Processes II (CS-351)

Agenda

- Process operations
- Fork
- Zombie and orphan
- IPC: shared memory, message passing, pipe and socket.
- UNIX System V IPC

Operations on Processes: Process Creation in Unix/Linux: fork()

- fork() system call is issued by a parent process to create a child process.
- Child process is a clone of a parent process.
- Both parent and child continue execution at the instruction immediately after fork():
  - In the child fork() returns 0
  - In the parent fork returns process id (pid) of the child.
  - fork() returns -1 on failure.

Operations on Processes: Process Creation in Unix/Linux: fork()

- The child process inherits:
  - The set of files opened by the parent process.
  - Other resources…

Questions:

Does "inherit" mean share, copy or else?

What resources the parent and child processes should share or copy?

What resources the parent and child processes should not share or copy?

Operations on Processes: Process Creation in Unix/Linux: exec()/wait()/exit()

- exec(…): replaces the program of the caller process with a new program.
- wait(…): waits until the child terminates.
- exit(int exitcode): terminates the caller process with the specified exit code.

Operations on Processes: Process Creation in Unix/Linux: exec() variants

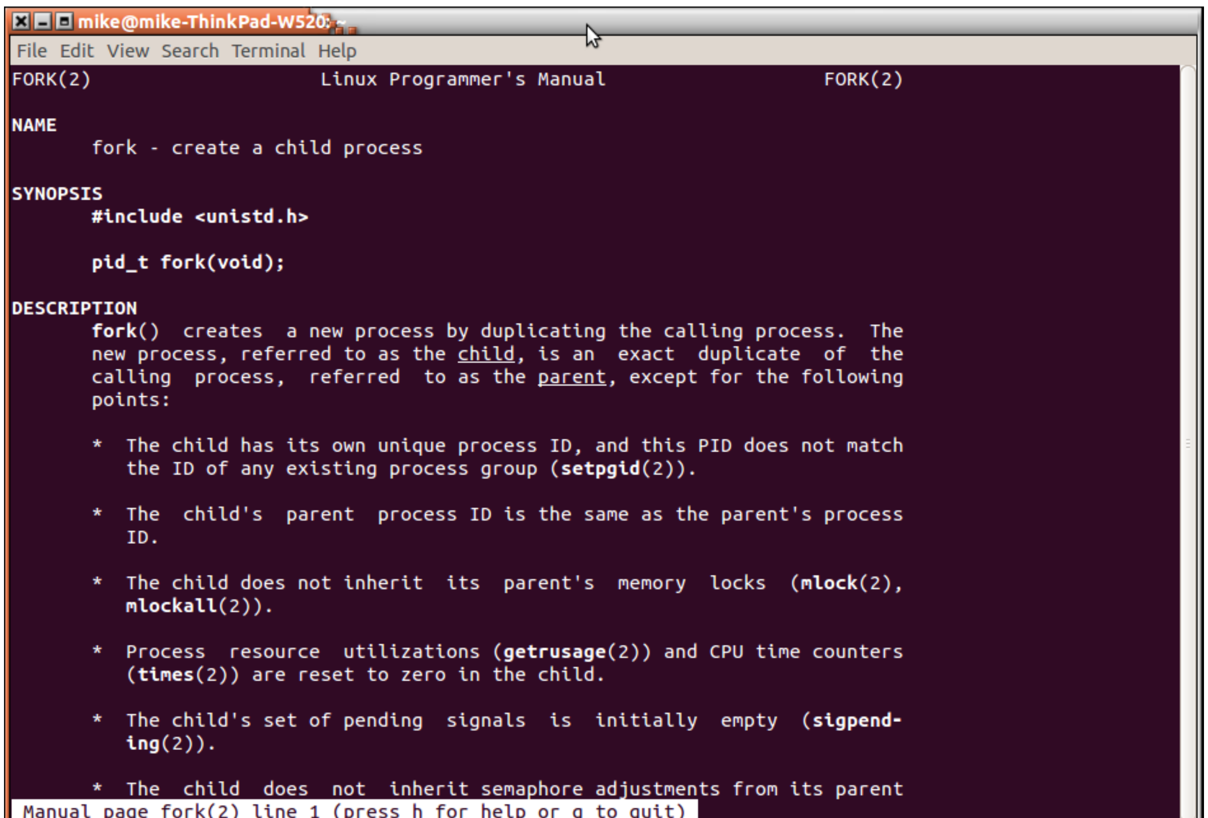
- int execl(const char **path*, const char **arg*, ...);
- int execlp(const char **file*, const char **arg*, ...);
- int execle(const char **path*, const char **arg*,..., char * const *envp*[]);
- int execv(const char **path*, char *const *argv*[]);
- int execvp(const char **file*, char *const *argv*[]);
- int execvpe(const char **file*, char *const *argv*[], char *const *envp*[]);
- Example: execlp(const char **file*, const char **arg*, ...);
  - file: the path of the executable image
  - arg0…argn: command line arguments to pass to the process.
- *All* return -1 on failure

Operations on Processes: Process Creation in Unix/Linux: wait() variants

- pid_t wait(int **status*);
- pid_t waitpid(pid_t *pid*, int **status*, int *options*);
- int waitid(idtype_t *idtype*, id_t *id*, siginfo_t **infop*, int *options*);
- wait() and waitpid() return the process id of the child.
- waitid() return 0 on success and -1 on faliure.

Operations on Processes: Process Creation in Unix/Linux: Manual Pages (man pages)

- For more technical details (or usage) of fork(), exec(), and wait() please see the man pages:
  - Google: man fork, or
  - In Linux terminal: e.g. man fork



Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

Start with a parent process

```
//Parent process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to
    complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

Parent process issues a fork() system call.

```
//Parent process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to
    complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

fork() clones the parent process. Both parent and child continue by executing the next instruction after fork().

```
//Parent process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

```
//Child process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

In parent, fork() returns process id of the child.

```
//Parent process
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
               fprintf(stderr, "Fork Failed");
               exit(-1);
     }
     else if (pid == 0) { /* child process */
               execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
     /* parent will wait for the child to
     complete */
               wait (NULL);
               printf ("Child Complete");
               exit(0);
     }
}
```

In child, fork() returns 0.

```
//Child process
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
               fprintf(stderr, "Fork Failed");
               exit(-1);
     }
     else if (pid == 0) { /* child process */
               execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
     /* parent will wait for the child to
     complete */
               wait (NULL);
               printf ("Child Complete");
               exit(0);
     }
}
```

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

Parent issues a wait() syscall to wait until the child terminates.

```c
//Parent process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to
    complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Child issues a execlp() syscall to replace its executable image with that of ls command

```c
//Child process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to
    complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

Parent waits (in wait()) for the child process to terminate.

ls command executes starting from the first instruction; original child code is destroyed.

```
//Parent process
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
               fprintf(stderr, "Fork Failed");
               exit(-1);
     }
     else if (pid == 0) { /* child process */
               execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
     /* parent will wait for the child to
     complete */
               wait (NULL);
               printf ("Child Complete");
               exit(0);
     }
}
```

```
//Child process
```

ls command code

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

wait() returns, and parent process executes the next instruction

ls command finishes execution and terminates

```
//Parent process
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
              fprintf(stderr, "Fork Failed");
              exit(-1);
     }
     else if (pid == 0) { /* child process */
              execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
     /* parent will wait for the child to
     complete */
              wait (NULL);
              printf ("Child Complete");
              exit(0);
     }
}
```

```
//Child process

ls command code
```

Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

Parent process issues an exit() syscall in order to self-terminate.

```
//Parent process
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
    /* parent will wait for the child to
    complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

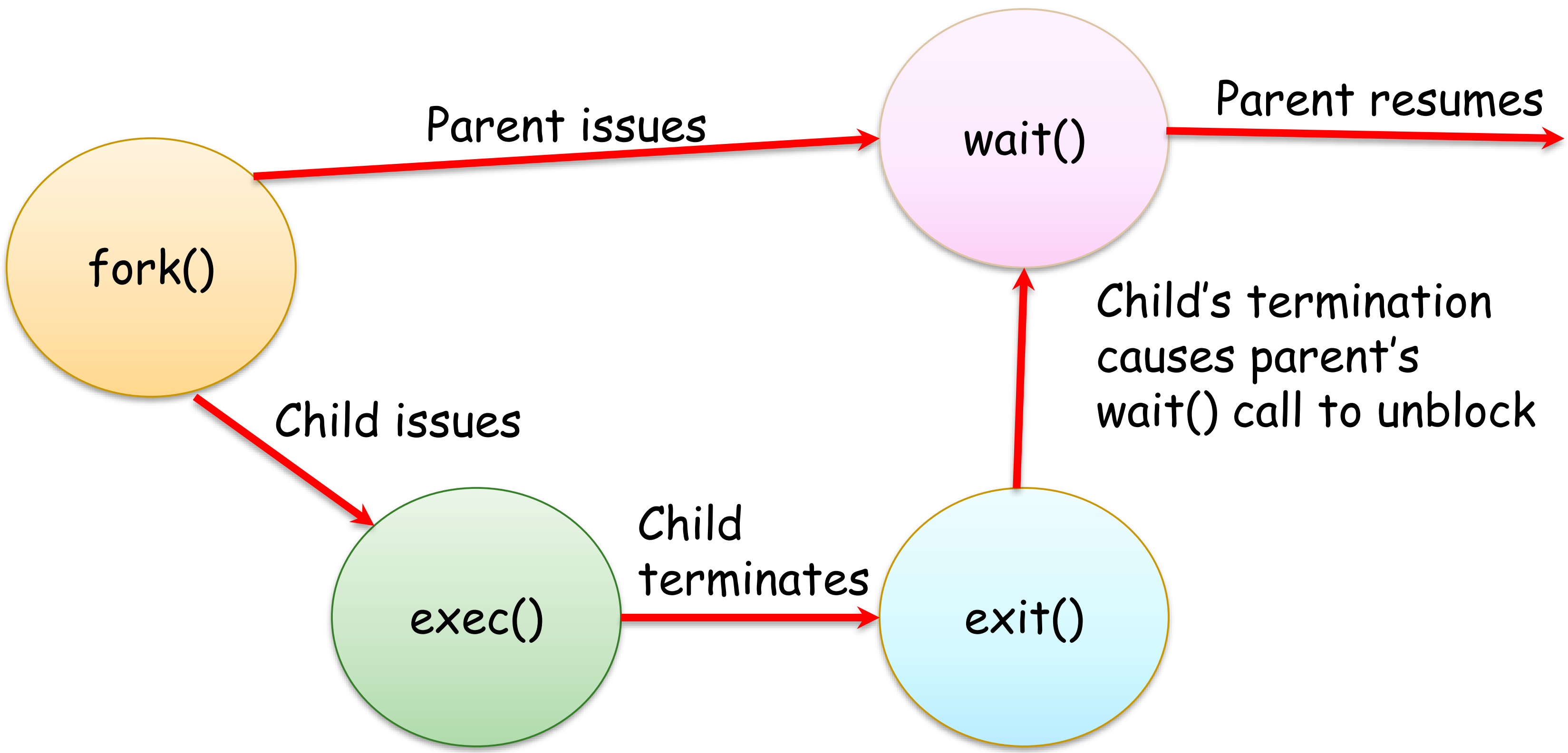
Operations on Processes: Process Creation in Unix/Linux: Putting it all Together

Parent process terminates.

```
//Parent process
int main()
{
pid_t pid;
	/* fork another process */
	pid = fork();
	if (pid < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed");
		exit(-1);
	}
	else if (pid == 0) { /* child process */
		execlp("/bin/ls", "ls", NULL);
	}
	else { /* parent process */
	/* parent will wait for the child to
	complete */
		wait (NULL);
		printf ("Child Complete");
		exit(0);
	}
}
```

Operations on Processes: Process Creation in Unix/Linux: Summary of fork()/exec()/wait()

- Process creation system call sequence.



Operations on Processes: Process Creation in Unix/Linux: zombie

- If a parent forks a child, but does not issue a wait() after the child terminates, the terminated child becomes a zombie process.
- Zombie process: a terminated process whose PCB was not deallocated i.e. PCB contains child's exit code e.g. the code returned by int main().
  - The child will remain a zombie until the parent calls wait().
- Child's exit code may be useful to the parent e.g. to see whether the child has exited with an error.

A C program to demonstrate Zombie Process.

```
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
// Fork returns process id
// in parent process
pid_t child_pid = fork();

// Parent process
if (child_pid > 0)
  sleep(50);

// Child process
else
  exit(0);
return 0; }
```

Operations on Processes: Process Creation in Unix/Linux: orphan

- What if the parent process terminates instead of calling wait() on the child?
  - The child becomes an orphan process.
  - init process becomes the new parent of the orphaned children.
  - init periodically calls wait() to collect the return statuses of orphans.

A C program to demonstrate Orphan Process

```c
// A C program to demonstrate Orphan Process.
// Parent process finishes execution while the
// child process is running. The child process
// becomes orphan.
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // Create a child process
    int pid = fork();
    if (pid > 0)
        printf("in parent process");

    // Note that pid is 0 in child process
    // and negative if fork() fails
    else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }
    return 0;
}
```

Orphans and Zombies (Demo)

- Zombies:
  - Compile and run zombie.cpp
  - Observe the behavior with ps
- Orphans:
  - Compile and run orphan.cpp
  - Run htop
  - Let the parent exit
  - Observe the changes in htop

Note that the above code may not work with online compilers as fork() is disabled

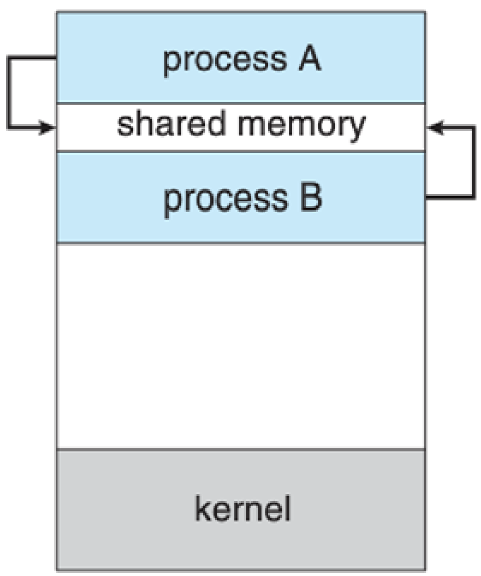
Interprocess Communications (IPC)

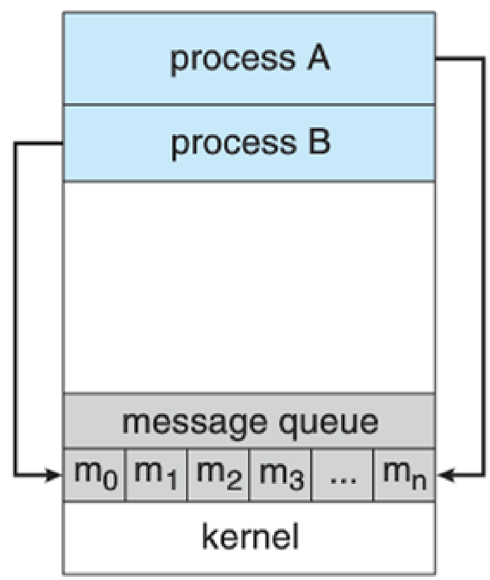
- A process can either be:
  - Independent: i.e. cannot affect or be affected by other processes.
  - Cooperating: process that can affect or be affected by other processes:
    - Example: if the process shares memory with other processes.
- Advantages of process cooperation:
  - Information sharing: e.g. exchanging data.
  - Computation speedup: e.g. break a task into subtasks and execute them concurrently on multiple processors.
  - Modularity: divide system functions into separate processes.
  - Convenience: working on many tasks at the same time e.g. editing, printing, etc.

Interprocess Communications

- Cooperating processes need a mechanism to exchange information i.e. interprocess communications (IPC).
- Fundamental IPC models:
  - Shared memory: cooperating processes exchange information by reading/writing data from/to a region of shared memory.
  - Message passing: cooperating processes exchange messages.

Shared memory





Message Passing

Interprocess Communications: Shared Memory vs. Message Passing

- Shared memory:
  - Faster than message passing: only requires intervention from the OS to establish a shared memory region.
  - Good for large transfers of information.
  - Disadvantage: requires process synchronization to ensure that e.g. no two processes write the same memory location at the same time.
- Message passing:
  - No need for synchronization.
  - Good for small information transfers.
  - Easier to implement than shared memory.
  - Disadvantage: usually requires OS intervention on every message transfer:
    - Can be slower than shared memory.

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Producer consumer problem: producer process produces information that is consumed by the consumer process
  - Example: webserver process produces HTML that is consumed by the web browser.
- Solution: use shared memory!
  - Approach 1: unbounded buffer
  - Approach 2: bounded buffer

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Unbounded buffer: no practical limits on the size of the shared buffer i.e. the size of shared memory.
  - The producer may produce items indefinitely.
  - Consumer waits until items are available.

How does the consumer know the items are available?
The length of the item?

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Bounded buffer: assumes a fixed size shared buffer.
  - Producer must wait if the buffer is full.
  - Consumer must wait if the buffer is empty.

Waiting time?

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Bounded buffer implementation (a wrap-around buffer):
  - Store the following variables in shared memory:

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];

int in = 0;    //First empty position in "buffer", producer's counter
int out = 0; //First full position in "buffer", consumer's counter
//The buffer is empty when in == out
//The buffer is full when ((in+1) % BUFFER_SIZE) == out.
```

