Advanced System Programming

Agenda

- IPC
- Process Scheduling
- Thread Scheduling

IPC – Message Queue

Message Queue

- Another IPC mechanism in Linux.

- Implemented both SYS V type and POSIX type and it's structure is somehow like pipes (FIFO).

- A process initializes a message queue and itself or other processes can put messages in this queue, knowing the MSQID of this message queue.

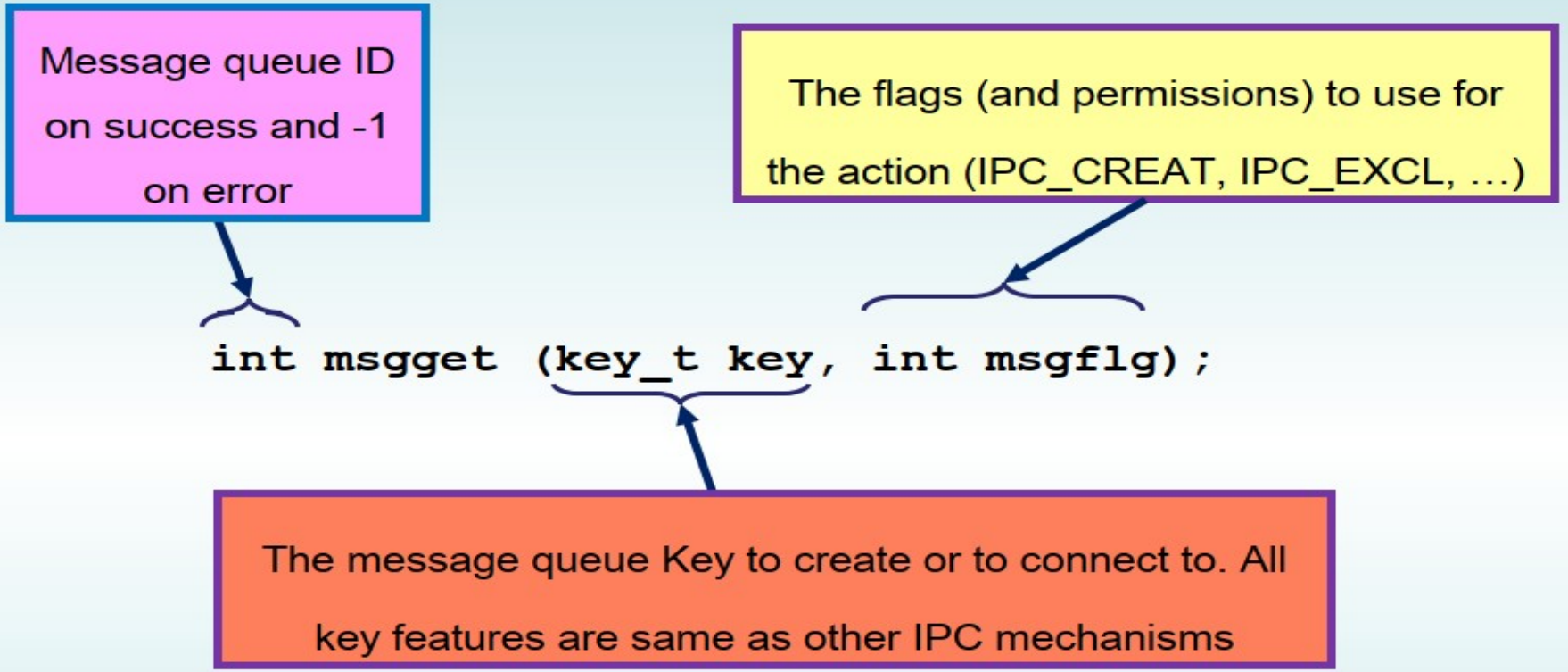
- After finishing the job, one process should deallocate the queue.

- If you don't remove a message queue it will remain even after process termination.

- You can use *ipcs* command to view the current message queues.

IPC – Message Queue

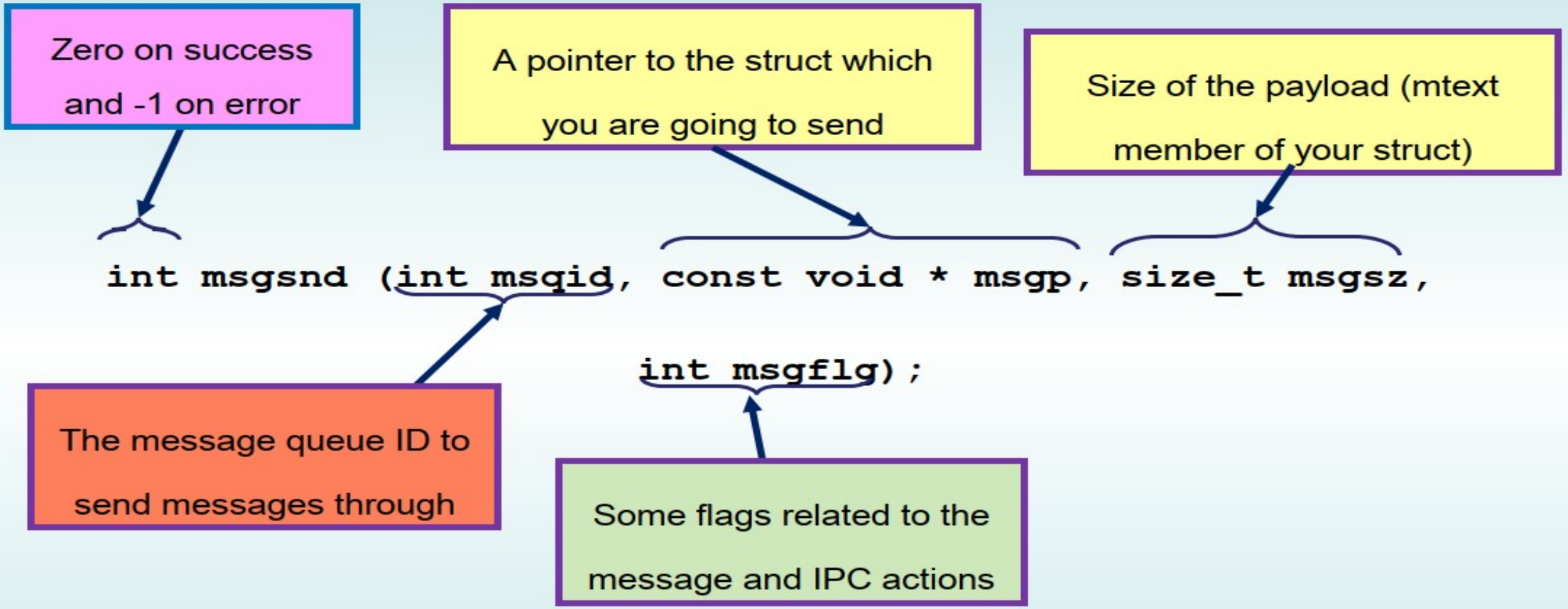
Message Queue



- You can connect to a queue and also create one, using *msgget()*.

