Boletín 4: Planificación de procesos y procesos de prioridad alta

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2024/2025

Implementación de procesos e hilos en xv6

- La unidad de aislamiento en `xv6` es el proceso
- La abstracción de proceso permite evitar que un proceso pueda acceder a los datos de otro proceso y del kernel
- Los mecanismos utilizados por el núcleo para implementar dicha abstracción incluye el flag de modo usuario/núcleo, los espacios de direcciones y la división en el tiempo de los hilos
- El núcleo de xv6 mantiene en la estructura `proc` la información relativa al estado de cada proceso. Entre los aspectos más importantes que se guardan en esta estructura está la tabla de páginas, su pila kernel y su estado de ejecución

Implementación de procesos e hilos en xv6 (cont.)

```
1  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

3  // Per-process state
   struct proc {
5    uint sz;                     // Size of process memory (bytes)
     pde_t* pgdir;                // Page table
7    char *kstack;                // Bottom of kernel stack for this process
     enum procstate state;        // Process state
9    int pid;                     // Process ID
     struct proc *parent;         // Parent process
11   struct trapframe *tf;        // Trap frame for current syscall
     struct context *context;     // swtch() here to run process
13   void *chan;                  // If non-zero, sleeping on chan
     int killed;                  // If non-zero, have been killed
15   struct file *ofile[NOFILE];  // Open files (file descriptors)
     struct inode *cwd;           // Current directory
17   char name[16];               // Process name (debugging)
   };
```

- Cada proceso tiene un hilo de ejecución que ejecuta las instrucciones del proceso
- La mayor parte del estado de un hilo (variables locales, dirección de retorno de la función) se almacena en las pilas del hilo

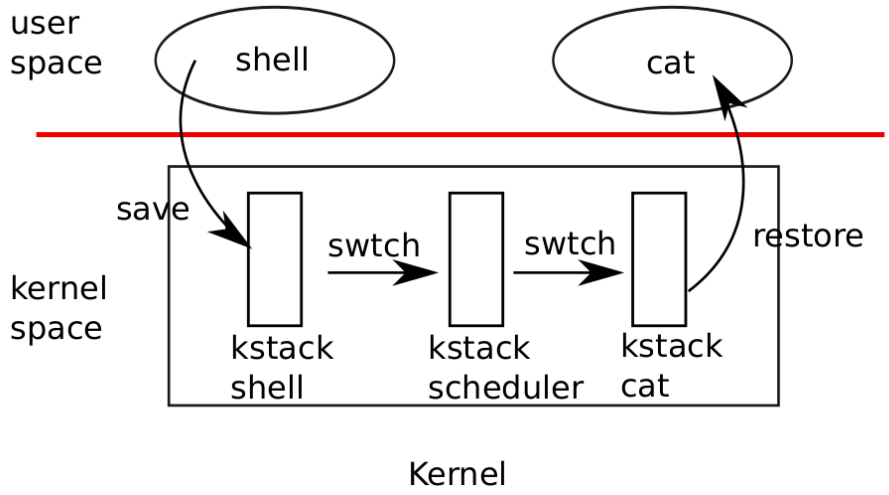
Implementación de procesos e hilos en xv6 (cont.)

- Cada proceso tiene dos pilas
  - Una pila de usuario que se usa cuando el proceso está en modo usuario
  - Una pila kernel que se usa cuando el proceso entra en el kernel

