Mutual Exclusion

Hardware Solutions

- Interrupt disabling
  - not all interrupts need to be disabled
  - might not work in multiprocessor environment
- Special machine instructions
  - test and set instruction (**`TSET`**)
  - exchange instruction (**`XCHG`**)
  - advantages
    - works on any number of processors sharing memory
    - simple and supports multiple critical sections
  - disadvantages
    - busy waiting is employed
    - starvation and deadlock are possible

Test-and-Set

```
bool testset(int &i)
{  if (*i == 0)
   {  *i = 1;
      return false;
   }
   else return true;
}
```

An atomic action
An Instruction

- Shared variable *b* (initialized to 0)
- Only the first P_i who sets *b* enters CS

int b;

testset(&b)

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
   while (testset(&lock)) ; // busy waiting
      critical section
   *lock = false;
      remainder section
}
```

Synchronization Hardware

- Atomically swap two variables.

```
void swap(boolean &a, boolean &b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

  boolean lock;

- Process P_i

```
do {
   key = true;
   while (key == true) swap(&lock,&key);
      critical section
   lock = false;
      remainder section
}
```

Concurrency Control

Mutual Exclusion and Synchronization

- Principles of Concurrency
- Mutual Exclusion
  - Software Solutions
  - Hardware Solutions
- Semaphores
- Producer/Consumer
- Bounded Buffer
- Message Passing
- Monitors

Mutual Exclusion Issues

- Software solutions are complex
  - must be used when no Hardware support
- Hardware instructions work
  - Still kind of complex
- Both make the processes busy-wait
  - Wastes CPU cycles

Semaphores

- A synchronization tool provided by the OS.
- Two or more processes can cooperate by using simple signals – a process can be forced to stop until it has received a specific signal.
- Semaphore operations:
  - initialize to a non-negative value
  - *wait* decrements the semaphore value – if negative, the process is blocked
  - *signal* increments the semaphore value – if not positive, then a process blocked by a *wait* operation is unblocked.

Semaphores

- Modifications to semaphore variable are atomic.
- Two forms of semaphores:
  - binary semaphore
    - easier to implement
    - also called a mutex
  - counting semaphore
    - *wait* - # of processes waiting on the resource
    - *signal* - # of resources available
- Blocked processes are held in waiting queues
  - strong semaphores unblock the longest waiting process
    - guarantee freedom from starvation
  - weak semaphores unblock without regard to order

The Critical Section Problem…

Shared semaphore: mutex = 1;

repeat
 wait(mutex)
 critical section
 signal(mutex)
 remainder section
until false

Will this work for n processes?

wait/signal

```
struct semaphore
{   int value;
    Queue processes;
}

wait(Semaphore s)
{   s.value = s.value - 1;
    if (s.value < 0)
    {   add this process to s.queue
        block; // the caller is BLOCKED
    }
}
signal(Semaphore s)
{   s.value = s.value + 1;
    if (s.value <= 0)
    {   remove a process P from s.queue
        wakeup(P); // make P READY
    }
}
```

Autonomy

- Critical
  - Semaphore operations must be atomic
- Uniprocessor
  - simply inhibit interrupts (normal user can't)
  - Use testset to create a mutex in the calls
- Multiprocessor
  - hardware must provide special support, or
  - use software solutions

```
wait(Semaphore s)
{   while(testset(&lock));
    s.value--;
    if (s.value < 0)
    {   add process to s.queue
        lock = FALSE;
        block;
    }
    else lock = FALSE;
}
```

Autonomy

- Commonly used ME mechanisms are:
- Semaphore, lock, shared variables, or conditional variables
- P/V:

  P(s) [while (s==0) wait; s = s-1]

  V(s) [s = s+1]
- lock/unlock
- wait/signal
- monitors, etc.

Consider…

semaphore S, Q; // initial value 1

P0:	P1:
P(S);	P(Q);
P(Q);	P(S);
.	.
.	.
.	.
V(S);	V(Q);
V(Q);	V(S);

Is there anything wrong with this?

