CE5045
Embedded System Design

Inter-process Communication

https://github.com/tychen-NCU/EMBS-NCU

Instructor: Dr. Chen, Tseng-Yi

Computer Science & Information Engineering

Outline

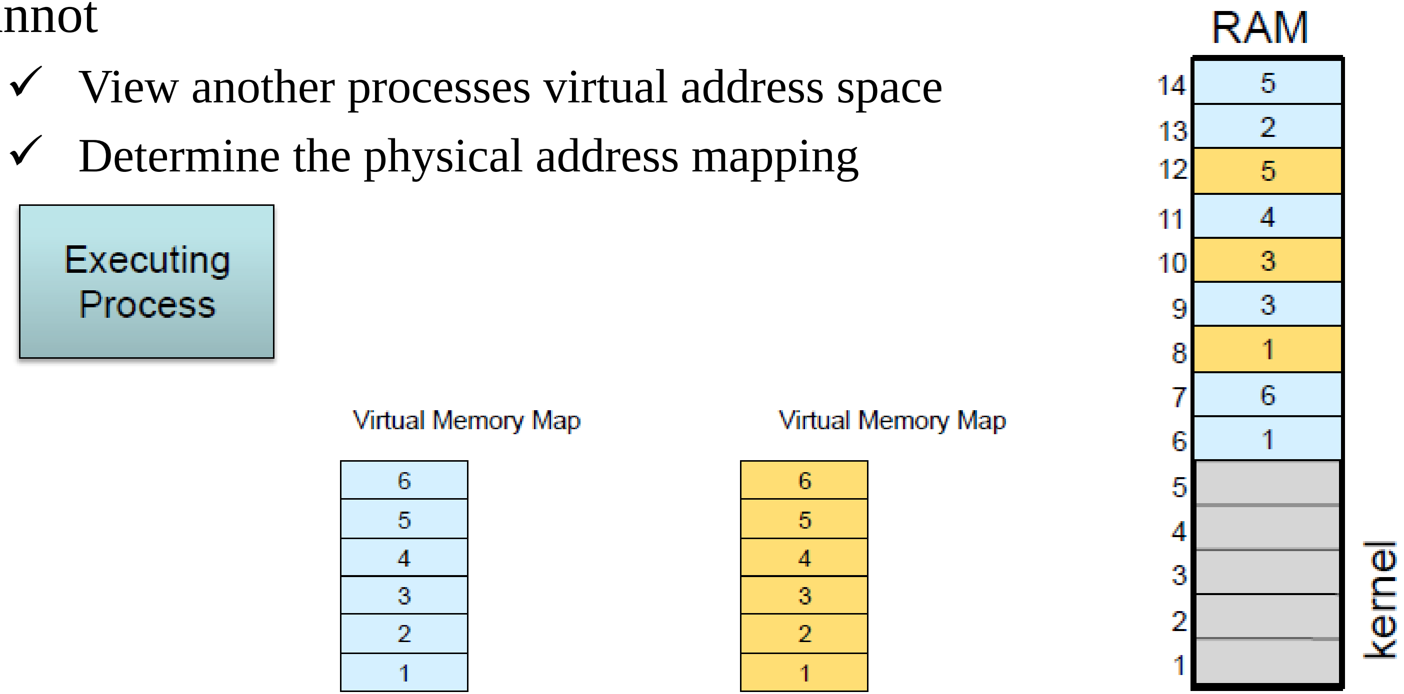
- Types of Inter-process Communication in Linux O.S.
  - ✓ Shared Memory
  - ✓ Message Passing
  - ✓ Signal
  - ✓ Pipes
  - ✓ Synchronization Issues and Solutions
- Android IPC Mechanism
  - ✓ IPC: The heart of Android
  - ✓ Design Patterns
  - ✓ Binder IPC Internals
  - ✓ Use case: Graphics

Outline

- Types of Inter-process Communication in Linux O.S.
  - Shared Memory
  - Message Passing
  - Signal
  - Pipes
  - Synchronization Issues and Solutions
- Android IPC Mechanism
  - IPC: The heart of Android
  - Design Patterns
  - Binder IPC Internals
  - Use case: Graphics

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

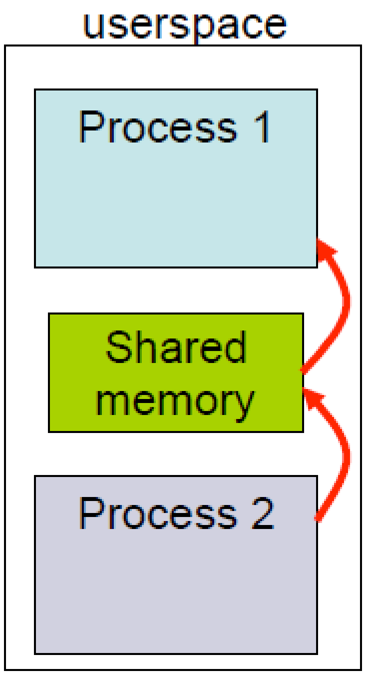


Shared Memory

- 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

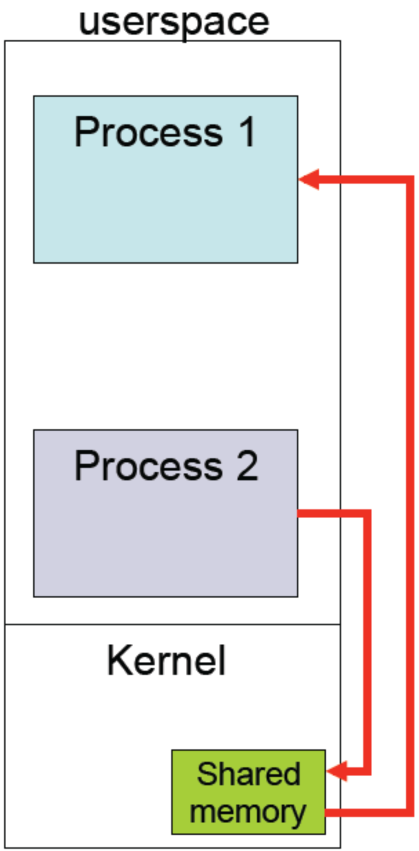
```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#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 = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        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 = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        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

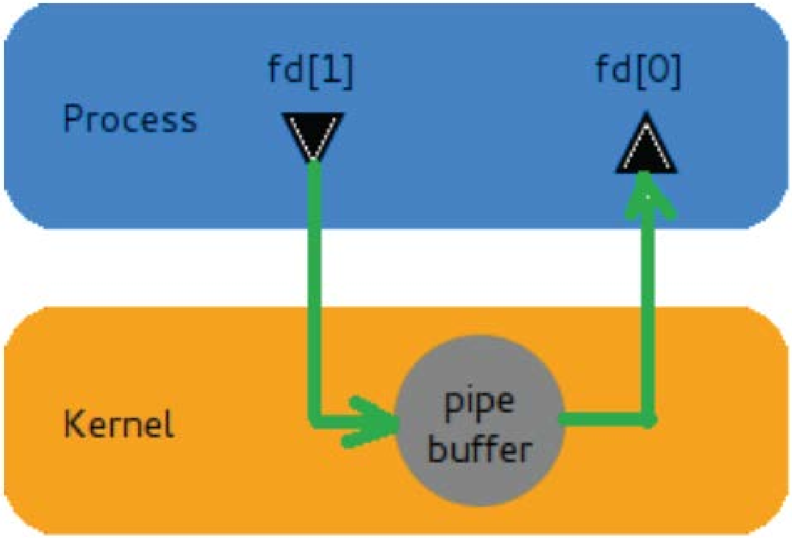


Pipes

- ➢ 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



Pipe Example

- Child process sending a string to parent

Child process

Parent process