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Bounded buffer implementation (a wrap-around buffer):
  - Producer code:

```
while (true)
{
        /* do nothing -- no free buffers, wait */
        while ((in + 1) % BUFFER_SIZE == out);

        //Produce an item
        …..
        //Save the produced item
        buffer[in] = item;
        //Compute the next free index
        in = (in + 1) % BUFFER_SIZE;
 }
```

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Bounded buffer implementation (a wrap-around buffer):
  - Consumer code:

```
while (true)
{
    //No items to consume, wait
    while (in == out);

   // Consume an item
   item = buffer[out];
   //Compute the index of the next item to consume
   out = (out + 1) % BUFFER SIZE;
   return item;
}
```

Interprocess Communications: Shared Memory: Producer Consumer Problem

- Bounded buffer implementation (a wrap-around buffer):
  - Problem: what if producer and consumer try to access the same buffer slot concurrently?
  - Solution: process synchronization (later in the course).

Interprocess Communications: Message Passing

- Message passing functions:
  - send(message): sends the message
  - receive(message): receives the message
- Messages can be either fixed-sized or variable-sized:
  - Fixed-sized: easier to implement, but imposes limitations.
  - Variable-sized: harder to implement, but is more flexible.
- Two processes exchange messages through an established link which can be implemented in many ways:
  - Direct or indirect communication.
  - Synchronous or asynchronous communication.
  - Automatic or explicit buffering.

Interprocess Communications: Message Passing: Direct Communication

- A process must explicitly name the sender or the receiver.
  - Symmetrical direct communication: both communicating processes must explicitly name the sender or the receiver:
    - send(P,message): send message "message" to process P.
    - receive(Q,message): receive a message from process Q.
    - A link is established automatically between all pairs of processes.
    - Each link is exactly between two processes.
    - Between each pair of processes there is only one link.

Interprocess Communications: Message Passing: Direct Communication

- A process must explicitly name the sender or the receiver.
  - Asymmetrical direct communication: similar to symmetrical, but only the sender must explicitly name the receiver.
    - send(P,message): send message to process P.
    - receive(id,message): receive the message from any process and save the sender's id in id.

Interprocess Communications: Message Passing: Direct Communication

- Problem: if the process changes the identifier, we must change the identifier in all places that use it.
  - Example: the receiver process saves all messages. If sender changes its identifier, receiver must change it in all saved messages.

Interprocess Communications: Message Passing: Indirect Communication

- Indirect communication: processes use mailboxes to send/receive messages:
  - send(A,message): send message to mailbox A.
  - receive(A,message): receive a message from mailbox A.
  - There is a link between two processes only if they share a mailbox.
  - A link may be associated with more than two processes.
  - Each pair of communicating processes must share a mailbox.

Interprocess Communications: Message Passing: Indirect Communication

- Problem: processes P_1, P_2, and P_3 share mailbox *A*:
  - Process P_1 places a message into *A*
  - Both P_2 and P_3 execute receive. Who should get the message?
- Solutions:
  - Restrict one link to at most two processes.
  - Allow only one process at a time to execute receive.
  - Select the receiver arbitrarily and notify the sender of the receiver's id.

Interprocess Communications: Message Passing: Synchronization

- How can we implement send() and receive()?
  - Blocking send: sender blocks until the receiver gets the message.
  - Nonblocking send: the sender sends the message and resumes operation.
  - Blocking receive: the receiver blocks until the message is available.
  - Nonblocking receive: the receiver retrieves either a valid message or a null.

Interprocess Communications: Message Passing: Synchronization

- When both send() and receive() are implemented as blocking, we say that there is a rendezvous between the sender and the receiver:
  - The receiver blocks until the message is available.
  - The sender blocks until the receiver gets the message.

Interprocess Communications: Message Passing: Synchronization: Buffering

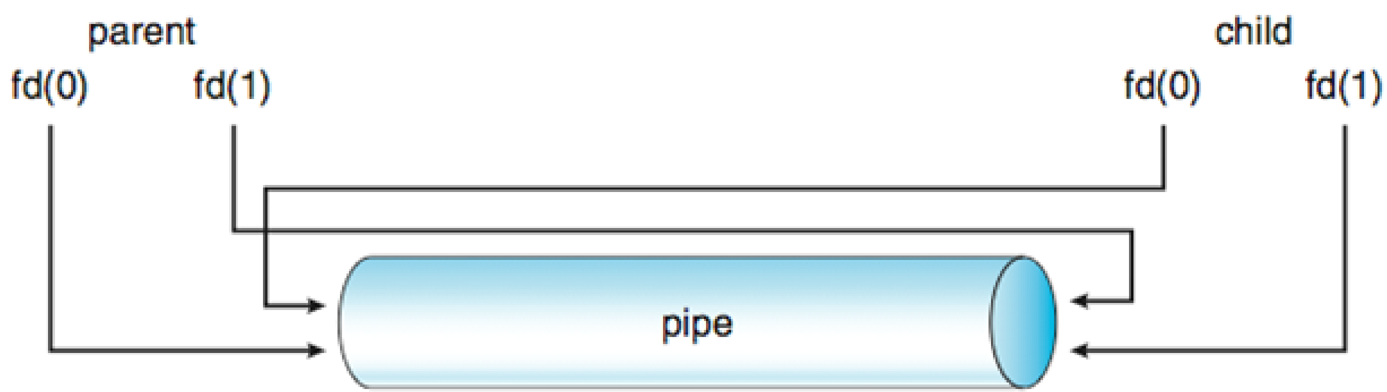
- Messages exchanged between processes must be placed in a temporary queue.
  - Question: how do we implement such queue?
- Implementing a queue:
  - Zero capacity (no buffering): the queue has a maximum length of zero; the sender must block until the message is received.
  - Bounded capacity: the queue has a finite capacity. When the capacity is exceeded the sender blocks.
  - Unbounded capacity: any number of messages can be placed in the queue; the sender never blocks.

Interprocess Communications: IPC examples: Pipes

- Pipe: acts as a channel between two processes utilizing standard input and output (**I/O**).
- Ordinary pipes: enable a straightforward, 1-way, producer-consumer communications.
  - Used by processes to exchange streams of unstructured data.
- A typical pipe comprises a front-end and a rear-end:
  - Producer: writes to the front-end of the pipe.
  - Consumer: reads the written information from the rear-end of the pipe.
- Bi-directional communications require two pipes.

Interprocess Communications: IPC examples: Pipes

- Example: in Unix ordinary pipes are used for communications between parents and children.
- pipe(fd) system call (where int fd[2]) creates a pipe where:
  - fd[0] is the read end
  - fd[1] is the write end
- A parent creates a pipe and forks a child.
- The child inherits the pipe because pipes are treated as files (recall: child inherits the state of files of the parent)
  - Example: if the file is opened in the parent at time of forking, then same file will also be opened in the newly created child).
- Parent and child should close the unused ends of the pipe.



