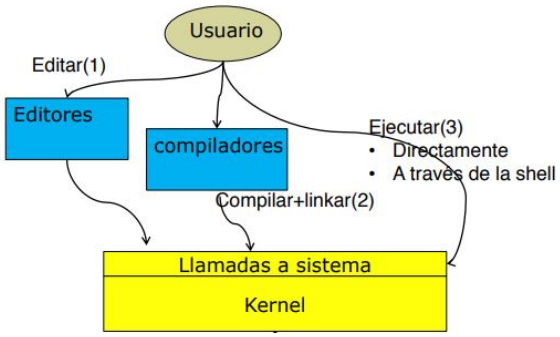
SO → software que controla los recursos disponibles del sistema hardware que queremos utilizar y que actúa de intermediario entre las aplicaciones y el hardware.

- Internamente: estructura datos para gestión HW y algoritmos de uso de HW.
- Externamente: conjunto funciones de funcionalidades o servicios de gestión de recursos.

> Componentes:



> Entorno de trabajo → shell o intérprete de comandos

- SHELL → bucle infinto que lee prompt y ejecuta
  - Prompt → símbolo sistema (dónde se escribe instrucción)

> SO debe ser:

- Usable: claro y útil para el usuario
- Seguro: el sistema no puede caer si tenemos algún fallo (pondrá un mensaje de error)
- Eficiente: cada proceso quiere tener una sensación exclusiva, es decir, que la CPU solo trabaja para él.

> Acceso a kernel → HW sabe en que modo estamos (flag de 1 bit)

- USER MODE: Por ejemplo, cuando hacemos llamamadas al sistema
- KERNEL MODE: Privilegiado. Por ejemplo, cuando ejecutamos llamadas al sistema (síncronas), cuando tenemos excepciones (asíncronas), interrupciones (síncronas), etc.

> Kernel → código guiado por eventos

- Ejecutar codigo kernel:
  - Llamada sist.
  - Excepciones (aplicaciones)
  - Interrupciones (disp. Externos)

!! SO Configura periódicamente interrupción de reloj para no perder control (y que user se quede todos los recursos).

> Llamadas a sistema (Syscalls) → funciones del kernel para acceder a sus servicios

- Requerimientos (programador).
  - Llamadas a función → usuario aislado de arquitectura.
  - No se puede modificar contexto de función.
- Requerimientos (kernel)

    - Ejecución en modo priviliegiado (seguridad).
    - Paso parametros y retorno de resultados entre modos diferentes (spoprte HW).
    - Portabilidad → direcciones de llamadas de sistema variables (soporte entre versiones de mismo kernel).
- Librerias de sistema con soporte HW → cumplen los requerimientos
    - Traducen función a petición de servicio explícito al sistema.
    - Pasa parámetros / recoge resultados de kernel
    - Invoca kernel (TRAP)
    - Homogeneiza resultados (llamadas a sistema en linux = -1 si hay error)
    - Portabilidad → identificador para todas las llamadas a sistema que se usa para indexar una tabla de llamadas a sistema (constante entre versiones)
        - Implica que no importa la dirección, ya que mantienen identificador y tabla.
    - Depende de la arquitectura → binario con lenguaje maquina.
    - Librerías → se ejecuta en modo usuario.

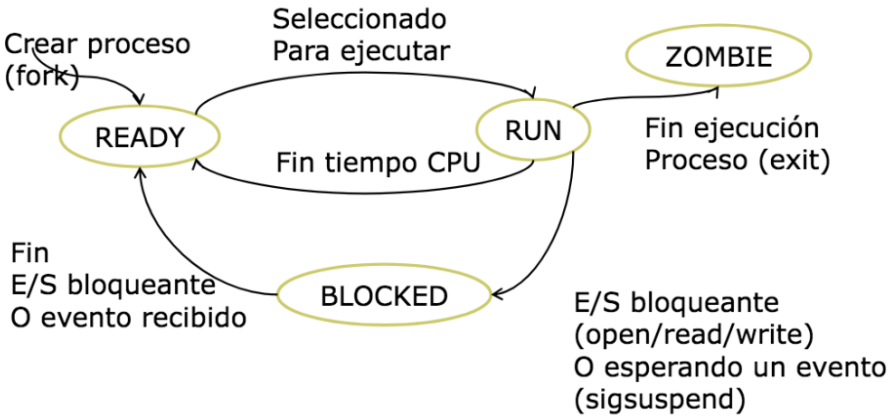
Procesos → programa en ejecución → recursos asignados por SO