```c
int main(void) {
  int pipefds[2];
  char *pin;
  char buffer[5];

  if(pipe(pipefds) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
  }

  pid_t pid = fork();

  if(pid == 0) { // in child process
    pin = "4821\0"; // PIN to send
    close(pipefds[0]); // close read fd
    write(pipefds[1], pin, 5); // write PIN to pipe

    printf("Generating PIN in child and sending to parent...\n");
    sleep(2); // intentional delay
    exit(EXIT_SUCCESS);
  }

  if(pid > 0) { // in main process
    wait(NULL); // wait for child process to finish
    close(pipefds[1]); // close write fd
    read(pipefds[0], buffer, 5); // read PIN from pipe
    close(pipefds[0]); // close read fd

    printf("Parent received PIN '%s'\n", buffer);
  }

  return EXIT_SUCCESS;
}
```

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

Signals: POSIX Standard

Signal	Portable number	Default action	Description
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGALRM	14	Terminate	Alarm clock
SIGBUS	N/A	Terminate (core dump)	Access to an undefined portion of a memory object
SIGCHLD	N/A	Ignore	Child process terminated, stopped, or continued
SIGCONT	N/A	Continue	Continue executing, if stopped
SIGFPE	8	Terminate (core dump)	Erroneous arithmetic operation
SIGHUP	1	Terminate	Hangup
SIGILL	4	Terminate (core dump)	Illegal instruction
SIGINT	2	Terminate	Terminal interrupt signal
SIGKILL	9	Terminate	Kill (cannot be caught or ignored)
SIGPIPE	13	Terminate	Write on a pipe with no one to read it
SIGPOLL	N/A	Terminate	Pollable event
SIGPROF	N/A	Terminate	Profiling timer expired
SIGQUIT	3	Terminate (core dump)	Terminal quit signal
SIGSEGV	11	Terminate (core dump)	Invalid memory reference
SIGSTOP	N/A	Stop	Stop executing (cannot be caught or ignored)
SIGSYS	N/A	Terminate (core dump)	Bad system call
SIGTERM	15	Terminate	Termination signal
SIGTRAP	5	Terminate (core dump)	Trace/breakpoint trap

- Portable number
  - ✓ For most signals the corresponding signal number is implementation-defined.
- Actions
  - ✓ **Terminate** – Abnormal termination of the process.
  - ✓ **Terminate (core dump)** – Abnormal termination of the process. Additionally, implementation-defined abnormal termination actions, such as creation of a core file, may occur

Signals: Example

```
#include <stdio.h>
#include <signal.h>
typedef void (*signal_handler)(int);

void signal_handler_fun(int signum) {
	printf("catch signal %d\n", signum);
}

int main(int argc, char *argv[]) {
	signal(SIGINT, signal_hander_fun);
	while(1);
	return 0;
}
```

- Catch INT signal
  - ✓ When `CTRL + C` are pressed

Results

```
catch signal 2
catch signal 2
catch signal 2
catch signal 2
```

Synchronization Problem

Shared variable

```
int counter=5;
```

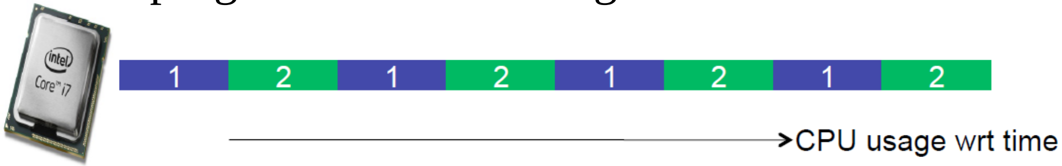
Program 0

```
{
    *
    *
    Counter ++;
    *
}
```

Program 1

```
{
    *
    *
    Counter --;
    *
}
```

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



Synchronization Problem

Shared variable

```
int counter=5;
```

Program 0

```
{
     *
     *
     Counter ++;
     *
}
```

Program 1

```
{
     *
     *
     Counter --;
     *
}
```

- What is the value of counter?
  - ✓ Expected to be 5
  - ✓ But could also be 4 and 6

Synchronization Problem

Program 0

```
{
    *
    *
    Counter ++;
    *
}
```

Shared variable

```
int counter=5;
```

Program 1

```
{
    *
    *
    Counter --;
    *
}
```

R1 ← counter
R1 ← R1 + 1
counter ←R1
context switch
R2 ← counter
R2 ← R2 - 1
counter ←R2

counter = 5

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

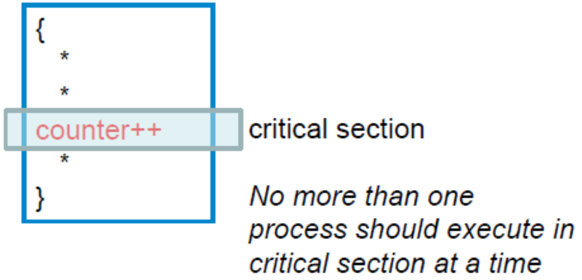
counter = 6

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

counter = 4

Recall: Race Conditions

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



Race Conditions in Multicore

shared variable

```
int counter=5;
```

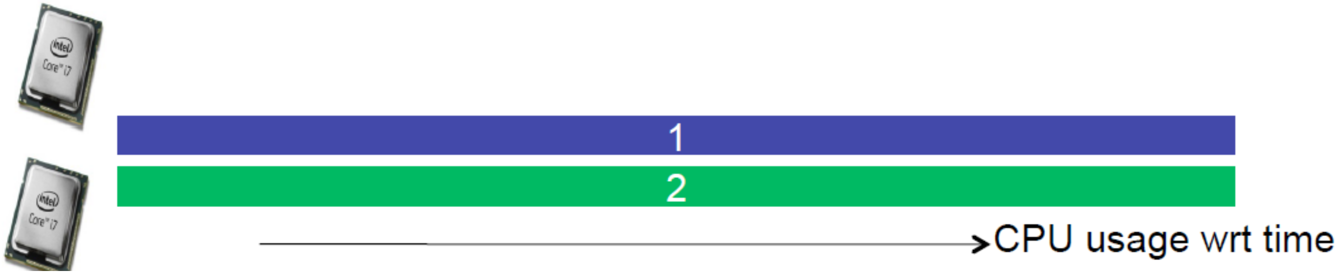
program 0

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

program 1

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

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



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

shared variable

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

program 0

```
{
   *
   *
lock(L)
counter++
unlock(L)
   *
}
```

program 1

```
{
   *
   *
lock(L)
counter--
unlock(L)
   *
}
```

- 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

Software Solution: Peterson Algo.

- 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