Interprocess Communications: IPC examples: Pipes

- In Unix data can be read from/written to the pipe, using read()/write() system calls.
- Example: (next slide)

Interprocess Communications: IPC examples: Pipes

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main( void )
{
    char write_msg[BUFFER_SIZE] =
    "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t  pid;

    /*create pipe */
    if (pipe(fd) == -1) {
            fprintf(stderr, "Pipe
    failed.\n");
                return 1;
    }
    /* fork a child process */
    pid = fork();
```

```
        if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed\n");
            return 1;
            }

            else if (pid > 0) { /* parent process */
                    close (fd[READ_END]);

                    write(fd[WRITE_END],
write_msg, strlen(write_msg)+1);

                    close (fd[WRITE_END]);

                    wait(NULL);   //Wait for the
child
            }

            else { /* child process */
            close (fd[WRITE_END]);

            read(fd[READ_END], read_msg,
BUFFER_SIZE);
            printf("Read from pipe: %s \n",
read_msg);

            close (fd[READ_END]);
            }
            return 0;
}
```

Interprocess Communications: IPC examples: Named Pipes

- Named pipes (or FIFOs in Unix): A pipe implemented through a **file** on the file system instead of standard input and output. Multiple processes can read and write to the file as a buffer for IPC data.
  - Support bi-directional relay of data (one direction at a time).
  - Persist after the processes that use them have terminated (unlike ordinary pipes).
  - Must be explicitly removed.
  - Can be used for IPC between unrelated processes (and more than one).
  - We can create a FIFO in Unix shell using the mkfifo command:
    - Example:
      - Create a FIFO: mkfifo myfifo
      - Write a string to the FIFO: echo "Hello" > myfifo
      - Read a string from the FIFO: cat myfifo.

Interprocess Communications: IPC examples: Named Pipes

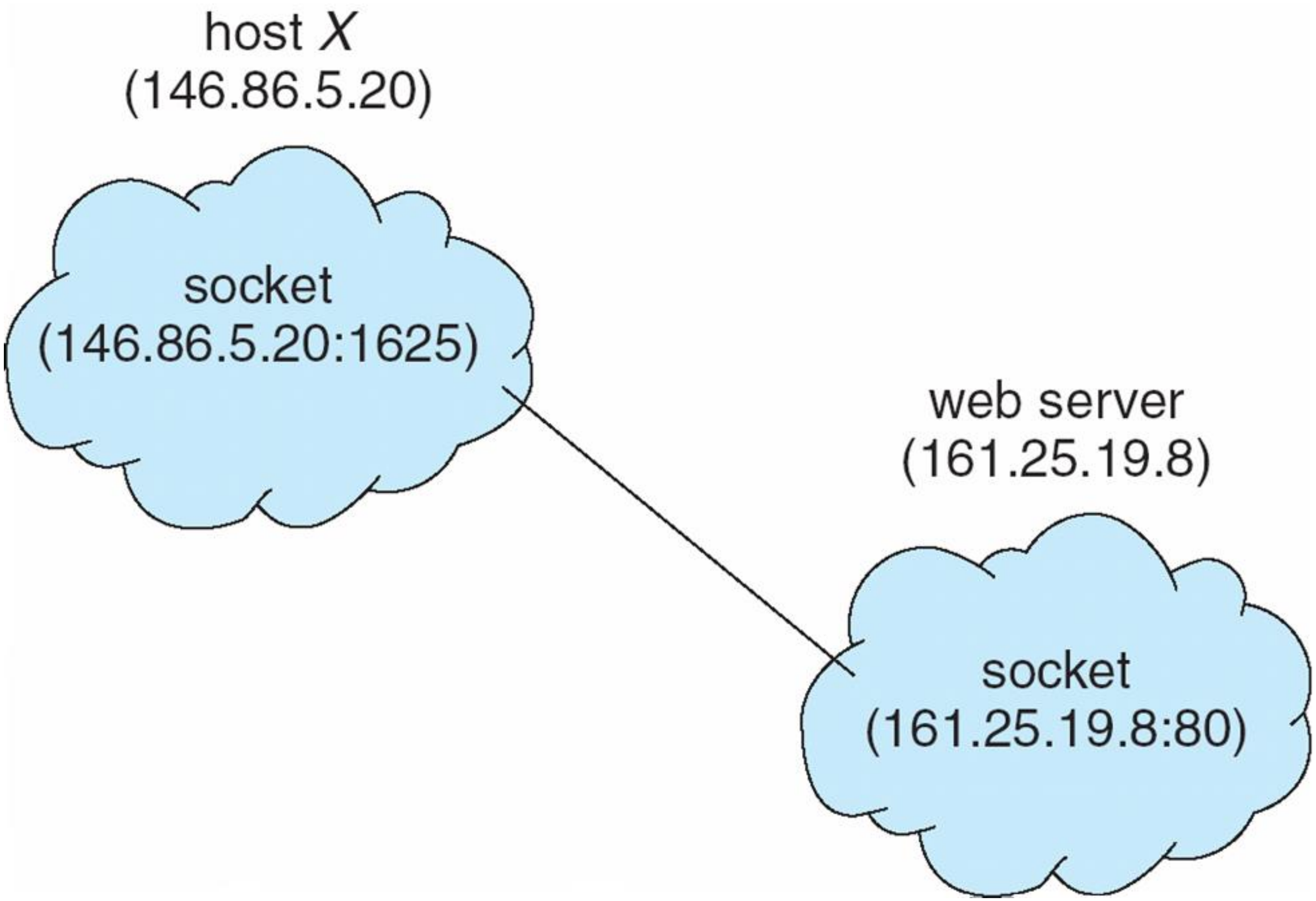
- Named pipes can also be created programmatically using mkfifo() system call.

- Example: mkfifo("myfifo", S_IWUSR | S_IRUSR)
  - Will create a FIFO called "myfifo".
  - The FIFO will be readable and writable by the user (i.e. the second parameter).
  - The FIFO can be read or written using fread(), fgets(), fstream, and other standard means of reading/writing files.

Interprocess Communications: IPC examples: Sockets

- Socket: an endpoint of communication.
- Pair of processes can communicate over the **network** using a pair of sockets.
- A socket is identified by concatenating an IP address of the system and a port number on the system.
- The mechanism works as follows:
  - A server listens on the port to which the client connects.
  - The server accepts the client's connection.
  - The server sets up a pair of sockets used for communications.
- Sockets provide means for low-level communications: unstructured byte stream.
- Reading/writing sockets is similar to pipes.

Interprocess Communications: IPC examples: Sockets Example



UNIX System V IPC

(pronounced: "**System** Five")

Interprocess Communications: IPC examples: System V Shared Memory

- Process allocates a shared memory region using shmget() (i.e. SHared MEmory GET) system call:
  - segment_id = shmget(key, size, S_IRUSR | S_IWUSR)
    - segment_id:
      - On success, unique identifier of the shared memory segment, or
      - -1 in case of error.
    - key: a key associated with the shared memory segment.
    - size: how much memory to allocate?
    - S_IRUSR | S_IWUSR flags: the memory is both readable and writable.
    - Other possible flags:
      - IPC_CREAT: create a memory segment with key key if the segment does not exist.
      - IPC_EXCL: exit with an error if IPC_CREAT flag is specified but the segment with key key already exists.

Interprocess Communications: IPC examples: System V Shared Memory

- Accessing a shared memory region:
  - shared_memory = (char*)shmat(segment_id, NULL,0);
    - segment_id: the segment id to attach to local memory.
    - shared_memory: a pointer to the beginning of the shared memory segment.

Interprocess Communications: IPC examples: System V Shared Memory

- Example: Processes P_1 and P_2 wish to communicate using shared memory. Assume that we want P_1 to be responsible for allocating the shared memory segment.
  - P_1:
    - Step 1: Create a shared memory region by invoking shmget() with key parameter of 123 and flag IPC_CREAT.