A simple example

Expression: (x*2) / (y-z) + sin(x)

Shared variables: t1, t2, t3, t4, t5;

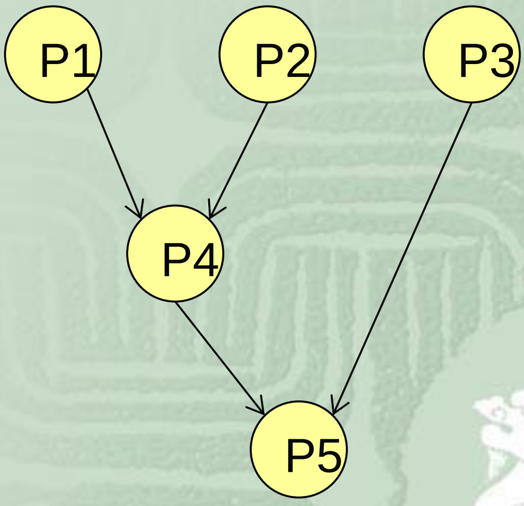
P1: t1 = x*2;

P2: t2 = y – z;

P3: t3 = sin(X);

P4: t4 = t1/t2;

P5: t5 = t4 + t3



Concurrency control

semaphore s1, s2, s3, s4; // initial 0s

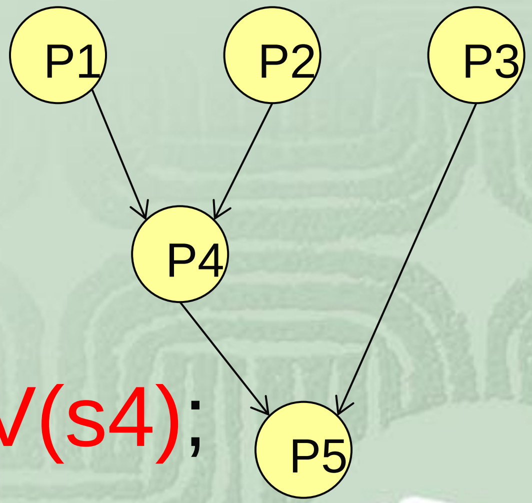
P1: t1 = x*2; V(s1);

P2: t2 = y – z; V(s2);

P3: t3 = sin(X); V(s3);

P4: P(s1); P(s2); t4 = t1/t2; V(s4);

P5: P(s3); P(s4); t5 = t4 + t3;



The Producer-Consumer Problem

```
int counter = 0;
Item buffer[n];
```

Producer

```
while(1){
    …
    produce an item in nextp
    …
    while(counter == n)
       do no-op
    buffer[in] = nextp
    in = (in + 1) mod n
    counter = counter + 1

}
```

Consumer

```
while(1){
    …
    while(counter == 0)
       do no-op
    nextc = buffer[out]
    out = (out + 1) mod n
    counter = counter -1
    …
    consume the item in nextc
    …
}
```

Bounded Buffer Solution

```
Shared semaphore: empty = n, full = 0;
Item buffer[n];
```

```
while(1){
  produce an item in nextp

  wait(empty);


  buffer[in] = nextp
  in = (in + 1) mod n


  signal(full);

}
```

Producer

```
while(1){
  wait(full);


  nextc = buffer[out]
  out = (out + 1) mod n


  signal(empty);

  consume the item in nextc
}
```

Consumer

Any problem?

Bounded Buffer Solution

```
Shared semaphore: empty = n, full = 0, mutex = 1;
Item buffer[n];
```

```
while(1){
  produce an item in nextp

  wait(empty);
  wait(mutex);

  buffer[in] = nextp
  in = (in + 1) mod n

  signal(mutex);
  signal(full);
}
```

Producer

```
while(1){
  wait(full);
  wait(mutex);

  nextc = buffer[out]
  out = (out + 1) mod n

  signal(mutex);
  signal(empty);

  consume the item in nextc
}
```

Consumer

Mutex + Synchronization: multiple producers and comusers

Message Passing

- A general method used for interprocess communication (IPC)
  - for processes inside the same computer
  - for processes in a distributed system
- Another means to provide process synchronization and mutual exclusion
- We have at least two primitives:
  - send(destination, message)
  - receive(source, message)
- May or may not be blocking

