# Memory and Interprocess Communication

# Fragmentation

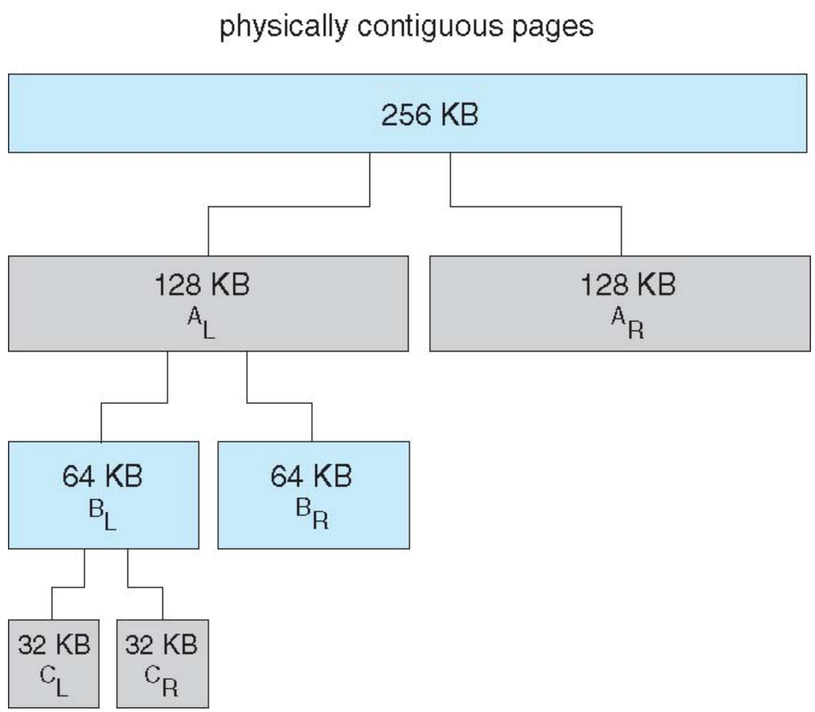
- External Fragmentation
  - Memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation
  - Allocated memory may be slightly larger than requested memory
  - The size difference is memory internal to a partition, but not being used

# Fragmentation

- Fixed partitions suffer from
  - Internal fragmentation
- Variable partitions suffer from
  - External fragmentation

# Buddy System

- Memory allocated using power-of-2 allocator
  - Satisfy requests in units of size power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available



# Buddy System

- Approach
  - Minimum allocation size = smallest frame
  - Use a bitmap to monitor frame use
  - Maintain freelist for each possible frame size
    - power of 2 frame sizes from min to max
  - Initially one block = entire buffer
  - If two neighboring frames (“buddies”) are free, combine them and add to next larger freelist

# Buddy System Example

| |
|---|
| 128 Free |

# Buddy System Example

Process A requests 16

| |
|---|
| 128 Free |

| | |
|---|---|
| 64 Free | 64 Free |

| | | |
|---|---|---|
| 32 Free | 32 Free | 64 Free |

| | | | |
|---|---|---|---|
| 16 A | 16 Free | 32 Free | 64 Free |

# Buddy System Example

Process B requests 32

| | | | |
|---|---|---|---|
| 16 A | 16 Free | 32 B | 64 Free |

# Buddy System Example

Process C requests 8

| | | | |
|---|---|---|---|
| 16 A | 16 Free | 32 B | 64 Free |

| | | | | |
|---|---|---|---|---|
| 16 A | 8 C | 8 | 32 B | 64 Free |

# Buddy System Example

Process A exits

| | | | | |
|---|---|---|---|---|
| 16 Free | 8 C | 8 | 32 B | 64 Free |

# Buddy System Example

Process C exits

| | | | | |
|---|---|---|---|---|
| 16 Free | 8 | 8 | 32 B | 64 Free |

| | | | |
|---|---|---|---|
| 16 Free | 16 Free | 32 B | 64 Free |

| | | |
|---|---|---|
| 32 Free | 32 B | 64 Free |

- Advantage
  - Minimizes external fragmentation
- Disadvantage
  - Internal fragmentation when not 2^n request