      - OS sees there is no shared memory region with key 123 and sees IPC_CREAT flag, so it allocates a new memory segment with key 123.
    - Step 2: Attach the allocated region by invoking shmat() with segment id returned by shmget() (in prev. step) as a parameter.
    - Step 3: Access shared memory through the pointer returned by shmat().
  - P_2: follows the same steps as process 1, except:
    - In step 1, no new memory region will be allocated; shmget() will return the segment ID of the region previously allocated by process 1 (OS knows that process 1 means that region, because process 2 invokes shmget() with the same key as process 1).

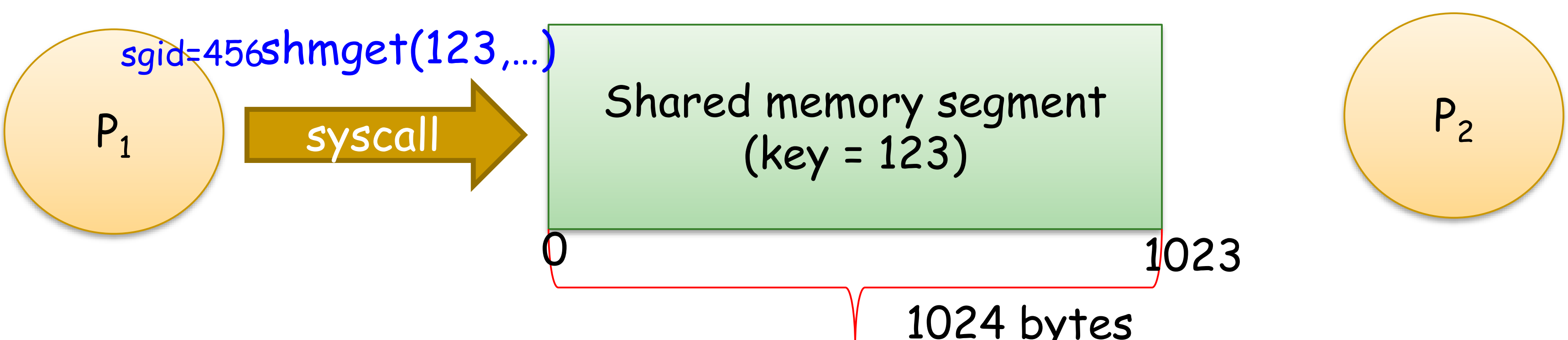
Interprocess Communications: IPC examples: System V Shared Memory

- Problem: how can we ensure that both communicating processes know the key of the shared memory segment?
- Solution: Both processes call ftok() function with the same arguments:
  - key_t key = ftok("/bin/ls", 'b');
    - Generates a key based on the random path e.g. "/bin/ls" and a random character e.g. 'b'.
    - Given the same path and character, will always generate the same key.

Interprocess Communications: IPC examples: System V Shared Memory: Visual Example

- Processes P_1 and P_2 would like to communicate:

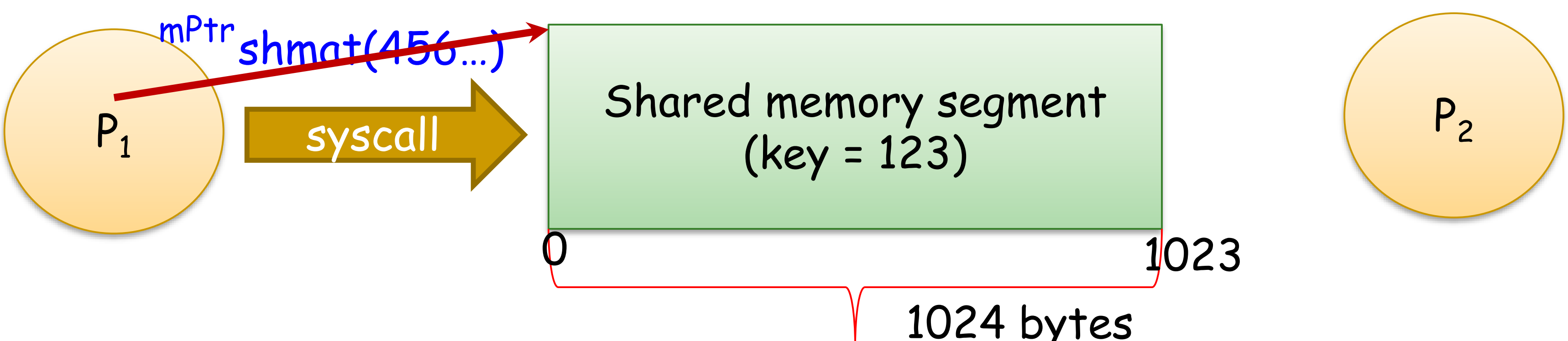
- Step 1: Process P_1 invokes e.g.,
  - sgid= shmget(key, size, S_IRUSR | S_IWUSR | IPC_CREAT) where key = 123 and size = 1024



Interprocess Communications: IPC examples: System V Shared Memory: Visual Example

- Processes P_1 and P_2 would like to communicate:

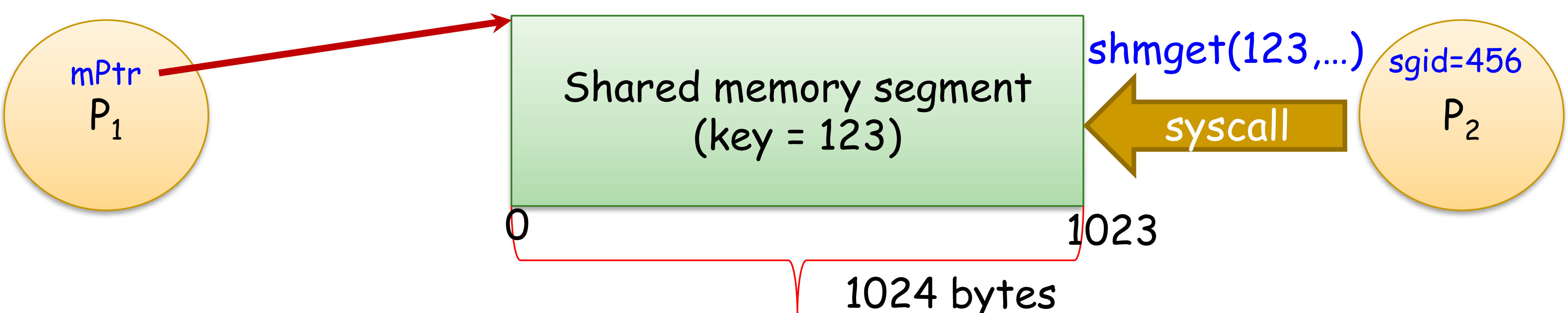
- Step 2: Process P_1 invokes e.g.,
  - mPtr= (char*)shmat(sgid, NULL,0);



Interprocess Communications: IPC examples: System V Shared Memory: Visual Example

- Processes P_1 and P_2 would like to communicate:

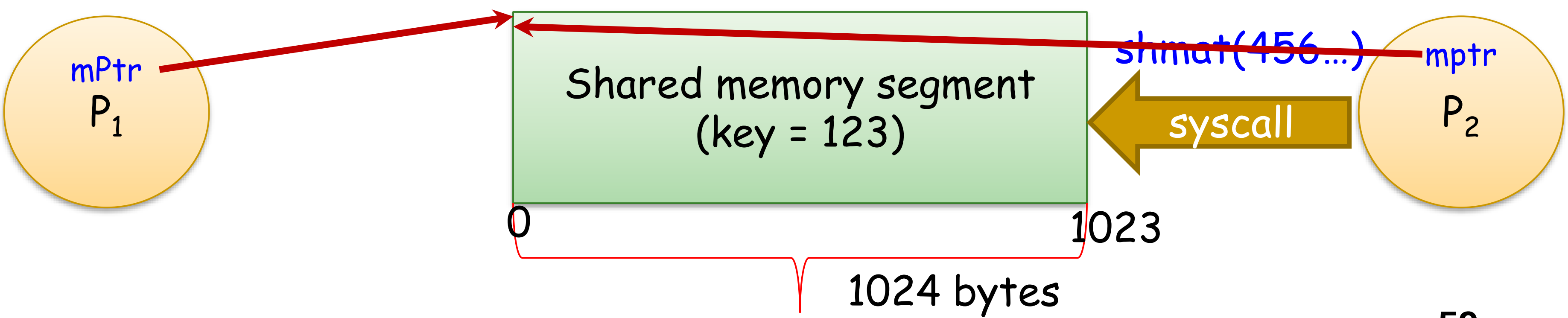
- **Step 4:** Process P_2 invokes e.g.,
  - sgid= shmget(key, size, S_IRUSR | S_IWUSR)
    where key = 123 and size = 1024



Interprocess Communications: IPC examples: System V Shared Memory: Visual Example

- Processes P_1 and P_2 would like to communicate:

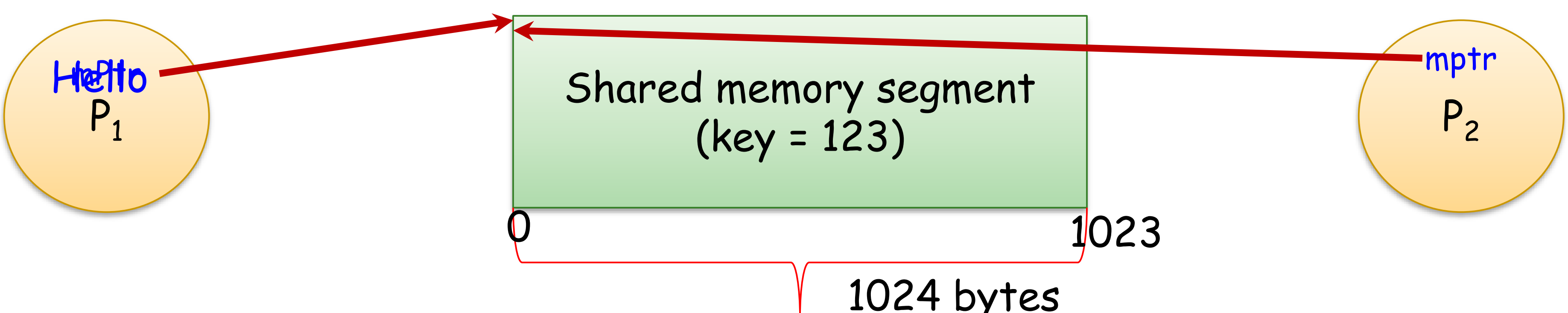
- Step 3: Process P_2 invokes e.g.,
  - sgid= shmget(key, size, S_IRUSR | S_IWUSR) where key = 123 and size = 1024



Interprocess Communications: IPC examples: System V Shared Memory: Visual Example

- Processes P_1 and P_2 would like to communicate:

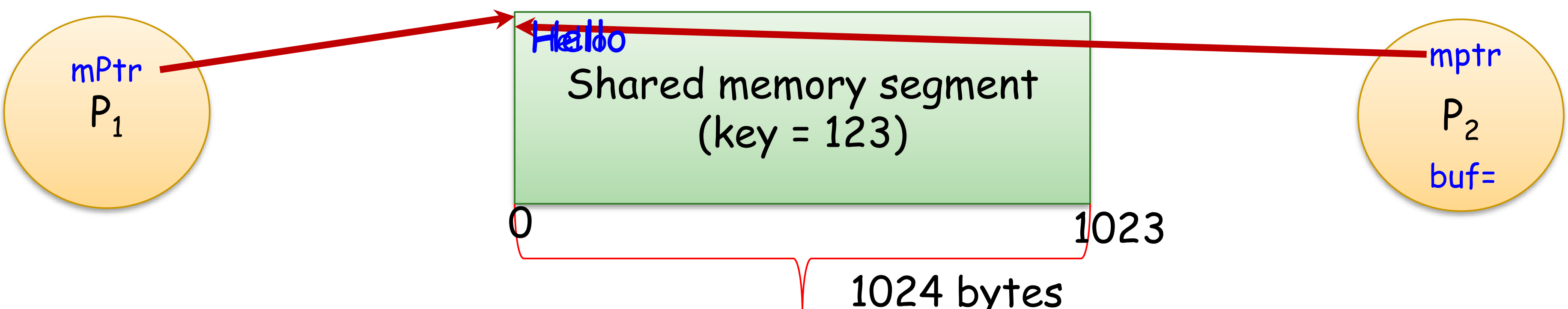
- Step 4: Process P_1 invokes e.g.,
  - strncpy(mPtr, "Hello", 6); // Copy 6 character string "Hello"
    // to shared memory



Interprocess Communications: IPC examples: System V Shared Memory: Visual Example

- Processes P_1 and P_2 would like to communicate:

- Step 5: Process P_2 invokes e.g.,
  - strncpy(buf, mPtr, 6); // Copy 6 character string "Hello"
    // from shared memory to local array
    // buff



Interprocess Communications: IPC examples: System V Shared Memory

- Writing to shared memory:
  - sprintf(shared_memory, "Hello world");
- Detaching shared memory:
  - shmdt(shared_memory);
- Deallocating shared memory segment:
  - shmctl(segment_id, IPC_RMID,…);

Interprocess Communications: How two processes can generate the same key?

- key_t key = ftok("/bin/ls", 'b'):
  - Generate a unique key based on the random path e.g., “/bin/ls“ and a random character e.g. ‘b’.
  - Given the same path and character, will always generate the same key.
  - Both processes agree upon the same file path and character and then use ftok() to generate the same key.

Interprocess Communications: IPC examples: System V Message Queues

- 1. Sender: Create message queue:
  - int msqid = msgget(key, S_IRUSR | S_IWUSR | IPC_CREAT);
    - Create a message queue with key key.
    - If the queue does not exist, then create it (IPC_CREAT flag).
    - S_IRUSR | S_IWUSR specifies the permissions (identical to as we seen in shared memory).
    - Returns the id of the created queue
      - Recall: same concept: key is similar to the file name and id is like the file handle you use for interacting with a file.
  - NOTE: either sender or the receiver create the queue. In the explanations that follow, we assume it is the sender.

Interprocess Communications: IPC examples: System V Message Queues

- 2. Sender: Create a message:
  - All messages are represented using struct.
  - The struct can have and can contain any elements.
  - However, the first element ***must be a long integer*** which will be used to represent the message type (to be explained soon)
  - Example:

