Programming for shared memory architectures with processes (Programação em Memória Partilhada com Processos)

Miguel Areias
(based on the slides of Ricardo Rocha)

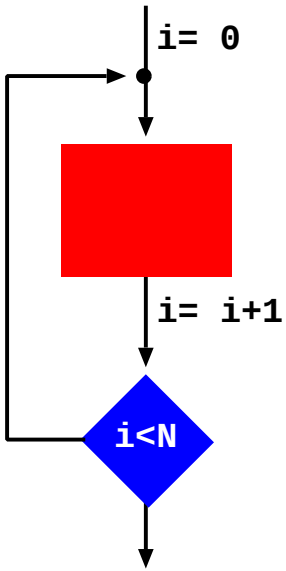
Computer Science Department
Faculty of Sciences
University of Porto

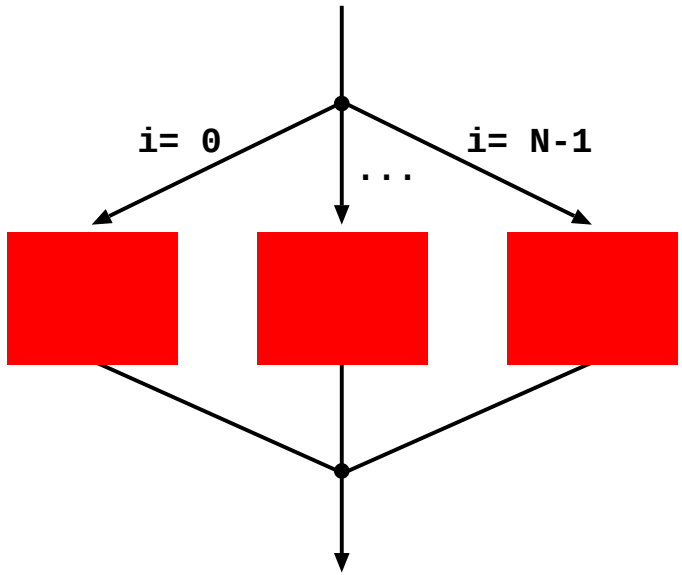
Parallel Computing 2018/2019

Data Parallelism

Data parallelism is one of the simplest techniques that exist to exploit parallelism. The key idea is to execute the same operation over the different components of the data:

- The data is usually organized in multidimensional arrays or matrices.
- Cycles are the main candidates to be parallelized.
- Frequent in scientific and engineering problems.

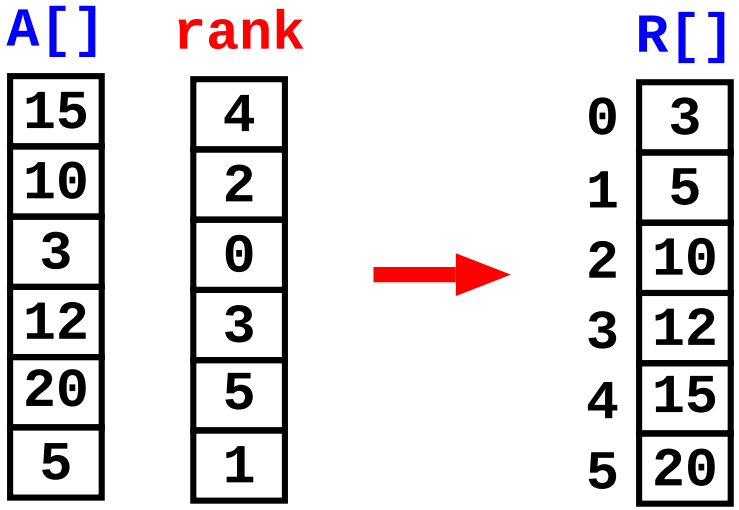




Rank Sort

Give an array `A[N]`, we want to build a new array `R[N]` with the sorted elements of `A[N]`:

- For each element in `A[k]` we will determine its relative position (**rank**) in the array `R[N]`. The position can be obtained by calculating the number of elements in `A[N]` that are lower than `A[k]`.
- As the calculation of the relative position is an independent task, the algorithm can the easily parallelizable.



Rank Sort

```
int A[N], R[N];

main() {
  ...
  for (k = 0; k < N; k++)
    compute_rank(A[k]);
  ...
}

compute_rank(int elem) {
  int i, rank = 0;
  for (i = 0; i < N; i++)
    if (elem > A[i])
      rank++;
  R[rank] = elem;
}
```

Rank Sort

```
int A[N], R[N];

main() {
  ...
  for (k = 0; k < N; k++)
    compute_rank(A[k]);
  ...
}

compute_rank(int elem) {
  int i, rank = 0;
  for (i = 0; i < N; i++)
    if (elem > A[i])
      rank++;
  R[rank] = elem;
}
```

Question: how can we parallelize the rank sort algorithm?

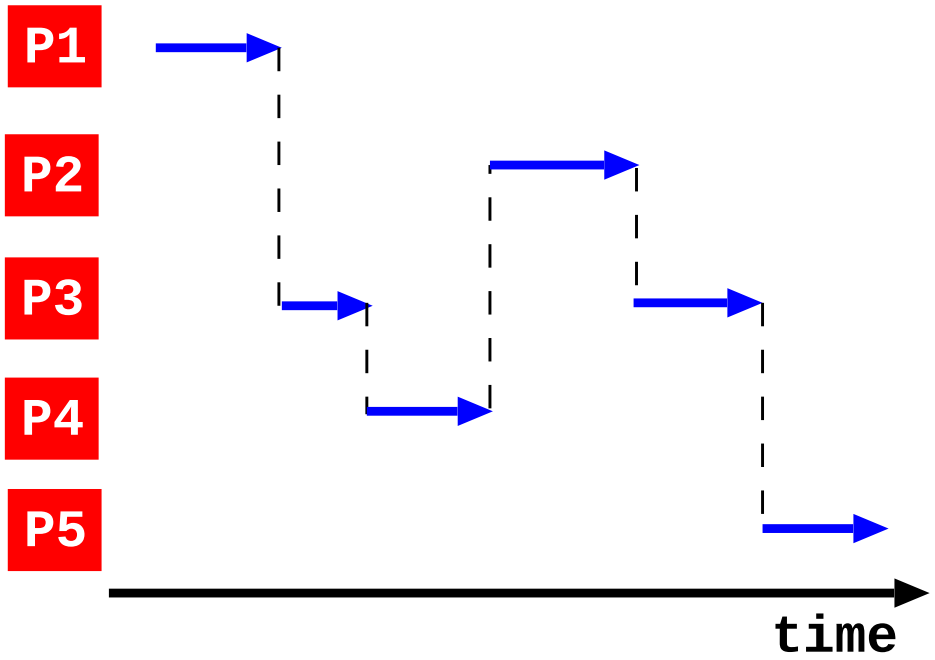
Processes

One process is an **abstraction of a program in execution**, which allows for a program to have multiple instances in execution.

Processes

One process is an **abstraction of a program in execution**, which allows for a program to have multiple instances in execution.

In uni-processor machines, in each instant of execution, only one process is in execution. However, as the processor time is sliced, several processes can be executed in a given fraction of time (higher than an instant). This gives to the user an illusion of parallelism.



Creating Processes

```
pid_t fork(void)
```

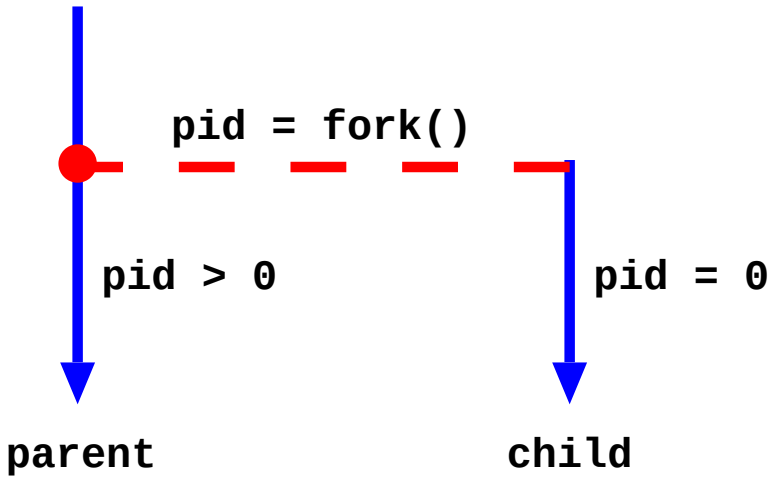
The system call **fork()** allows the creation of new processes. It returns the PID of the newly created process (**child process**) to the process that has made the call (**parent process**) and returns 0 to the child process.

