Operating Systems 412

Pete Keleher

Process state

Process State Example

- more realistic, with I/O:

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Figure 4.4: **Tracing Process State: CPU and I/O**

- I/O completion enqueues process on *ready queue*
  - might not run immediately
- still no time-slicing

Process State *process control block (PCB)*

- A generic PCB (teaching OS called "xv6"):

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                   // Start of process memory
    uint sz;                     // Size of process memory
    char *kstack;                // Bottom of kernel stack
                                 // for this process
    enum proc_state state;       // Process state
    int pid;                     // Process ID
    struct proc *parent;         // Parent process
    void *chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;           // Current directory
    struct context context;      // Switch here to run process
    struct trapframe *tf;        // Trap frame for the
                                 // current interrupt
};
```


Process State *context*

- *Intel*

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

GeekOS

- GeekOS
  - A complete OS written at MD
  - We've removed most of the interesting bits
  - Runs on bare metal, but we use docker containers
- demo!

How to run a program *without losing control?*

- OS *timeshares* physical CPU
  - want program to run full speed
  - w/o losing control…
- Limited direct execution

OS	**Program**
1. Create entry on process list 2. Allocate program memory 3. Load program into memory 4. Init stack with `argc` / `argv` 5. Clear registers 6. Execute `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

- Control?

"All Your Bases are Belong to Us"

- Define "restricted" operations, such as
  - I/O requests
  - resource allocation
  - creating and destroying processes
  - accessing the file system
- Use protected control transfer
  - user mode: limited applications for apps
  - kernel mode: full access

How?

- **trap** instructions:
  - enter kernel
  - raise privilege level to kernel mode

- **return-from-trap** instruction
  - reduce privilege level to user mode
  - return to calling program

Limited Direct Execution

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	- tell hardware address of table - fill table w/ syscall handler addresses	

OS @ process startup (kernel mode)	Hardware	Program (user mode)
- Create entry for process list - Allocate memory for program - Load program into memory - Init user stack with argv - Fill kernel stack with reg/PC **- return-from -trap**		
	- restore user regs from kernel stack - move to user mode - jump to main	
		- Run main() … - Call system **- trap** into OS

Limited Direct Execution

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	- save regs to kernel stack - move to kernel mode - jump to trap handler	
- Handle trap - Do work of syscall **- return-from-trap**		
	- restore regs from kernel stack - move to user mode - jump to PC after trap	
		… - return from main - trap (via `exit()`)
- Free memory of process - Remove from process list		

Limited Direct Execution

- How does OS regain control?
  - cooperative:
    - apps voluntarily *yield* the CPU, or
    - OS grabs control at system calls
  - non-cooperative
    - *timer interrupts* etc.
    - faults (divide by zero, illegal access to memory)
  - reboot the machine

Timer Interrupts

- Periodic interrupts
  - OS starts timer during boot sequence
  - raised every *n* msecs
  - when raised:
    - currently running process halted
    - process state saved by kernel
    - pre-configured OS timer interrupt handle called
      - often used to *context switch* to another process

Context Switch

- low-level assembly code:
  - save some registers
    - general purpose registers
    - PC
    - kernel stack pointer
    - user stack pointer
  - restore a few for next process
  - switch to kernel stack of next process

Timer Interrupt

OS @ boot (kernel mode)	Hardware
initialize trap table	remember addresses of … - syscall handler - timer handler - …
start interrupt timer	- start timer - interrupt CPU in X ms

Timer Interrupt

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A …
	timer interrupt • regs(A) —> k-stack(A) • move to kernel mode • jump to trap handler	
Handle the trap Decide to *context-switch* Call `switch()` routine: • regs —> proc_struct(A) • regs <— proc_struct(B) • switch to k-stack(B) **return-from-trap (into B)**		
	• restore regs(B) <— k-stack(B) • move to user mode • jump to B's PC	
		Process B …

Context switch code

- /src/geekos/lowlevel.asm

```
Handle_Interrupt:
    ; macro defined above to push registers and create Interrupt_State
    Save_Registers


    ; Ensure that we're using the kernel data segment
    mov ax, KERNEL_DS
    mov ds, ax
    mov es, ax

    ; Get the address of the C handler function from the
    ; table of handler functions.
    mov eax, g_interruptTable   ; get address of handler table
    mov esi, [esp+REG_SKIP]     ; get interrupt number
    mov ebx, [eax+esi*4]        ; get address of handler function

    test ebx,ebx                ; if handler is null (ebx & ebx == 0), set ZF
    jz .bail_no_handler         ; if ZF, halt for debugging.

