Interprocess Communication

Chester Rebeiro
IIT Madras
Virtual Memory View

- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping

Executing Process

Virtual Memory Map

<table>
<thead>
<tr>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Virtual Memory Map

<table>
<thead>
<tr>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

RAM

<table>
<thead>
<tr>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

kernel
Virtual Memory View

- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping
Virtual Memory View

- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping
Inter Process Communication

• Advantages of Inter Process Communication (IPC)
  – Information sharing
  – Modularity/Convenience

• 3 ways
  – Shared memory
  – Message Passing

  – Signals
Shared Memory

• One process will create an area in RAM which the other process can access
• Both processes can access shared memory like a regular working memory
  – Reading/writing is like regular reading/writing
  – Fast
• Limitation: Error prone. Needs synchronization between processes
Shared Memory in Linux

- **int shmget (key, size, flags)**
  - Create a shared memory segment;
  - Returns ID of segment: `shmid`
  - `key`: unique identifier of the shared memory segment
  - `size`: size of the shared memory (rounded up to the PAGE_SIZE)

- **int shmat(shmid, addr, flags)**
  - Attach `shmid` shared memory to address space of the calling process
  - `addr`: pointer to the shared memory address space

- **int shmdt(shmid)**
  - Detach shared memory
Example

server.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE 27 /* Size of shared memory */

main()
{
  char c;
  int shmid;
  key_t key;
  char *shm, *s;
  key = 5678; /* some key to uniquely identifies the shared memory */

  /* Create the segment. */
  if ((shmid = shmat(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
    perror("shmat");
    exit(1);
  }

  /* Attach the segment to our data space. */
  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
  }

  /* Now put some things into the shared memory */
  s = shm;
  for (c = 'a'; c <= 'z'; c++)
    *s++ = c;
  *s = 0; /* end with a NULL termination */

  /* Wait until the other process changes the first character */
  /* * to '*'; the shared memory */
  while (*shm != '*')
    sleep(1);

  exit(0);
}
```

client.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE 27

main()
{
  int shmid;
  key_t key;
  char *shm, *s;
  /* We need to get the segment named "5678", created by the server */
  key = 5678;

  /* Locate the segment. */
  if ((shmid = shmat(key, SHMSIZE, 0666)) < 0) {
    perror("shmat");
    exit(1);
  }

  /* Attach the segment to our data space. */
  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
  }

  /* read what the server put in the memory. */
  for (s = shm; *s != 0; s++)
    putchar(*s);
  putchar(\n');

  /* Finally, change the first character of the */
  /* segment to '*', indicating we have read */
  /* the segment. */
  *shm = '*';
  exit(0);
```
Message Passing

- Shared memory created in the kernel
- System calls such as `send` and `receive` used for communication
  - Cooperating: each send must have a receive
- **Advantage**: Explicit sharing, less error prone
- **Limitation**: Slow. Each call involves marshalling / demarshalling of information
Pipes

- Always **between parent and child**
- Always **unidirectional**
- Accessed by two associated file descriptors:
  - `fd[0]` for reading from pipe
  - `fd[1]` for writing to the pipe
Pipes for two way communication

- Two pipes opened: pipe0 and pipe1
- Note the unnecessary pipes
- Close the unnecessary pipes
Example

(child process sending a string to parent)
Signals

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - eg. 9: kill, 11: segmentation fault
- Send a signal to a process
  - `kill(pid, signum)`
- Process handler for a signal
  - `sighandler_t signal(signum, handler);`
  - Default if no handler defined

ref: http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html
Synchronization

Chester Rebeiro
IIT Madras
Motivating Scenario

- **Single core**
  - Program 1 and program 2 are executing at the same time but sharing a single core

```c
program 0
{
    *
    *
    counter++
    *
}
```

```c
program 1
{
    *
    *
    counter--
    *
}
```

shared variable
int counter=5;