Creating Processes

```
pid_t fork(void)
```

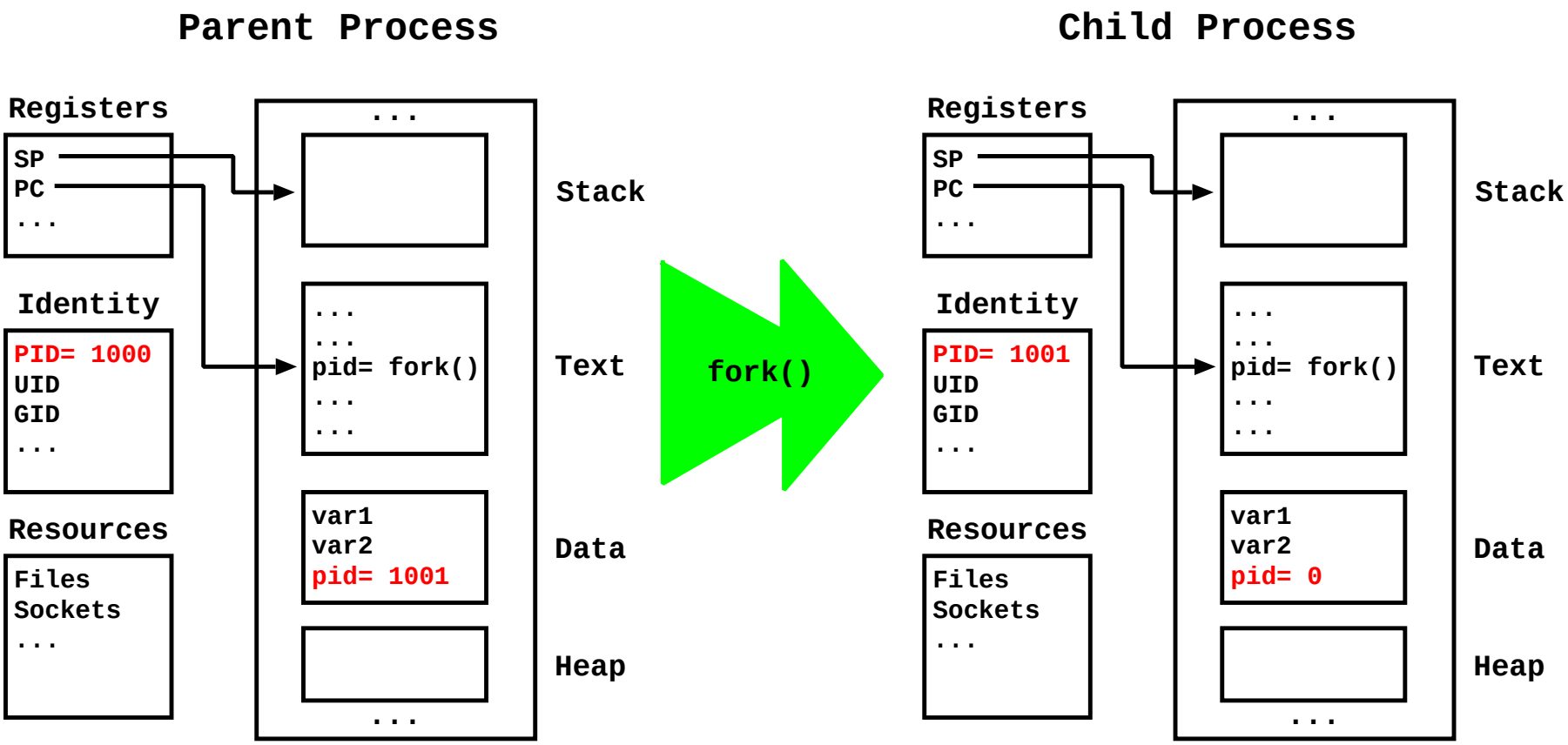
The system call **fork()** allows the creation of new processes. It returns the PID of the newly created process (**child process**) to the process that has made the call (**parent process**) and returns 0 to the child process.

How can we distinguish the execution of both processes (parent and child)?




```
pid_t pid;
...
pid = fork();
if (pid == 0) {
  ... // child code after fork
} else {
  ... // parent code after fork
}
// common code after fork
```

Creating Processes



Parallel Rank Sort (proc-ranksort.c)

```
main() {
  ...
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  // parent waits for all children to complete
  for (k = 0; k < N; k++)
    wait(NULL);
  // parent shows result
  for (k = 0; k < N; k++)
    printf("%d ", R[k]);
  printf("\n");
  ...
}
```

Parallel Rank Sort (proc-ranksort.c)

```
main() {
  ...
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  // parent waits for all children to complete
  for (k = 0; k < N; k++)
    wait(NULL);
  // parent shows result
  for (k = 0; k < N; k++)
    printf("%d ", R[k]);
  printf("\n");
  ...
}
```

Question: what is the output of the program?

Parallel Rank Sort

Launch N child processes, in which, each process executes `compute_rank()` on the different element in `A[k]`:

- Each child process inherits one copy of the variables of the parent process. However, the changes made to those variables are not visible to the parent process.
- As the changes made in `R[]` are not visible, the parent process writes a sequence of zeros!

Parallel Rank Sort

Launch N child processes, in which, each process executes `compute_rank()` on the different element in `A[k]`:

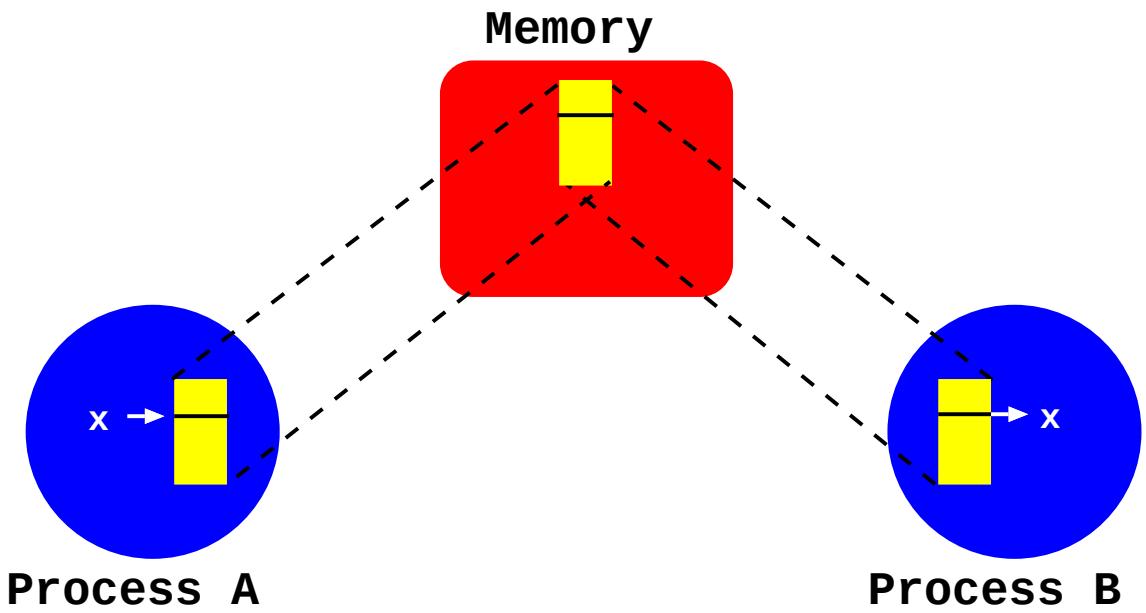
- Each child process inherits one copy of the variables of the parent process. However, the changes made to those variables are not visible to the parent process.
- As the changes made in `R[]` are not visible, the parent process writes a sequence of zeros!

Solution: the array `R[]` must be shared!

Shared Memory Segments

One of the simplest methods of **Inter-Process Communication (IPC)** is the usage of shared memory segments:

- The segment is known by both processes and when one of processes writes in the segment, the other also sees the change.
- The access to shared memory segments is as efficient as the access to non-shared segments and their manipulation is similar.



Shared Memory Segments

How to create and use a shared memory segment:

- The processes begin by **allocating the segment**.
- Then, each process must **map the segment** in a memory address, so that, it can use the segment.
- After its usage, each process must **release the mapping done in the previous step**.
- Finally, one of the processes must **remove the segment**.

Allocating a Shared Memory Segment

```
int shmget(key_t key, int size, int flags)
```

`shmget()` allocates a new shared memory segment and returns its id. If it is not possible to allocate the segment then it returns -1.

Allocating a Shared Memory Segment

```
int shmget(key_t key, int size, int flags)
```

`shmget()` allocates a new shared memory segment and returns its id. If it is not possible to allocate the segment then it returns -1.

- `key` is the identifier of the requested segment. Other processes can access the same segment if they present the same key (`IPC_PRIVATE` ensures that a new segment is created).

Allocating a Shared Memory Segment

```
int shmget(key_t key, int size, int flags)
```

`shmget()` allocates a new shared memory segment and returns its id. If it is not possible to allocate the segment then it returns -1.

- `key` is the identifier of the requested segment. Other processes can access the same segment if they present the same key (`IPC_PRIVATE` ensures that a new segment is created).
- `size` defines the amount of memory of the request, rounded to a multiple of the operating system's page size (usually 4KB – `getpagesize()` to obtain the exact value).