IPC – Message Queue

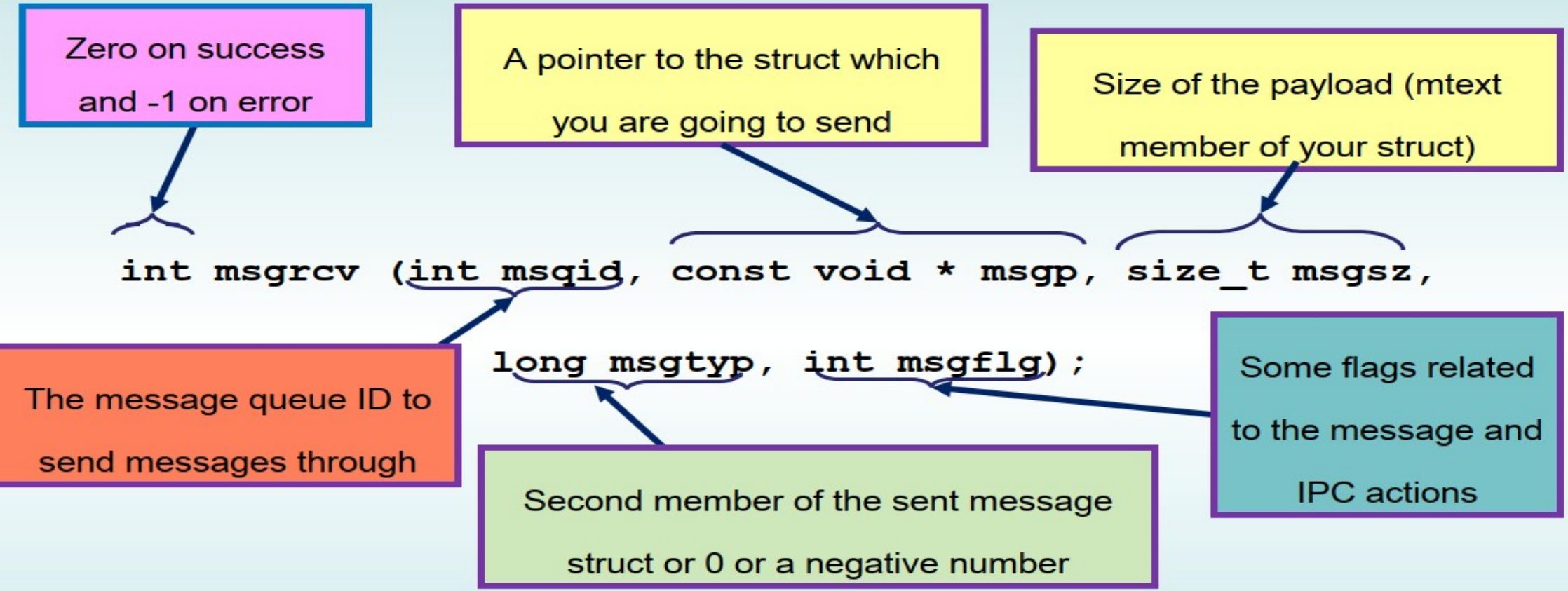
Message Queue



- The message you are going to send should be a struct containing two members: m_type and m_text.

IPC – Message Queue

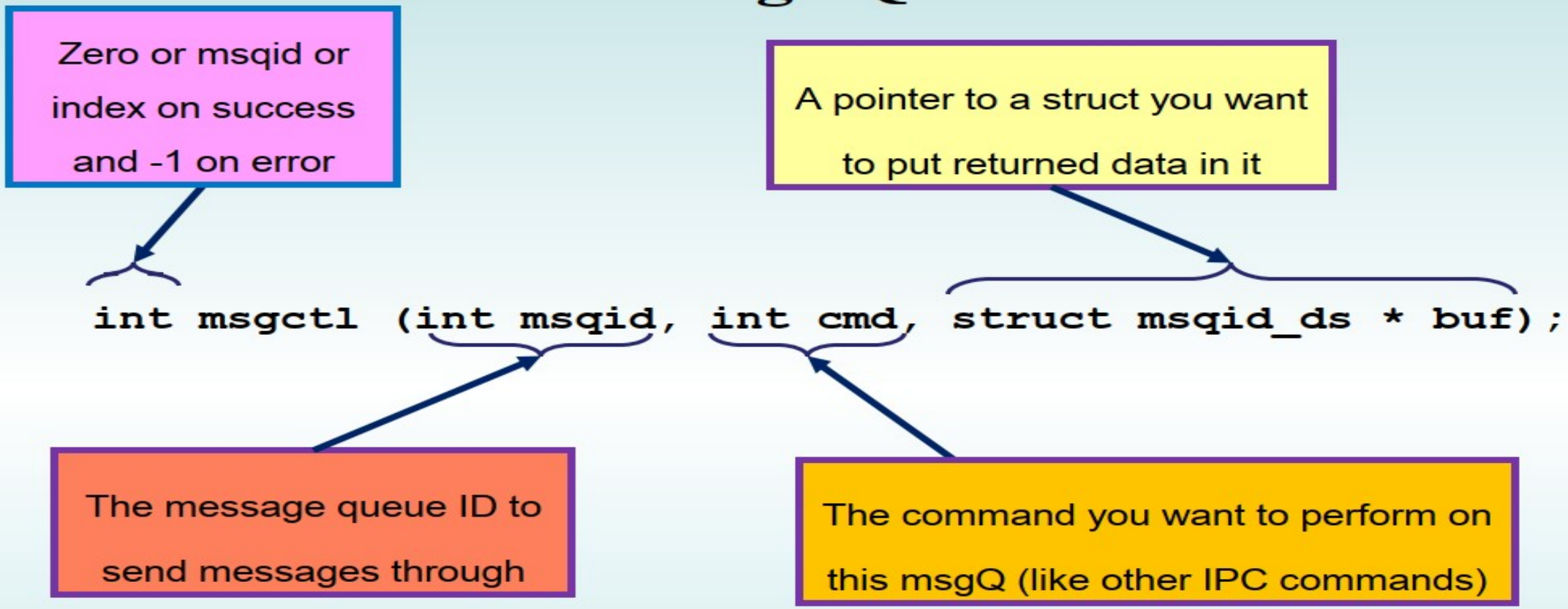
Message Queue



- If the message has size bigger than what is said here, the truncated message could be lost (MSG_NOERROR flag)

IPC – Message Queue

Message Queue



- The msqid_ds type is described in sys/msg.h and contains information about desired message queue.