```
/* Message Buffer */
struct msgBuff
{
	/* All message buffers must start with this long (name does not
	 *  matter). It's used by the receiver for message selection */
	long mtype;

	/* The actual data we want to send */
	 int someInt;
	 char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

- 3. Sender: Create an instance of the message buffer structure and populate it:
  - Set the first long integer (named mtype in this case) to a positive value.
    - This integer represents the ***message type***.
    - We will see that the receiver will use this value when checking for messages.
  - Populate other data fields to contain whatever data you want the message to carry.
  - Example:

```
msgBuff msg;    //Create an instance
msg.mtype = 2;  //Set the message type (e.g., 2)
//Set the data fields
msg.someInt = 123;
strncpy(msg.data, "Hello World", 12);
```

```
/* Message Buffer */
struct msgBuff
{
   /* All message buffers
    * must start with this
    * long (name  does not
    * matter). It's  used
by
    * the receiver  for
    * message selection
    */
   long mtype;

    /* The actual data we
     * want to send  */
   char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

- 4. Sender: place the message into the queue:
  - msgsnd(msqid, &msg, sizeof(msgBuff) - sizeof(long), 0);
    - msqid: the id of the message queue into which to place a message.
    - msg: the message to send
    - sizeof(msgBuff) - sizeof(long): the size of the payload (i.e., total message size – the size of mtype).
    - Example:

Note: sizeof(arg) in C/C++ returns the size in bytes of arg. E.g., on 32-bit system sizeof(int) is 4, sizeof(long) is 4, sizeof(char) is 1.

sizeof(long)

//Payload size

sizeof(msgBuff) - sizeof(long)

```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

sizeof(msgBuff)

    - 0: miscellaneous flags (can leave as 0).

Interprocess Communications: IPC examples: System V Message Queues

- 4. Sender: place the message into the queue:
  - msgsnd(msqid, &msg, sizeof(msgBuff) - sizeof(long), 0);
    - msqid: the id of the message queue into which to place a message.
    - msg: the message to send
    - sizeof(msgBuff) - sizeof(long): the size of the payload (i.e., total message size – the size of mtype).
    - Example:

sizeof(mtype) is 4
sizeof(int) is 4
sizeof(data) is 100 (i.e., sizeof(char) * 100)
sizeof(msgBuff) = 4 + 4 + 100 = 108

```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

sizeof(long)

//Payload size
sizeof(msgBuff) - sizeof(long)

sizeof(msgBuff)

    - 0: miscellaneous flags (can leave as 0).

Interprocess Communications: IPC examples: System V Message Queues

- 4. Sender: place the message into the queue:
  - msgsnd(msqid, &msg, sizeof(msgBuff) - sizeof(long), 0);
    - msqid: the id of the message queue into which to place a message.
    - msg: the message to send
    - sizeof(msgBuff) - sizeof(long): the size of the payload (i.e., total message size – the size of mtype).
    - Example:

sizeof(mtype) is 4
sizeof(int) is 4
sizeof(data) is 100 (i.e., sizeof(char) * 100)
sizeof(msgBuff) = 4 + 4 + 100 = 108

```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

sizeof(long)

//Payload size
sizeof(msgBuff) - sizeof(long)
108 - 4 = 104

sizeof(msgBuff)

    - 0: miscellaneous flags (can leave as 0).