```
//flag[] is boolean array; and turn is an integer
flag[0]   = false;
flag[1]   = false;
int turn;
```

```
P0: flag[0] = true;
    turn = 1;
    while (flag[1] == true && turn == 1)
    {
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[0] = false;
```

```
P1: flag[1] = true;
    turn = 0;
    while (flag[0] == true && turn == 0)
    {
        // busy wait
    }
    // critical section
    ...
    // end of critical section
    flag[1] = false;
```

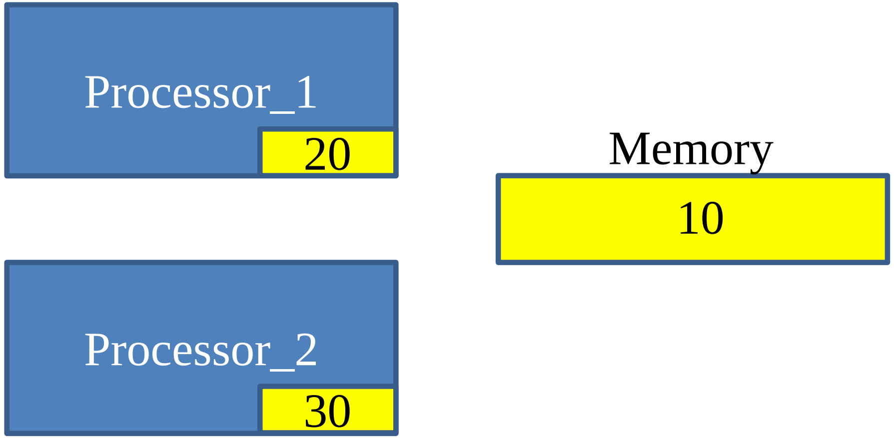
Hardware Solution: Intel xchg Instruction

- Write to a memory location, return its old value

atomic

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

equivalent software representation (the entire function is executed atomically)



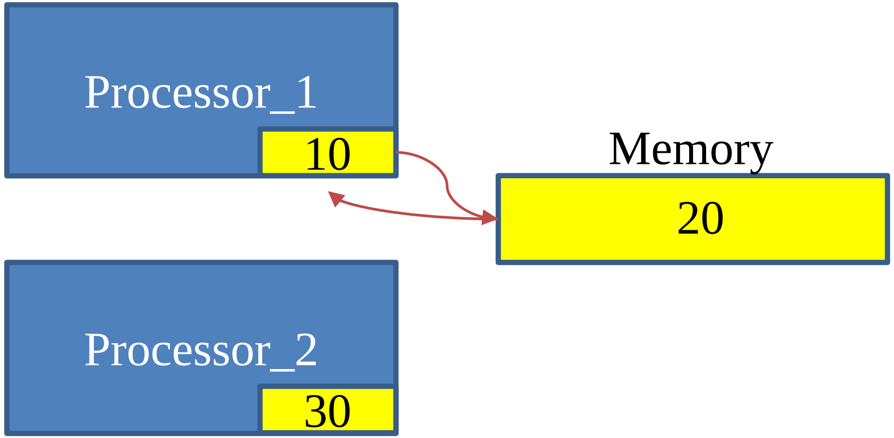
Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

Hardware Solution: Intel xchg Instruction

- Write to a memory location, return its old value


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

equivalent software representation (the entire function is executed atomically)



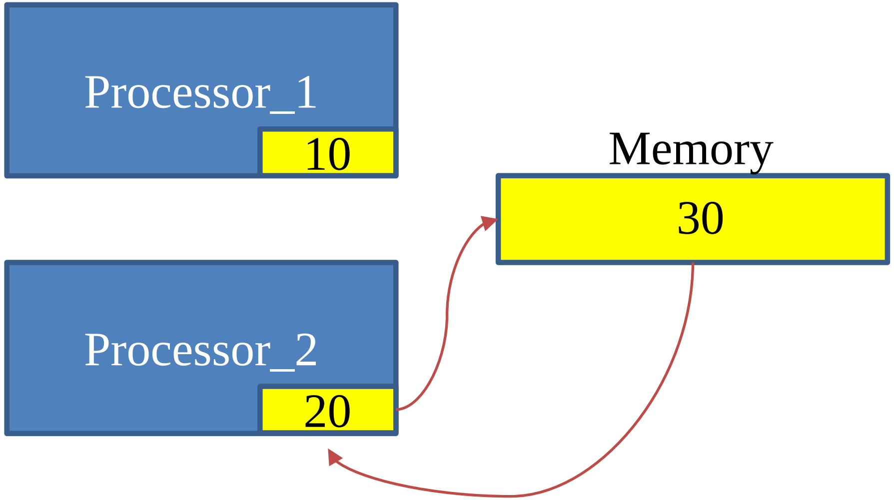
Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

Hardware Solution: Intel xchg Instruction

➢ Write to a memory location, return its old value


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

equivalent software representation (the entire function is executed atomically)



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.

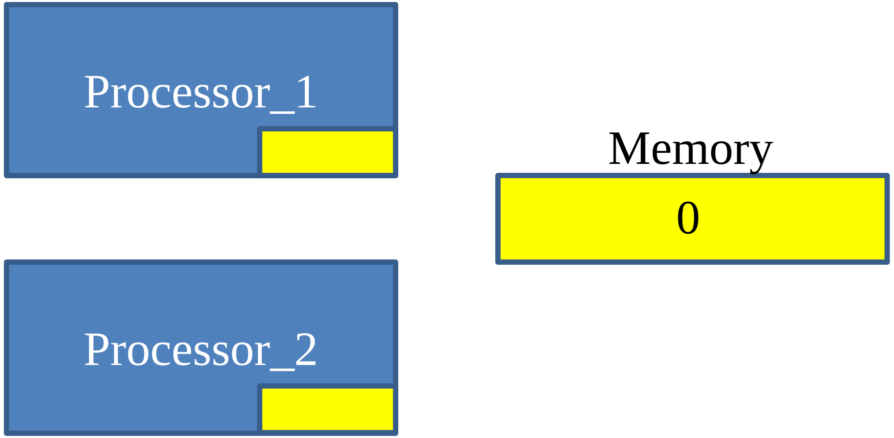
Hardware Solution: Intel xchg Instruction

- Write to a memory location, return its old value

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;
}
```

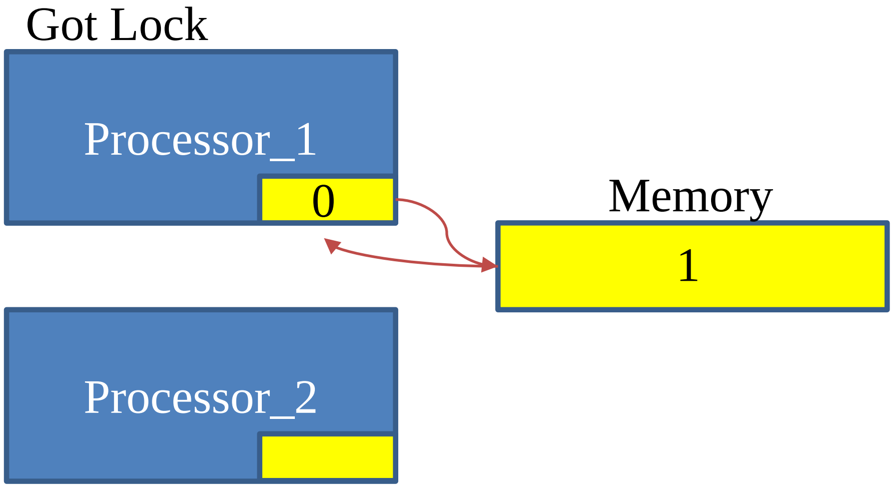
Hardware Solution: Intel xchg Instruction

