ASI

Primera Parte: S.O.

3º Ingeniería de Telecomunicaciones — UPV/EHU
"Under-promise and over-deliver."
Javier de Martín – 2016/17

Gestión de Procesos

Un **proceso** es, básicamente, un programa en ejecución. Es definido como una entidad que representa la unidad básica de información implementada por el sistema.

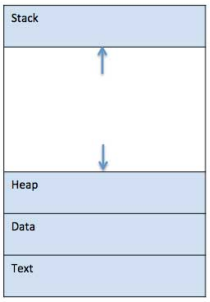


Figura 1: Modelo simplificado de un proceso en la memoria principal

Cuando un programa se carga en memoria y se convierte en proceso, puede ser dividido en 4 secciones:

- **Stack**: Contiene datos temporales, como parámetros de funciones, variables locales y direcciones de retorno.
- **Heap**: Memoria dinámicamente alojada para un proceso en su runtime.
- **Text**: Incluye la actividad actual representada por el valor del program counter y el contenido de los registros del procesador.
- **Data**: Contiene las variables globales y estáticas.

Todos los procesos tienen tres estados principales:

- **Bloqueo**: Un proceso espera un recurso externo o petición necesaria para su ejecución. No sigue la ejecución hasta que ocurre un evento externo.
- **Listo**: Estado en el que están los procesos que están preparados para ser ejecutados.
- **Ejecución**: Estado en el que están los procesos que se están ejecutando usando la CPU.

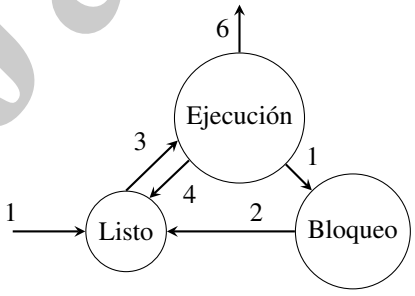


Figura 2: Modelo Básico de 3 estados

1. **Start**: Estado inicial cuando un proceso es creado/iniciado por primera vez.
2. **Listo**: El proceso espera ser asignado a un procesador por el SO y que puedan ejecutarse. Los procesos llegan a este estado después del estado de start o mientras lo están corriendo pero son interrumpidos por el scheduler para asignar la CPU a otro proceso.
3. **Ejecución**: Una vez que el proceso ha sido asignado a un procesador por el scheduler del SO, el estado cambia a running y el procesador ejecuta sus instrucciones.
4. **Bloqueo**: Un proceso se mueve al estado de bloqueo si tiene que esperar por un recurso, como la input del usuario, o a que un archivo esté disponible.
5. **Terminated**: Una vez que el proceso finaliza la ejecución o es terminado por el SO, se mueve a este estado donde espera a ser eliminado de la memoria principal.

Para que un **proceso** pueda ser **ejecutado** necesita varios requisitos:

- Una entrada en una tabla de procesos para que el planificador pueda gestionar su memoria y ejecución.
- Una zona de memoria reservada para este proceso.
- Una comunicación entre otros procesos para peticiones de datos.

En la tabla de procesos hay un array de estructuras con una estructura para cada proceso. Cada estructura de la tabla de procesos contiene cierta información y se llama **PCB** (*Process Control Block*). Es una estructura de datos mantenida por el SO para cada proceso. Es identificada por un entero que representa el ID del proceso (PID).

- **Estado del Proceso**: Estado actual del proceso, independientemente del estado.
- **Privilegios del Proceso**: Requerido para permitir o no acceso a los recursos del sistema.
- **Process ID**: Identificación única para cada proceso en el SO.
- **Puntero**: Puntero al proceso padre.
- **Contador de Programa**: Puntero a la dirección de la siguiente instrucción a ser ejecutada por el proceso.
- **Registros de CPU**: Donde el proceso necesita ser almacenado para ejecución en el estado 'listo'.
- **CPU Scheduling Info**: Prioridad de proceso y otra información que se necesite para schedule el proceso.
- **Información de gestión de memoria**: Page table, límites de memoria...
- **Accounting Information**: Incluye la cantidad de CPU utilizada para la ejecución del proceso, límite de tiempos...
- **IO Status Info**: Lista de dispositivos de IO allocated para el proceso.

Cuando se produce una **interrupción** se producen ciertas acciones:

- Se guardan los registros (PC).
- Se carga el PC de atención a interrupción de ese tipo.
- Se ejecuta en modo kernel y enmascaran las interrupciones.
- Se eejcuta el planificador.
- Se cargan los registros del proceso a ejecutar.
- Se pasa a modo usuario y se da el control al proceso a ejecutar.