Interprocess Communications: IPC examples: System V Message Queues

- 5. Receiver: get the handle to the message queue created by the sender in step 1.
  - int msqid = msgget(key, S_IRUSR | S_IWUSR);
    - Get the id of the message queue associated with key key.
    - DO NOT create the queue if it does not exist (hence, no IPC_CREAT flag here, unlike in step 1)

Interprocess Communications: IPC examples: System V Message Queues

- 6. Receiver: Declare a message buffer to store the received message:
  - Use the same message buffer structure as the sender.
  - Declare an instance of the message buffer: msgBuff msg;

```
/* Message Buffer */
struct msgBuff
{
    /* All message buffers
     * must start with this
     * long (name  does not
     * matter). It's  used by
     * the receiver  for
     * message selection
     */
    long mtype;

     /* The actual data we
      * want to send  */
     char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

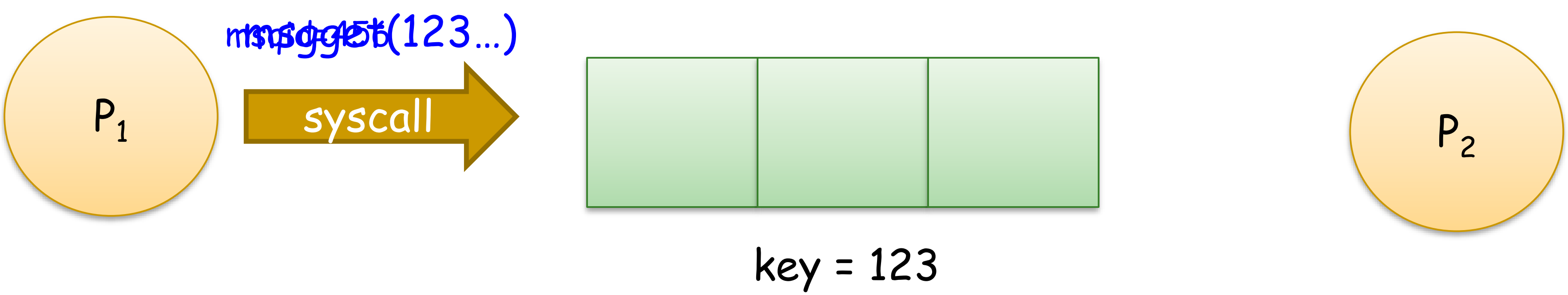
- 7. Receiver: retrieve the message from the queue:
  - msgrcv(msqid, &msg, sizeof(msgBuff) - sizeof(long), 2, 0):
    - msqid: the id of the queue from which to retrieve the message.
    - msg: the buffer where to store the received message.
    - sizeof(msgBuff) - sizeof(long): the size of payload (i.e. total message size – the size of mtype).
    - 2: the mtype of the message to retrieve. Must match the mtype in the message specified by the sender.
    - 0: miscellaneous flags (can leave as 0).

Interprocess Communications: IPC examples: System V Message Queues

- For example, two processes would like to communicate through System V message queues

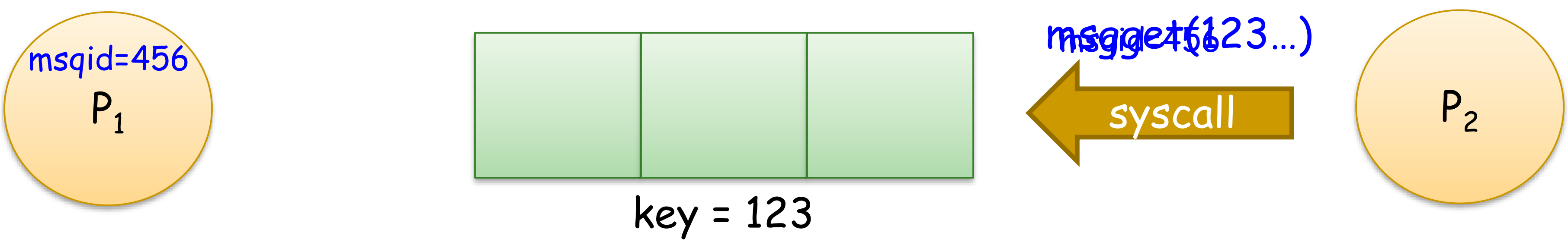
Interprocess Communications: IPC examples: System V Message Queues

- Step 1. P_1 issues e.g. system call
  - int msqid = msgget(key, S_IRUSR | S_IWUSR | IPC_CREAT);
    where key = 123



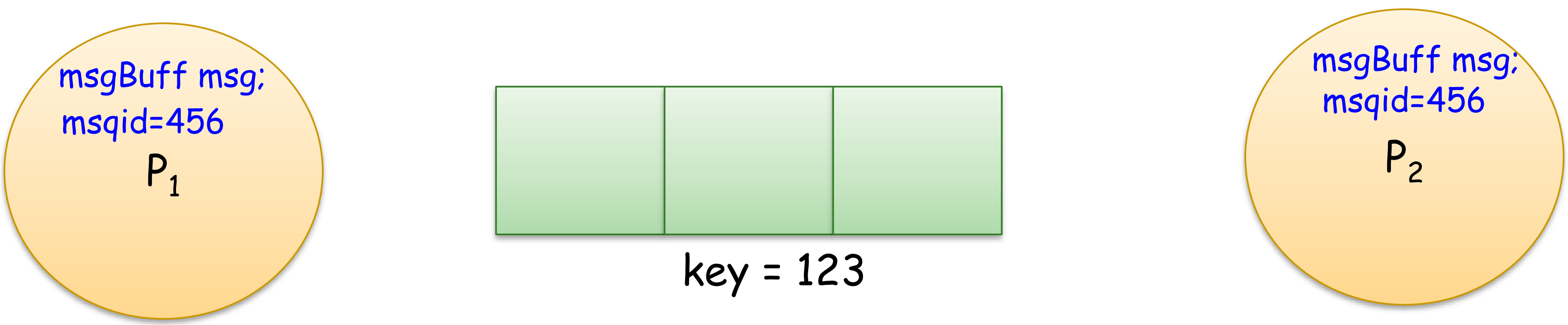
Interprocess Communications: IPC examples: System V Message Queues

- Step 2. P_2 issues e.g. system call
  - int msqid = msgget(key, S_IRUSR | S_IWUSR);
    where key = 123



Interprocess Communications: IPC examples: System V Message Queues

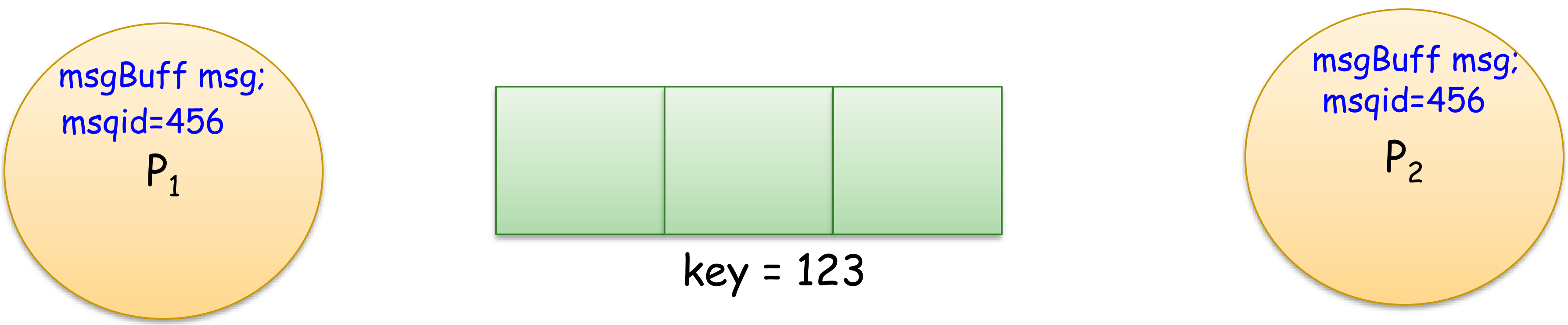
- Step 3. P_1 and P_2 both declare an instance of the message buffer struct (which, assume has the structure as shown below).