CPU usage wrt time
Motivating Scenario

- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6
Motivating Scenario

Program 0

```
{  
  *  
  *  
  counter++  
  *  
}
```

Program 1

```
{  
  *  
  *  
  counter--  
  *  
}
```

Shared variable counter = 5;

```
R1 ← counter  
R1 ← R1 + 1  
counter ← R1  
R2 ← counter  
R2 ← R2 - 1  
counter ← R2  
R2 ← R2 - 1  
counter ← R2
```

```
R1 ← counter  
R2 ← counter  
R2 ← R2 - 1  
counter ← R2  
R2 ← R2 - 1  
counter ← R2
```

```
R2 ← counter  
R2 ← counter  
R2 ← R2 + 1  
counter ← R2  
R2 ← R2 - 1  
counter ← R2
```

Context switch

counter = 5  
counter = 6  
counter = 4
Race Conditions

• Race conditions
  – A situation where several processes access and manipulate the same data (*critical section*)
  – The outcome depends on the order in which the access take place
  – Prevent race conditions by synchronization
    • Ensure only one process at a time manipulates the critical data

```plaintext
{  
  *  
  *  

  counter++

}
```
Race Conditions in Multicore

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores

shared variable
int counter=5;

program 0
{
  *
  *
  counter++
  *
}

program 1
{
  *
  *
  counter--
  *
}

CPU usage wrt time
Critical Section

• Any solution should satisfy the following requirements
  – **Mutual Exclusion**: No more than one process in critical section at a given time
  – **Progress**: When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay
  – **No starvation (bounded wait)**: There is an upper bound on the number of times a process enters the critical section, while another is waiting.
Locks and Unlocks

- **lock(L)**: acquire lock L exclusively
  - Only the process with L can access the critical section
- **unlock(L)**: release exclusive access to lock L
  - Permitting other processes to access the critical section

```c
int counter=5;
lock_t L;

program 0
{
    *
    *
    lock(L)
    counter++
    unlock(L)
    *
}

program 1
{
    *
    *
    lock(L)
    counter--
    unlock(L)
    *
}
```
When to have Locking?

• Single instructions by themselves are atomic
  
  eg. add %eax, %ebx

• Multiple instructions need to be explicitly made atomic
  – Each piece of code in the OS must be checked if they need to be atomic
How to Implement Locking
(Software Solutions)

Chester Rebeiro
IIT Madras
Using Interrupts

- **Simple**
  - When interrupts are disabled, context switches won’t happen
- **Requires privileges**
  - User processes generally cannot disable interrupts
- **Not suited for multicore systems**
Software Solution (Attempt 1)

- Achieves mutual exclusion
- Busy waiting – waste of power and time
- Needs to alternate execution in critical section

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2; // unlock
    other code
}
```

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```
Problem with Attempt 1

- Had a common turn flag that was modified by both processes
- This required processes to alternate.
- Possible Solution: Have two flags – one for each process

```c
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2; // unlock
    other code
}
```

```c
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```
Software Solution (Attempt 2)

- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

```c
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

```c
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

p2_inside = False, p1_inside = False

shared

lock

unlock
## Attempt 2: No mutual exclusion

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU</th>
<th>p1_inside</th>
<th>p2_inside</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>while(p2_inside == True);</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>context switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>while(p1_inside == True);</code></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>p2_inside = True;</code></td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>context switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>p1_inside = True;</code></td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Both p1 and p2 can enter into the critical section at the same time

```c
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

```c
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```
Problem with Attempt 2

- The flag \((p1_{\text{inside}}, p2_{\text{inside}})\), is set after we break from the while loop.

```c
while(1){
  while(p2_{\text{inside}} == True);
  p1_{\text{inside}} = True;
  \text{critical section}
  p1_{\text{inside}} = False;
  \text{other code}
}
```

```c
while(1){
  while(p1_{\text{inside}} == True);
  p2_{\text{inside}} = True;
  \text{critical section}
  p2_{\text{inside}} = False;
  \text{other code}
}
```
Software Solution (Attempt 3)

- Achieves mutual exclusion
- Does not achieve progress (could deadlock)
Attempt 3: No Progress