Allocating a Shared Memory Segment

```
int shmget(key_t key, int size, int flags)
```

`shmget()` allocates a new shared memory segment and returns its id. If it is not possible to allocate the segment then it returns -1.

- `key` is the identifier of the requested segment. Other processes can access the same segment if they present the same key (`IPC_PRIVATE` ensures that a new segment is created).
- `size` defines the amount of memory of the request, rounded to a multiple of the operating system's page size (usually 4KB – `getpagesize()` to obtain the exact value).
- `flags` specifies the type of allocation: `IPC_CREAT` indicates that the new segment must be create (if it does not exist); `IPC_EXCL` indicates that segment must be exclusive (fails otherwise); `S_IRUSR`, `S_IWUSR`, `S_IROTH` and `S_IWOTH` indicate the read/write permissions.

Mapping a Shared Memory Segment

```
void *shmat(int shmid, void *addr, int flags)
```

`shmat()` allows the mapping of a shared memory segment from a memory address within the address space of the process. Returns the address of memory in which the segment was mapped, or return -1 if it is not possible to map the segment.

Mapping a Shared Memory Segment

```
void *shmat(int shmid, void *addr, int flags)
```

`shmat()` allows the mapping of a shared memory segment from a memory address within the address space of the process. Returns the address of memory in which the segment was mapped, or return -1 if it is not possible to map the segment.

- `shmid` is the integer that identifies the segment (obtained with `shmget()`)

Mapping a Shared Memory Segment

```
void *shmat(int shmid, void *addr, int flags)
```

`shmat()` allows the mapping of a shared memory segment from a memory address within the address space of the process. Returns the address of memory in which the segment was mapped, or return -1 if it is not possible to map the segment.

- `shmid` is the integer that identifies the segment (obtained with `shmget()`)
- `addr` is the desired memory address (multiple of the operating system's page size), or `NULL` if we allow the operating system to choose the address.

Mapping a Shared Memory Segment

```
void *shmat(int shmid, void *addr, int flags)
```

`shmat()` allows the mapping of a shared memory segment from a memory address within the address space of the process. Returns the address of memory in which the segment was mapped, or return -1 if it is not possible to map the segment.

- `shmid` is the integer that identifies the segment (obtained with `shmget()`)
- `addr` is the desired memory address (multiple of the operating system's page size), or `NULL` if we allow the operating system to choose the address.
- `flags` specifies the options of the mapping: for example, `SHM_RDONLY` forces the segment to be read-only.

Freeing a Shared Memory Segment

```
int shmdt(void *addr)
```

`shmdt()` frees the mapping, thus that the correspondent shared memory segment is no longer associated with a memory address (the operating system decrements in one unit the number of mappings associated with the segment). Returns 0 if it succeeds, or -1 otherwise.

- `addr` is the initial memory address associated with the segment to be freed.

Removing a Shared Memory Segment

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

`shmctl()` removes the shared memory segment and does not allow any further mappings (the segment is only really removed when the number of mappings is zero). Returns 0 if it succeeds, or -1 otherwise.

- `shmid` is the integer that identifies the segment.
- `cmd` should be `IPC_RMID` (remove an IPC identifier).
- `buf` should be `NULL`.

Removing a Shared Memory Segment

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

`shmctl()` removes the shared memory segment and does not allow any further mappings (the segment is only really removed when the number of mappings is zero). Returns 0 if it succeeds, or -1 otherwise.

- `shmid` is the integer that identifies the segment.
- `cmd` should be `IPC_RMID` (remove an IPC identifier).
- `buf` should be `NULL`.

The number of shared memory segments allowed is limited. When a process ends its execution, it frees automatically the mapping. However, it does not remove the segment. `shmctl()` must be explicitly called by one of the processes.

- Command `ipcs` allows to check which segments are in use.
- Command `ipcrm` allows the removal of a segment.

Basic Step Sequence

```
int shmid, shmsize;
char *shared_memory;
...
shmsize = getpagesize();
shmid = shmget(IPC_PRIVATE, shmsize, S_IRUSR | S_IWUSR);
shared_memory = (char *) shmat(shmid, NULL, 0);
...
sprintf(shared_memory, "Hello World!");
...
shmdt(shared_memory);
shmctl(shmid, IPC_RMID, NULL);
```

Parallel Rank Sort (proc-rankshm.c)

```
int A[N], *R;

main() {
  ...
  // allocate and map a shared segment for R[]
  shmid = shmget(IPC_PRIVATE, N * sizeof(int), S_IRUSR | S_IWUSR);
  R = (int *) shmat(shmid, NULL, 0);
```

Parallel Rank Sort (proc-rankshm.c)

```
int A[N], *R;

main() {
  ...
  // allocate and map a shared segment for R[]
  shmid = shmget(IPC_PRIVATE, N * sizeof(int), S_IRUSR | S_IWUSR);
  R = (int *) shmat(shmid, NULL, 0);
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  for (k = 0; k < N; k++) wait(NULL);
  for (k = 0; k < N; k++) printf("%d ", R[k]); printf("\n");
```

Parallel Rank Sort (proc-rankshm.c)

```
int A[N], *R;

main() {
  ...
  // allocate and map a shared segment for R[]
  shmid = shmget(IPC_PRIVATE, N * sizeof(int), S_IRUSR | S_IWUSR);
  R = (int *) shmat(shmid, NULL, 0);
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  for (k = 0; k < N; k++) wait(NULL);
  for (k = 0; k < N; k++) printf("%d ", R[k]); printf("\n");
  // free and remove shared segment
  shmdt(R);
  shmctl(shmid, IPC_RMID, NULL);
}
```

Mapping of Files in Memory

The communication between processes using shared memory, can also be obtained through **shared files**.

Mapping of Files in Memory

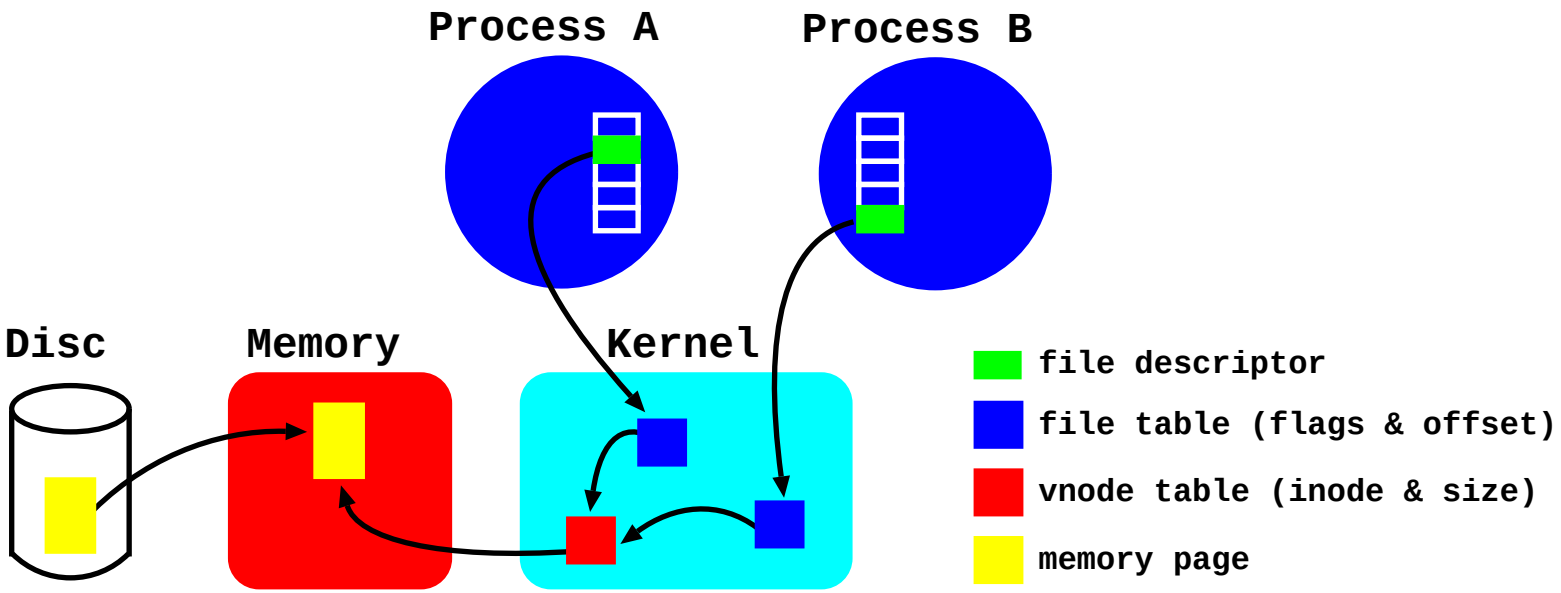
The communication between processes using shared memory, can also be obtained through **shared files**.