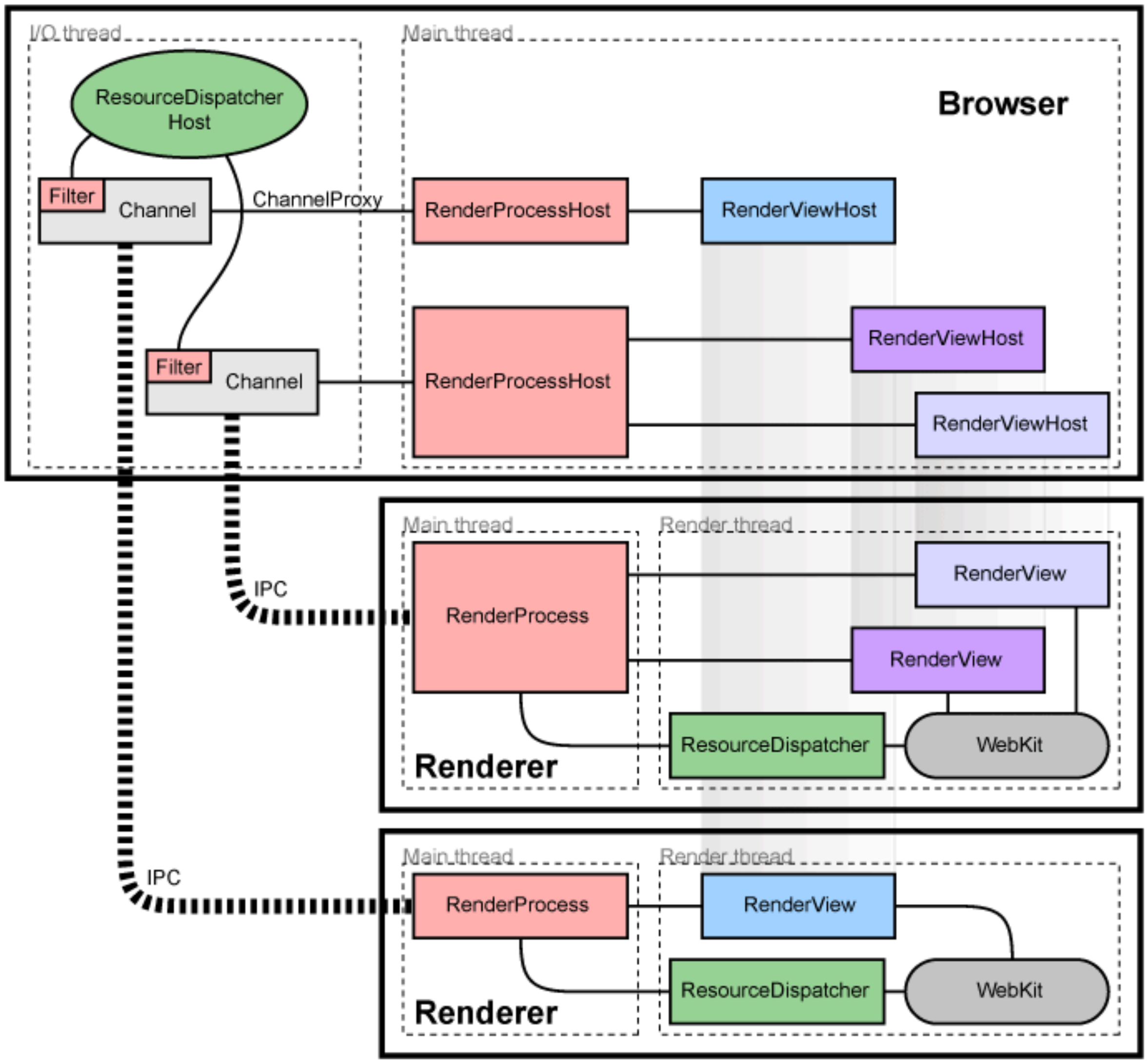
# Interprocess Communciation

- What is IPC?
  - Mechanisms to transfer data between processes
- Why is it needed?
  - Not all important procedures can be easily built in a single process

