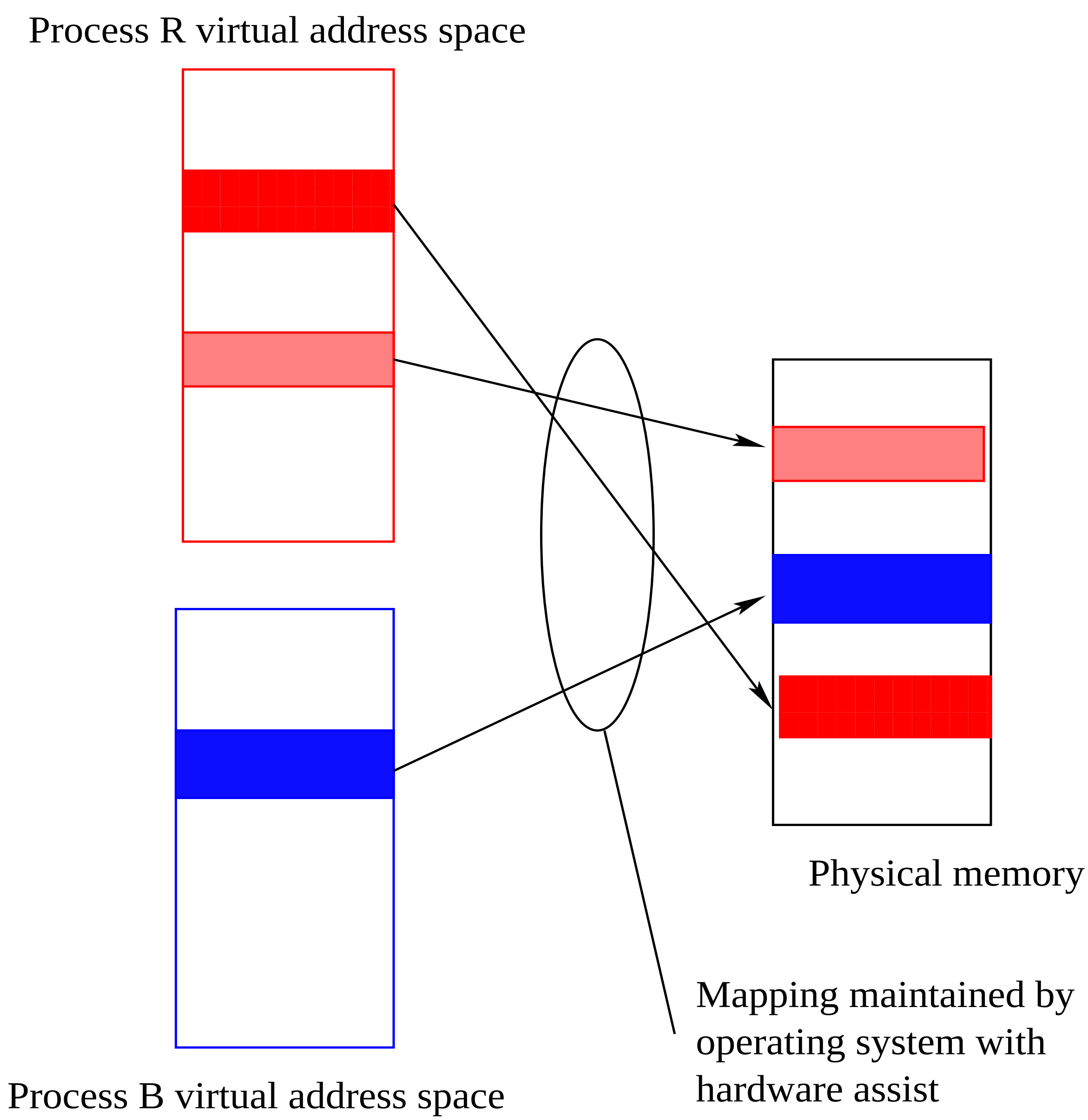
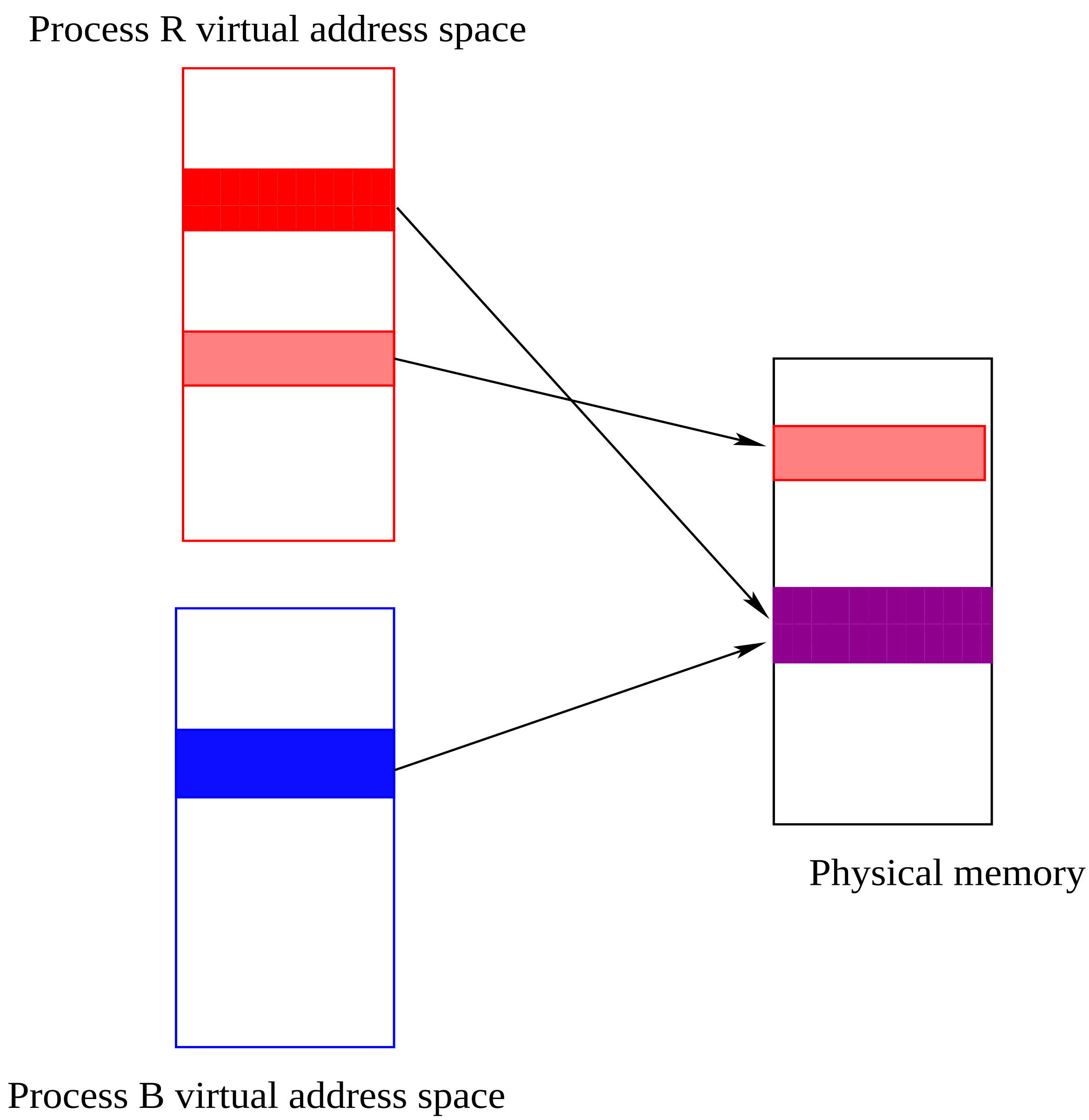
7. Shared Memory & Semaphores