IPC – Posix Message Queue

POSIX supports asynchronous, indirect message passing through the notion of message queues

A message queue can have many readers and many writers

Priority may be associated with the queue

Intended for communication between processes (not threads)

Message queues have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.

An attribute object is used to set the queue attributes when the queue is created

IPC – Posix Message Queue

Message queues are given a name when they are created

To gain access to the queue, requires an mq_open name

mq_open is used to both create and open an already existing queue (also mq_close and mq_unlink)

Sending and receiving messages is done via mq_send and mq_receive

Data is read/written from/to a character buffer.

If the buffer is full or empty, the sending/receiving process is blocked unless the attribute O_NONBLOCK has been set for the queue (in which case an error return is given)

If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified

Shared Memory

- Is the fastest form of interprocess communication and is also called: Fast Local Communication.

- It allows two or more processes to access the same memory.

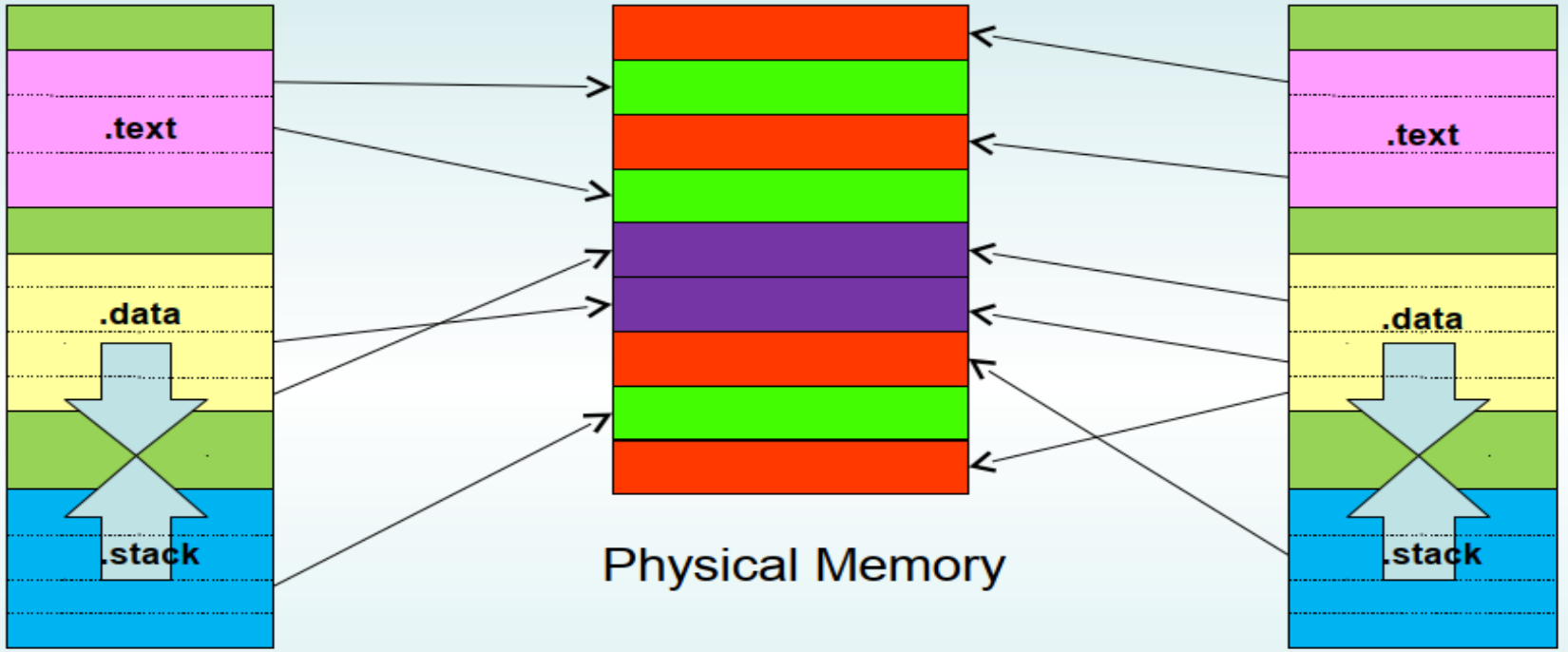
- Linux kernel does not take care of synchronization between processes to access the shared memory.

- Process semaphores are a suitable way of synchronization.

- Using shared memory for each process is like calling *malloc.*

IPC – Shared Memory

Shared Memory



Process A
Memory mapping

Process B
Memory mapping

Shared Memory

Using Shared Memory

- To use shared memory in Linux, one process must allocate the segment.

- Each process desiring to access the segment, must attach the segment and after finishing its job must detach the segment.

- One process at the end must deallocate the segment.

- Allocating a new shared memory segment, causes virtual memory pages to be created.

Using Shared Memory

- Allocating an existing segment, does not create new pages, but returns and identifier to the existing ones.

- All shared memory segments are allocated as integral multiples of the system's page size.

- On Linux systems, page size is 4KB.

- You should obtain the page size by calling the *getpagesize()* function.

Using Shared Memory

- You can use *ipcs -m* command to see currently assigned shared segments.

- Using *ipcrm -m* command, you can remove unused shared segments left behind by processes.

- There are some limitations on using shared memory in linux.

- *SHMALL, SHMMAX, SHMMIN, SHMMNI* are corresponding values to these limitations which are located under /proc/sys/kernel/.

Using Shared Memory

- SHMALL: system-wide maximum of shared memory pages.

- SHMMAX: maximum size in bytes for a shared memory segment.

- SHMMIN: minimum size in bytes for a shared memory segment.

- SHMMNI: system-wide maximum number of shared segments.

If the minimum size of a shared memory segment is equal to *page size*, then what is *SHMMIN*?

Using Shared Memory

- A process may allocate a shared memory segment using *shmget()*.

- Its first argument is a key specified to the shared segment.

- Other processes can access to the same shared memory segment using the same key.

- To ensure that the key is not previously used, you can use the special constant IPC_PRIVATE.

IPC – Shared Memory

Using Shared Memory

```
int shmget ( key_t key, size_t size, int shmflg )
```

A valid segment identifier on success and -1 on error.

The key you wish to specify for the shared segment.

The segment size (will be rounded up to the multiple of page size).

Permission and other specifications of the shared segment.

Using Shared Memory

- shmflg is the logical OR of flags. The most useful flags are:

  • IPC_CREAT: Is used to create a new shared segment.

  • IPC_EXCL: Is used with IPC_CREAT to ensure failure if the segment already exists.

  • Mode flags (see the manual page of stat.h for details).

