operations
  - special operations cannot be interrupted
  - operating system carries out special operations

Semaphores

- s: semaphore variable
- special operations:
  - P (wait): when entering CS: `mutex_begin`
  - V (signal): when leaving CS: `mutex_end`

```
P(s):                V(s):
if (s > 0)           if(anyone_waiting_on_s)
  s=s-1;               activate_next_in_line;
else                 else
  wait_on_s;           s=s+1;
```

Semaphores

- take on integer values (>=0)
- created through a special system call
- assigned an initial value
- binary semaphore:
  - can be 0/1
  - used for CS
- counting semaphore:
  - can be integers >=0

Example: Observer – Reporter

```
global shared variables:
  counter: integer;
  sem: semaphore;

process P1:              process P2:
  observe;                  ...
  P(sem);                   P(sem);
    counter++;                print(counter);
  V(sem);                     counter=0;
  ....                      V(sem);

main_program:
  sem=1; counter=0;
  activate(P1);
  activate(P2);
```

Example: Observer – Reporter

sample run:

P1: P(sem) ... sem=0;

P2: P(sem) ... sem=0 so P2 is suspended

P1: V(sem) ... P2 is waiting for sem; activate P2

P2: V(sem) ... no one waiting; sem=1

Synchronization with Semaphores

- a process may require an event to proceed – process is suspended
  - e.g. process waiting for input
- another process detecting the occurence of event wakes up suspended process

⇒ "suspend – wake-up" synchronization

Synchronization with Semaphores

- solution:

```
event:semaphore; event=0;

process P1:            process P2:
  ...                    ...
  P(event);              ...
  ...                    V(event);
                 ...
```

- more than two processes may be synchronized

Semaphores

Initial value for semaphore:

- =1 for mutual exclusion
- =0 for synchronization

Semaphores

- possible deadlock scenario:

```
x, y: semaphore;
x=1; y=1;

process 1:      process 2:
  ...               ...
  P(x);         P(y);
  ...               ...
  P(y);         P(x);
  ...               ...
  V(x);         V(y);
  V(y);         V(x);
  ...               ...
```

Pay attention to the order of P and V!