> PCB (Process Control Block) → estructura de datos para gestión de un proceso. Formado por:

- Espacio de direcciones: código, datos, pila…
- Contexto de ejecución:
    - SW: PID, información para la planificación, información sobre el uso de dispositivos, estadísticas, ppid…
    - HW: tabla de páginas, PC…

> Uso de CPU

- Concurrencia → capacidad de ejecutar varios procesos de forma

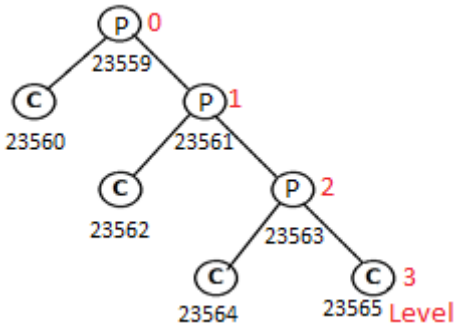


  simultánea.
    - Paralelismo → varios procesos concurrentes se ejecutan de forma simultánea.
    - Quantum → unidad de tiempo de ejecución del proceso en CPU.
- Secuenciales → procesos se ejecutan uno después del otro.
    - Sincronizaciones → con waitpid o signals.

> Estados de un proceso
- En un sistema multiproceso (múltiples procesos activos)

> Propiedades PCB (Linux)
- Identidad: PID, credenciales (USERID, GROUPID)
- Entorno: parámetros (argv, HOME, PATH,...)
- Contexto: estado, recursos, ... (durante ejecución

> Creación de procesos → procesos tienen un padre
- int fork()
  - Padre recibe pid_hijo
  - Hijo recibe 0
- PC hijo = PC padre → hijo comparte código, pero ejecuta después de el fork() que lo ha creado.
- SO decide los recursos que comparte, su planificación y su espacio de direcciones.
- Ejecución concurrente padre-hijo.
- Hereda:
  - Código, datos, pila
  - Programación de signals
  - Máscara de signals
  - Dispositivos virtuales (?)
  - userID y groupID
  - Variables de entorno
- No hereda:
  - PID, PPID
  - Contadores internos
  - Alarmas y signals pendientes



> Fin de ejecución de un proceso
- Voluntariamente (void exit(int)) → proceso libera recursos y estructura kernel.
- Involuntariamente (signals)
- Sincronización padre-hijo
  - int waitpìd (pid_t pid, int *status, int options) → padre se bloquea hasta fin del hijo.
  - pid_t pid =
    - -1 → padre espera cualquier hijo.
    - pid_hijo → padre espera a hijo con pid_hijo.
  - int *status = exit_code [exit(int)]
    - SO almacena exit_code hasta consulta padre.
    - Mientras → Hijo: PCB no se libera y estado ZOMBIE.
      - Si proceso muere sin liberar PCB de sus hijos → init (proceso padre de sistema) los libera.
    - Tratamiento exit_code
      - WIFEXITED(status) → muerte por exit
      - WTERMSIG(status) → muerte por signal
  - int options = 0
  - return pid_h esperado o -1 si error

> Mutación de procesos → cambio imagen del proceso (ejecución de otro código)

- int execlp (const char *file, const char *arg, ...)
- Cambia → Espacio de direcciones y ejecutable (PC a 1ª instrucción).
- Mantiene → Identidad del proceso (Contadores internos, signals pendientes, máscara de signals bloqueados ...)
  - Tabla programación signals definida por defecto.

> Resumen Syscalls de procesos

- fork → crea un nuevo proceso (si es el hijo, devuelve 0, si es el padre, devuelve el PID del hijo)
- exec → muta a otro programa (cambia el espacio de direcciones)
- exit → termina el proceso
- wait / waitpid → espera a que muera un proceso hijo (RUN → BLOCKED / WAITING)
- getpid → te da el PID
- getppid → te da el PPID

> Comunicación entre procesos

- Beneficios
  - Compartir información
  - Acelerar la computación que realizan
  - Modularidad
- 2 modos principales
  - Shared memory → procesos utilizan variables que pueden leer/escribir.
  - Message passing → procesos utilizan funciones para enviar/recibir datos.

Signals → notificaciones (enviadas por kernel o otro proceso de mismo usuario) que informan a un proceso de que ha sucedido un evento.

> Usos principales

- Sincronización procesos
- Control de tiempos → alarmas

> Evento → signal asociado. Excepto SIGUSR1 y SIGUSR2.