- The access to files is usually done using specific functions, such as `open()`, `read()`, `write()`, `lseek()` e `close()`.
- The atomicity in reading and in writing a file is granted by the operations of `read()` and `write()`, which synchronize the data structure **vnode** associated with the file.

Mapping of Files in Memory

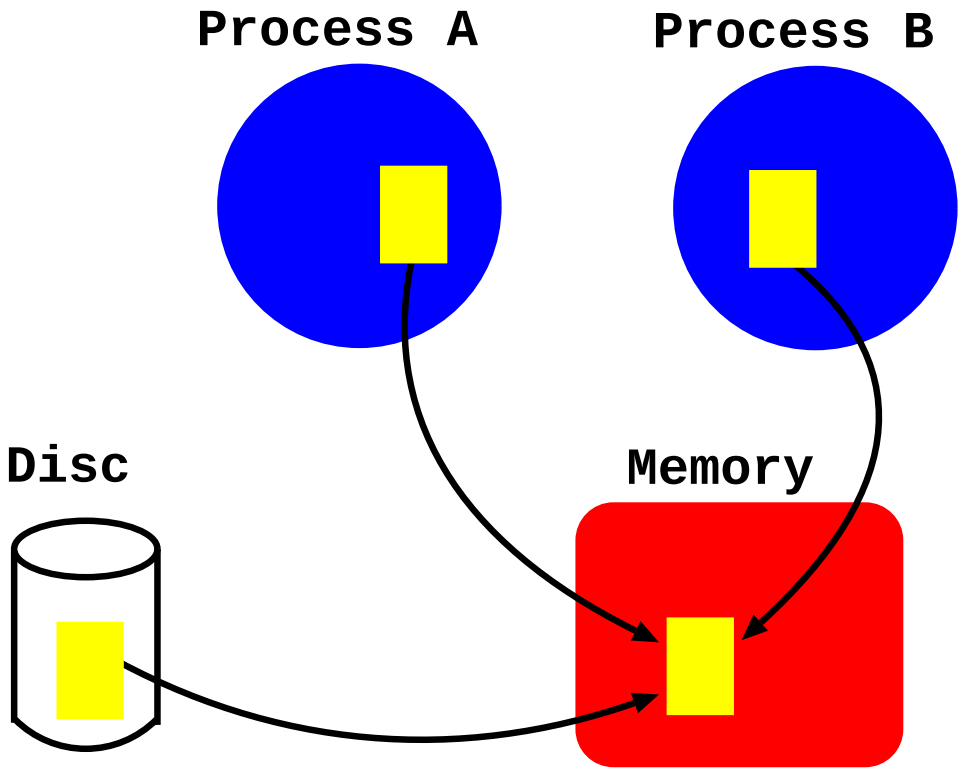
The communication between processes using shared memory, can also be obtained through **shared files**.

- The access to files is usually done using specific functions, such as `open()`, `read()`, `write()`, `lseek()` e `close()`.
- The atomicity in reading and in writing a file is granted by the operations of `read()` and `write()`, which synchronize the data structure **vnode** associated with the file.



Mapping of Files in Memory

Allows a process to map regions of a file directly within its address space, such that, the read and the write operations are completely transparent.



Mapping of Files in Memory

How to map a file in to an address space:

- Initially, the processes must obtain the **descriptor of the file** to be mapped.
- Next, each process, must **map the file** in to an address space.
- And finally, after using the mapping, each process must free it.

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

`mmap()` allows the mapping of a region of a file from a memory address within a process address space. Returns the memory address in which the region was mapped, or -1 if it is not possible to do the mapping.

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

`mmap()` allows the mapping of a region of a file from a memory address within a process address space. Returns the memory address in which the region was mapped, or -1 if it is not possible to do the mapping.

- `start` is the initial memory address, where we want to map the file region (multiple of the operating system's page size) or `NULL` if we allow the operating system to choose the address.

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

`mmap()` allows the mapping of a region of a file from a memory address within a process address space. Returns the memory address in which the region was mapped, or -1 if it is not possible to do the mapping.

- `start` is the initial memory address, where we want to map the file region (multiple of the operating system's page size) or `NULL` if we allow the operating system to choose the address.
- `length` is the size of the mapping (in bytes).

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

`mmap()` allows the mapping of a region of a file from a memory address within a process address space. Returns the memory address in which the region was mapped, or -1 if it is not possible to do the mapping.

- `start` is the initial memory address, where we want to map the file region (multiple of the operating system's page size) or `NULL` if we allow the operating system to choose the address.
- `length` is the size of the mapping (in bytes).
- `prot` specifies the read and write permissions of the mapping: `PROT_READ` and `PROT_WRITE`.

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

- `flags` specifies the attributes of the mapping: `MAP_FIXED` forces the usage of `start` to map the region; `MAP_SHARED` indicates that the write operation changes the file; `MAP_PRIVATE` indicates that the write operations are not propagated to the file (usually used for debugging).

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

- `flags` specifies the attributes of the mapping: `MAP_FIXED` forces the usage of `start` to map the region; `MAP_SHARED` indicates that the write operation changes the file; `MAP_PRIVATE` indicates that the write operations are not propagated to the file (usually used for debugging).
- `fd` is the descriptor of the file to mapped.

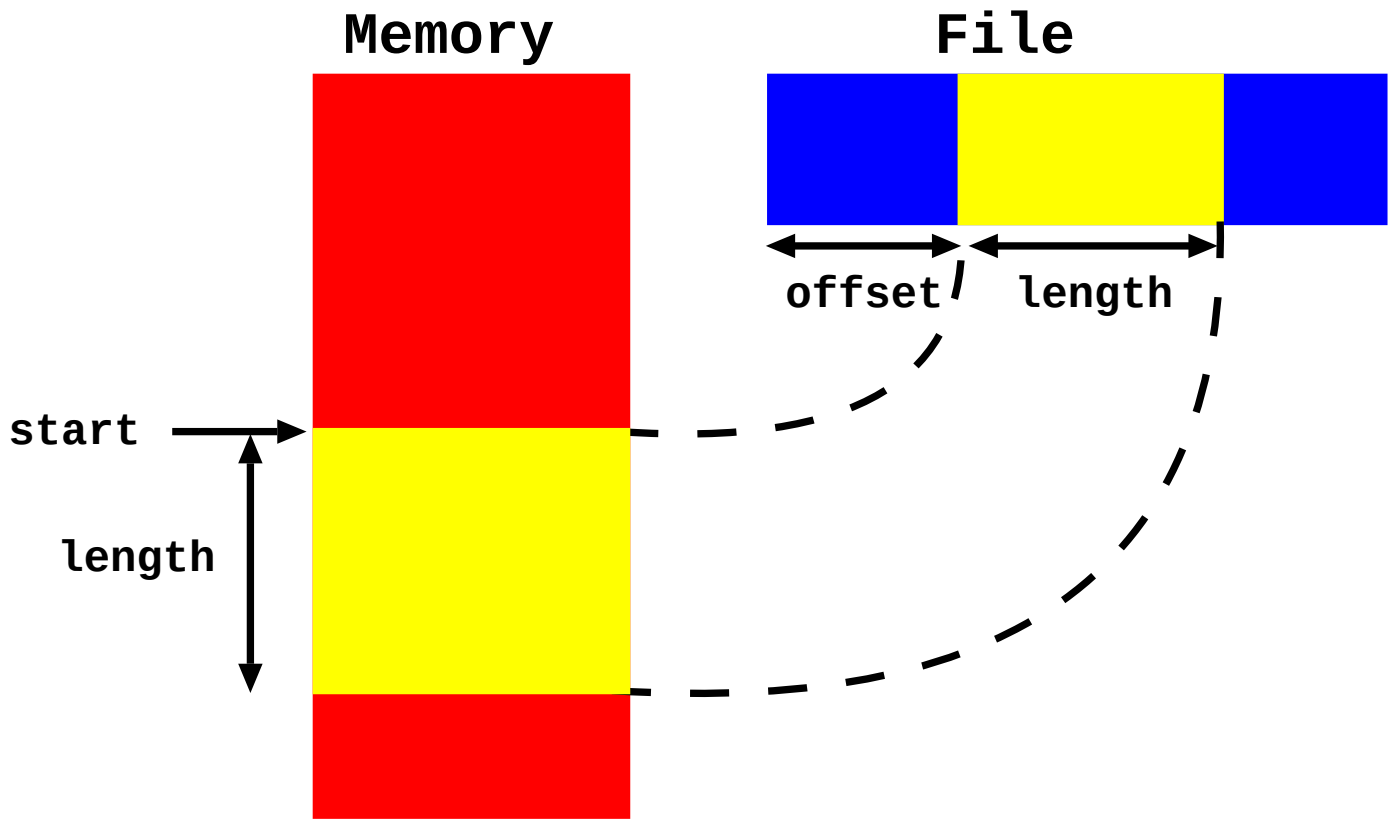
Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

- `flags` specifies the attributes of the mapping: `MAP_FIXED` forces the usage of `start` to map the region; `MAP_SHARED` indicates that the write operation changes the file; `MAP_PRIVATE` indicates that the write operations are not propagated to the file (usually used for debugging).
- `fd` is the descriptor of the file to mapped.
- `offset` is displacement within the region of the file to be mapped (multiple of the operating system's page size).