Planificación de procesos en xv6

- `xv6` proporciona a cada proceso la ilusión de que tiene su propio procesador virtual mediante la multiplexación de los procesos sobre los procesadores existentes
- `xv6` implementa la multiplexación cambiando de proceso en dos situaciones:
  - Los mecanismos `sleep` y `wakeup` permiten dicho cambio cuando un proceso espera a que un dispositivo o tubería finalice, está esperando a que termine un hijo o se encuentra esperando en la llamada al sistema `sleep()`
  - De forma periódica `xv6` fuerza dicho cambio cuando un proceso está ejecutando instrucciones de usuario
- La figura de la siguiente transparencia muestra, para el caso de un único procesador, los pasos necesarios para cambiar de un proceso de usuario a otro
  - Una transición de modo usuario a kernel (llamada al sistema o interrupción) al hilo del núcleo del proceso antiguo
  - Un cambio de contexto al hilo de planificación de la CPU local
  - Otro cambio de contexto al hilo del núcleo del nuevo proceso
  - Un regreso de *trap* al modo usuario del proceso

Planificación de procesos en xv6 (cont.)



- `xv6` usa dos cambios de contexto porque el planificador tiene su propia pila, lo que simplifica el proceso
- El cambio de un hilo a otro implica salvar los registros de CPU del hilo antiguo y restaurar los registros, previamente salvados, del nuevo hilo
- El hecho de guardar y restaurar los registros `%esp` y `%eip` implica que cambia el código que se está ejecutando
- `swtch()` no tiene noción acerca de los hilos, solamente guarda y restaura un conjunto de registros, denominados contextos

Planificación de procesos en xv6 (cont.)

- Cada contexto se representa por una `struct context*`, un puntero a una estructura almacenada en la pila kernel implicada

```
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- Cuando un proceso tiene que ceder la CPU, el hilo *kernel* del proceso llama a `swtch()`, implementada en el fichero `swtch.S`, para salvar su propio contexto y regresar al contexto del planificador

Planificación de procesos en xv6 (cont.)

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Planificación de procesos en xv6 (cont.)

- Los pasos a seguir cuando un proceso quiere liberar la CPU son los siguientes:
  - Adquirir el cerrojo que protege la tabla de procesos (`ptable.lock`)
  - Liberar cualquier otro cerrojo que hubiera adquirido
  - Actualizar su propio estado (`myproc()->state`)
  - Llamar a `sched()`
- Tanto `yield()` como `sleep()` y `exit()` siguen esta convención

```
// Give up the CPU for one scheduling round
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- `sched()` comprueba de nuevo dichas condiciones y además comprueba que las interrupciones están deshabilitadas (ya que se ha adquirido un cerrojo)
- Finalmente, `sched()` llama a `swtch()` para almacenar el contexto actual en `proc->context` y cambiar al contexto del planificador en `cpu->scheduler`

Planificación de procesos en xv6 (cont.)

```
1  void
   sched(void)
3  {
     int intena;
5    struct proc *p = myproc();

7    if(!holding(&ptable.lock))
       panic("sched ptable.lock");
9    if(mycpu()->ncli != 1)
       panic("sched locks");
11   if(p->state == RUNNING)
       panic("sched running");
13   if(readeflags()&FL_IF)
       panic("sched interruptible");
15   intena = mycpu()->intena;
     swtch(&p->context, mycpu()->scheduler);
17   mycpu()->intena = intena;
   }
```

- Al producirse el cambio de contexto se regresa a `scheduler()` en el punto en que se llamó a `swtch()`
- La función `scheduler()` continua ejecutando el bucle `for` buscando un proceso que ejecutar y cambiando a dicho proceso

Planificación de procesos en xv6 (cont.)

- La función `scheduler()` adquire el cerrojo `ptable.lock` para la mayoría de sus acciones, pero lo libera (y habilita las interrupciones) una vez por cada iteración de su bucle externo
- Esto es importante en el caso en que la CPU está desocupada para permitir que otras CPU adquieran la tabla de procesos y puedan marcar alguno de ellos como `RUNNABLE` o para permitir que se procesen las interrupciones de E/S
- Una vez encontrado un proceso, actualiza la variable `proc` y cambia a la tabla de páginas del proceso a través de `switchuvm()`, marca el proceso como `RUNNING` y llama a `swtch()` para empezar su ejecución

Planificación de procesos en xv6 (cont.)

```
// Per-CPU process scheduler
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor
    sti();

    // Loop over process table looking for process to run
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
```