<table>
<thead>
<tr>
<th>CPU</th>
<th>p1_inside</th>
<th>p2_inside</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1_wants_to_enter = True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>context switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2_wants_to_enter = True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved
Each process is waiting for the other
this is a deadlock

```c
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```
Deadlock

<table>
<thead>
<tr>
<th>CPU</th>
<th>p1_inside</th>
<th>p2_inside</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1_wants_to_enter = True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>context switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2_wants_to_enter = True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved
Each process is waiting for the other
this is a deadlock
Problem with Attempt 3

- Deadlock
  - Have a way to break the deadlock
Peterson’s Solution

Break the deadlock with a ‘favored’ process

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

If the second process wants to enter, favor it. (be nice !!!)

lock

while(1){
    p1_wants_to_enter = True
    favored = 2
    while (p2_wants_to_enter AND favored = 2);
    critical section
    p1_wants_to_enter = False
other code
}

unlock

favored is used to break the tie when both p1 and p2 want to enter the critical section.
favored can take only two values : 1 or 2
(* the process which sets favored last looses the tie *)
Peterson’s Solution

• Deadlock broken because favored can only be 1 or 2.
  – Therefore, tie is broken. Only one process will enter the critical section
• Solves Critical Section problem for two processes
Bakery Algorithm

- Synchronization between \( N > 2 \) processes
- By Leslie Lamport

Simplified Bakery Algorithm

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}

unlock(i){
    num[i] = 0;
}
```

This is at the doorway!!!
It has to be atomic to ensure two processes do not get the same token
Simplified Bakery Algorithm (example)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```plaintext
lock(i){
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

```plaintext
unlock(i){
    num[i] = 0;
}
```

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Simplified Bakery Algorithm (example)

Processes numbered 0 to N-1
num is an array N integers (initially 0).
Each entry corresponds to a process

lock(i){
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}

unlock(i){
    num[i] = 0;
}

critical section

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Simplified Bakery Algorithm
(why atomic doorway?)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```c
lock(i){
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

Unlock(i){
    num[i] = 0;
}

This is at the doorway!!!
Assume it is not atomic

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

P4 and P5 can enter the critical section at the same time.
Original Bakery Algorithm (making MAX atomic)

- Without atomic operation assumptions
- Introduce an array of N Booleans: choosing, initially all values False.

```c
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}

unlock(i){
    num[i] = 0;
}
```

Critical section

Choosing ensures that a process is not at the doorway, i.e., the process is not ‘choosing’ a value for num

\((a, b) < (c, d)\) which is equivalent to: \((a < c)\) or \(((a == c) \text{ and } (b < d))\)
Original Bakery Algorithm (making MAX atomic)

- Without atomic operation assumptions
- Introduce an array of N Booleans: choosing, initially all values False.

```c
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

```c
unlock(i){
    num[i] = 0;
}
```

critical section

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))
Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: `choosing`, initially all values False.

```c
lock(i)
{
    choosing[i] = True
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
unlock(i){
    num[i] = 0;
}
```

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))
Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: choosing, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))
How to Implement Locking
(Hardware Solutions and Usage)
Analyze this

- Does this scheme provide mutual exclusion?

Process 1
```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

Process 2
```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No

- lock = 0
- P1: while(lock != 0);
- P2: while(lock != 0);
- P2: lock = 1;
- P1: lock = 1;
- .... Both processes in critical section

context switch
If only…

• We could make this operation atomic

```c
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

Make atomic

Hardware to the rescue….
Hardware Support
(Test & Set Instruction)

- Write to a memory location, return its old value

```c
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)
Hardware Support
(Test & Set Instruction)

- Write to a memory location, return its old value

```c
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)
Hardware Support
(Test & Set Instruction)

• Write to a memory location, return its old value

```c
int test_and_set(int *L){
  int prev = *L;
  *L = 1;
  return prev;
}
```

equivalent software representation (the entire function is executed atomically)

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
Hardware Support
(Test & Set Instruction)

• Write to a memory location, return its old value

```c
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
Hardware Support  
(Test & Set Instruction)

- Write to a memory location, return its old value

```c
int test_and_set(int *L){
  int prev = *L;
  *L = 1;
  return prev;
}
```

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
Hardware Support
(Test & Set Instruction)

• Write to a memory location, return its old value

```c
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

```c
while(1){
    while(test_and_set(&lock) == 1);
    critical section
    lock = 0; // unlock
    other code
}
```

Usage for locking

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts. So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0.
Intel Hardware Support (xchg Instruction)

- Write to a memory location, return its old value

```c
int xchg(int *L, int v){
    int prev = *L;
    *L = v;
    return prev;
}
```

Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.
Intel Hardware Support (using xchg instruction)

typical usage:
\texttt{xchg \textit{reg}, \textit{mem}}

Note. \%eax is returned

```c
int xchg(addr, value){
    \%eax = value
    xchg \%eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

void release(int *locked){
    locked = 0;
}
```
Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage:

xchg reg, mem

int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

void release(int *locked){
    locked = 0;
}
Intel Hardware Support
(using xchg instruction)

**Typical usage:**

```
xchg reg, mem
```

```c
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

void release(int *locked){
    locked = 0;
}
```

Note. `%eax` is returned
Intel Hardware Support (using xchg instruction)

Note. \%eax is returned

typical usage:
\texttt{xchg reg, mem}

```c
int xchg(addr, value){
  \%eax = value
  xchg \%eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
  locked = 0;
}
```
Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage:

\[ \text{xchg reg, mem} \]

```c
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

void release(int *locked){
    locked = 0;
}
```
High Level Constructs

- Spinlock
- Mutex
- Semaphore
Spinlocks Usage

Process 1
acquire(&locked)  
critical section  
release(&locked)

Process 2
acquire(&locked)  
critical section  
release(&locked)

• One process will acquire the lock
• The other will wait in a loop repeatedly checking if the lock is available
• The lock becomes available when the former process releases it

int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

void release(int *locked){
    locked = 0;
}

See spinlock.c and spinlock.h in xv6 [15]
Issues with Spinlocks

xchg %eax, X

• No compiler optimizations should be allowed
  – Should not make X a register variable
    • Write the loop in assembly or use volatile

• Should not reorder memory loads and stores
  • Use serialized instructions (which forces instructions not to be reordered)
  • Luckily xchg is already implements serialization
More issues with Spinlocks

- No caching of (X) possible. All xchg operations are bus transactions.
  - CPU asserts the LOCK, to inform that there is a ‘locked’ memory access
- acquire function in spinlock invokes xchg in a loop…each operation is a bus transaction …. huge performance hits

```
xchg %eax, X
```
A better acquire

void acquire(int *locked) {
    reg = 1
    while(1)
        if(xchg(locked, reg) == 0)
            break;
}

Original.
Loop with xchg.
Bus transactions.
Huge overheads

Better way
Outer loop changes the value of locked
inner loop only reads the value of locked. This allows caching of locked.
Access cache instead of memory.

int xchg(addr, value) {
    %eax = value
    xchg %eax, (addr)
}
Spinlocks (when should it be used?)

• Characteristic: busy waiting
  – Useful for short critical sections, where much CPU time is not wasted waiting
    • eg. To increment a counter, access an array element, etc.

  – Not useful, when the period of wait is unpredictable or will take a long time
    • eg. Not good to read page from disk.
    • Use mutex instead (…mutex)
Spinlock in pthreads

```c
#include <pthread.h>
#include <stdio.h>

int global_counter;
pthread_spinlock_t splk;

void *thread_fn(void *arg){
    long id = (long) arg;
    while(1){
        pthread_spin_lock(&splk);
        if (id == 1) global_counter++;
        else global_counter--;
        pthread_spin_unlock(&splk);
        printf("%ld(%ld)\n", id, global_counter);
        sleep(1);
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_spin_init(&splk, PTHREAD_PROCESS_PRIVATE);
    pthread_create(&t1, NULL, thread_fn, (void *)1);
    pthread_create(&t2, NULL, thread_fn, (void *)2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_spin_destroy(&splk);
    printf("Exiting main\n");
    return 0;
}
```
Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process

```c
int xchg(addr, value) {
    %eax = value
    xchg %eax, (addr)
}

void lock(int *locked) {
    while(1) {
        if(xchg(locked, 1) == 0)
            break;
        else
            sleep();
    }
}

void unlock(int *locked) {
    locked = 0;
    wakeup();
}
```

Ref: wakeup(2864), sleep(2803)
Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```c
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void lock(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
        else
            sleep();
    }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```
Thundering Herd Problem

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```c
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void lock(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
        else{
            // add this process to Queue
            sleep();
        }
    }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```
pthread Mutex

• pthread_mutex_lock
• pthread_mutex_unlock
Locks and Priorities

• What happens when a high priority task requests a lock, while a low priority task is in the critical section
  – Priority Inversion
  – Possible solution
    • Priority Inheritance