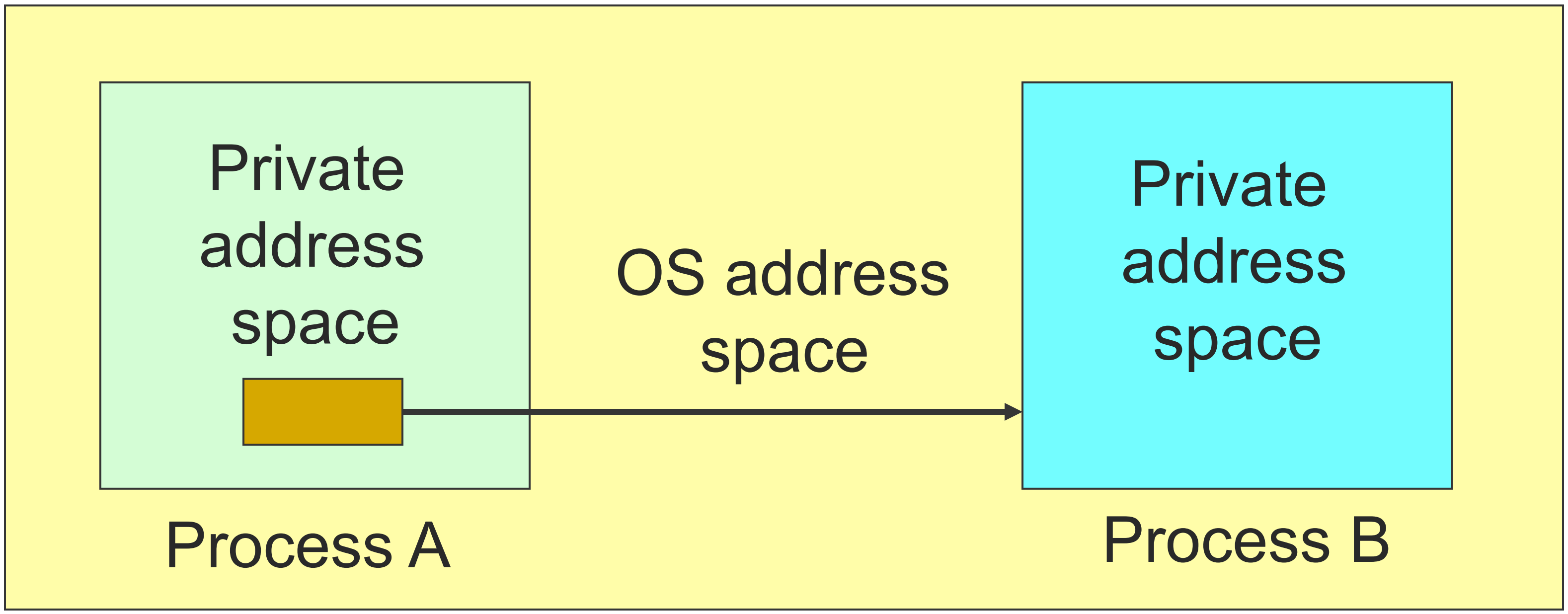
# Interprocess Communication

- Cooperating processes
  - Can affect or be affected by other processes, including sharing data
  - Benefits
    - Information sharing
    - Computation speedup
    - Modularity
    - Convenience
- Need interprocess communication (IPC)
  - Shared memory
  - Message passing

# Google Chrome architecture (figure borrowed from Google)



# IPC Communications Model




- Each process has a private address space
- No process can write to another process's space
- How can we get data from process A to process B?

# IPC Solutions

- Two options
  - Support some form of shared address space
    - Shared memory
  - Use OS mechanisms to transport data from one address space to another
    - Files, messages, pipes

# Shared Memory

- Processes share the same segment of memory directly
  - Memory is mapped into the address space of each sharing process
- Mutual exclusion must be provided by processes using the shared memory

# POSIX Shared Memory

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int
   shmflg);
```

- Create shared memory segment

```
id = shmget(key, size, 0644 | IPC_CREAT);
```

```
void *shmat(int shmid, const void
   *shmaddr, int shmflg);
```

- Access to shared memory requires an attach

```
shared_memory = (char *) shmat(id, (void
   *) 0, 0);
```

# POSIX Shared Memory

- Write to the shared memory using normal system call

```
sprintf(shared_memory, "Writing to shared
   memory");
```

```
int shmdt(const void *shmaddr);
```

- Detach the shared memory from its address space

```
shmdt(shared_memory);
```

# Shared Memory example

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

#define SHM_SIZE 1024  /* a 1K shared memory segment */

int main(int argc, char *argv[]) {
    key_t key;
    int shmid;
    char *data;
    int mode;
```

# Shared Memory example

```
/* make the key: */
if ((key = ftok("shmdemo.c", 'R')) == -1) {
    perror("ftok");
    exit(1);
}
/* connect to (and possibly create) the segment: */
if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
    perror("shmget");
    exit(1);
}
/* attach to the segment to get a pointer to it: */
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1)) {
    perror("shmat");
    exit(1);
}
```