Planificación de procesos en xv6 (cont.)

```
30          switchkvm();

32          // Process is done running for now.
            // It should have changed its p->state before coming back
34          proc = 0;
        }
36      release(&ptable.lock);

38    }
  }
```

Implementación a realizar

- Actualmente el planificador implementa una versión sencilla de un Round-Robin. Tendrás que implementar un planificador por colas de prioridad similar al de UNIX, que permita ascender y descender en dichas colas. También tendrás que añadir llamadas al sistema para permitir variar la prioridad de los procesos. Por tanto, modifica la tabla de procesos y la función `scheduler()` para que contemple lo siguiente:
  - Admitiremos como prioridad un número entre 0 y 9 ambos inclusive, tal y como permite Linux, siendo 0 la máxima prioridad y 9 la mínima, lo que implica implementar 10 colas, cada una de las cuales se comportará como un Round-Robin entre los procesos de la misma prioridad. La estructura de datos utilizada será un array de listas de procesos.
  - Cualquier proceso con prioridad alta siempre se ejecuta antes que cualquier proceso con prioridad menor.
  - Todos los procesos reciben la CPU de forma equitativa (round-robin), pero ahora entre los que tienen la misma prioridad.

Implementación a realizar (cont.)

- **EJERCICIO 1**:
  1. Añade una tabla con tantas entradas como prioridades a la estructura de datos que contiene la tabla de procesos y el candado correspondiente. Cada elemento de esta tabla corresponde a la prioridad de su índice y contendrá un puntero la entrada del primer proceso de esa prioridad y otro puntero a la entrada del último proceso.
  2. Modifica la estructura de datos del proceso para añadir una variable de tipo `unsigned int` que guarde la prioridad, y un puntero `siguiente` a entradas de la propia estructura de proceso. De esta forma, las colas de procesos de la misma prioridad se forman usando las propias entradas de la tabla de procesos. Si preferís no usar punteros, podéis utilizar dos enteros en su lugar como índices a la tabla de procesos.

Implementación a realizar (cont.)

3. Implementa una funcion para insertar al final de la lista un proceso, y otra para quitar el proceso del comienzo de la lista.
4. Implementa el planificador de procesos que usa dicho array de prioridades.
5. Recorre el código del kernel buscando aquellos lugares donde el estado de proceso cambia, y ver si hay que insertarlo en la lista o eliminarlo de ella.
6. Ten en cuenta los siguientes detalles:
   - Cada proceso que se cree tendrá asignada la prioridad normal (5).
   - A través de llamadas al sistema como las que se piden en el ejercicio 2, se deberá poder modificar y obtener la prioridad de un proceso.
   - Al realizar un `fork()`, el proceso hereda la prioridad del padre.

Implementación a realizar (cont.)