If IPC_CREAT is used without IPC_EXCL, and the segment key already exists, the existing segment's id will be returned and no error will occur.

Using Shared Memory

- Permission flags are:

Mode bit	Meaning
S_IRWXU	R, W, X by owner
S_IRUSR	Read by owner
S_IWUSR	Write by owner
S_IXUSR	Execute by owner
S_IRWXG	R, W, X by group
S_IRGRP	Read by group
S_IWGRP	Write by group
S_IXGRP	Execute by group
S_IRWXO	R, W, X by other
S_IROTH	Read by other
S_IWOTH	Write by other
S_IXOTH	Execute by other

Using Shared Memory

- To use a shared memory segment, a process must attach it.

- *shmat()* is used to attach to a shared memory segment with given segment identifier.

- You can tell *shmat()* where in your process address space to map the shared memory.

- If you call *fork()* the child will inherit the shared memory.

- When you finished with shared memory, you can detach it using *shmdt()*.

Using Shared Memory

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

On success, will return the address of attached shared memory. On error, -1 is returned and errno is set.

The address in your process address space in which you want the shared memory be mapped.

The segment ID (returned by *shmget()*).

Could be ***SHM_RND,*** ***SHM_RDONLY,*** ***SHM_REMAP*** *(Linux specific)*

Using Shared Memory

Any process who knows the key, can

access to this shared memory segment

(regarding to permission flags)

Using Shared Memory

- The *shmctl()* returns information about a shared memory segment and can modify it.

- Using *shmctl()* you can also deallocate a shared memory segment.

- Each shared memory segment should be deallocated explicitly.

- The *shmctl()* fills the *shmid_ds* type structure.

IPC – Shared Memory

Using Shared Memory

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

On success depends on cmd. On error, -1 is returned and errno is set.

The segment ID (returned by *shmget()*).

IPC_STAT, IPC_SET, IPC_RMID, ***RPC_INFO*** *(Linux specific),* ***SHM_INFO*** *(Linux specific),* ***SHM_STAT*** *(Linux specific),* ***SHM_LOCK*** *(Linux specific),* ***SHM_UNLOCK*** *(Linux specific).*

An structure which contains the information you want to be set or to be read.

Process Semaphore

- Processes must coordinate access to shared memory.

- Process semaphores like thread semaphores are kind of counter with two operations: *POST* & *WAIT*.

- Process semaphores come in sets.

- The last process using a semaphore set, must explicitly remove it.

- Unlike shared memory, removing a semaphore set causes Linux to deallocate immediately.

Process Semaphore

- To use a semaphore set, you should first allocate it using *semget()*.

- The *semget()* system call, will return a semaphore ID regarding to the key you gave it before.

- After allocating the semaphore, you should initialize it using *semctl()* .

- After initializing the semaphore set, you can do POST or WAIT on it using *semop()* system call.

- The last process must invoke *semctl()* to remove the semaphore.

Process Semaphore

- Allocating an existing semaphore will return it's semaphore ID.

- *semget()* flags behave the same way as *shmget()* does.

- You can use *ipcs –s* command to view information about existing semaphore sets.

- Using *ipcrm –s* you may remove the semaphore sets.

- Each semaphore in a set has the following associated values:

```
unsigned short semval;      /* semaphore value */
unsigned short semzcnt;     /* # waiting for zero */
unsigned short semncnt;     /* # waiting for increase */
pid_t sempid;               /* PID that did last OP */
```

Allocating a semaphore set

```
int semget ( key_t key, int nsems, int semflg )
```

A valid semaphore identifier on success and -1 on error.

The key you wish to specify for the semaphore set.

The number of semaphores you wish to have in this set.

Permission and other specifications of the semaphore set.

Initializing Semaphores

- To initialize a semaphore set, you must do:
  - Set the semaphore value of all members to desired values.
  - Set the last change time of all members.
  - Set other specifications of semaphore set members.
- To do so, you can use *semctl()* function.
- As mentioned in *semctl()* manual page, the calling program must define a *semun* union.


IPC – Semaphore

Initializing Semaphores

```
int semctl ( int semid, int semnum, int cmd, union semun args )
```

On success depends on cmd. On error, -1 is returned and errno is set.

Number of desired semaphore in set.

Use of fourth argument is depended on CMD (might be ignored).

The semaphore ID (returned by *semget()*).

IPC_STAT, IPC_SET, IPC_RMID, ***IPC_INFO*** *(Linux specific),* ***SEM_INFO*** *(Linux specific),* ***SEM_STAT*** *(Linux specific),* ***GETALL, SETALL, GETVAL, SETVAL, GETNCNT, GETZCNT, GETPID***

Initializing Semaphores

- Depending on CMD, you may need to provide the fourth argument.

- If so, you must define the *union semun* yourself like below:

```
union semun
{
        int val;    /* Value for SETVAL */
        struct semid_ds * buf;    /* Buffer for IPC_STAT, IPC_SET */
        unsigned short int * array;   /* Array for GETALL, SETALL */
        struct seminfo * __buf; /* Buffer for IPC_INFO (linux specific) */
};
```

semid_ds and *ipc_perm* (an structure in *semid_ds*) are filled with information about semaphores


Wait and post operation

- To do *wait* and *post* on a semaphore in a set, you can use *semop()*.

- *semop()* performs desired operation on selected semaphores in a set.

- *semop()* takes an array of operation structure.

- Each operation structure is related to one semaphore in a set.

- Operation structure is of type *sembuf* and contains:

```
unsigned short sem_num;   /* Semaphore number */
short sem_op;    /* Semaphore operation */
short sem_flg;    /* Semaphore flags */
```