Comunicación entre Procesos

Hay recursos a los que sólo puede acceder un único proceso a la vez, hay que evitar la **condición de carreras**.

La condición de carrera ocurre cuando dos o más procesos acceden a un recurso compartido sin control, de manera que el resultado del acceso depende del orden de llegada. Esa zona compartida se llama **sección crítica**. Para solucionar este problema el SO ha de prohibir el acceso a más de un proceso a una zona compartida al mismo tiempo.

Condiciones para una buena solución:

1. Dos procesos no podrán estar a la vez dentro de sus secciones críticas.
2. No se deben asumir suposiciones sobre velocidades relativas de los procesos.
3. Ningún proceso fuera de la sección crítica puede bloquear otros procesos.
4. Ningún proceso esperará indefinidamente para entrar en su sección crítica.

Exclusión Mutua con Espera Activa

Mientras un proceso está ocupado actualizando memoria compartida en su región crítica ningún otro entrará en la suya correspondiente.

Hay 5 formas:

1. **Inhabilitación de interrupciones**: Inhabilitar las interrupciones después de entrar en la región crítica y habilitarlas después de abandonarla para evitar que el planificador le quite el control. Esta solución es inviable debido a que no se le puede entregar el control de interrupciones a un proceso, es tarea del SO.
2. **Variables candado** (lock): Es una solución por software. Se utiliza una variable compartida `lock` (inicialmente `0`). Cuando un proceso quiere entrar en su sección crítica:
   - `lock = 0` → El proceso pone `lock = 1` y entra en la región crítica.
   - `lock = 1` → El proceso espera a que `lock = 0`.

   No cumple la condición 4, puede que tenga que esperar indefinidamente.
3. **Alternancia Estricta**: Solución por software. Cuando un proceso va a entrar en la sección crítica pasa el control a otro proceso para que bloquee los procesos que quieran entrar con una variable. Evita condiciones de carrera pero no cumple la condición 3.
4. **Solución Peterson**: Solución por software. Se definen 2 funciones de entrada y salida de la región crítica. Permite a dos o más procesos utilizar un recurso de único uso sin conflicto utilizando únicamente memoria compartida para comunicarse.
5. **Instrucción TSL** (*TEST and SET LOCK*: Solución por software con ayuda por hardware. Se utilizan 4 instrucciones de nivel ensamblador. E primer lugar, se copia el contenido de la variable en un registro y se pone la variable a 1. Se comprueba si el valor guardado es 1. Si es 0, se entra en la sección crítica. Si es 1, realiza de nuevo la comprobación. Debido a que se usan los registros, esta versión cumple todos los requisitos ya que si se diese el caso en el que en mitad de la ejecución de la reserva se produjese una interrupción dichos registros se guardarían.

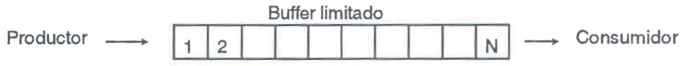
Dormir y Despertar

Peterson y TSL son soluciones buenas pero tiene como defectos la espera activa y otros más.

Primitivas de comunicación: Operaciones para conseguir la condición de exlusión mutua. Bloquean el proceso cuando no se les permite entrar en su región crítica sin consumir tiempo de CPU.
Para llevar a cabo este método son necesarias 2 primitivas de comunicación:

- **Sleep**: Llamada que suspende el proceso que la realiza hasta que otro proceso la despierte.
- **Wake-up**: Llamada que tiene por parámetro el proceso a despertar.

Surge el **problema productor-consumidor** (*ring buffer, bounded buffer*)



Intercambio de información entre processo por medio de un mecanismo de comunicación (buffer limitado de datos). Los productores se bloquean cuando el mecanismo (buffer) está lleno. Los consumidores se bloquean cuando el buffer está vacío.

Se da una situación de condición de carrera, el acceso a los elementos del buffer no está restringido → para solucionarlo añadir un bit de espera (wakeup).

Semáforos

Resuelven el problema de pérdidas de llamadas de Wake Up. Se utiliza una variable entera y definen dos operaciones:

- **DOWN**: Comprueba si el valor del semáforo `>0`. Si es así decrementa una unidad su contenido. Si es no (es cero) el proceso se duerme y queda bloqueado.
- **UP**: Comprueba si el valor del semáforo es `>0`. Si es así incrementa una unidad su contenido. Si es no y no hay ningún proceso dormido pone a `1` el semáforo. Si es no y hay algún proceso dormido aleatoriamente despierta a alguno y el semáforo permanece a `0`.