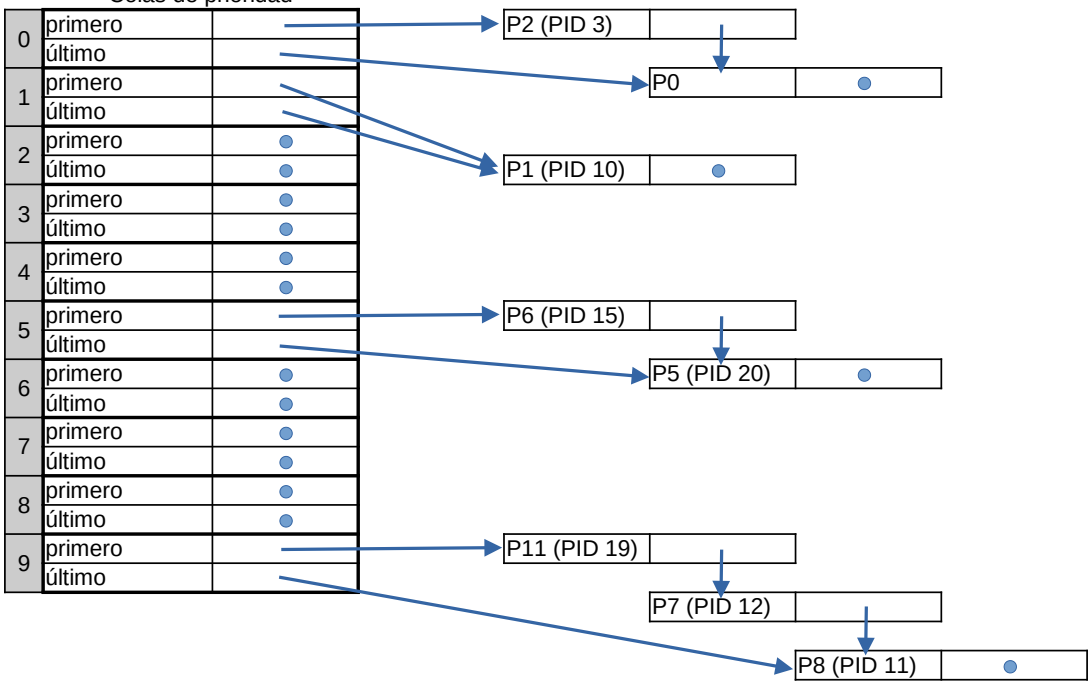
- Esquema de cómo sería la estructura de datos necesaria (la tabla de procesos está simplificada) antes de que el planificador seleccione al siguiente proceso que se ejecutará. IMPORTANTE: Date cuenta que la lista de la derecha está realmente dentro de la tabla de procesos de la izquierda, simplemente es la representación de cómo quedaría. La tabla de la derecha solamente guarda los punteros a la entrada de la tabla de procesos que están en primer y último lugar:

Implementación a realizar (cont.)

Tabla de procesos

	PID	Priori	Status	siguiente
0	PID 9	0	RUNNABLE	NULL
1	PID 10	1	RUNNABLE	NULL
2	PID 3	0	RUNNABLE	0
3	PID 8	1	*SLEEPING*	*NULL*
4	PID 4	1	*SLEEPING*	*NULL*
5	PID 20	5	RUNNABLE	NULL
6	PID 15	5	RUNNABLE	5
7	PID 12	9	RUNNABLE	8
8	PID 11	9	RUNNABLE	NULL
9	PID 7	5	*SLEEPING*	*NULL*
10	PID 6	9	*SLEEPING*	*NULL*
11	PID 19	9	RUNNABLE	7
12	PID 5	9	*SLEEPING*	*NULL*
…				
…				
…				
N-1				