Mapping a Region of a File in Memory

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```



Freeing a Region from Mapped Memory

```
int munmap(void *start, size_t length)
```

`munmap()` frees the mapping made and the correspondent region of memory is no longer associated with a memory address. Returns 0 if OK or (-1) otherwise.

- `start` is the initial address of the memory region to be freed.
- `length` is the amount of memory to freed.

Basic Step Sequence

```
int fd, mapsize;
void *mapped_memory;
...
mapsize = getpagesize();
fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
lseek(fd, mapsize, SEEK_SET);
write(fd, "", 1);
mapped_memory = mmap(NULL, mapsize, PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, 0);
...
sprintf(mapped_memory, "Hello World!");
...
munmap(mapped_memory, mapsize);
```

Parallel Rank Sort (proc-rankmmap.c)

```
int A[N], *R;

main() {
  ...
  // map a file into a shared memory region for R[]
  fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  lseek(fd, N * sizeof(int), SEEK_SET);
  write(fd, "", 1);
  R = (int *) mmap(NULL, N * sizeof(int), PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, 0);
```

Parallel Rank Sort (proc-rankmmap.c)

```
int A[N], *R;

main() {
  ...
  // map a file into a shared memory region for R[]
  fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  lseek(fd, N * sizeof(int), SEEK_SET);
  write(fd, "", 1);
  R = (int *) mmap(NULL, N * sizeof(int), PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, 0);
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  for (k = 0; k < N; k++) wait(NULL);
  for (k = 0; k < N; k++) printf("%d\n", R[k]); printf("\n");
```

Parallel Rank Sort (proc-rankmmap.c)

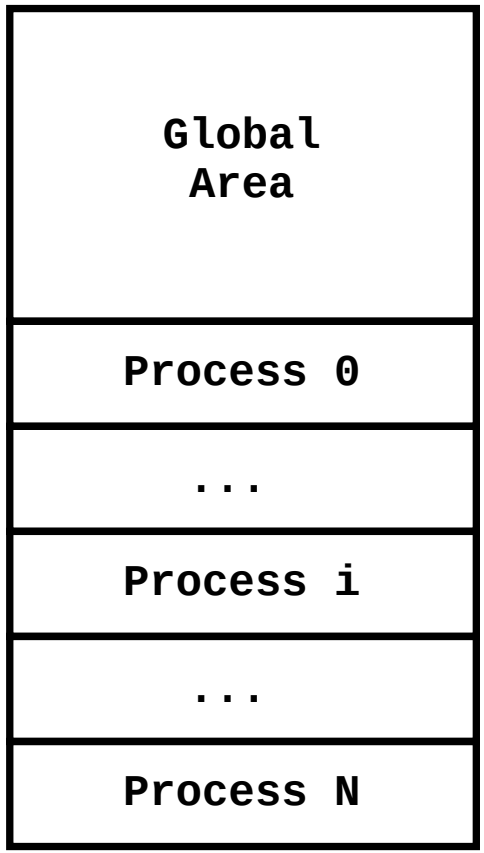
```
int A[N], *R;

main() {
  ...
  // map a file into a shared memory region for R[]
  fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  lseek(fd, N * sizeof(int), SEEK_SET);
  write(fd, "", 1);
  R = (int *) mmap(NULL, N * sizeof(int), PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, 0);
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  for (k = 0; k < N; k++) wait(NULL);
  for (k = 0; k < N; k++) printf("%d\n", R[k]); printf("\n");
  // unmap shared memory region
  munmap(R, N * sizeof(int));
}
```

Advanced Techniques in Memory Mapping

Consider the mapping of a shared memory segment according with the figure:

- Each process has a local area and all processes shared the same global area.
- The sharing of tasks is obtained through the synchronization of the states of the processes in the different parts of the computation.
- This synchronization corresponds in practice to the copy of segments of memory from one process to another process.



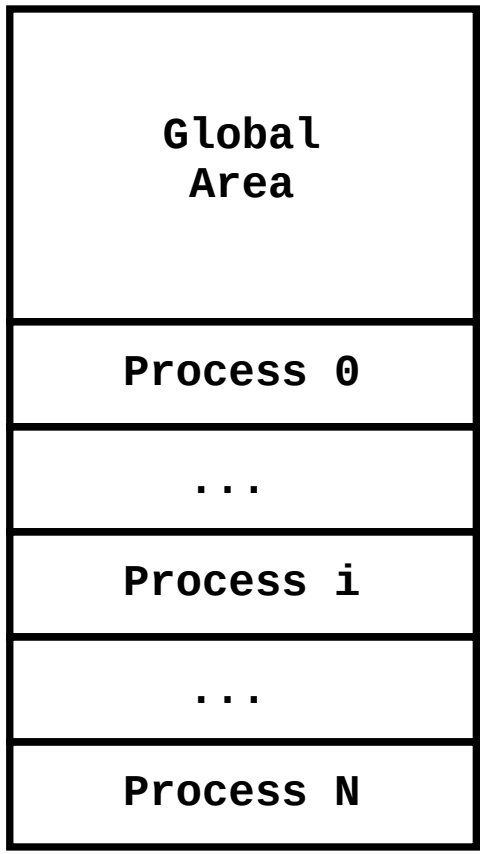
Advanced Techniques in Memory Mapping

Problem: the copy of segments of memory between the processes requires the reallocation of addresses, so that they can make sense in the new address space.

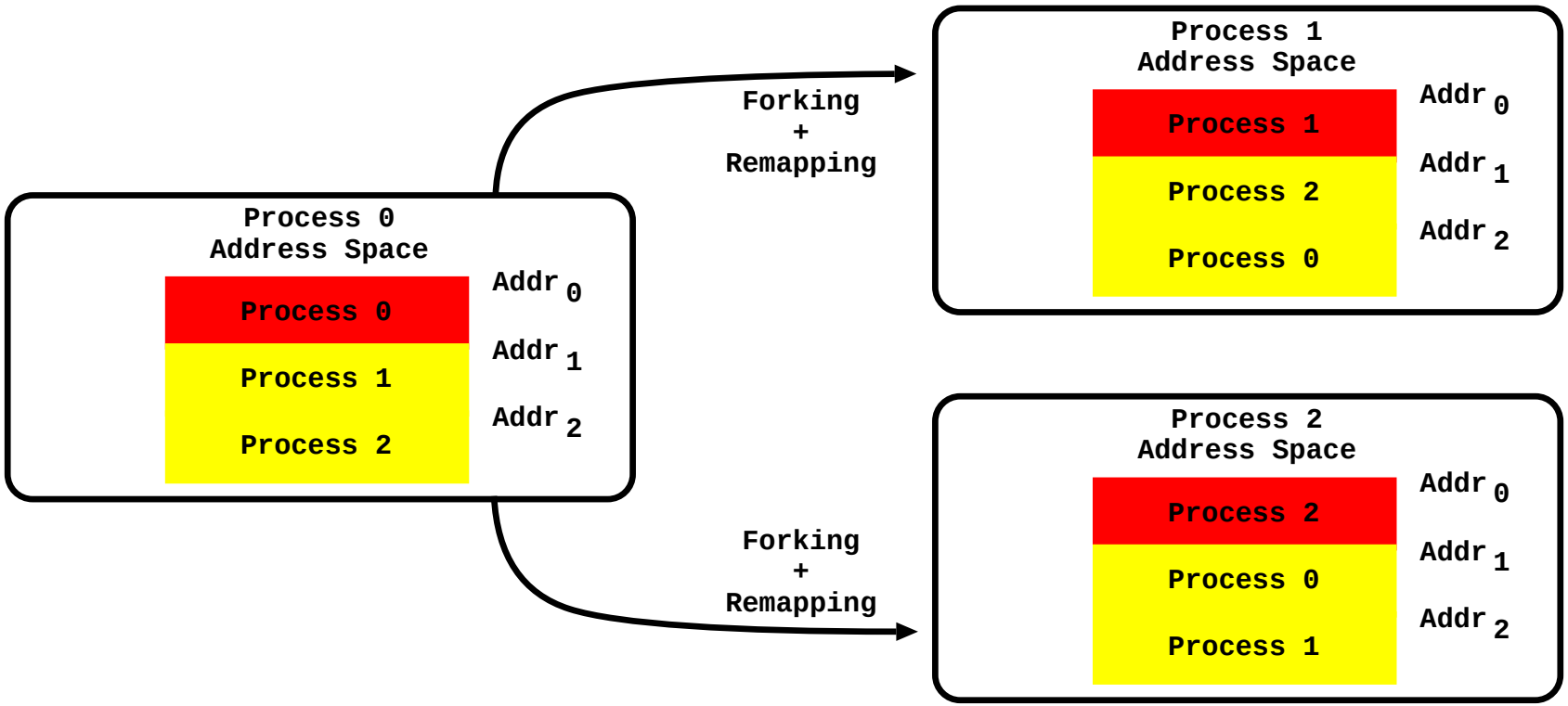
Advanced Techniques in Memory Mapping

Problem: the copy of segments of memory between the processes requires the reallocation of addresses, so that they can make sense in the new address space.