El SO inhabilita las interrupciones durante las operaciones a un semáforo. Para que este sistema funcione, las llamadas `UP` y `DOWN` deben de ser atómicas, el SO no puede quitar el control al proceso hasta que se ejecute.

Este método consigue evitar la condición de carrera y todas sus condiciones de implementación. El problema es que las llamadas `UP` y `DOWN` las hace el programa cliente. Si éste se equivoca de orden al realizar las llamadas puede llegar a provocar una excepción de bloqueo a todos los procesos que quieren acceder a la zona de acceso crítico. Para mejorarlo se definió el método de monitores.

Contadores de Eventos

Soluciona el problema de que el productor no puede escribir en el buffer ya que está lleno y se tiene que dormir. Se necesita saber el tamaño del almacén. No necesita exclusión mutua. utiliza un tipo especial de variable llamada contador de eventos y 3 operaciones:

- `READ (E)`: Devuelve el contador de eventos `E`.
- `ADVANCE (E)`: Incrementa el valor de `E` en `1` (atómicamente).
- `AWAIT (E,v)`: Espera hasta que $E \geq v$.

Los contadores de eventos **siempre crecen**, nunca decrecen y empiezan desde `0`.

Cuando el **productor** desea colocar un elemento en el buffer verifica si hay espacio por medio del `WAIT`. El **consumidor** espera a que el número de elementos que el productor ha colocado se hayan retirado.

Monitores

Permiten el bloqueo de procesos dentro del monitor utilizando operaciones atómicas como `WAIT` y `SIGNAL` y una variable de control. Sus **propiedades**:

1. En un monitor sólo un proceso puede estar activo en cualquier instante → exclusión mutua.
2. Permite el bloqueo de procesos dentro del monitor (porque no puedan proseguir → espera activa) a partir de la utilización de variables de condiciones y de las operaciones `WAIT` y `SIGNAL`:
   - `WAIT`: Sobre una variable de condición → el proceso se bloquea permitiendo a otro proceso entrar en el monitor. Ya no se necesitan llamadas a `UP` y `DOWN`.
   - `SIGNAL`: Sobre la variable de condición que ha dormido el proceso, despierta al proceso.

Forma de evitar que dos procesos permanezcan activos dentro del monitor después de una operación `SIGNAL`:

- **Hoare**: Continúa el proceso recién despertado, suspendiendo el otro.
- **Hansen**: El proceso que realiza `SIGNAL` deberá abandonar el monitor, `SIGNAL` aparecerá únicamente al final de una función monitor.

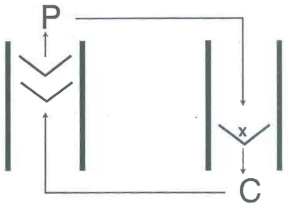
Transferencia de Mensajes

Existe un lugar donde almacenar los mensajes hasta que el consumidor los recoja. Hay dos primitivas de intercomunicación entre procesos:

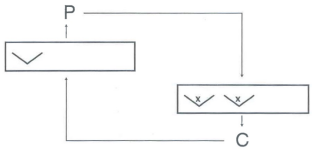
- `SEND (dest, msg)`: Envía un mensaje a un destino dado.
- `RECEIVE(origin, msg)`: Recibe un mensaje de un origen dado, si no hay ningún mensaje disponible el receptor se bloquea hasta que llegue uno.

Soluciones al **problema productor-consumidor**:

- **Pipes**: todos los mensajes son del mismo tamaño. El SO almacena mensajes enviados y no recibidos. Se puede bloquear si el consumidor no encuentra mensajes llenos y el productor no encuentra mensajes vacíos.

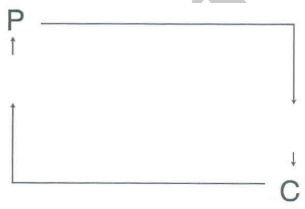


- **Buzón**: Estructura de datos. Zona de memoria donde cabe un cierto número de mensajes. Con buzones, los parámetros de dirección empleados en las llamadas de `SEND` y `RECEIVE` son buzones en lugar de procesos.