Interesting Read: Mass Pathfinder
Semaphores
Producer – Consumer Problems

- Also known as *Bounded buffer Problem*
- Producer produces and stores in buffer, Consumer consumes from buffer
Producer – Consumer Problems

• Also known as *Bounded buffer Problem*
• Producer produces and stores in buffer, Consumer consumes from buffer
• Trouble when
  – Producer produces, but buffer is full
  – Consumer consumes, but buffer is empty
Producer-Consumer Code

Buffer of size N
int count=0;
Mutex mutex, empty, full;

```c
void producer()
{
    while(TRUE)
    {
        item = produce_item();
        if (count == N) sleep(empty);
        lock(mutex);
        insert_item(item); // into buffer
        count++;
        unlock(mutex);
        if (count == 1) wakeup(full);
    }
}

void consumer()
{
    while(TRUE)
    {
        if (count == 0) sleep(full);
        lock(mutex);
        item = remove_item(); // from buffer
        count--;
        unlock(mutex);
        if (count == N-1) wakeup(empty);
        consume_item(item);
    }
}
```
Producer-Consumer Code

Buffer of size N
int count=0;
Mutex mutex, empty, full;

```c
void producer()
{
    while(TRUE)
    {
        item = produce_item();
        if (count == N) sleep(emptv);
        lock(mutex);
        insert_item(item); // into buffer
        count++;
        unlock(mutex);
        if (count == 1) wakeup(full);
    }
}

void consumer()
{
    while(TRUE)
    {
        if (count == 0) sleep(full);
        lock(mutex);
        item = remove_item(); // from buffer
        count--;
        unlock(mutex);
        if (count == N-1) wakeup(emptv);
        consume_item(item);
    }
}
```

Read count value
Test count = 0
Lost Wakeups

- Consider the following context of instructions
- Assume buffer is initially empty

```plaintext
3 read count value // count ← 0
3 item = produce_item();
5 lock(mutex);
6 insert_item(item); // into buffer
7 count++; // count = 1
8 unlock(mutex)
9 test (count == 1) // yes
9 signal(full);
3 test (count == 0) // yes
3 wait();
```

Note, the wakeup is lost. Consumer waits even though buffer is not empty. Eventually producer and consumer will wait infinitely

consumer still uses the old value of count (ie 0)
Semaphores

- Proposed by Dijkstra in 1965
- Functions down and up must be atomic
- down also called P (Proberen Dutch for try)
- up also called V (Verhogen, Dutch form make higher)
- Can have different variants
  - Such as blocking, non-blocking
- If S is initially set to 1,
  - Blocking semaphore similar to a Mutex
  - Non-blocking semaphore similar to a spinlock

```c
void down(int *S){
    while( *S <= 0);
    *S--;
}

void up(int *S){
    *S++;
}
```
Producer-Consumer with Semaphores

Buffer of size N
int count;

void producer()
{
    while(TRUE){
        item = produce_item();
        down(empty);
        wait(mutex);
        insert_item(item); // into buffer
        signal(mutex);
        up(full);
    }
}

void consumer()
{
    while(TRUE){
        down(full);
        wait(mutex);
        item = remove_item(); // from buffer
        signal(mutex);
        up(empty);
        consume_item(item);
    }
}

full = 0, empty = N
POSIX semaphores

- sem_init
- sem_wait
- sem_post
- sem_getvalue
- sem_destroy
Dining Philosophers Problem
Dining Philosophers Problem

- Philosophers either think or eat
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.

**Problem Statement**: Develop an algorithm where no philosopher starves.
What happens if only philosophers P1 and P3 are always given the priority? P2, P4, and P5 starves… so scheme needs to be fair

```c
#define N 5
int forks = {1,2,3,4,5,1};

void philosopher(int i){
    while(TRUE){
        think(); // for some_time
        take_fork(i);
        take_fork(i + 1);
        eat();
        put_fork(i);
        put_fork(i + 1);
    }
}
```
What happens if all philosophers decide to pick up their left forks at the same time? Possible starvation due to deadlock.

```c
#define N 5
int forks = {1,2,3,4,5,1};

void philosopher(int i){
    while(TRUE){
        think(); // for some_time
        take_fork(i);
        take_fork(i + 1);
        eat();
        put_fork(i);
        put_fork(i + 1);
    }
}
```
Deadlocks

• A situation where programs continue to run indefinitely without making any progress
• Each program is waiting for an event that another process can cause
Second try

- **Take fork i, check if fork (i+1) is available**
- Imagine,
  - All philosophers start at the same time
  - Run simultaneously
  - And think for the same time
- This could lead to philosophers taking fork and putting it down continuously. A deadlock.