- Write to a memory location, return its old value

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;
}
```

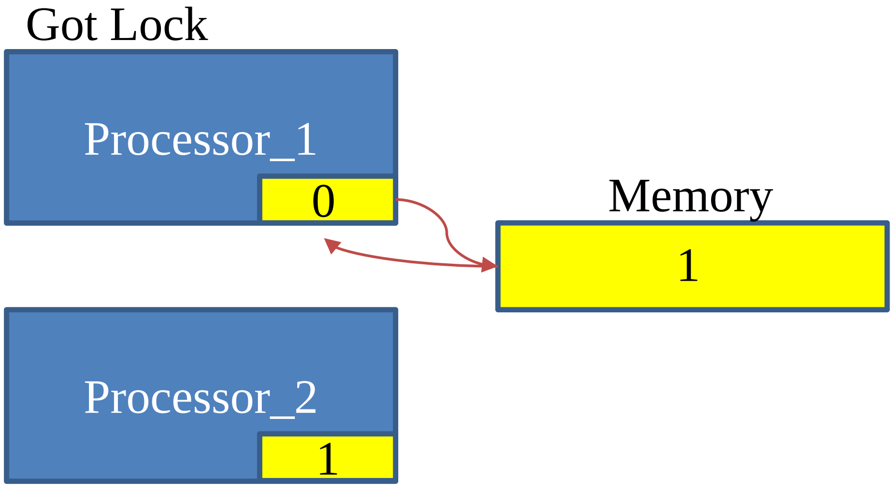
Hardware Solution: Intel xchg Instruction

- Write to a memory location, return its old value

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;
}
```

Hardware Solution: Intel xchg Instruction

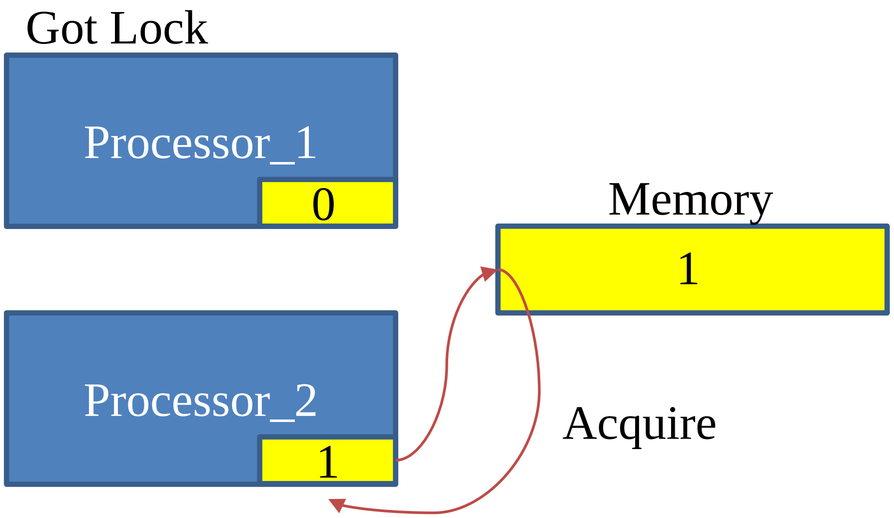
- Write to a memory location, return its old value

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;
}
```



Hardware Solution: Intel xchg Instruction

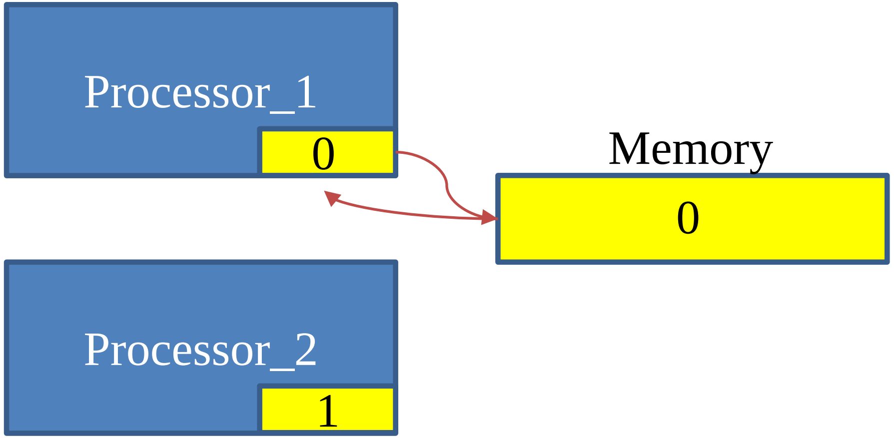
- Write to a memory location, return its old value

Note %eax is returned

typical usage :

`xchg reg, mem`

Release Lock



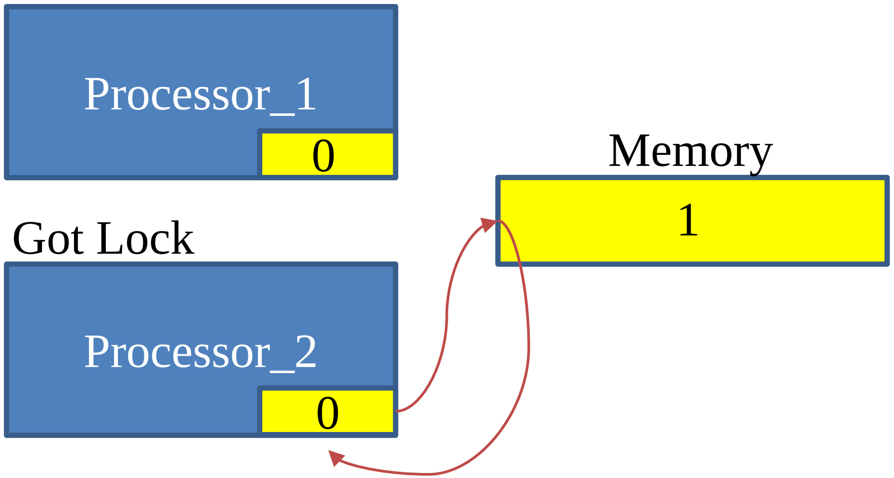