El buzón de destino coge los mensajes que han sido enviados al proceso de destino pero que aún no han sido recibidos. Cuando un proceso intenta hacer un envío a un buzón que está lleno, es suspendido hasta que se extrae un mensaje.

- **Rendez vous**: Elimina cualquier forma de almacenamiento. Si se hace `SEND` antes que `RECEIVE` el proceso `SEND` se bloquea. Cuando se produce `RECEIVE` se copia el mensaje directamente del proceso que lo envía al que lo recibe.



Planificación

El **planificador** es la perte del SO que decide el orden de ejecución de los distintos procesos activos del sistema. Existen varios algoritmos de planificación y tienen que tener las siguientes **características**:

1. **Justo**: Asegurarse de que todos los procesos tengan un reparto justo de la CPU.
2. **Eficiencia**: Mantener la CPU activa el 100 % del tiempo.
3. **Tiempo de respuesta**: Minimizar el tiempo de respuesta para los procesos activos.
4. **Rendimiento**: Minimizar el tiempo que los usuarios deben esperar para la salida.
5. **Throughput**: Maximizar el número de trabajos procesados.

Para poder realizar estas funciones, es necesario un reloj interno que produzca interrupciones cada cierto tiempo para que el SO tome el control y decida cuál es el siguiente proceso a ejecutar.

Round Robin

A cada proceso la CPU le asigna un tiempo de ejecución. Es cíclico, la asignación del procesador es rotatoria. Con este algoritmo todos los procesos tienen la misma prioridad. A cada proceso se le asigna un tiempo determinado de ejecución (*quantum*).

En el caso de que finalize el quantum y el proceso no haya finalizado la ejecución se le quitará el control al proceso y se le dará el siguiente en la cola.

Los *quantum* pueden tener distintas duraciones:

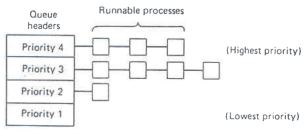
- Corto (20ms): Muchos cambios de contexto, da lugar a poco rendimiento.
- Largo (500ms): Respuesta pobre en entornos interactivos, puede tardar mucho en ejecutar un proceso si existen muchos en el sistema.

Prioridades

A cada proceso se le asigna una prioridad. El proceso con mayor prioridad es el que se ejecuta. Asignación de prioridades:

- **Estática**: No cambia en ejecución. Fácil de implementar. Es poco sensible a los cambios de entorno.
- **Dinámica**: Usa el algoritmo $1/f$, si un proceso tiene un *quantum* de 100ms y un proceso usa 2ms se le asigna una prioridad de 50. f es la fracción del último *quantum* utilizado en el proceso. Es bastante difícil de implementar.

Hay veces que interesa agrupar los procesos por clases de prioridad, utilizando planificación de prioridades entre las clases y planificación round robin entre los procesos de cada clase.



Si hay varios procesos con el mismo nivel de prioridad se suele usar un mecanismo adicional para decidir.

Colas Múltiples

Variante de la planificación en clases de prioridad de forma dinámica. Se desarrolla en los primeros sistemas CTSS en los que sólo se podía mantener un proceso en memoria. Los procesos de mayor prioridad son asignados con 1 quantum de ejecución, las demás clases inferiores tendrán 2^{n-1} quantums.

Con este mecanismo se consigue priorizar los procesos de ejecución rápida. Si un proceso consume la totalidad de su tiempo de ejecución se le baja una clase.

Hay 4 clases de prioridad:

- Terminal
- I/O
- Quantum Corto
- Quantum Largo

Algoritmo de planificación:

1. Proceso bloqueado en espera entrada terminal despertado → prioridad más alta.
2. Proceso bloqueado en espera I/O despertado → prioridad I/O
3. Proceso permanece ejecutable cuando se termina su quantum → quantum corto
4. Proceso emplea varias veces su quantum sin bloquearse por el terminal o por I/O → quantum largo

Primero Trabajo Más Corto

Algoritmo indicado para sistemas batch en los que son conocidos los tiempos de ejecución. Consiste en ejecutar los procesos de menor tiempo de ejecución en primer lugar. De esta manera, se consiguen menores tiempos medios de respuesta. Para saber el tiempo aproximado de ejecución se utilizan técnicas de *aging*. Se estima el siguiente valor tomando una medida ponderada de los valores medidos y uno estimado previamente. Tiene una dificultad, conocer a priori la duración del trabajo. Este sistema de planificación es óptimo únicamente cuando todos los trabajos están disponibles simultáneamente.