- A better alternative
  - Philosophers wait a random time before `take_fork(i)`
  - Less likelihood of deadlock.
  - Used in schemes such as Ethernet

```c
#define N 5
int forks = {1,2,3,4,5,1};

void philosopher(int i){
    while(TRUE){
        think();
        take_fork(i);
        if (available((i+1)){
            take_fork((i + 1));
            eat();
        }else{
            put_fork(i);
        }
    }
}
```
Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
  - Only one philosopher can eat at a time

```c
#define N 5
int forks = {1,2,3,4,5,1};
void philosopher(int i){
    while(TRUE){
        think(); // for some_time
        wait(mutex);
        take_fork(i);
        take_fork((i + 1));
        eat();
        put_fork(i);
        put_fork((i + 1));
        signal(mutex);
    }
}
```
Solution to Dining Philosophers

Uses N semaphores (s[0], s[1], ...., s[N-1]) all initialized to 0, and a mutex.

Philosopher has 3 states: HUNGRY, EATING, THINKING

A philosopher can only move to EATING state if neither neighbor is eating.

```c
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}

void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}

void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    unlock(mutex);
}

void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
        state[i] = EATING;
        up(s[i]);
    }
}
```
Deadlocks
Consider this situation:
Deadlocks

A Deadlock Arises:
Deadlock: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
Conditions for Resource Deadlocks

1. **Mutual Exclusion**
   - Each resource is either available or currently assigned to exactly one process

2. **Hold and wait**
   - A process holding a resource, can request another resource

3. **No preemption**
   - Resources previously granted cannot be forcibly taken away from a process

4. **Circular wait**
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain

All four of these conditions must be present for a resource deadlock to occur!!
Deadlocks: (A Chanced Event)

- Ordering of resource requests and allocations are probabilistic, thus deadlock occurrence is also probabilistic.
No dead lock occurrence (B can be granted S after step q)
Should Deadlocks be handled?

• Preventing / detecting deadlocks could be tedious
• Can we live without detecting / preventing deadlocks?
  – What is the probability of occurrence?
  – What are the consequences of a deadlock? (How critical is a deadlock?)
Handling Deadlocks

• Detection and Recovery
• Avoidance
• Prevention
Deadlock detection

• How can an OS detect when there is a deadlock?
• OS needs to keep track of
  – Current resource allocation
    • Which process has which resource
  – Current request allocation
    • Which process is waiting for which resource
• Use this information to detect deadlocks
Deadlock Detection

- Deadlock detection with **one resource of each type**
- Find cycles in resource graph
Deadlock Detection

- Deadlock detection with multiple resources of each type

\[ \sum_{i=1}^{n} C_{ij} + A_{j} = E_{j} \]

\[ E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix} \]
\[ A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix} \]

Current Allocation Matrix
Who has what!!

\[ C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \]

Request Matrix
Who is waiting for what!!

\[ R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \]

Process \( P_i \) holds \( C_i \) resources and requests \( R_i \) resources, where \( i = 1 \) to 3

Goal is to check if there is any sequence of allocations by which all current requests can be met. If so, there is no deadlock.
Deadlock Detection

- Deadlock detection with multiple resources of each type

\[
E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}
\]

Existing Resource Vector

Resources Available

\[
\sum_{i=1}^{n} C_{ij} + A_j = E_j
\]

\[
P_1 \text{ cannot be satisfied}
\]

\[
P_2 \text{ cannot be satisfied}
\]

\[
P_3 \text{ can be satisfied}
\]

Process \( P_i \) holds \( C_i \) resources and requests \( R_i \) resources, where \( i = 1 \) to 3
Deadlock Detection

- Deadlock detection with multiple resources of each type

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong> = (4 2 3 1)</td>
<td><strong>A</strong> = (2 1 0 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Existing Resource Vector

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tape drives</td>
<td>Plotters</td>
<td>Scanners</td>
<td>CD Roms</td>
</tr>
</tbody>
</table>

Resources Available

\[
P_1 \quad P_2 \quad P_3
\]

Current allocation matrix

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix}
\]

Request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]

\[P_3\) runs and its allocation is (2, 2, 2, 0)

On completion it returns the available resources are \(A = (4 2 2 1)\)

Either \(P_1\) or \(P_2\) can now run.