```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

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

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

Hardware Solution: Intel xchg Instruction

- Write to a memory location, return its old value

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;
}
```

Hardware Solution: Intel xchg Instruction

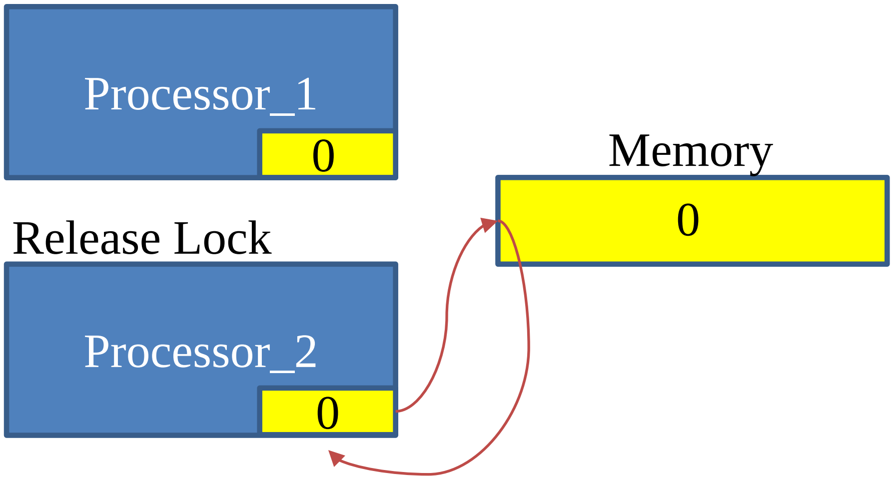
- ➢ Write to a memory location, return its old value

Note %eax is returned

typical usage :

```
xchg reg, mem
```

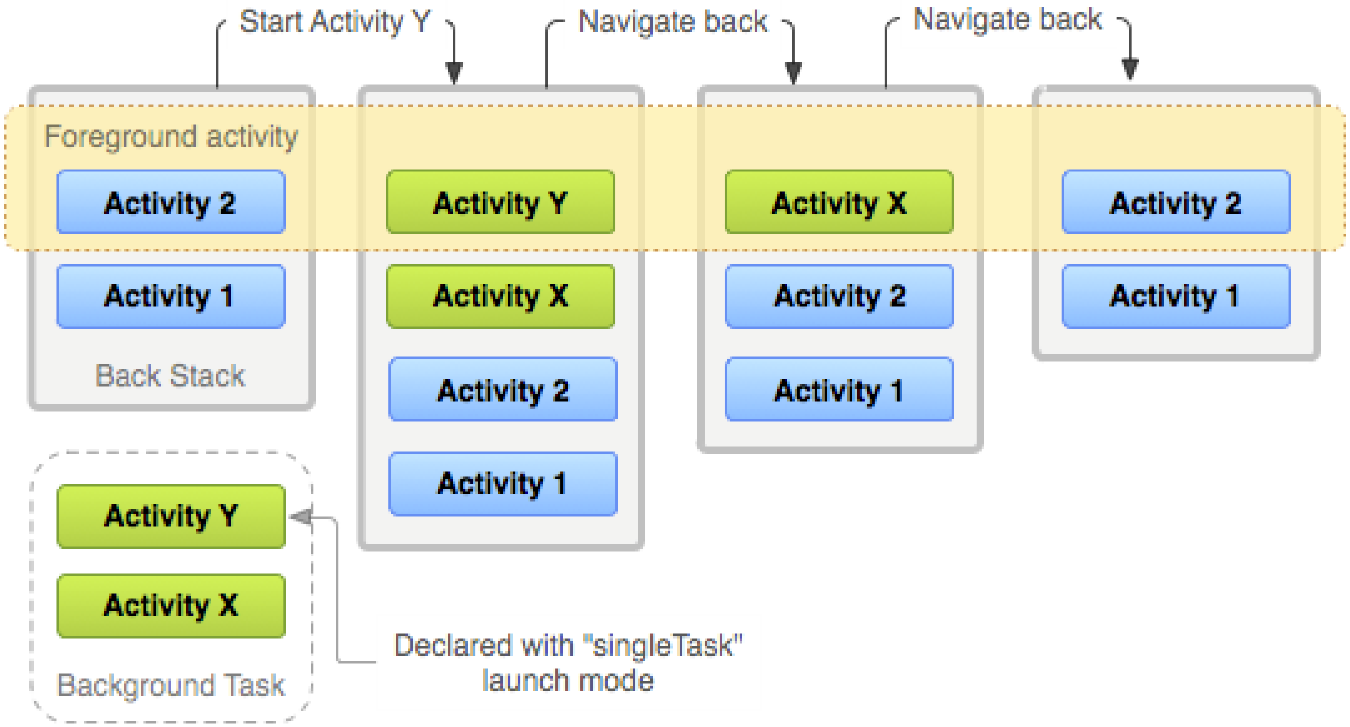
```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

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

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



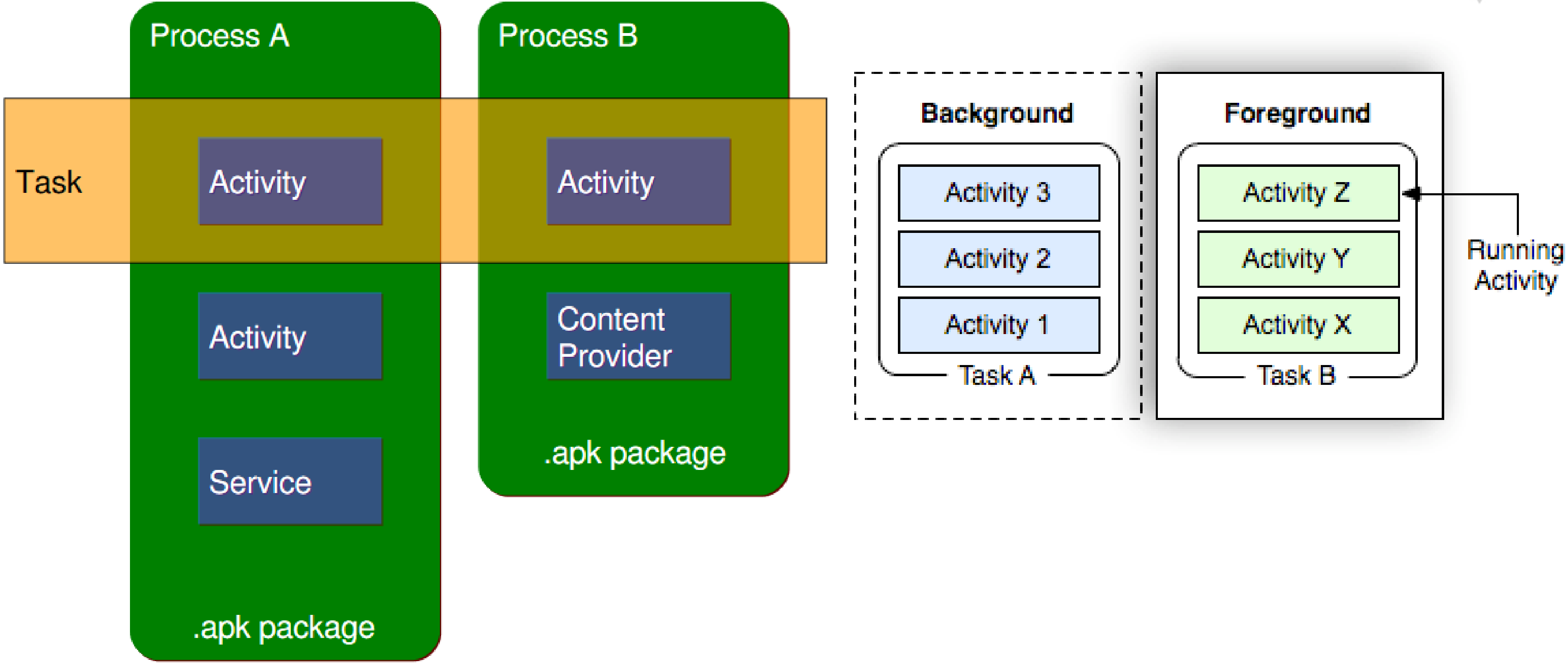
Outline

- Types of Inter-process Communication in Linux O.S.
  - ✓ Shared Memory
  - ✓ Message Passing
  - ✓ Signal
  - ✓ Pipes
  - ✓ Synchronization Issues and Solutions
- Android IPC Mechanism
  - ✓ IPC: The heart of Android
  - ✓ Design Patterns
  - ✓ Binder IPC Internals
  - ✓ Use case: Graphics