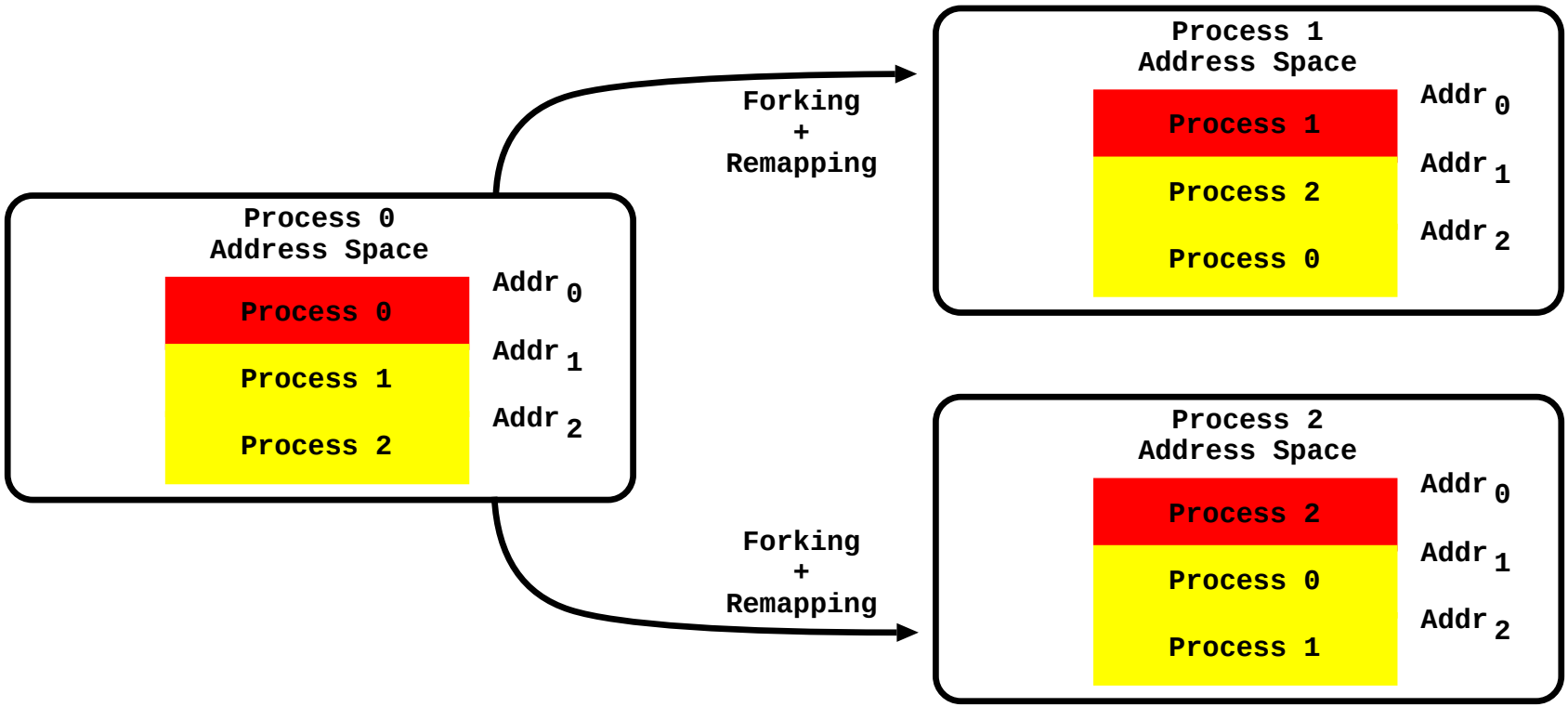
Solution: map the memory in such a way that all processes can see their own areas in the same address. In other words, the **address space of each process**, from the individual point of view, **begins in the same address**.



Advanced Techniques in Memory Mapping



Advanced Techniques in Memory Mapping



This technique allows the copying operations to be very efficient, since it **avoids the reallocation of addresses**. Suppose that, for example, the process 2 wants to copy to process 1, a memory segment that begins in the address `Addr` (for the point of view of the process 2). Then, the destination address should be $\texttt{Addr} + (\texttt{Addr}_2 - \texttt{Addr}_0)$.

Advanced Techniques in Memory Mapping

```
map_addr = mmap(NULL, global_size + n_procs * local_size,
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 0; i < n_procs; i++)
  proc(i) = map_addr + global_size + local_size * i;
```

Advanced Techniques in Memory Mapping

```
map_addr = mmap(NULL, global_size + n_procs * local_size,
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 0; i < n_procs; i++)
  proc(i) = map_addr + global_size + local_size * i;
for (p = 1; p < n_procs; p++)
  if (fork() == 0) {
    // unmap local regions
    remap_addr = map_addr + global_size;
    munmap(remap_addr, local_size * n_procs);
```

Advanced Techniques in Memory Mapping

```
map_addr = mmap(NULL, global_size + n_procs * local_size,
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 0; i < n_procs; i++)
  proc(i) = map_addr + global_size + local_size * i;
for (p = 1; p < n_procs; p++)
  if (fork() == 0) {
    // unmap local regions
    remap_addr = map_addr + global_size;
    munmap(remap_addr, local_size * n_procs);
    // remap local regions
    for (i = 0; i < n_procs; i++) {
      proc(i) = remap_addr + local_size * ((n_procs + i - p) % n_procs);
      mmap(proc(i), local_size, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_FIXED, fd, global_size + local_size * i);
    }
    break;
  }
```

Advanced Techniques in Memory Mapping

```
map_addr = mmap(NULL, global_size + n_procs * local_size,
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 0; i < n_procs; i++)
  proc(i) = map_addr + global_size + local_size * i;
for (p = 1; p < n_procs; p++)
  if (fork() == 0) {
    // unmap local regions
    remap_addr = map_addr + global_size;
    munmap(remap_addr, local_size * n_procs);
    // remap local regions
    for (i = 0; i < n_procs; i++) {
      proc(i) = remap_addr + local_size * ((n_procs + i - p) % n_procs);
      mmap(proc(i), local_size, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_FIXED, fd, global_size + local_size * i);
    }
    break;
  }
```

The memory copy of process 2 to process 1 from `Addr` would have the destination address `Addr + (proc(1) - proc(2))`.

Synchronization in Shared Memory

In **Parallel Rank Sort** the processes are independent and do not need to synchronize in the shared memory access. However, when processes update shared data structures (**critical region**), it is necessary to use mechanisms that guarantee **mutual exclusion**, i.e., guarantee that two processes are never simultaneously within the same critical region.

Synchronization in Shared Memory

In **Parallel Rank Sort** the processes are independent and do not need to synchronize in the shared memory access. However, when processes update shared data structures (**critical region**), it is necessary to use mechanisms that guarantee **mutual exclusion**, i.e., guarantee that two processes are never simultaneously within the same critical region.

Besides granting mutual exclusion, a good and correct solution to the **critical region problem** should also verify the following conditions:

- Processes outside the critical region cannot block other processes.
- No process should wait indefinitely to enter in the critical region.
- The CPU frequency or the number of CPU's available should not be relevant.

Synchronization in Shared Memory

In **Parallel Rank Sort** the processes are independent and do not need to synchronize in the shared memory access. However, when processes update shared data structures (**critical region**), it is necessary to use mechanisms that guarantee **mutual exclusion**, i.e., guarantee that two processes are never simultaneously within the same critical region.

Besides granting mutual exclusion, a good and correct solution to the **critical region problem** should also verify the following conditions:

- Processes outside the critical region cannot block other processes.
- No process should wait indefinitely to enter in the critical region.
- The CPU frequency or the number of CPU's available should not be relevant.

Next, we will see two synchronization mechanisms:

- **Spinlocks** – busy waiting
- **Semaphores** – no busy waiting

Atomic Instructions

One way to grant an efficient mutual exclusion is to protect the critical regions through the usage of **atomic instructions**:

- **Test and Set Lock (TSL)** – modifies the content of a memory position to a pre-determined value and returns the previous value.
- **Compare And Swap (CAS)** – tests and swaps the content of a memory position according with an expected value.

The implementation of this type of atomic instructions requires the **support of the hardware**. Nowadays, modern hardware architectures support atomic instructions TSL/CAS or its variants.