> Signals principales

SIGNAL	TRATAMIENTO (por defecto)	DESCRIPCIÓN
SIGCHLD	Ignorar	El proceso hijo ha muerto o ha sido parado
SIGCONT	-	Continua si estaba parado
SIGSTOP	Stop	Para el proceso
SIGSEGV	Core	Referencia inválida a memoria (Segmentation Fault)
SIGINT	Terminar	Ctrl + C
SIGALRM	Terminar	La alarma ha sonado
SIGKILL	Terminar	Terminar el proceso
SIGUSR1	Terminar	Definido por el proceso / usuario
SIGUSR2	Terminar	Definido por el proceso / usuario

> Tratamiento de signals

- Proceso irrumpe ejecución código para tratamiento → si sobrevive se continua dónde estaba.
- CADA proceso → tratamiento asociado a cada signal.
  - Por defecto
  - Capturado → modificación del tratamiento. Excepto SIGKILL y SIGSTOP.

- Procesos → bloquear/desbloquear recepción signals
  - Excepto: SIGKILL y SIGSTOP o SIGFPE, SIGILL y SIGSEGV provocados por excepción.
  - Signal bloqueado → no se recibe → si se envía, SO marca para tratarlo
    - bitmap del proceso → recuerda recibimiento de una signal de cada tipo.
    - Signal desbloqueado → tratamiento si bitmap indica que ha llegado.

> Enviar signals → int kill (int pid, int signum)

> Capturar signals → int sigaction (int signum, struct sigaction *tratamiento, struct sigaction *tratamiento_antiguo)

- Inicializar un struct sigaction
  - sa_handler
    - SIG_IGN -> ignorar el signal
    - SIG_DFL -> tratamiento por defecto
    - my_func -> función con cabecera: void my_func(int s)
  - sa_mask
    - vacía -> sólo se añade el signal que se está capturando
    - Al salir se restaura la anterior
  - sa_flags
    - 0 -> configuración por defecto
    - SA_RESETHAND -> después de tratar el signal se restaura el tratamiento por defecto
    - SA_RESTART -> Si estás haciendo una llamada a sistema y recibes un signal, se reinicia la llamada a sistema.

> Gestión de signals → por proceso

- Tabla de programación de signals (1 entrada/signal) → acción a realizar cuando se reciba el evento.
- bitmap de eventos pendientes (1 bit/signal)

- Un temporizador para la alarma → si programamos 2 veces la alarma solo queda la última.
- Máscara de bits → indica signals a tratar.
- Gestión real:
  - El proceso A apunta un signal en el PCB del proceso B.
  - El kernel ejecuta el código (ya sea el por defecto o el que ha especificado el proceso) para tratar el signal de B.

> Máscara de bits → estructura de datos que permite determinar qué signals puede recibir un proceso en un momento determinado de ejecución.

- int sigemptyset (sigset_t *mask) → init máscara sin signals.
- int sigfillset (sigset_t *mask) → init máscara con todos los signals.
- int sigaddset (sigset_t *mask, int signum) → añade el signal a la máscara.
- int sigdelset (sigset_t *mask, int signum) → elimina el signal de la máscara.
- int sigismember (sigset_t *mask, int signum) → cierto si signal está en la máscara.

> Bloquear/Desbloquear signals

- int sigprocmask (int operacion, sigset_t *mascara, sigset_t *vieja_mascara)
- int operación =
  - SIG_BLOCK → bloquea los signals de la máscara que le pases.
  - SIG_UNBLOCK → desbloquea los signals de la máscara que le pases.
  - SIG_SETMASK → intercambia las máscaras.

> Esperar un evento → bloquear proceso hasta que llega un signal que no es ignorado (no SIG_IGN)

- int sigsuspend (sigset_t *mascara)
  - mascara → signals bloqueados → control signal que saca de bloqueo.
  - salir del sigsuspend se restaura la máscara anterior.

> Sincronización de procesos

- Espera activa → proceso consume cpu para comprobar si ha llegado o no el evento. Normalmente comprobando una variable.
  - Recomendado para tiempo espera corto → No compensa la sobrecarga necesaria para ejecutar el bloqueo del proceso y el cambio de contexto.
  - Ej: while(!recibido)
- Bloqueo: El proceso libera la cpu (se bloquea) y será el kernel quien le despierte a la recepción de un evento.
  - Recomendado para tiempo espera largo → CPU libre para otros procesos.