Predeterminada

Consiste en adquirir un compromiso con el usuario y cumplirlo. Se realiza una comparación entre el uso de la CPU y la CPU asignada al proceso. Después, el algoritmo se encarga de ejecutar los procesos que tengan una relación más baja hasta que posean una relación por encima de su competidor más próximo.

Tiene las siguientes dificultades:

1. El sistema ha de ejecutar el trabajo sin degradar el servicio a otros usuarios.
2. El sistema debe planificar la asignación de recursos completamente hasta el *deadline*. Pueden llegar trabajos que introduzcan demandas que el sistema no puede predecir.
3. Varios trabajos *deadline* a la vez → introducir sofisticados métodos de overhead.

Planificación a Dos Niveles

Los tiempos de intercambio entre memoria principal y disco son de uno o dos órdenes de magnitud superiores a los de intercambio entre los procesos que se encuentran en memoria principal.

Hay dos formas, una de ellas mantiene los procesos en la memoria principal mientras se ejecuta y otra es que los procesos se mantienen en el disco hasta que puedan ser ejecutados.

- Planificador de bajo nivel → Procesos en MP
- Planificador de alto nivel → Desplazamiento entre MP y disco

Criterios de decisión de un planificador de alto nivel:

1. ¿Cuánto ha pasado desde que el proceso fue intercambiado?
2. ¿Cuánto tiempo de CPU ha consumido el proceso recientemente?
3. ¿Cómo de grande es el proceso?
4. ¿Cómo de alta es la prioridad del proceso?

Evaluación

Métodos analíticos: Consisten en aplicar el algoritmo seleccionado a una determinada carga del sistema y producir una fórmula o número que evalúa la eficacia del algoritmo para esa carga.

- Modelo determinista: Se toma una carga particular establecida con anterioridad y se determina la eficacia de cada algoritmo para esa carga. Produce números exactor y permite la comparación de algoritmos. Requiere números exactos en sus datos de entrada y los resultados obtenidos se pueden aplicar únicamente a esos datos.
- Modelo de colas: Según este modelo el ordenador se describe como un servidor, donde se determinan las distribuciones de probabilidad tanto del tiempo de llegada entre procesos como el tiempo de empleo de CPU. A partir de estas dos distribuciones se pueden calcular los valores medios de *throughput*, utilización, tiempo de espera... Las distribuciones de llegada y servicio se definen por relaciones matemáticas que muchas veces suponen hacer simplificaciones que restan exactitud al método, por ser éstas sólo una aproximación a los sistemas reales.

Simulación: Permite conseguir una evaluación más precisa de los algoritmos de evaluación (siempre refiriéndose a los niveles de confianza). Supone realizar una serie de tareas.

Implementación: Única forma exacta de evaluar un algoritmo de planificación. Consiste en introducir el algoritmo en el SO para ver cómo responde bajo condiciones operativas reales.

Tratamiento Procesos MINIX

Comprobar si entra, en los apuntes de Carmen no está

Hilos

Un **hilo** es una secuencia de tareas encadenada muy pequeña que puede ser ejecutada por un SO. Un hilo se **diferencia** de un proceso en que los últimos son, generalmente, independientes y actúan sólo a través de mecanismos de comunicación dados por el sistema.

Gestión de Procesos

Introducción

Se define un **proceso**

Gestión de E/S

Gestión de Memoria

Gestión de Archivos

ASI

Prácticas de Laboratorio

3º Ingeniería de Telecomunicaciones — UPV/EHU

"Under-promise and over-deliver."

Javier de Martín – 2016/17

Práctica 2

IPC permite comunicar datos entre procesos y sincronizarlos. Hay diferentes mecanismos para comunicar datos y sincronizar los procesos: Archivos, Pipes y FIFOs, colas de mensajes, semáforos...

Un **pipe** es un mecanismo de comunicación y sincronización con
Trabajaremos con pipes system5, no POSIX
pipes son fifo sin nombre,

Nunca usar printf dentro de signal handlers
http://stackoverflow.com/questions/25663492/user-defined-signal-1

Pipes sin nombre, creación de un pipe:

```
int pipe(int fildes[2]);
```

- `fildes[0]`: Descriptor para lectura del PIPE
- `fildes[1]`: Descriptor para escritura del PIPE

Hay que llamar a `pipe()` antes de crear los procesos que requieren compartir el PIPE para que puedan compartir los descriptores de lectura y escritura.