Atomic Instructions

```
// test and set lock
boolean TSL(boolean *target) {
  boolean aux = *target;
  *target = TRUE;
  return aux;
}
```

Atomic Instructions

```
// test and set lock
boolean TSL(boolean *target) {
  boolean aux = *target;
  *target = TRUE;
  return aux;
}

// compare and swap
boolean CAS(int *target, int expected, int new) {
  if (*target != expected)
    return FALSE;
  *target = new;
  return TRUE;
}
```

Atomic Instructions

```
// test and set lock
boolean TSL(boolean *target) {
  boolean aux = *target;
  *target = TRUE;
  return aux;
}

// compare and swap
boolean CAS(int *target, int expected, int new) {
  if (*target != expected)
    return FALSE;
  *target = new;
  return TRUE;
}
```

The execution of the `TSL()` and `CAS()` instructions must be indivisible, i.e., no other process can access the memory position which is being refereed by `target` before the instruction completes it execution.

Mutual Exclusion with TSL

Question: how to use the TSL instruction to ensure mutual exclusion when accessing a critical region?

Mutual Exclusion with TSL

Question: how to use the TSL instruction to ensure mutual exclusion when accessing a critical region?

Solution: associate a shared variable (**mutex lock**) to the critical region and repeatedly execute the TSL instruction on that variable until it returns the value `FALSE`. A process accesses only the critical region when the instruction returns `FALSE`, which guarantees the mutual exclusion.

Mutual Exclusion with TSL

Question: how to use the TSL instruction to ensure mutual exclusion when accessing a critical region?

Solution: associate a shared variable (**mutex lock**) to the critical region and repeatedly execute the TSL instruction on that variable until it returns the value `FALSE`. A process accesses only the critical region when the instruction returns `FALSE`, which guarantees the mutual exclusion.

```
#define INIT_LOCK(M)     M = FALSE
#define ACQUIRE_LOCK(M) while (TSL(&M))
#define RELEASE_LOCK(M) M = FALSE

INIT_LOCK(mutex);
... // non-critical section
ACQUIRE_LOCK(mutex);
... // critical section
RELEASE_LOCK(mutex);
... // non-critical section
```

Mutual Exclusion with CAS

```
#define INIT_LOCK(M)     M = 0
#define ACQUIRE_LOCK(M) while (!CAS(&M, 0, 1))
#define RELEASE_LOCK(M) M = 0

INIT_LOCK(mutex);
... // non-critical section
ACQUIRE_LOCK(mutex);
... // critical section
RELEASE_LOCK(mutex);
... // non-critical section
```

Spinlocks

When a solution to implement mutual exclusion requires **busy waiting**, the mutex lock is called **spinlock**.

Spinlocks

When a solution to implement mutual exclusion requires **busy waiting**, the mutex lock is called **spinlock**.

Busy waiting can be a problem because:

- Wastes CPU time that another process could be using to do useful work.
- If the process holding the lock is interrupted (change of context) then no other process can access the lock and so it will be useless to give CPU time to another process.
- Does not satisfy the condition that no process should wait indefinitely to enter in a critical region.

Spinlocks

When a solution to implement mutual exclusion requires **busy waiting**, the mutex lock is called **spinlock**.

Busy waiting can be a problem because:

- Wastes CPU time that another process could be using to do useful work.
- If the process holding the lock is interrupted (change of context) then no other process can access the lock and so it will be useless to give CPU time to another process.
- Does not satisfy the condition that no process should wait indefinitely to enter in a critical region.

On the other hand, when the **time holding the lock is too short** it is expected to be **more advantageous than doing a context switch**:

- Usual in multiprocessor/multicore systems, where a process holds a lock and the remaining processes remain in busy waiting.

Spinlocks in Linux (include/linux/spinlock.h)

Initialize the spinlock:

```
spin_lock_init(spinlock_t *spinlock)
```

Busy waiting until obtaining the spinlock:

```
spin_lock(spinlock_t *spinlock)
```

Tries to obtained the spinlock, but does not wait if it is not possible:

```
spin_trylock(spinlock_t *spinlock)
```

Free the spinlock:

```
spin_unlock(spinlock_t *spinlock)
```

Read-Write Spinlocks

Sometimes, the necessity of granting mutual exclusion in the access to a critical region is only (or mostly) associated with reading operations.

Non-exclusive read operations never lead to inconsistency of data, only write operations cause this problem.

Read-write spinlocks provide an alternative solution, since they allow **multiple simultaneous reading operations and one single write operation** to occur in the same critical region.

Read-Write Spinlocks in Linux (include/linux/rwlock.h)

Initialize the spinlock:

```
rwlock_init(rwlock_t *rwlock)
```

Busy waiting until all writing operations are complete:

```
read_lock(rwlock_t *rwlock)
```

Busy waiting until all read and write operations are complete:

```
write_lock(rwlock_t *rwlock)
```

Read-Write Spinlocks in Linux (include/linux/rwlock.h)

Try to obtain a spinlock, but does not wait if it is not possible:

```
read_trylock(rwlock_t *rwlock)
write_trylock(rwlock_t *rwlock)
```

Free a spinlock:

```
read_unlock(rwlock_t *rwlock)
write_unlock(rwlock_t *rwlock)
```

Spinlocks

Advantages and disadvantages:

- **(+)** Simple and easy to verify
- **(+)** Can be used by an arbitrary number of processes
- **(+)** Supports multiple critical regions
- **(–)** With a high number of processes, busy waiting can be a problem
- **(–)** When we have multiple critical regions, it is possible to have **deadlocks** between processes.

Semaphores

They were introduced by Dijkstra in 1965 and they allow a synchronized access to **shared resources** that can be defined by a **finite number of instances**.

Semaphores

They were introduced by Dijkstra in 1965 and they allow a synchronized access to **shared resources** that can be defined by a **finite number of instances**.

A semaphore can be seen as a **non-negative integer** that represents the number of instances available on the respective resource:

- It is not possible to read or write the value of a semaphore directly, except to set its initial value
- It cannot be negative, because when it reaches the value of 0 (which means that all instances are in use), the processes which want to use the resource remain blocked until the semaphore gets back to a value which is higher than 0

Semaphores

They were introduced by Dijkstra in 1965 and they allow a synchronized access to **shared resources** that can be defined by a **finite number of instances**.

A semaphore can be seen as a **non-negative integer** that represents the number of instances available on the respective resource:

- It is not possible to read or write the value of a semaphore directly, except to set its initial value
- It cannot be negative, because when it reaches the value of 0 (which means that all instances are in use), the processes which want to use the resource remain blocked until the semaphore gets back to a value which is higher than 0

There are two types of semaphores:

- **Counting Semaphores** – can have any value
- **Binary Semaphores** – can only have the value of 0 or 1

Operations over Semaphores

The semaphores can be accessed through two **atomic operations**:

- **DOWN** (or **SLEEP** or **WAIT**) – waits for the semaphore to be positive and then decrements it in one unit
- **UP** (or **WAKEUP** or **POST** or **SIGNAL**) – increments the semaphore in one unit

Operations over Semaphores

The semaphores can be accessed through two **atomic operations**:

- **DOWN** (or **SLEEP** or **WAIT**) – waits for the semaphore to be positive and then decrements it in one unit
- **UP** (or **WAKEUP** or **POST** or **SIGNAL**) – increments the semaphore in one unit

```
down(semaphore S) {
  if (S == 0)
    suspend(); // suspend current process
  S--;
}

up(semaphore S) {
  S++;
  if (S == 1)
    wakeup(); // wakeup one waiting process
}
```

Implementation of Semaphores

The implementation must ensure that two operations **DOWN** and/or **UP** are never performed simultaneously on the same semaphore:

- Simultaneous operations of **DOWN** cannot decrement the semaphore below zero.
- One cannot loose one increment **UP** if another **DOWN** occurs simultaneously.

Implementation of Semaphores

The implementation must ensure that two operations **DOWN** and/or **UP** are never performed simultaneously on the same semaphore:

- Simultaneous operations of **DOWN** cannot decrement the semaphore below zero.
- One cannot loose one increment **UP** if another **DOWN** occurs simultaneously.

The implementation of semaphores is based in synchronization mechanisms that try to **minimize the time spent in busy waiting**.
There are two approaches to minimize the time spent in busy waiting:

- In uniprocessores, by deactivating the interrupts.
- In multiprocessors/multicores, by combining deactivation interrupts with atomic instructions.