Colas de prioridad

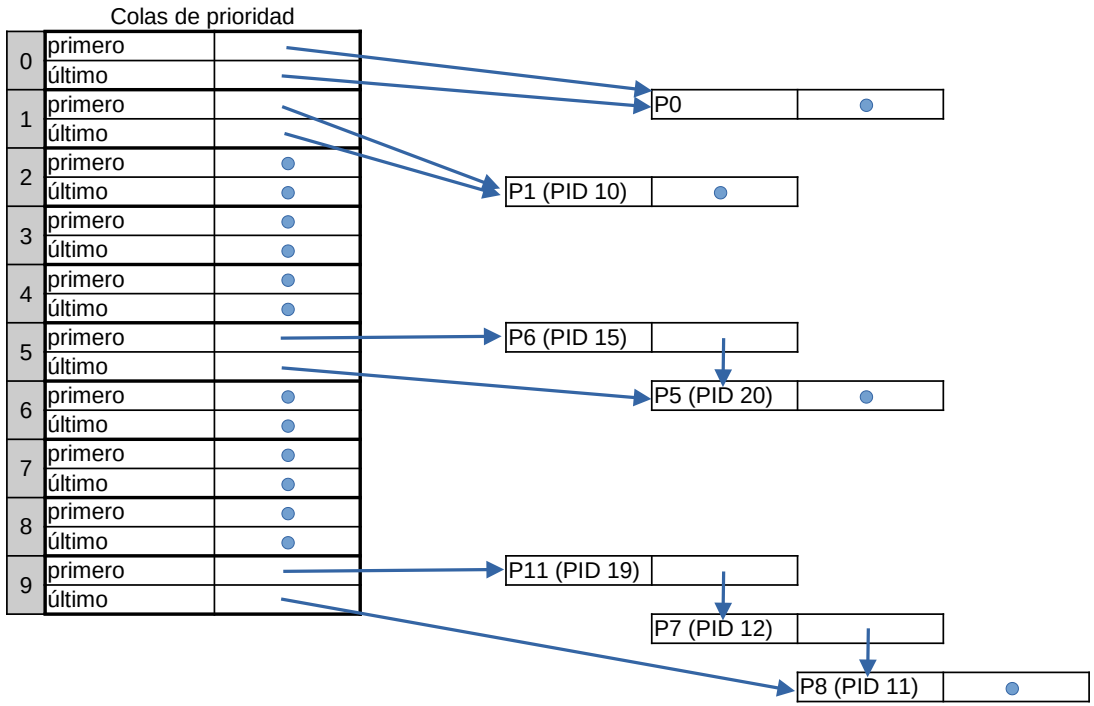


Implementación a realizar (cont.)

- Una vez que se ha seleccionado al proceso del comienzo de la cola de mayor prioridad, se quitará de dicha cola y se le cambiará el estado a RUNNING:

Tabla de procesos

	PID	Priori	Status	siguiente
0	PID 9	0	RUNNABLE	NULL
1	PID 10	1	RUNNABLE	NULL
2	PID 3	0	**RUNNING**	**NULL**
3	PID 8	1	*SLEEPING*	*NULL*
4	PID 4	1	*SLEEPING*	*NULL*
5	PID 20	5	RUNNABLE	NULL
6	PID 15	5	RUNNABLE	5
7	PID 12	9	RUNNABLE	8
8	PID 11	9	RUNNABLE	NULL
9	PID 7	5	*SLEEPING*	*NULL*
10	PID 6	9	*SLEEPING*	*NULL*
11	PID 19	9	RUNNABLE	7
12	PID 5	9	*SLEEPING*	*NULL*
…				
…				
…				
N-1				



Implementación a realizar (cont.)

- **EJERCICIO 2**: Añade a xv6 dos nuevas llamadas al sistema:

```
1    int getprio (int pid);
     int setprio (int pid, unsigned int proc_prio);
```

- Si el PID pasado como parámetro no existe, o el número de la prioridad no está en el rango permitido, devolverán -1 como error.

Salida del programa de prueba

- Para la realización de las pruebas, se tiene que establecer el número de CPUS a 1. Para ello, modifica la variable `CPUS` del fichero `Makefile` al valor 1
- Programa de prueba `tprio`. Este proceso genera 3 procesos con prioridad 9 y uno de prioridad 5, que muestran un carácter diferente cada uno por pantalla, resultando en un patrón alternante de caracteres. Como el shell tiene prioridad 5, podremos ejecutar órdenes, mostrando la salida por medio del patrón de caracteres repetidos. Cuando se está ejecutando el proceso de prioridad 5 junto con el shell, se intercalarán los caracteres de las ordenes ejecutadas con los asteriscos. A los pocos segundos se crearán 2 procesos de prioridad 0, que al ser de la mayor prioridad, monopolizarán la ejecución, no pudiendo ejecutarse órdenes en el shell. Si intentamos ejecutar un `ls`, no se ejecutará hasta que los procesos de alta prioridad no terminen, apareciendo la salida a continuación. Después deberá continuar el patrón de caracteres de los procesos de baja prioridad hasta que terminen su ejecución (ver vídeo en aula virtual).