**NO Deadlock!!!**
Deadlock Detection

- Deadlock detection with multiple resources of each type

\[ E = (4 \hspace{1em} 2 \hspace{1em} 3 \hspace{1em} 1) \]

\[ A = (2 \hspace{1em} 1 \hspace{1em} 0 \hspace{1em} 0) \]

\[ \sum_{i=1}^{n} C_{ij} + A_j = E_j \]

Process \( P_i \) holds \( C_i \) resources and requests \( R_i \) resources, where \( i = 1 \) to \( 3 \)

Deadlock detected as none of the requests can be satisfied.
Deadlock Recovery

What should the OS do when it detects a deadlock?

• **Raise an alarm**
  – Tell users and administrator

• **Preemption**
  – Take away a resource temporarily (frequently not possible)

• **Rollback**
  – Checkpoint states and then rollback

• **Kill low priority process**
  – Keep killing processes until deadlock is broken
  – (or reset the entire system)
Deadlock Avoidance

- System decides in advance if allocating a resource to a process will lead to a deadlock.

Note: unsafe state is not a deadlocked state.
Deadlock Avoidance

Is there an algorithm that can always avoid deadlocks by conservatively make the right choice.

- Ensures system never reaches an unsafe state

- **Safe state** : A state is said to be safe, if there is some scheduling order in which every process can run to completion even if all of them suddenly requests their maximum number of resources immediately

- An unsafe state *does not have to* lead to a deadlock; *it could* lead to a deadlock
Example with a Banker

• Consider a banker with 4 clients ($P_1, P_2, P_3, P_4$).
  – Each client has certain credit limits (totaling 20 units)
  – The banker knows that max credits will not be used at once, so he keeps only 10 units

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Total : 10 units
free : 3 units

– Clients declare **maximum** credits in advance. The banker can allocate credits provided no unsafe state is reached.
Safe State

Allocate 2 units to B

Allocate 5 to C

This is a safe state because there is some scheduling order in which every process executes.
## Unsafe State

This is an unsafe state because there exists NO scheduling order in which every process executes.

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Free: 2 units**

Allocate 2 units to B

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Free: 0 units**

B completes

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Free: 4 units**
Banker’s Algorithm
(with a single resource)

When a request occurs
  – If(is_system_in_a_safe_state)
    • Grant request
  – else
    • postpone until later

Please read Banker’s Algorithm with multiple resources from Modern Operating Systems, Tanenbaum
Deadlock Prevention

- Deadlock avoidance not practical, need to know maximum requests of a process
- Deadlock prevention
  - Prevent at-least one of the 4 conditions
  1. Mutual Exclusion
  2. Hold and wait
  3. No preemption
  4. Circular wait
Prevention

1. **Preventing Mutual Exclusion**
   - Not feasible in practice
   - But OS can ensure that resources are optimally allocated

2. **Hold and wait**
   - One way is to achieve this is to require all processes to request resources before starting execution
     - May not lead to optimal usage
     - May not be feasible to know resource requirements

3. **No preemption**
   - Pre-empt the resources, such as by virtualization of resources (e.g., Printer spools)

4. **Circular wait**
   - One way, process holding a resource cannot hold a resource and request for another one
   - Ordering requests in a sequential / hierarchical order.
Hierarchical Ordering of Resources

- Group resources into levels
  (i.e. prioritize resources numerically)
- A process may only request resources at higher levels than any resource it currently holds
- Resource may be released in any order
- eg.
  - Semaphore $s_1$, $s_2$, $s_3$ (with priorities in increasing order)
    \[
    \text{down}(S_1); \text{down}(S_2); \text{down}(S_3); \rightarrow \text{allowed}
    \]
    \[
    \text{down}(S_1); \text{down}(S_3); \text{down}(S_2); \rightarrow \text{not allowed}
    \]