Implementation of Semaphores in Uniprocessors

```
typedef struct { // semaphore data structure
  int value;     // semaphore value
  PCB *queue;    // associated queue of waiting processes
} semaphore;
```

Implementation of Semaphores in Uniprocessors

```
typedef struct { // semaphore data structure
  int value;     // semaphore value
  PCB *queue;    // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) {
  S.value = 1;
  S.queue = EMPTY;
}
```

Implementation of Semaphores in Uniprocessors

```
typedef struct { // semaphore data structure
  int value;     // semaphore value
  PCB *queue;    // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) {
  S.value = 1;
  S.queue = EMPTY;
}

down(semaphore S) {
  disable_interrupts();
  if (S.value == 0) { // avoid busy waiting
    add_to_queue(current_PCB, S.queue);
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    S.value--;
    enable_interrupts();
  }
}
```

Implementation of Semaphores in Uniprocessors

```
up(semaphore S) {
  disable_interrupts();
  if (S.queue != EMPTY) {
    // keep semaphore value and wakeup one waiting process
    waiting_PCB = remove_from_queue(S.queue);
    add_to_queue(waiting_PCB, OS_ready_queue);
  } else {
    S.value++;
  }
  enable_interrupts();
}
```

Implementation of Semaphores in Multiprocessors

```
typedef struct { // semaphore data structure
  boolean mutex; // to guarantee atomicity
  int value;     // semaphore value
  PCB *queue;    // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) {
  INIT_LOCK(S.mutex);
  S.value = 1;
  S.queue = EMPTY;
}
```

Implementation of Semaphores in Multiprocessors

```
down(semaphore S) {
  disable_interrupts();
  ACQUIRE_LOCK(S.mutex); // short busy waiting time
  if (S.value == 0) {
    add_to_queue(current_PCB, S.queue);
    RELEASE_LOCK(S.mutex);
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    S.value--;
    RELEASE_LOCK(S.mutex);
    enable_interrupts();
  }
}
```

Implementation of Semaphores in Multiprocessors

```
up(semaphore S) {
  disable_interrupts();
  ACQUIRE_LOCK(S.mutex); // short busy waiting time
  if (S.queue != EMPTY) {
    // keep semaphore value and wakeup one waiting process
    waiting_PCB = remove_from_queue(S.queue);
    add_to_queue(waiting_PCB, OS_ready_queue);
  } else {
    S.value++;
  }
  RELEASE_LOCK(S.mutex);
  enable_interrupts();
}
```

POSIX Semaphores

The POSIX semaphores are available in two versions:

- **Named Semaphores** – they are accessed by their name and they can be used by all processes that know that name
- **Unnamed Semaphores** – exist only in memory and therefore can only be used by the processes that share the same address space.

Both versions work in the same way, they differ only on the way that they are initialized and freed.

Creating a Named Semaphore

```
sem_t *sem_open(char *name, int oflag)
sem_t *sem_open(char *name, int oflag, mode_t mode, int value)
```

`sem_open()` creates a new semaphore or opens one that already exists and returns the address of the semaphore. In case of error, it returns `SEM_FAILED`.

Creating a Named Semaphore

```
sem_t *sem_open(char *name, int oflag)
sem_t *sem_open(char *name, int oflag, mode_t mode, int value)
```

`sem_open()` creates a new semaphore or opens one that already exists and returns the address of the semaphore. In case of error, it returns `SEM_FAILED`.

- `name` is the name that identifies the semaphore (by convention, the first character of the name is '/' and does not have any further '/')

Creating a Named Semaphore

```
sem_t *sem_open(char *name, int oflag)
sem_t *sem_open(char *name, int oflag, mode_t mode, int value)
```

`sem_open()` creates a new semaphore or opens one that already exists and returns the address of the semaphore. In case of error, it returns `SEM_FAILED`.

- `name` is the name that identifies the semaphore (by convention, the first character of the name is '/' and does not have any further '/')
- `oflag` specifies the create/open options: `O_CREAT` creates a new semaphore; `O_EXCL` if the semaphore is exclusive; `0` to open a semaphore that already exists.

Creating a Named Semaphore

```
sem_t *sem_open(char *name, int oflag)
sem_t *sem_open(char *name, int oflag, mode_t mode, int value)
```

`sem_open()` creates a new semaphore or opens one that already exists and returns the address of the semaphore. In case of error, it returns `SEM_FAILED`.

- `name` is the name that identifies the semaphore (by convention, the first character of the name is '/' and does not have any further '/')
- `oflag` specifies the create/open options: `O_CREAT` creates a new semaphore; `O_EXCL` if the semaphore is exclusive; `0` to open a semaphore that already exists.
- `mode` specifies the access options (important only when we create one new semaphore with option `O_CREAT`).
- `value` specifies the initial value of the semaphore (important when we create one new semaphore with option `O_CREAT`).

Closing a Named Semaphore

```
int *sem_close(sem_t *sem)
```

`sem_close()` closes the access to the semaphore and frees all of the resources of the process associated with the semaphore (the value of the semaphore is not affected). Returns 0 if OK, -1 otherwise.

- `sem` is the address that identifies the semaphore to be closed

By default, the resources associated with a semaphore, which was opened by a process, are released when the process ends (similar to what happens with the files opened in the context of a process).

Removing a Semaphore with Name

```
int *sem_unlink(char *name)
```

`sem_unlink()` removes the semaphore's name from the system (i.e., it is no longer possible to open the semaphore with `sem_open()`) and, if there are no references to close to the semaphore, the semaphore is also destroyed. Otherwise, the semaphore is only destroyed when there are no references to close. Returns 0 if OK, -1 otherwise.

- `name` is the name that identifies the semaphore to be removed

Creating a Unnamed Semaphore

```
int sem_init(sem_t *sem, int pshared, int value)
```

`sem_init()` creates an unnamed semaphore to be shared between processes. Returns 0 if OK, -1 otherwise.

- `sem` is the address that identifies the unnamed semaphore.
- `pshared` states if the semaphore is to be shared between threads (`0`) or between processes (`1`).
- `value` states the initial value of the semaphore.

Creating a Unnamed Semaphore

```
int sem_init(sem_t *sem, int pshared, int value)
```

`sem_init()` creates an unnamed semaphore to be shared between processes. Returns 0 if OK, -1 otherwise.

- `sem` is the address that identifies the unnamed semaphore.
- `pshared` states if the semaphore is to be shared between threads (`0`) or between processes (`1`).
- `value` states the initial value of the semaphore.

To share a semaphore between processes, it must be located in a **memory region which is shared among all processes**.

Freeing a Unnamed Semaphore

```
int sem_destroy(sem_t *sem)
```

`sem_destroy()` destroys the unnamed semaphore. Returns 0 if OK, -1 otherwise.

- `sem` is the address that identifies the semaphore to be destroyed

Destroying a semaphore that other processes can still be using leads to an unknown behavior, unless that, in the meantime the semaphore is again created by another call to the function `sem_init()`.

Operations over Semaphores with/without Name

```
int sem_post(sem_t *sem)
int sem_wait(sem_t *sem)
int sem_trywait(sem_t *sem)
```

`sem_post()` increments the value of the semaphore, while the `sem_wait()` and `sem_trywait()` decrement the value of the semaphore. `sem_wait()` blocks while the semaphore has the value 0, while the `sem_trywait()` avoids the blocking by returning the value of error instead of blocking. All operations return 0 if OK, -1 otherwise.

- `sem` is the address that identifies the semaphore to be incremented or decremented.

Basic Steps to Use a Named Semaphore

```c
#define SEM_NAME "/mysem"

int main() {
  sem_t *sem;
  sem = sem_open(SEM_NAME, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 1);
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_close(sem); // close semaphore
  sem_unlink(SEM_NAME); // destroy semaphore name
}
```

Basic Steps to Use an Unnamed Semaphore

```
sem_t sem; // unnamed semaphore to be used with threads

int main() {
  sem_init(&sem, 0, 1); // create semaphore
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_destroy(&sem); // destroy semaphore
}
```

Basic Steps to Use an Unnamed Semaphore

```c
sem_t sem; // unnamed semaphore to be used with threads

int main() {
  sem_init(&sem, 0, 1); // create semaphore
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_destroy(&sem); // destroy semaphore
}
```

```c
sem_t *sem; // unnamed semaphore to use with processes

int main() {
  sem = (sem_t *) shmget(...); // allocate shared memory for semaphore
  sem_init(sem, 1, 1); // create semaphore
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_destroy(sem); // destroy semaphore
}
```

Sleeping Barber Problem

The Sleeping Barber problem is a classic IPC problem:

- A barber shop has a number of barbers and a number (`NCHAIRS`) for clients waiting to be attended.
- Whenever a barber does not have clients to attend, he takes a little sleep.
- When a customer arrives at the barber shop, he has to wake up a barber to attend him.
- If a client arrives and all barbers are occupied, then he should wait to be attended (if there are free chairs) or should leave the barber shop without having a haircut (if all the chairs are occupied).

Sleeping Barber Problem

```
int waiting = 0; semaphore clients = 0, barbers = 0, mutex = 1;
client() {
  down(mutex); // get access to the Chair's Waiting Room (CWR)
  if (waiting >= NCHAIRS) // check for empty chairs
    { up(mutex); exit(1); } // leave without a haircut
  waiting++; // get one of the chairs
  up(clients); // wakeup (or notify) a barber if necessary
  up(mutex); // release access to the CWR
  down(barbers); // wait if there are no barbers available
  get_hair_cut();
}

barber() {
  while(1) { // infinite loop to receive multiple clients
    down(clients); // sleep if there are no clients
    down(mutex); // awake - get access to the CWR
    waiting--; // free one of the chairs
    up(barbers); // ready to cut hair
    up(mutex); // release access to the CWR
    cut_hair();
  }
}
```