Ordinary virtual memory (no sharing):



Depiction

Shared memory:



Shared Memory API

UNIX has had several shared memory APIs

How N processes use "`shm`" interface ...

1. Some process creates shared memory segment with `shmget(2)`

2. That process somehow communicates segment ID to other N-1 processes

3. All processes "attach" with `shmat(2)`

4. All processes access shared physical segment via their virtual addresses

5. When a process is finished, it "detaches" with `shmdt(2)`

shmget

1. Create shared memory segment with `shmget(2)`

```
int shmget(..., size, ...)
```

Returned `int` is ID of shared memory segment

Segment is initialized to all zeroes

shmat

3. Other processes "attach" with `shmat(2)`

`shmat(ID, address, ...)`

`address` is location of start of segment in this process's address space

Sharing

4. All processes access shared physical segment via their virtual addresses

If "`start`" is start of shared memory segment in this address space, then `start[0]` is first byte, `start[99]` is 100th byte, etc.

shmdt

5. When a process is finished, it "detaches" with `shmdt(2)`

`shmdt(address)`

Garbage collection: use "control" interface function `shmctl(2)` to indicate whether segment ID remains or is deallocated after last process detaches

Semaphore

Q: How to coordinate access to shared memory?

A: Later in the course we will study the **semaphore** mechanism

Summary

MECHANISM	COMMUNICATION	COORDINATION
Pipe	unidirectional, stream, ancestor/descendant processes	receiver blocks waiting for sender
FIFO (named pipe)	unidirectional, stream, arbitrary processes	receiver blocks waiting for sender
Message queue	unidirectional, discrete messages	receiver blocks waiting for sender
Socket	bidirectional, stream, can be across machines	receiver blocks waiting for sender
Wait for multiple descriptors	N/A	receiver can block or not, receiver can use select(2) to wait for multiple descriptors at once
Signal	one int (signal type)	sender interrupts receiver; receiver's signal handler runs
Shared memory	share arbitrary amount without copying	access to shared memory must be controlled; semaphore is common control mechanism

Moral of the Story

Plain old UNIX (no threads) provides abstractions for concurrent programming

- Schedulable unit: single-threaded process
- Scheduler: operating system
- Scheduling algorithm:
  - Preemptive descheduling
  - Scheduling algorithm: process can provide "advice," but programmer must assume that process can be descheduled at any time
- Inter-process communication:
  - Copy data to other address space: pipe, FIFO, message, socket
  - Shared memory
  - Signal – coordination only, no data communication

Reasons to Use Threads Instead of Processes

1. Need many schedulable units — threads faster to create, destroy, switch

2. Much shared data — inefficient to pass it among address spaces using system calls like `read`, `write`

Thread Creation

```
int pthread_create(pthread_t *ID,
                   pthread_attr_t *attr,
                   void *(*function)(void *),
                   void *arg);
```


Return Values

UNIX system calls:

- Return 0 on success, -1 on any error
- `errno` specifies which error

Pthread library calls:

- Return 0 on success, error code otherwise
- No concept like `errno` — static variable foils reentrancy