Interprocess Communication and Synchronization: 4/10/19

Why do we need IPC?

- each process operates sequentially
    - all is fine until processes want to share data change data between multiple processes
    - allow processes to navigate critical regions
    - Maintain proper sequencing of actions in multiple processes
- These issues apply to threads as well
    - threads can share data easily (same address space)
    - other two issues apply to threads

Example: bounded buffer problem:

- shared variable
- producer
- consumer
- assumed atomic
- not a great solution:
    - one is constantly pulling, waste of CPU time
    - race conditions

Problem: race conditions:

- cooperating processes share storage (memory)
- both may read and write the shared memory
- problem: can't guarantee that read followed by write is atoic
    - ordering matters!
- this can result in erroneous results
- we need to eliminate race conditions…
- atomic: indivisible

Critical Regions:

- use critical regions to provide mutual exclusion and help fix race conditions
- critical regions exist in an address space
- only shared variables should you use this technique
- four conditions must hold to provide mutual exclusion:
    - 1. no two processes may simultaneously be in critical region
    - 2. no assumptions may be made about speeds or number of CPUs
        - "the wall clock doesn't exist"
        - can't assume that an instructions will finish within a certain time
    - 3. no prices running outside its critical region may block another process

        - can only block when in a critical region
    - 4. a process may not wait forever to enter its critical region

Busy waiting:strict alternation:

- Use a shared variable (turns) to keep track of whose turn it is
- waiting process continually reads the variable to see it it can proceed
    - called a spin lock: the waiting process “spins” in a tight loop reading the variable
- avoids race conditions. but doesn’t satisfy criterion 3 for critical regions
- value either 1 or 0
- cost: quantum of spin
    - not optimal: if a process gets interrupted before turning the other process runs for another quantum
- each process get a “time slice” when they only get the CPU for a limited time: called quantum
    - a process runs until its quantum runs out

Bakery algorithm for many processes:

- notation used:
    - << is lexicographical orger on (ticket_number, process_id)
    - (a, b) <<< (c, d) if (a <<< c) or ((a == c) and (b < d)
    - Ect…
- choosing a monotonic number
- doing the processes in order
- when no one is interested the number is set to 0

Hardware for synchronization:

- proper methods work, but..
    - may be somewhat complex
    - require busy waiting: process spins in a loop waiting for something to happen, wasting CPU time
- solution: use hardware
- several hardware methods
    - test and set: test a variable and set it in one instruction
    - atomic swap: switch register and memory in one instruction
        - compare-and-swap supported on x86
    - turn off interrupts: process won’t be switched out unless it asks to be suspended
        - can’t have a context switch if you don’t have interrupts

Mutual exclusion using hardware:

- single shared variable
- two versions:
    - test and set
    - swap
- works for ant number of process
- still requires busy waiting

Eliminating busy waiting:

- semaphores:
    - synchronization mechanism that doesn't require busy waiting during entire critical section
- implementation
    - semaphore S accessed by two atomic operations:
        - Down (S) : while (S <= 0) {}; S <-- S-1
            - sit and spin if true
        - Up(S): S <— S + 1
    - Down() is another name for P()
    - Up() is another name for V()
    - modify implementation to eliminate busy wait fro Down()
        - add a system call called yield, process goes to sleep
            - awakened when 1 or more
        - blocked queue: an event S happens they are no longer blocked
        - S is the number of process waiting to go to critical region

Critical sections using semaphores

- Define a class called Semaphore
    - class allows more complex implementations for semaphores
    - details hidden from processes
- Code for individual process is simple

Implementing semaphores with blocking

- Assume two (existing) operations:
    - sleep(): suspends current process
    - Wakeup(P): allows process P to resume execution
- Semaphore is a class
    - track value of semaphore
    - keep a list of processes waiting for the semaphore
- Operations still atomic
- if this variable is less than or equal to 0? then put it to sleep

```
class Semaphore{
   int value;
   ProcessList pl;
   void down();
   void up();
};

Semaphore code:
Semaphore::down(){
   value -=1;
   if(value <0){
   //add this process tool
   Sleep();
   }
}
Semaphore::up () {
Process *P;
   value +=1
   if (value <= 0 ) {
      //remove a process P
      //from pl
      Wakeup(P);
   }
}
```

Types of Semaphores

- two different types of semaphores
    - counting semaphores
    - binary semaphores
- counting semaphore
    - value can range over an unrestricted range
- Binary semaphore
    - only two values possible
        - value 1 means the semaphore is available
        - value 0 means the processes acquired the semaphore
    - may be simpler implement
- possible to implement one type using the other

Monitors:

- a monitor is another kind of high-level synchronization primitive
    - one monitor has multiple entry points

        - only one process may be in the monitor at any time
        - enforces mutual exclusion: better at avoiding programming errors
    - Monitors provided by high-level language
        - variables belonging to monitor are protected from simultaneous access
        - procedures in monitor are guaranteed to have mutual exclusion
    - Monitor implementation
        - language/complier handles implementation
        - can be implemented using semaphores
    - box where only one thread can be in
        - when it leaves another thread can come in

    - only one process can be executing in wither func1 or func2 at any time

```
monitor mon {
   int foo;
   int bar;
   double arr[100];
   void func1(…) {
   }
   void func2(…){
   }
   void mon() //initialization code
   }
}
```

Condition variables in monitors

- problem: how can a process wait inside a monitor?
    - can’t simply sleep: there’s no way for anyone else to enter
    - solution:use a condition variable
- condition variables support two operations
    - Wait(): suspend this process until signaled
    - Signal(): wake up exactly one process waiting on this condition variable
        - if no process is waiting, signal has no effect
        - signals on condition variables aren’t “saved up"
- condition variables are only usable within monitors
    - process must be in monitor to signal on a condition variable
- if a thread can't do what it wanted inside the monitor, put it to sleep and allow another thread to go inside the monitor and run. When the new thread finishes and leaves the monitor the first thread can wake up

Locks and condition variables

- anything you can do with a semaphore you can do with a monitor, vice versa
- monitors require native language support
- Instead, provide monitor support using special data types and procedures
    - locks (Acquire(), Release())
    - Condition variables (Wait(), Signal())
- Lock Usage
    - acquiring a lock <-> entering a monitor
    - releasing a lock <-> leaving a monitor
- Condition variable usage
    - each condition variable is associated with exactly one lock
    - lock must be held to use condition variable
    - waiting on a condition variable releases the lock implicitly
    - returning from Wait() on a condition variable requires the lock

Message passing

- synchronize by exchanging messages
- two primitives:
    - send: send a message
    - receive: receive a message
    - both may specify a “channel” to use
- Messages are mathematically the same as semaphores and monitors

Barriers:

- used for synchronizing multiple processes
- processes wait at a “barrier” until all in the group arrive
- after all have arrived, all processes can proceed
- may be implemented using locks and condition variables

Deadlock and starvation:

- deadlock: two or more processes are waiting indefinitely for an event that can only by caused by a waiting process
    - P0 gets A, needs B
    - P1 gets B, needs A
    - each process waiting for the other to signal
- starvation: indefinite blocking
    - process is never removed from the semaphore queue in which its suspended

    - may be caused by ordering in queues (priority)
- Doesn’t happen very happen
- would not happen with monitors