> Control del tiempo

- int alarm (num_secs) → programa envío (por el kernel) de SIGALRM.
- Comportamiento típico

```
void configurar_esperar_alarma() {
   sigemptyset(&mask);
   sigaddset(&mask, SIGALRM);
   sigprocmask(SIG_BLOCK, &mask, NULL);
}
```

```
void esperar_alarma() {
   sigfillset(&mask);
   sigdelset(&mask, SIGALRM);
   sigsuspend(&mask);
}
```

Gestión interna de procesos

> Estructuras de datos: PCB y threads

- PCB → información asociada con cada proceso
  - PID
  - userID y groupID
  - Estado: RUN, READY, ...
  - Espacio para salvar los registros de la CPU
  - Gestión de signals
  - Información sobre la planificación
  - Información sobre la gestión de memoria
  - Información sobre la gestión de E/S
  - Accounting (recursos consumidos)

> Estructuras de gestión → organizan los PCB's en función de su estado y de necesidades de organización del sistema → SO distribuye.

- Procesos con mismo estado → organizados en colas o listas para mantener un orden.
- Ej: Cola procesos, cola procesos en ready, cola dispositivos

> Política de planificación (scheduler) → gestión de estructura de datos y de gestión.

- Planificación rápida → Cada 10ms tiene una interrupción de reloj para que un solo proceso no pueda acaparar la CPU.
- Clasificación de eventos:
  - Eventos preemptivos → política quita la cpu al proceso.
  - Eventos no preemptivos → política no quita la cpu al proceso, el la libera.
- Tipos de proceso:
  - Procesos de cálculo → ráfagas de computación.
  - Procesos de E/S → ráfagas de acceso a E/S.
- Context switch → cambio de contexto en CPU.
  1. Ejecutando proceso A
  2. Fin del quantum → interrupción del reloj
  3. Guardar Contexto de A -> PCB de A
  4. Planificador decide ejecutar B
  5. PCB de B -> Restaurar Contexto de B
  6. Ejecutando proceso B
  7. Repetir
- Tiempo total de ejecución de un proceso (Turnaround time) → tiempo total (todos los estados) desde inicio hasta final del proceso.
- Tiempo de espera de un proceso → Tiempo que el proceso pasa en estado ready.

> Round Robin → política preemptiva, procesos organizados según estado y orden de llegada.

- Eventos de activación:
  1. Un proceso se bloquea (no preemtivo)
  2. Un proceso termina (no preemtivo)
  3. Cuando termina el quantum (preemtivo)
- Proceso en run deja CPU y seleccionamos primero en cola ready. Según evento:
  1. Proceso en cola bloqueados hasta fin acceso a disp.
  2. Proceso zombie.
  3. Proceso a final cola ready.
- Tiempo espera màximo = (N.º procesos – 1) * Quantum

Relación syscalls-kernel

> Fork

1. Busca PCB libre y lo reserva
2. Inicializar datos (PID...)
3. Se aplica política de Gestión de memoria
4. Se actualizan las estructuras de Gestión de E/S
5. (RR) Se añade a la cola de READY

> Exec

1. Código / Datos / Pila -> NUEVO
2. Se inicializan las tablas de signals, contexto, ...
3. Se actualizan las variables de entorno, argv, registros, ...

> Exit

1. Se liberan los recursos del proceso
2. Se guarda el estado de finalización en el PCB
3. Se elimina de la cola de READY
4. Se aplica la política de planificación

> Waitpid

1. Se busca el proceso en la lista de PCB's para conseguir su estado de finalización
2. Si está ZOMBIE → el PCB se libera y se devuelve el estado de finalización al padre
3. Si NO está ZOMBIE → el proceso padre pasa de RUN → BLOCKED
4. Se aplica la política de planificación

Protección → problema interno al sistema

> Identificación usuario/contraseña (userID)
> GroupID → conjunto de usuarios

- Protección a Lectura/Escritura/Ejecución (rwx).
- Procesos → usuario determina sus derechos.

> ROOT → excepción → acceder a cualquier objeto y ejecuta operaciones privilegiadas.
> Setuid → un mecanismo para que un usuario pueda ejecutar un programa con los privilegios de otro usuario.

Seguridad → ataques externos

1. Físico → Poner las máquinas en habitaciones / edificios seguros.
2. Humanos → Controlar quien accede al sistema
3. SO → Evitar que un proceso sature el sistema, asegurar que siempre funcione, asegurar que ciertos puertos de acceso no están operativos, controlar que los procesos no se salgan de su espacio de direcciones.
4. RED → Es el más atacado