# Shared Memory example

```
    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```

# Message-based IPC

- Message system
  - Enables communication without resorting to shared variables
- To communicate, processes P and Q must
  - Establish a communication link between them
  - Exchange messages
- Two operations
  - send(message)
  - receive(message)

# Direct Message Passing

- Processes must name each other explicitly
  - `send (P, message)`
    - Send a message to process `P`
  - `receive(Q, message)`
    - Receive a message from process `Q`
  - `receive(&id, message)`
    - Receive a message from any process
- Link properties
  - Established automatically
  - Associated with **exactly** one pair of processes
  - There exists **exactly** one link between each pair
- Limitation
  - **M**ust know the name or ID of the process(es)

# Indirect Message Passing

- Process names a mailbox (or port)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Link properties
  - Established only if processes share a common mailbox
  - May be associated with **many** processes
  - Each pair of processes may share **multiple** links
  - Link may be unidirectional or bi-directional

# Mailbox Ownership

- Process
  - Only the owner receives messages through mailbox
  - Other processes only send.
  - When process terminates, any “owned” mailboxes are destroyed.
- System
  - Process that creates mailbox owns it (and so may receive through it) but may transfer ownership to another process.

# Indirect Message Passing

- Mailboxes are a resource
  - Create and Destroy
- Primitives
  - **`send(A, message)`**
    - Send a message to mailbox **`A`**
  - **`receive(A, message)`**
    - Receive a message from mailbox **`A`**

# Indirect Message Passing

- Mailbox sharing
  - **`P1`**, **`P2`**, and **`P3`** share mailbox **`A`**
  - **`P1`**, sends; **`P2`** and **`P3`** receive
  - Who gets the message?
- Options
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to arbitrarily select the receiver and notify the sender

# IPC and Synchronization

- Blocking
  - Blocking send
    - Sender blocks until the message is received
  - Blocking receive
    - Receiver blocks until a message is available
- Non-blocking
  - Non-blocking send
    - Sender sends the message and continues
  - Non-blocking receive
    - Receiver receives a valid message or null

# Buffering

- IPC message queues
  1. Zero capacity
     - No messages may be queued
     - Sender must wait for receiver
  2. Bounded capacity
     - Finite buffer of n messages
     - Sender blocks if link is full
  3. Unbounded capacity
     - Infinite buffer space
     - Sender never blocks

# Buffering

- Is a buffer needed?

```
P1:   send(P2, x)        P2: receive(P1, x)
      receive(P2, y)         send(P1, y)
```

- Is a buffer needed?

```
P1:   send(P2, x)        P2:     send(P1, x)
      receive(P2, y)         receive(P1, y)
```

# Example: Message Passing

```
void Producer() {
    while (TRUE) {
    /* produce item */
    build_message(&m, item);
    send(consumer, &m);
    receive(consumer, &m); /* wait for ack */
    }
}

void Consumer {
    while(TRUE) {
        receive(producer, &m);
        extract_item(&m, &item);
        send(producer, &m); /* ack */
        /* consume item */
    }
}
```

# UNIX Pipes

```
#include <unistd.h>
int pipe(int fildes[2]);
```

- Creates a message pipe
  - Anything can be written to the pipe, and read from the other end in the order it came in
  - OS enforces mutual exclusion: only one process at a time
  - Accessed by a file descriptor, like an ordinary file
  - Processes sharing the pipe must have same parent in common
- Returns a pair of file descriptors
  - **`fildes[0]`** is connected to the write end of the pipe
  - **`fildes[1]`** is connected to the read end of the pipe

# UNIX Pipe Example

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

# UNIX Pipe Example: `ls | wc -l`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    int pfds[2];

    pipe(pfds);
        if (!fork()) {
            close(1);       /* close stdout */
            dup(pfds[1]);   /* make stdout pfds[1] */
            close(pfds[0]); /* don't need this */
            execlp("ls", "ls", NULL);
        } else {
            close(0);        /* close stdin */
            dup(pfds[0]);    /* make stdin pfds[0] */
            close(pfds[1]); /* don't need this */
            execlp("wc", "wc", "-l", NULL);
        }
        return 0;
}
```

# FIFOs

- A pipe disappears when no process has it open
- FIFOs = named pipes
  - Special pipes that persist even after all the processes have closed them
  - Actually implemented as a file

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkfifo("/home/cnd/mod_done",
                S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH);
```

# FIFO Example: Producer-Consumer

- Producer
  - Writes to fifo
- Consumer
  - Reads from fifo
  - Outputs data to file
- Fifo
  - Ensures atomicity of write

# FIFO Example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"

int main (int argc, char *argv[]) {
    int requestfd;

    if (argc != 2) { /* name of consumer fifo on the command line */
        fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
        return 1;
    }
```

# FIFO Example

```
/* create a named pipe to handle incoming requests */
if ((mkfifo(argv[1], S_IRWXU | S_IWGRP| S_IWOTH) == -1)
     && (errno != EEXIST)) {
     perror("Server failed to create a FIFO");
     return 1;
}

/* open a read/write communication endpoint to the pipe */
if ((requestfd = open(argv[1], O_RDWR)) == -1) {
     perror("Server failed to open its FIFO");
     return 1;
}
/* Write to pipe like you would to a file */
...
}
```