Android Task, Process, and Activity

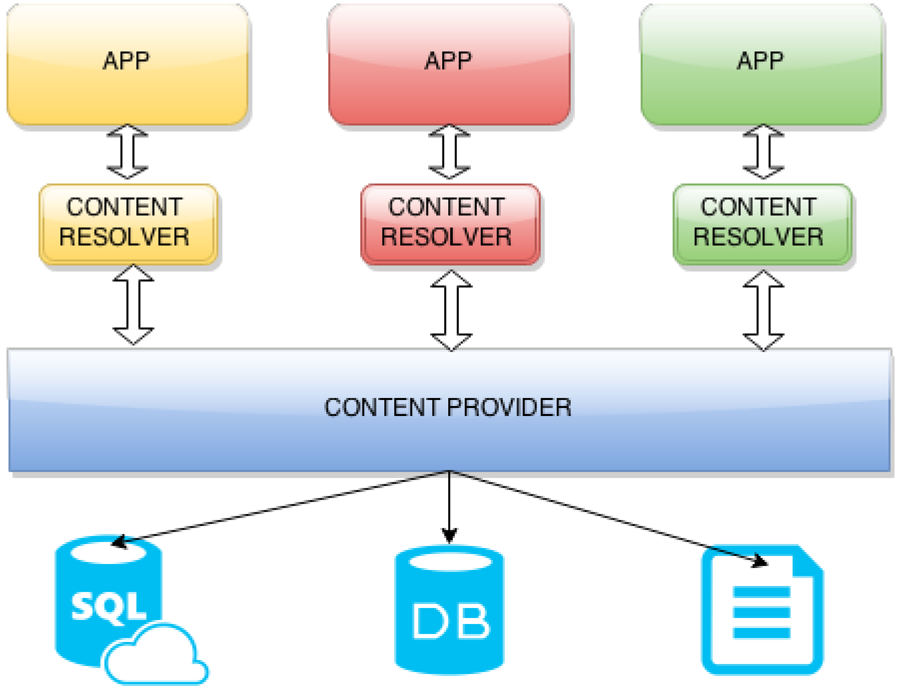


Android Task, Process, and Activity



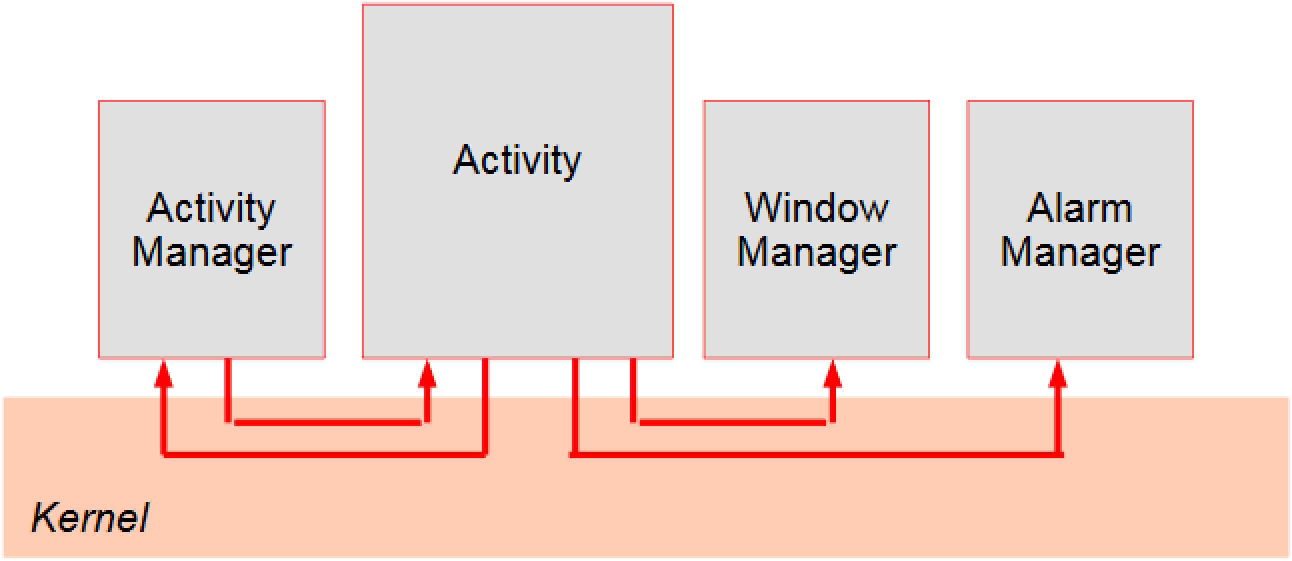
Component View

- Different component types
  - Activity
    - An activity is the **entry point** for **interacting** with the user.
  - Service
    - A **general-purpose** entry point for keeping an app running in the **background** for all kinds of reasons.
  - Content provider
    - Any **persistent storage location** that your app can access.
  - Broadcast receiver



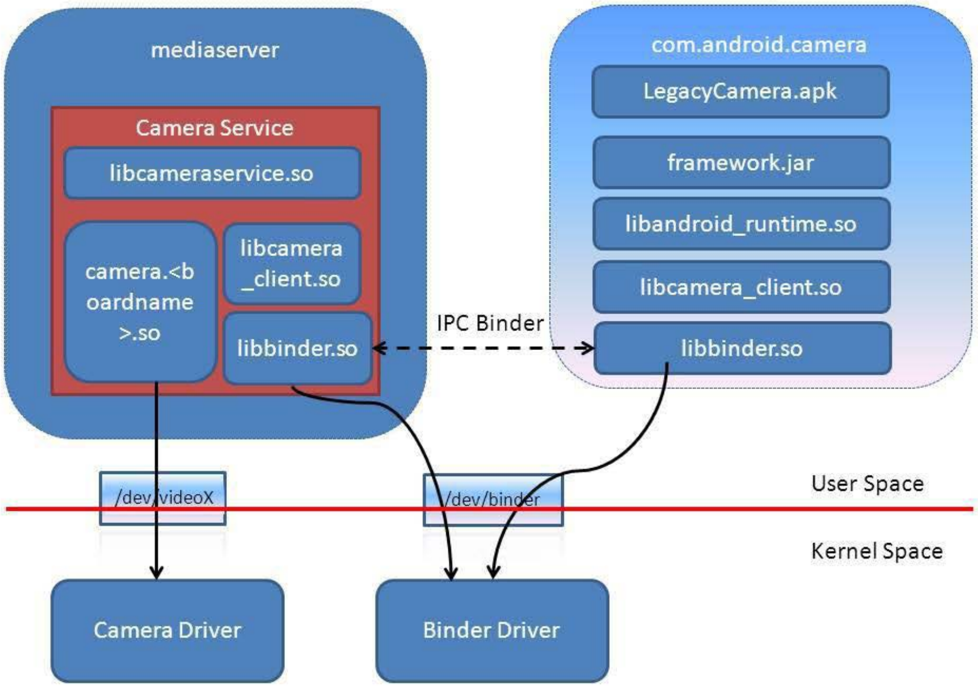
Inter-Process Communication

- Simple view



Revisit: Why IPC?