Semaphore Deallocation

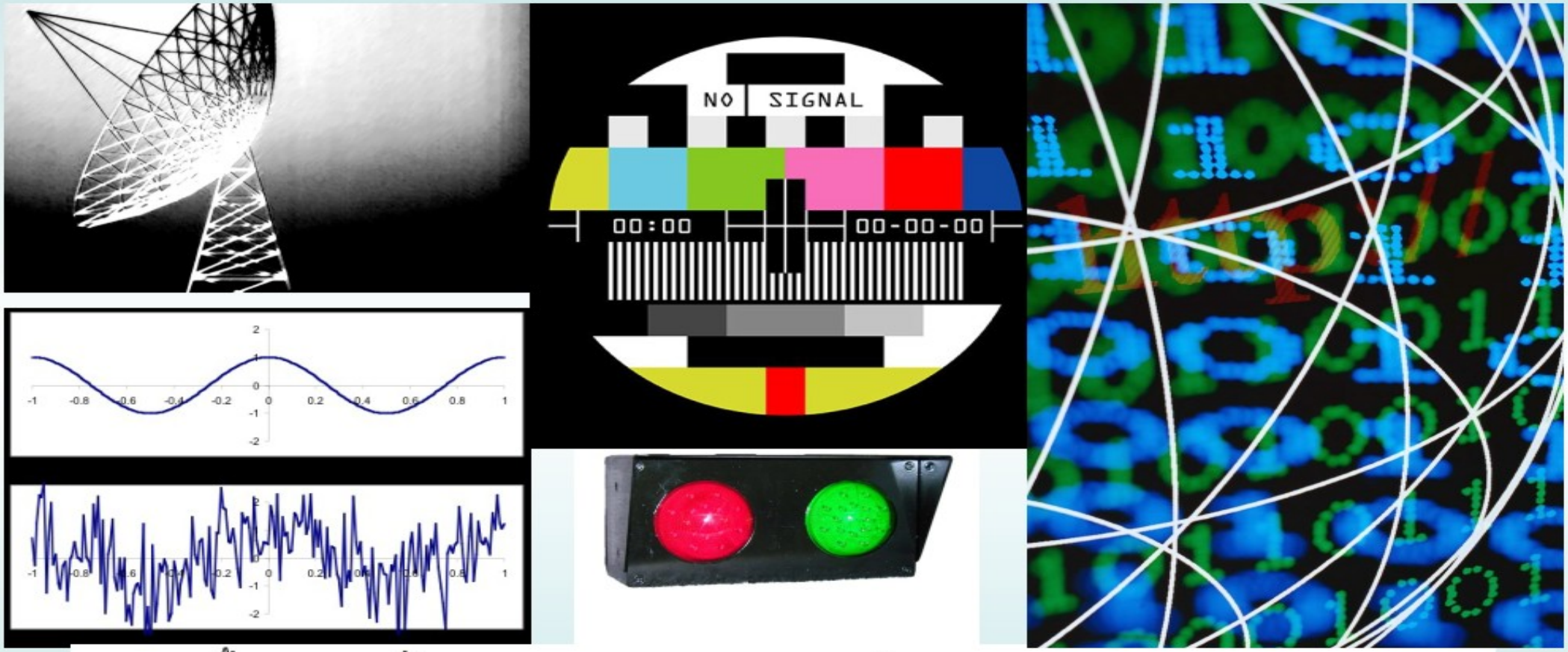
- The semaphore operations are performed atomically.

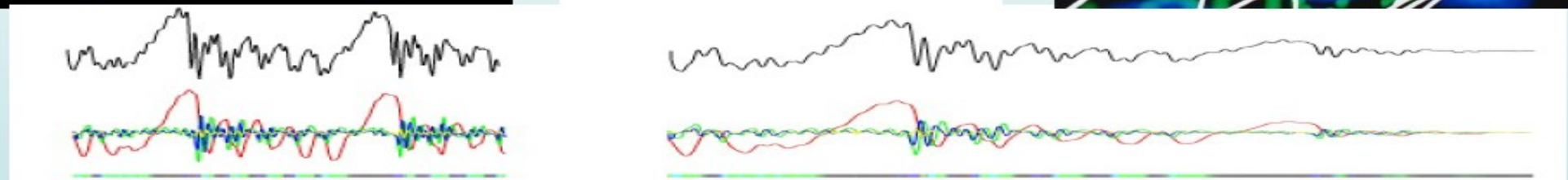
- If the *SEM_UNDO* flag is set in operation structure, the action will be undone when process terminates.

- You must deallocate the semaphore set when you finnished.

- Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.

Signals





(a) PCM audio data (top row) is pre-filtered in 4 frequency subbands (shown as red, green, blue and yellow signals in the middle row) and stored as 1D RGBA textures (bottom row). (b) Texture resampling can be used to compress or dilate the signal to simulate Doppler shift effects. Color modulation is used to alter the frequency content of the signal.

Signals

- Signals are mechanisms for communicating with and manipulating processes in Linux.

- Signals are asynchronous software interrupts.

- In Linux, each signal has it's specific number.

- Signal names and numbers are defined in "/usr/include/bits/signum.h"

- A program may receive signals from *OS Itself, Other processes or users.*

IPC – Signals

```
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/IPCSemaphore$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/IPCSemaphore$
```

Signals

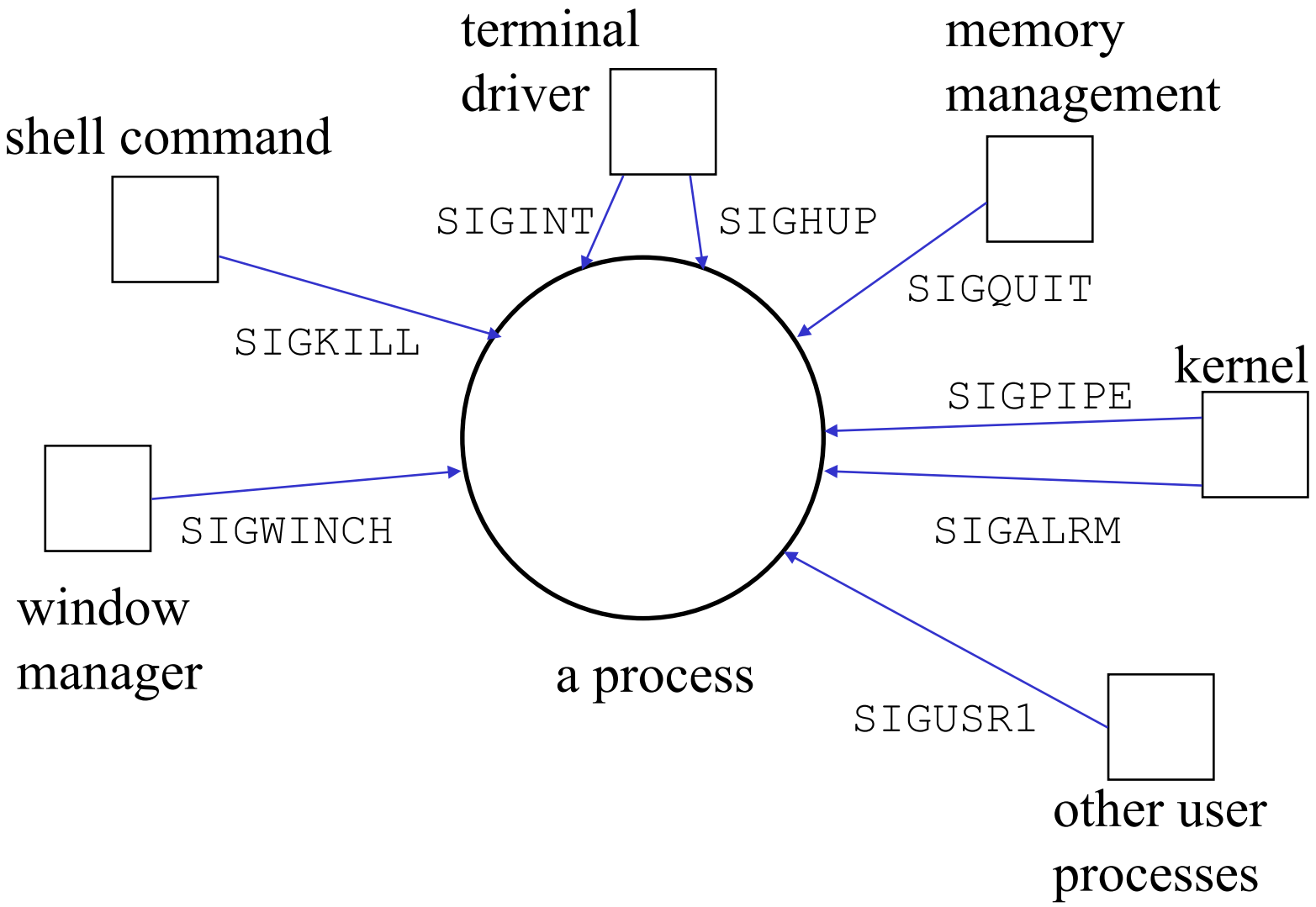
- A program may do one of lots of things when it receives a signal.