```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

- Step 4. P_1 populates the message structure and sets the first long (i.e., the mtype field) to e.g., 2

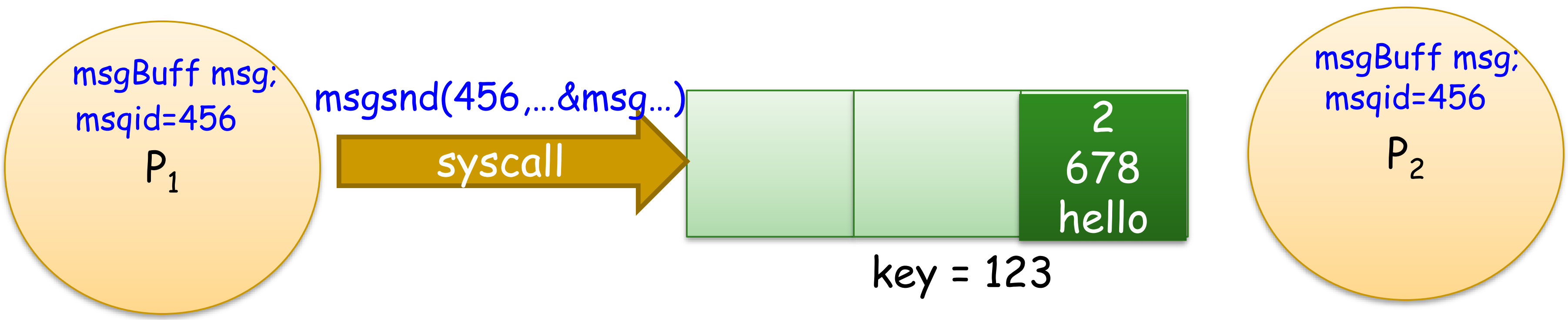



```
msgBuff msg;
msg.mtype = 2
msg.someInt = 678
strncpy(msg.data, "hello", 6);
```

```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

- Step 5. P_1 invokes msgsnd() to plate the message into the message queue:
  - msgsnd(msqid, &msg, sizeof(msgBuff) – sizeof(long), 0);

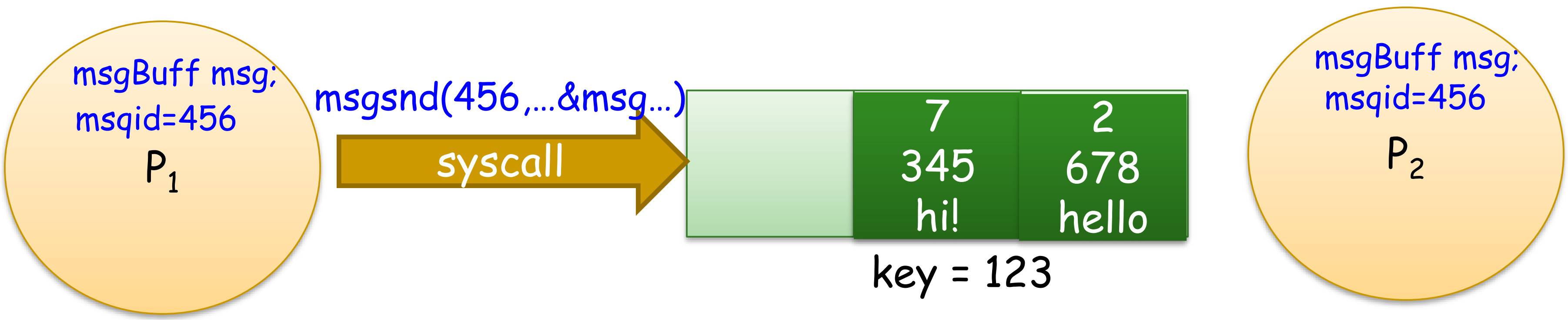



```
msgBuff msg;
msg.mtype = 2
msg.someInt = 678
strncpy(msg.data, "hello", 6);
```

```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

- **Step 6.** Suppose P_1 invokes msgsnd() again, but with a different message that has a different message type:
  - msgsnd(msqid, &msg, sizeof(msgBuff) – sizeof(long), 0);

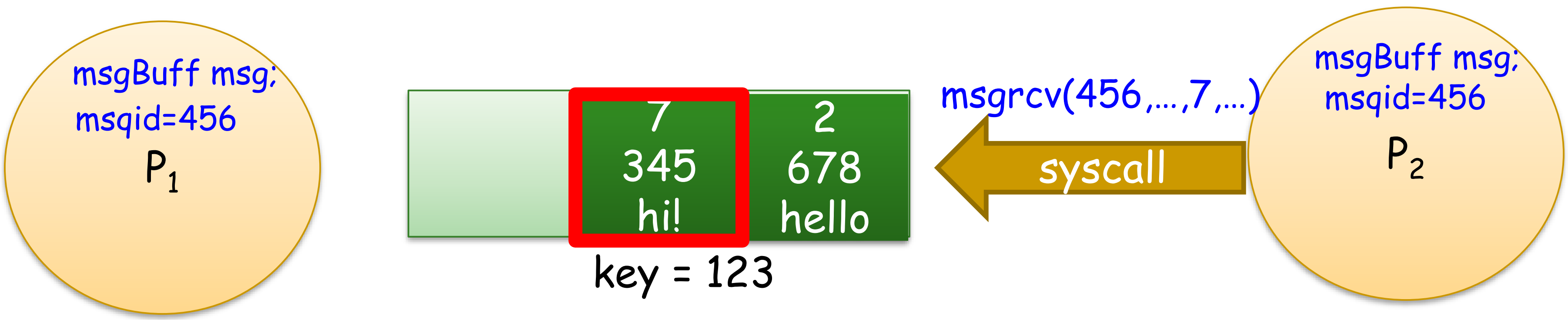



```
msgBuff msg;
msg.mtype = 7
msg.someInt = 345
strncpy(msg.data, "hi!", 4);
```

```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

Interprocess Communications: IPC examples: System V Message Queues

- Step 7. P_2 invokes msgrcv() with message type of e.g., 7:
  - msgrcv(msqid, &msg, sizeof(msgBuff) – sizeof(long), 7, 0);

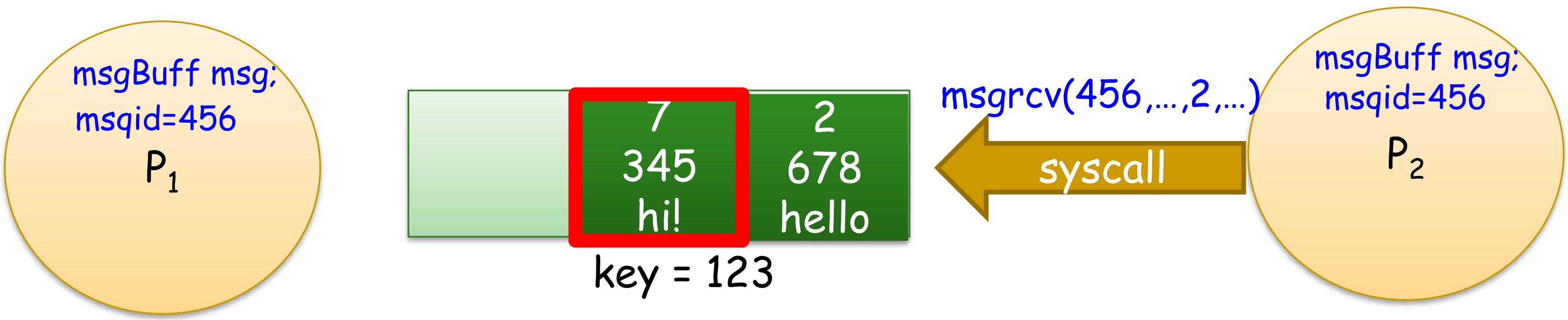



```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

OS searchers the message queue for a message whose first field is set to the value of 7…

Interprocess Communications: IPC examples: System V Message Queues

- Step 7. P_2 invokes msgrcv() with message type of e.g., 7:
  - msgrcv(msqid, sizeof(msgBuff) – sizeof(long), &msg, 7, 0);




```
struct msgBuff
{
    long mtype;
    int someInt;
    char data[100];
};
```

msg.mtype = 7
msg.someInt = 345
msg.data = "hi!"

The found message is copied into the instance of the message buffer of P2…

Interprocess Communications: IPC examples: System V Message Queues

- Deallocating a message queue:
  - msgctl(msqid, IPC_RMID, NULL);
    - msqid: the message queue id.
    - IPC_RMID: flag indicating that we would like to deallocate a message queue.
    - NULL: last argument can always be set to NULL when removing message queues.

Interprocess Communications: IPC examples: Mach

- Mach: an OS developed at Carnegie Mellon.
- All communications are based on message passing:
  - Even system calls are messages.
  - Each task gets two mailboxes at creation: Kernel and Notify
  - Three system calls used for message transfer:
    - msg_send()
    - msg_receive()
    - msg_rpc()
  - Mailboxes needed for communication, are created using port_allocate().

Interprocess Communications: IPC examples: Windows XP

- Messages are passed via local procedure call (LPC) facility.
- Uses ports to establish and maintain connections between processes.
- Two types of ports:
  - Connection ports: visible to all processes; used to set up communication ports.
  - Communication ports: used for actual communications.

Interprocess Communications: IPC examples: Windows XP

- The mechanism works as follows:
  - The client opens a handle (i.e. interface) to the server's (published) connection port object.
  - The client sends a connection request.
  - The server creates two private communications ports and returns the handle to one of them to the client.
  - The client and server use the corresponding port handles to send messages.