- Each process has its own address space
- Provides data isolation
- Prevents harmful direct interaction between two different processes
  - Sometimes, communication between processes is required for modularization

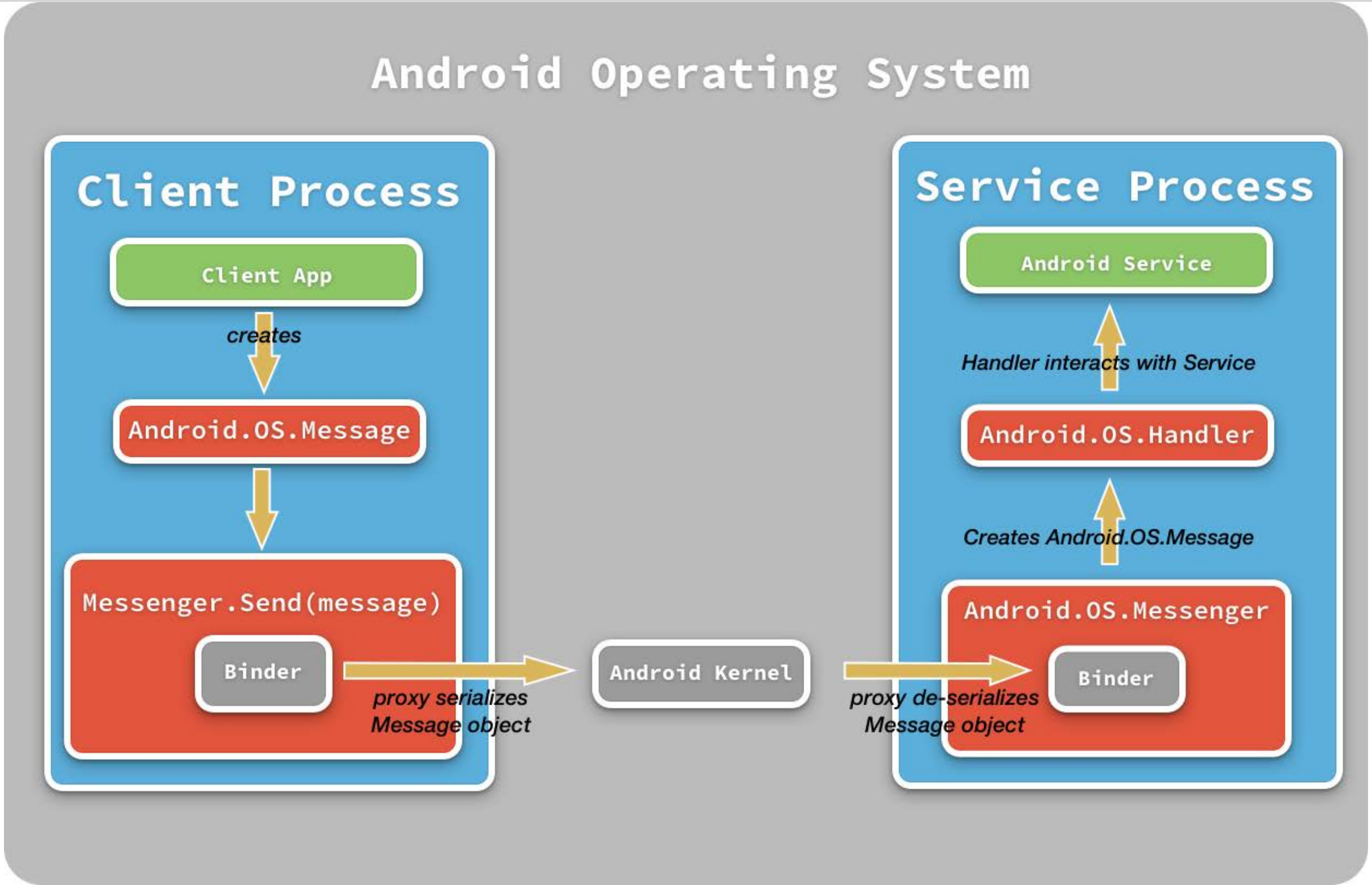


Revisit: IPC Mechanisms

- In GNU/Linux
  - Signal
  - Pipe
  - Socket
  - Semaphore
  - Message queue
  - Shared memory

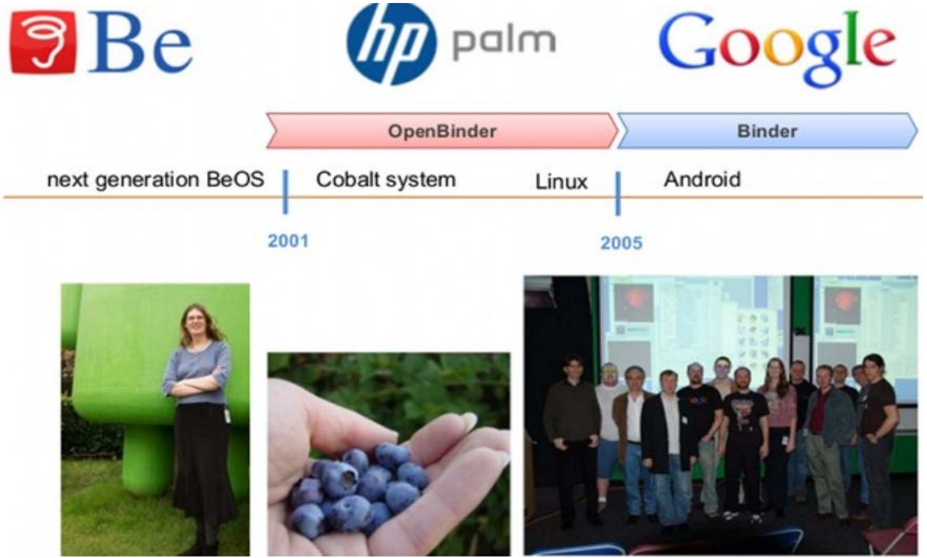
- In Android
  - Binder: lightweight RPC (Remote Procedure Communication) mechanism

Lightweight RPC



Binder History

- Developed under the name ***OpenBinder*** by Palm Inc. under the leadership of Dianne Hackborn
- Android Binder is the customized **re-implementation** of OpenBinder, which provides **bindings to functions and data** from one execution environment to another



Background Problems

- Applications and Services may run in **separate processes** but must communicate and **share data**