- For each signal there is a *Default Disposition* which determines the default behavior of a program when it receives this signal (If the program does not specify some specific action).

- There are some ways to handle the signals in Linux.

-The *SIGACTION* function can be used to change the action taken by a process on receipt of a specific signal.

IPC – Signals


terminal
driver
memory
management
shell command
SIGINT
SIGHUP
SIGQUIT
SIGKILL
kernel
SIGPIPE
SIGWINCH
SIGALRM
window
manager
a process
SIGUSR1
other user
processes

IPC – Signals

Generating a Signal

kill -9 <PID>

kill()

int kill(pid_t pid, int signo);

pause()

Signals

Signal number

(only trappable signals)

New signal disposition

```
int sigaction( int signum, const struct sigaction * act,
struct sigaction * oldact )
```

If is not NULL, the old signal disposition will go here

It's better to use signal NAMES instead of NUMBERs here. (Mapping is in /usr/include/asm/signal.h)

Signals

- The sigaction structure is something like this:

SIG_IGN, SIG_DFL or a handler function which takes an integer as the signal number

```
struct sigaction
{
        void (*sa_handler) (int);
        void (*sa_sigaction) (int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        void (* sa_restorer) (void);
}
```

If you want more information about the received signal, you can set this function instead of *sa_handler*.

Synchronization mechanism – Critical Section

Critical Section

- The ultimate cause of most bugs involving threads is that they are accessing the same data at the same time.

- The section of code which is responsible to access the shared data, is called *Critical Section* .

- A critical section is part of code that should be executed completely or not at all (a thread should not be interrupted when it is in this section)

- If you do not protect the *Critical Section*, your program might crash because of *Race Condition*.

Synchronization mechanism – Race Condition

Race Condition

- Race Condition is a condition in which threads are racing each other to change the same data structure.

- Because there is no way to know when the system scheduler will interrupt one thread and execute the other one, the buggy program may crash once and finish regularly next time.

- To eliminate race conditions, you need a way to make operations *atomic* (uninterruptible).

Synchronization mechanism – Threads

- Counting Semaphores
  - Permit a limited number of threads to execute a section of the code
- Binary Semaphores - Mutexes
  - Permit only one thread to execute a section of the code
- Condition Variables
  - Communicate information about the state of shared data

Synchronization mechanism – PosixSemaphore

Data type

Semaphore is a variable of type **sem_t**

Include **<semaphore.h>**

Atomic Operations

int sem_init(sem_t *sem, int pshared, unsigned value);

int sem_destroy(sem_t *sem);

int sem_post(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_wait(sem_t *sem);

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);

Initialize an semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

pshared:

0: only threads of the creating process can use the semaphore

Non-0: other processes can use the semaphore

value:

Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!

Synchronization mechanism – PosixSemaphore

- Sharing semaphores between threads within a process is easy, use **pshared==0**
- A non-zero **pshared** allows any process that can access the semaphore to use it Places the semaphore in the global (OS) environment Forking a process creates copies of any semaphore it has

Synchronization mechanism – PosixSemaphore

sem_init can fail

On failure

sem_init returns -1 and sets **errno**

errno	cause
EINVAL	**Value > sem_value_max**
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
  perror("Failed to initialize semaphore semA");
```

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_destroy(sem_t *sem);

Destroy an semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

Notes

Can destroy a **sem_t** only once

Destroying a destroyed semaphore gives undefined results

Destroying a semaphore on which a thread is blocked gives undefined results

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_post(sem_t *sem);

Unlock a semaphore - same as signal

Returns

0 on success

-1 on failure, sets **errno** (== **EINVAL** if semaphore doesn't exist)

Parameters

sem:

Target semaphore

sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented

sem == 0: one blocked thread will be allowed to run

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_wait(sem_t *sem);

Lock a semaphore

Blocks if semaphore value is zero

Returns

0 on success

-1 on failure, sets **errno** (== **EINTR** if interrupted by a signal)

Parameters

sem:

Target semaphore

sem > 0: thread acquires lock

sem == 0: thread blocks

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_trywait(sem_t *sem);

Test a semaphore's current condition

Does not block

Returns

0 on success

-1 on failure, sets **errno** (== **AGAIN** if semaphore already locked)

Parameters

sem:

Target semaphore

sem > 0: thread acquires lock

sem == 0: thread returns

Synchronization mechanism – Mutex

A typical sequence in the use of a mutex

1. Create and initialize **mutex**
2. Several threads attempt to lock **mutex**
3. Only one succeeds and now owns **mutex**
4. The owner performs some set of actions
5. The owner unlocks **mutex**
6. Another thread acquires **mutex** and repeats the process
7. Finally **mutex** is destroyed

Synchronization mechanism – Mutex

Creating a mutex

#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

Initialize a pthread mutex: the mutex is initially unlocked

Returns

0 on success

Error number on failure

EAGAIN: The system lacked the necessary resources; ENOMEM: Insufficient memory ;
EPERM: Caller does not have privileges; EBUSY: An attempt to re-initialise a mutex;
EINVAL: The value specified by attr is invalid

Parameters

mutex: Target mutex

attr:

NULL: the default mutex attributes are used

Non-NULL: initializes with specified attributes

Synchronization mechanism – Mutex

#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);

Destroy a pthread mutex

Returns

0 on success

Error number on failure

EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

Parameters

mutex: Target mutex

Synchronization mechanism – Mutex

Locking/unlocking a mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Returns

0 on success

Error number on failure

EBUSY: already locked; EINVAL: Not an initialised mutex; EDEADLK: The current thread already owns the mutex; EPERM: The current thread does not own the mutex

Synchronization mechanism – Conditional Variable

- Used to communicate information about the state of shared data
  - Execution of code depends on the state of
    - A data structure or
    - Another running thread
- Allows threads to synchronize based upon the actual value of data
- Without condition variables
  - Threads continually poll to check if the condition is met

Synchronization mechanism – Conditional Variable

- Signaling, not mutual exclusion
  - A mutex is needed to synchronize access to the shared data
- Each condition variable is associated with a single mutex
  - Wait atomically unlocks the mutex and blocks the thread
  - Signal awakens a blocked thread

Synchronization mechanism – Conditional Variable

Similar to pthread mutexes

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Synchronization mechanism – Conditional Variable

Waiting

Block on a condition variable.

Called with **mutex** locked by the calling thread

Atomically release **mutex** and cause the calling thread to block on the condition variable

On return, **mutex** is locked again

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

Synchronization mechanism – Conditional Variable

Signaling

int pthread_cond_signal(pthread_cond_t *cond);

unblocks at least one of the blocked threads

int pthread_cond_broadcast(pthread_cond_t *cond);

unblocks all of the blocked threads

Signals are not saved

Must have a thread waiting for the signal or it will be lost

Process Scheduling

Although Linux is a preemptively multitasked operating system, it also provides a system call that allows processes to explicitly yield execution and instruct the scheduler to select a new process for execution:

```
#include <sched.h>

int sched_yield (void);
```

A call to `sched_yield( )` results in suspension of the currently running process, after which the process scheduler selects a new process to run, in the same manner as if the kernel had itself preempted the currently running process in favor of executing a new process. Note that if no other runnable process exists, which is often the case, the yielding process will immediately resume execution. Because of this uncertainty, coupled with the general belief that there are generally better choices, use of this system call is not common.

Process Scheduling

Linux provides several system calls for retrieving and setting a process' nice value. The simplest is `nice( )`:

```
#include <unistd.h>

int nice (int inc);
```

A successful call to `nice( )` increments a process' nice value by `inc`, and returns the newly updated value. Only a process with the `CAP_SYS_NICE` capability (effectively, processes owned by root) may provide a negative value for `inc`, decreasing its nice value, and thereby increasing its priority. Consequently, nonroot processes may only lower their priorities (by increasing their nice values).

On error, `nice( )` returns `-1`. However, because `nice( )` returns the new nice value, `-1` is also a successful return value. To differentiate between success and failure, you can zero out `errno` before invocation, and subsequently check its value. For example:

Process Scheduling

A preferable solution is to use the `getpriority( )` and `setpriority( )` system calls, which allow more control, but are more complex in operation:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

These calls operate on the process, process group, or user, as specified by `which` and `who`. The value of `which` must be one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, in which case `who` specifies a process ID, process group ID, or user ID, respectively. If `who` is `0`, the call operates on the current process ID, process group ID, or user ID, respectively.

A call to `getpriority( )` returns the highest priority (lowest numerical nice value) of any of the specified processes. A call to `setpriority( )` sets the priority of all specified processes to `prio`. As with `nice( )`, only a process possessing `CAP_SYS_NICE` may raise a process' priority (lower the numerical nice value). Further, only a process with this capability can raise or lower the priority of a process not owned by the invoking user.

Like `nice( )`, `getpriority( )` returns `-1` on error. As this is also a successful return value, programmers should clear `errno` before invocation if they want to handle error conditions. Calls to `setpriority( )` have no such problem; `setpriority( )` always returns `0` on success, and `-1` on error.

Process Scheduling

Processes can manipulate the Linux scheduling policy via `sched_getscheduler( )` and `sched_setscheduler( )`:

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */


};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
                        int policy,
                        const struct sched_param *sp);
```

A successful call to `sched_getscheduler( )` returns the scheduling policy of the process represented by `pid`. If `pid` is `0`, the call returns the invoking process' scheduling policy. An integer defined in `<sched.h>` represents the scheduling policy: the first in, first out policy is `SCHED_FIFO`; the round-robin policy is `SCHED_RR`; and the normal policy is `SCHED_OTHER`. On error, the call returns `-1` (which is never a valid scheduling policy), and `errno` is set as appropriate.

Thread priority

pthread_attr_setschedparam()

Set a thread's scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(
            pthread_attr_t * attr,
            const struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see *pthread_attr_init()*.

param

A pointer to a `sched_param` structure that defines the thread's scheduling parameters.

Thread priority

pthread_attr_getschedparam()

Get thread scheduling parameters attribute

Synopsis:

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedparam(
                    const pthread_attr_t * attr,
                    struct sched_param * param );
```

Arguments:

attr

A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see *pthread_attr_init()*.

param

A pointer to a `sched_param` structure where the function can store the current scheduling parameters.

Processor Affinity

```
#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize,
                       const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize,
                       cpu_set_t *set);
```

Processor Affinity

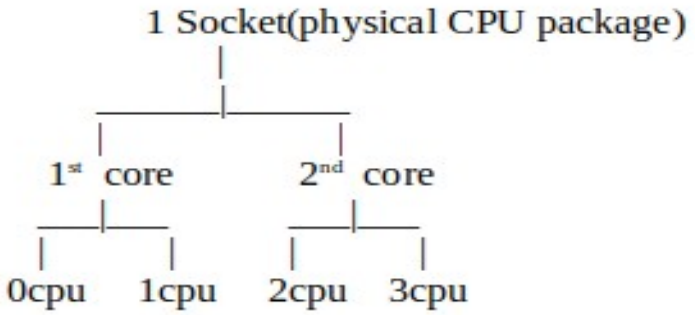
```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);          /* clear all CPUs */
CPU_SET (0, &set);        /* allow CPU #0 */
CPU_CLR (1, &set);        /* forbid CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_setaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

Processor Affinity

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_getaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

Linux Performance – CPU Isolation

- In Multi-processor architecture processors is directly get compared with processing power. Common notion is more processors means better performance but that is not always the case. In Most of Embedded chip-set multiple processing units are merged so one single chip can give you multiple processor functionality, processing units often referred as cores. To get an clear picture let's take an example. In general 4 core machine we can see below cpu arrangement.



- machine is having 2 cores and 4 logical processing unit which mean 4 parallel thread can run. When we say CPU Isolation then we are isolating one or more cpu.