Escribir en el PIPE:

```
ssize_t write(int fd, const void * buf, size_t n);
```

Leer del PIPE sin nombre

```
ssize_t read(int fd, const void * buf, size_t n);
```

Cualquier proceso que comparte el PIPE puede leerlo. Incluso el que escribe, de forma que si un proceso lee un dato, el resto de procesos no van a volver a leer ese dato.

PIPES con nombre (FIFOs) permiten comunicar procesos que no guardan ninguna relación de parentesco. Es un fichero *especial* que ocupa una entrada en un directorio y se accede a él a través de una ruta. La FIFO se puede crear utilizando una de la siguientes funciones:

```
int mknod (char * path, __mode_t mode, __dev_t dev);
int mkfifo (char * path, __mode_t mode);
```

Para crear FIFOs son equivalentes pero `mknod` permite crear además ficheros normales pero para ello requiere de privilegios root. La FIFO se puede borrar con la función `unlink()`.

Una vez creada una FIFO se puede utilizar como un fichero, se abre con `open()`, se escribe con `write()` y se cierra con `close()`. En este caso, `read()` se queda a la espera de que se haga `write()` en el otro lado de la pipe. Se puede trabajar también abriendo con `fopen()`[1], se lee con `fread()` y se cierra con `fclose()`. En este caso, el primer `fopen()` se queda a la espera de que se haga el `fopen()` en el otro lado.

[1] No se recomienda trabajar con estas funciones, son de nivel superior y realizan otras tareas

En un proceso se puede utilizar un juego de funciones y en el otro lado de la FIFO uno diferente.

Los **pipes** tienen unas **limitaciones**:

- Comunicación unidireccional.
- Prioridades, al tratarse de una FIFO hay que leer los datos en el orden en el que se han escrito. No hay 'categorías'de datos.
- Sincronización, el bloquear la lectura hasta que se produzca la escritura puede ser un inconveniente. Si se utiliza la opción de no bloqueo `O_NONBLOCK` se peude desbloquear pero la gestión del buffer no es sencilla.
- Falta de control de la estructura FIFO: No hay forma de conocer el número de bytes que hay en la FIFO
- TERMINAR ESTA DIAPOSITIVA

Colas de Mensajes

Otro sistema de información entre procesos. Se trata de listas enlazadas dentro del espacio de direccionamiento del núcleo. Los mensajes se envían a la cola y se pueden recuperar de formas diferentes. Cada cola se identifica por un único IPC. A diferencia de pipes y FIFOs: permiten el intercambio bidireccional, permiten establecer prioridades en los mensajes y 'tipos'de mensajes.

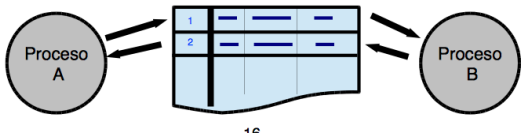


Figura 3: Cola de mensajes

Los mensajes que se guardan en la cola de mensajes son estructuras definidas por el usuario. El único requisito es que el primer campo sea un dato de tipo `long` cuyo valor será el tipo de mensaje.

```
typedef struct Mi_Mensaje {
    long idMensaje;
    char mensaje[100];
}
```

MIRA ESTO JAVI: http://stackoverflow.com/questions/5677163/message-queue-dynamic-message-size-c
UNIX SV ofrece una interface para la gestión de las colas de mensajes incluidas en `<sys/msg.h>`:

```
int msgget(key_t key, int msgflg);
```

Crea u obtiene una cola de mensajes existente a partir de la clave `key` y con los flags `msgflag`. Para crearla es necesario incluir el flag `IPC_CREAT`. Retorna un id de la cola que se va a utilizar en el resto de funciones.

```
int msgctl(int msqid, int cmd, struct msqid_ds * buf);
```

Realiza operaciones indicadas mediante `cmd` sobre la cola `msqid`. Si `cmd` es `IPC_RMID` destruye la cola de mensajes.

```
int msgsnd(int msqid, const void * msgp, size_t msgsz, i
```

Escribe un mensaje en la cola.

```
ssize_t msgrcv(int msqid, void * msgp, size_t msgsz,
    long msgtyp, int msgflg);
```

Lee un mensaje en la cola. En `msgtyp` se puede especificar un tipo de mensaje. La función de lectura puede esperar o no si hay mensajes en la cola en función de los `msgflag`.

Las colas de mensajes se pueden consultar con el comando `ipcs`.

La clave única que se necesita para crear la cola de mensajes se hace utilizando `ftok( )`