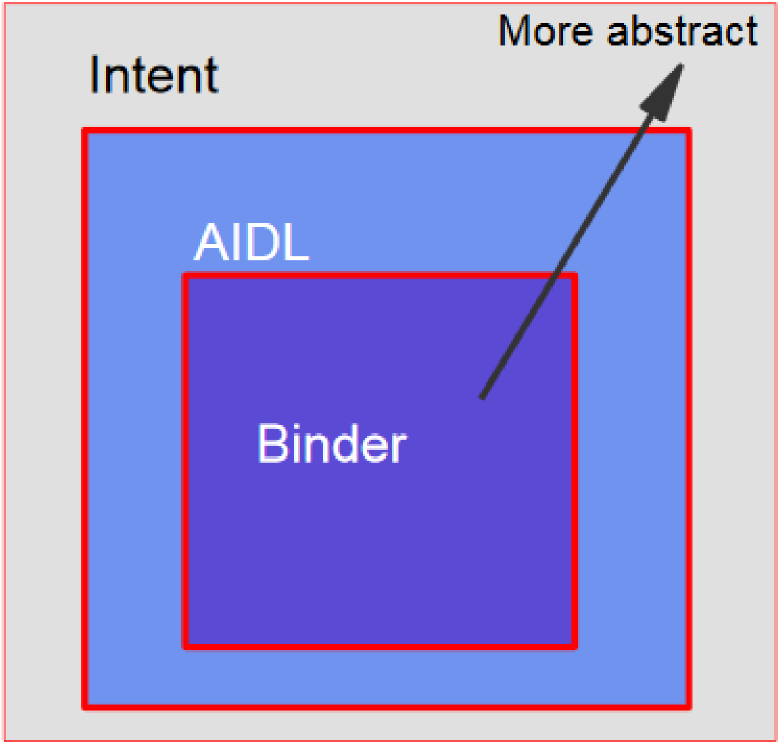
- IPC can introduce significant processing overhead and **security holes**

Binder: Android's Solution

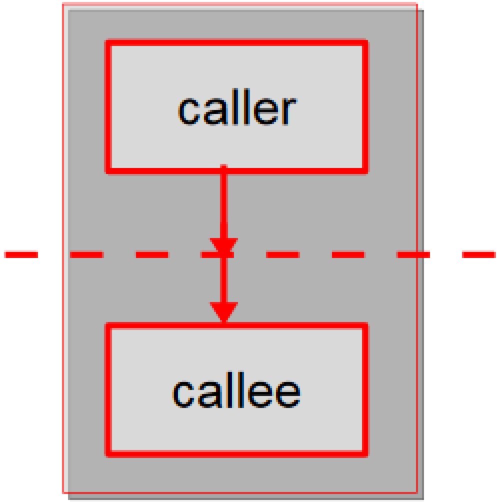
- Driver to facilitate **inter-process communication**
- High performance through **shared memory**
- Per-process **thread pool** for processing requests
- **Reference counting**, and mapping of object references across processes
- **Synchronous calls** between processes

IPC Abstraction

- Intent
  - The highest level abstraction
- Inter process method invocation
  - **AIDL**: Android Interface Definition Language
- Binder: kernel driver
- ashmem: shared memory



Method Invocation



In the same process

```
void func(int a, int b)
{
    ...
}
int main(void)
{
    func(100,200);
    ...
}
```

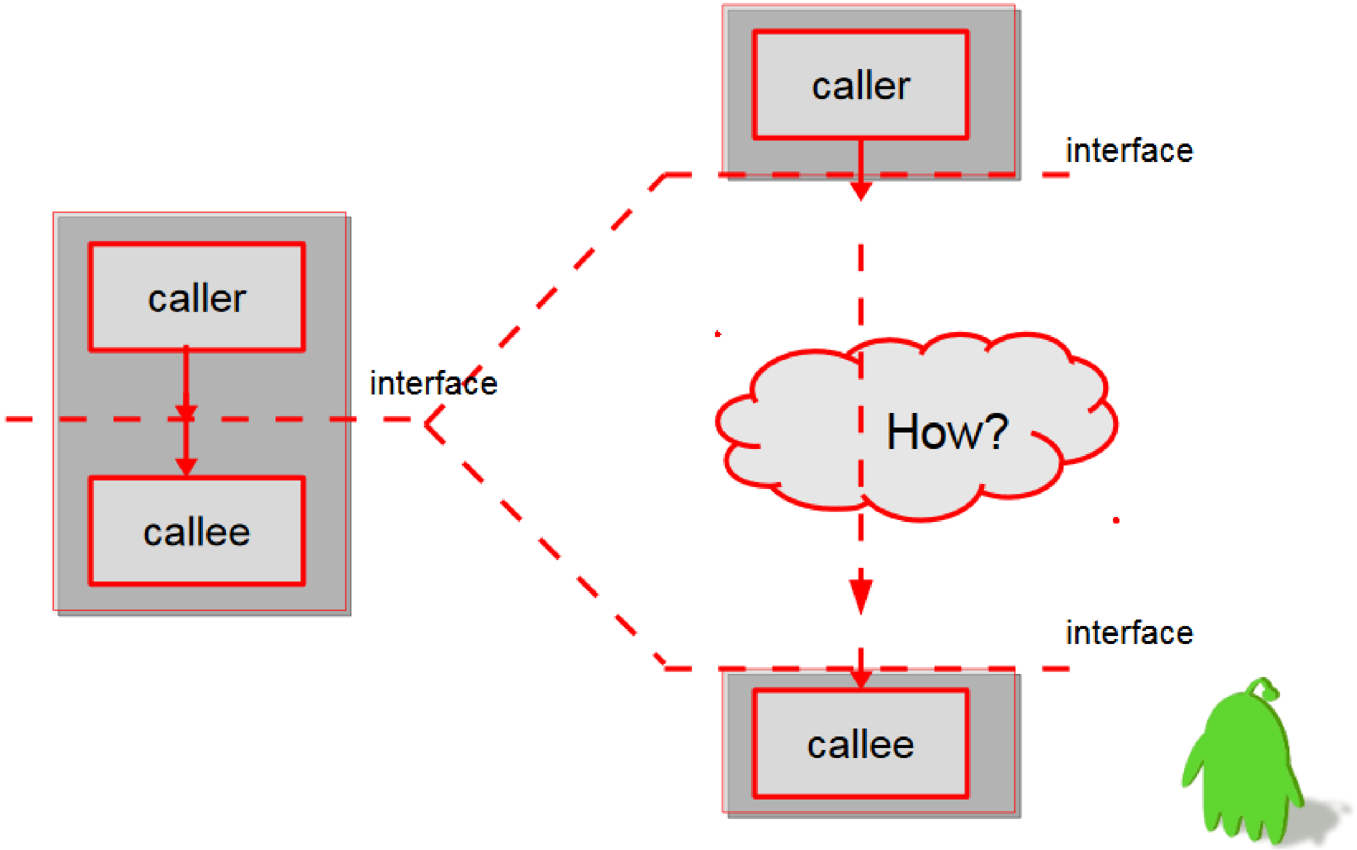
callee

parameters

caller

arguments

Inter-process Method Invocation



Inter-process Method Invocation