Synchronization

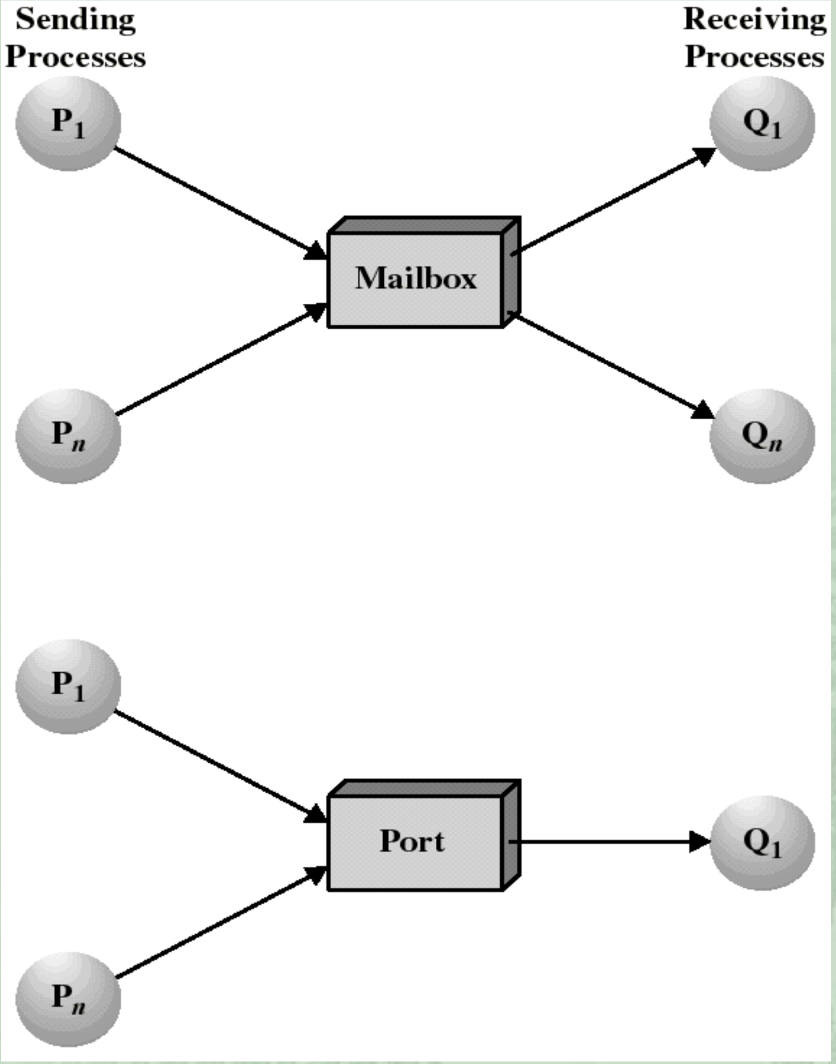
- For the sender: it is more natural not to be blocked
  - can send several messages to multiple destinations
  - sender usually expects acknowledgment of message receipt (in case receiver fails)
  - *PostMessage*() is asynchronous – returns immediately
  - *SendMessage*() is synchronous –block until message delivered and processed
- For the receiver: it is more natural to be blocked after issuing *ReceiveMessage*()
  - the receiver usually needs the info before proceeding
  - but could be blocked indefinitely if sender process fails before sending reply

Addressing in message passing

- Direct addressing:
  - when a specific process identifier is used for source/destination
  - but it might be impossible to specify the source ahead of time (ex: a print server)
- Indirect addressing (more convenient):
  - messages are sent to a shared mailbox which consists of a queue of messages
  - senders place messages in the mailbox, receivers pick them up

Mailboxes and Ports

- A mailbox can be private
  - one sender/receiver pair
- A mailbox can be shared among several senders and receivers
  - OS may then allow the use of message types (for selection)
- Port: a mailbox associated with one receiver and multiple senders
  - used for client/server application: the receiver is the server



Ownership of ports and mailboxes

- A port is usually owned and created by the receiving process
- The port is destroyed when the receiver terminates
- The OS creates a mailbox on behalf of a process (which becomes the owner)
- The mailbox is destroyed at the owner’s request or when the owner terminates

Message format

- Consists of header and body of message
- Control information:
  - what to do if run out of buffer space
  - sequence numbers
  - priority...
- **Queuing discipline: usually FIFO but can also include priorities**