Linux Performance – CPU Isolation

Scenario before CPU Isolation:

- When scheduler code get loaded on processor it put a spin lock and search for process which is ready to execute. Once it found it will load that process and remove lock form process table. Like wise on every processor scheduler code will get loaded

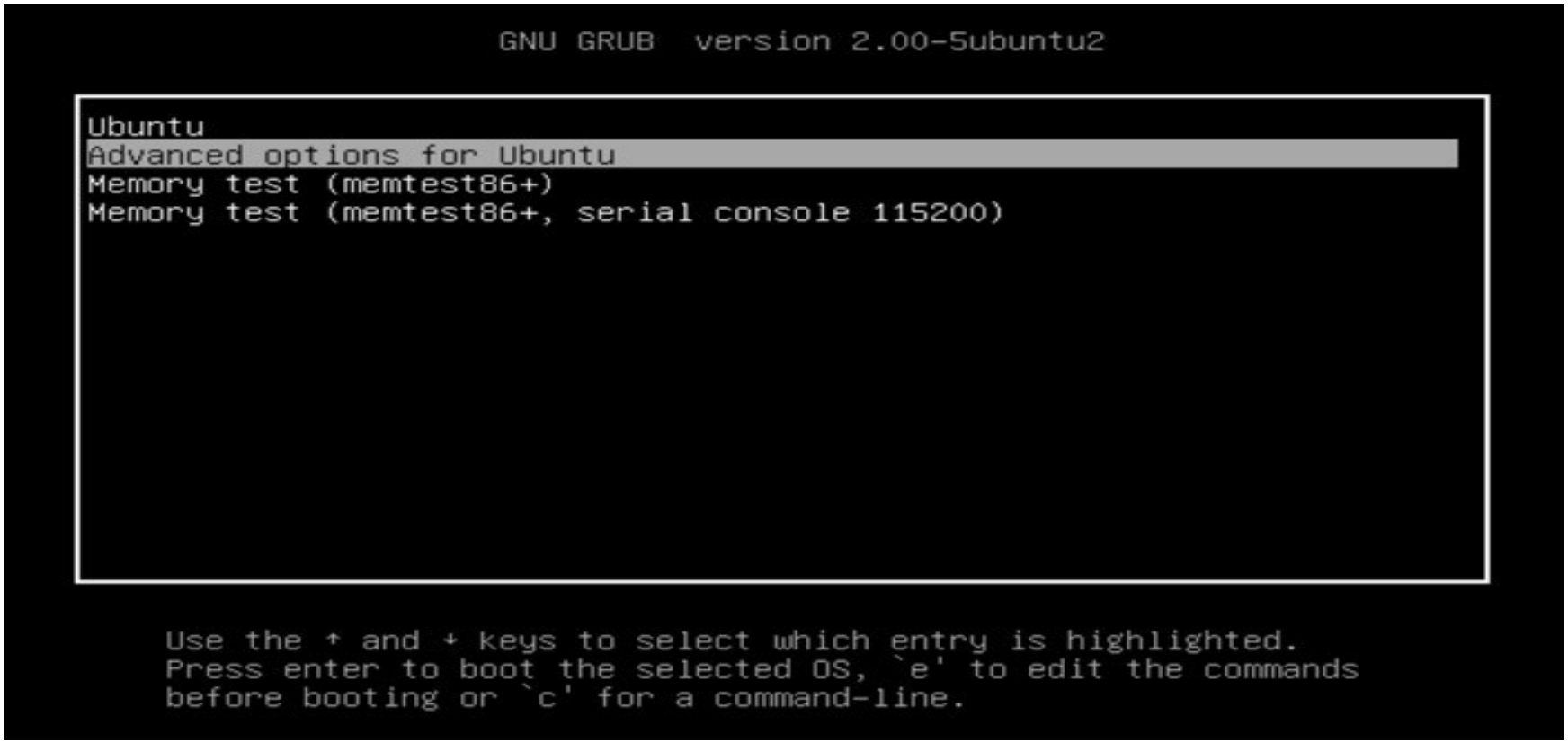
Scenario After CPU Isolation:

- If CPU isolation is activated for one of core then scheduler code will be never get loaded on isolated processor so no other process will get scheduled, until explicitly we tell to scheduler to load particular process on that processor no process will be loaded over it. Or in some architecture isolated CPU are removed/ Ignored in scheduler algorithms. it is basically Design dependent that how CPU is isolated from the process to be get loaded

Linux Performance – CPU Isolation

Steps to isolate CPU

- To isolate cpu temporarily or for single boot we can do it by below steps,while Linux booting by grub loader user will see grub menu as shown in below picture.



Linux Performance – CPU Isolation

Steps to isolate CPU

- To isolate cpu temporarily or for single boot we can do it by below steps,while Linux booting by grub loader user will see grub menu as shown in below picture.Here we can press ‘e’ to edit the kernel boot command. Once we press ‘e’ a new window will appear. Which is actual grub script to load application. There generally you will see line like below.

linux bootvmlinuz{kernel-version} root=UUID=.... quite splash isolcpus=2

above command loads kernel image there we are manually passing **“isolcpus=2”** parameter to isolate CPU number 2. after this change we can boot the kernel by pressing F10 or as mentioned in grub note. After this once system boots cpu2 will be isolated and no user/kernel process is loaded. Only few cpu bounded process will run over it.

Linux Performance – CPU Isolation

- To check CPU isolation is done or not. Fire below command

cat /sys/devices/system/cpu/isolated Isolated CPU number you will get as output. If No Cpu is isolated then you will get an empty Output

Linux Performance – CPU Affinity

- There are two types of CPU affinity:

- **soft affinity**: also called natural affinity, is the tendency of a scheduler to try to keep processes on the same CPU as long as possible.scheduler, which has poor CPU affinity. This behavior results in the ping-pong effect. The scheduler bounces processes between multiple processors each time they are scheduled and rescheduled.

- **Hard affinity:** on the other hand, is what a CPU affinity system call provides. It is a requirement, and processes must stick to a specified hard affinity. If a processor is bound to CPU zero, for example, then it can run only on CPU zero. It will not run on any processor Even if other CPUs are free and cpu0 is busy. It will wait until cpu0 gets free.

Linux Performance – CPU Affinity

Benefit:

- CPU affinity is optimizing cache performance.when processes bounce between processors they constantly cause cache invalidation, and the data they want is never in the cache when they need it. Thus, cache miss rates grow very large. CPU affinity protects against this and improves cache performance.Scheduling that process to execute on the same processor improves its performance by reducing performance-degrading events such as cache misses.
  - CPU isolation can help you to observe your code performance and process switch latency on different processor scenarios.
  - Once CPU is isolated, Isolated CPU is completely free to execute specific Pinned process. When we will pin a specific process to isolated CPU then cpu will execute solely pinned process only. Doing this will reduce the latency for pinned process and throughput will increase as no other process is there to disturb or take cpu away. For critical and important process we can do this way which will help to reduce latency and increase through put.
  - CPU affinity is optimizing cache performance

Thank you