    ; Call the handler.
    ; The argument passed is a pointer to an Interrupt_State struct,
    ; which describes the stack layout for all interrupts.
    push esp                    ; struct Interrupt_State *
    call ebx
    add esp, 4                  ; clear 1 argument

    ; If preemption is disabled, then the current thread
    ; keeps running.
    mov ebx, [APIC_BASE+APIC_ID]                ;; load id of local APC (which is cpuid)
    shr ebx, 24-2                               ;; id is in high 24 bits of register, but need id <<2
    cmp [g_preemptionDisabled+ebx], dword 0
    jne .tramp_restore

        ;;  nspring - check if kthreadLock is; if so, skip preemption.
        ;;  this is a hack.  it can help, but is not reliable (we are
        ;;  not acquiring the lock, but another thread might.
;;; TODO: move this into eax to leave ebx untouched to simplify the needReschedule comparison.
    mov ebx, [kthreadLock]              ;; the lock value at the front of the spinlock.
    jne .tramp_restore

    ; See if we need to choose a new thread to run.
    mov ebx, [APIC_BASE+APIC_ID]                ;; load id of local APC (which is cpuid)
    shr ebx, 24-2                               ;; id is in high 24 bits of register, but need id <<2
    cmp [g_needReschedule+ebx], dword 0
    je  .tramp_restore

    ; Put current thread back on the run queue
    Push_Current_Thread_PTR
    call        Make_Runnable
    add esp, 4                  ; clear 1 argument

    ; Save stack pointer in current thread context, and
    ; clear numTicks field.
    Get_Current_Thread_To_EAX
    test eax,eax
    jne .ok
    jmp .bail_null_current_thread
.tramp_restore:
    jmp .restore
.bail_no_handler:
    call Hardware_Shutdown
.ok:
    mov [eax+0], esp            ; esp field
    mov [eax+4], dword 0        ; numTicks field

    ; Pick a new thread to run, and switch to its stack
    call        Get_Next_Runnable
    mov ebx, eax               ; save new thread into ebx
    test eax, eax              ; possibly redundant setting of the flags.
    jne .ok2
    jmp .bail_null_runnable_thread
.ok2:
    Set_Current_Thread_From_EBX
    mov esp, [ebx+0]                ; load esp from new thread

    ; Clear "need reschedule" flag
```

Interrupts during interrupts *interrupt or trap handling*

- Prevent by:
  - disabling interrupts during interrupt processing

  - locking mechanisms to protect kernel data
    - necessary for performant multiple cores

Operating Systems 412

Pete Keleher

Scheduling

Scheduling *introduction*

- Simplistic workload assumptions:
  - Each job runs for the same amount of time
  - All jobs arrive at the same time
  - All jobs are compute-bound (no I/O)
  - Run-time of each job known a priori

Scheduling *performance metrics*

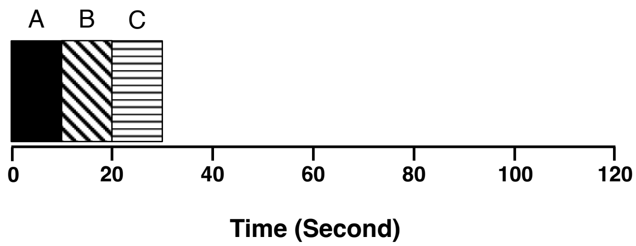
- Turnaround time
  - From job arrival to job completion

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- Fairness
  - Performance and fairness often conflict

Scheduling *FIFO*

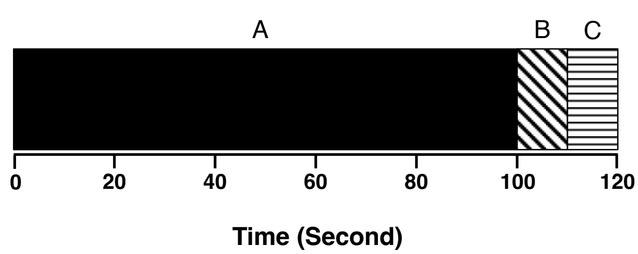
- First Come, First Served (FCFS)
  - non-preemptive, easy
- Example:
  - A ordered before B, before C
  - Each job runs 10 seconds



Average turnaround time = $\frac{10 + 20 + 30}{3} = \mathbf{20}$ seconds

Scheduling *FCFS*

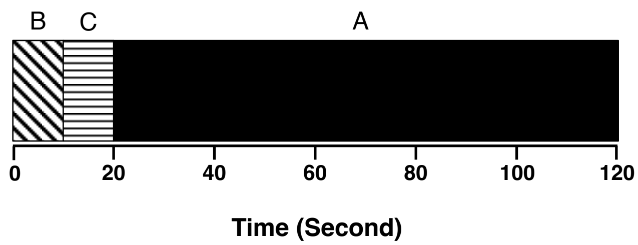
- Why not great?
  - *convoy-ing*
- Example:
  - Assume A runs for 100 seconds, B and C still 10 seconds



Average turnaround time = $\frac{100 + 110 + 120}{3} = \mathbf{110}$ seconds

Scheduling *SJF*

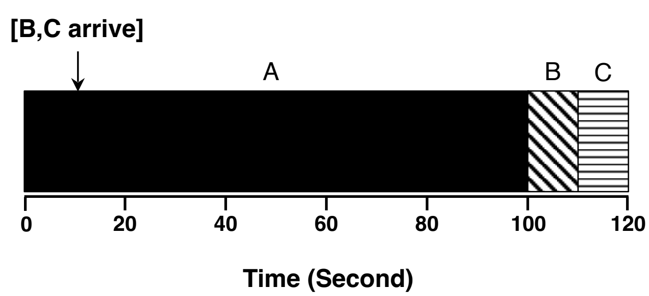
- Shortest-Job-First
  - always chooses shortest available job
- Example: assume A ordered last:
  - still A runs for 100 seconds, B and C still 10 seconds
  - still non-preemptive



Average turnaround time = $\frac{10 + 20 + 120}{3}$ = **50** seconds

Scheduling *SJF*

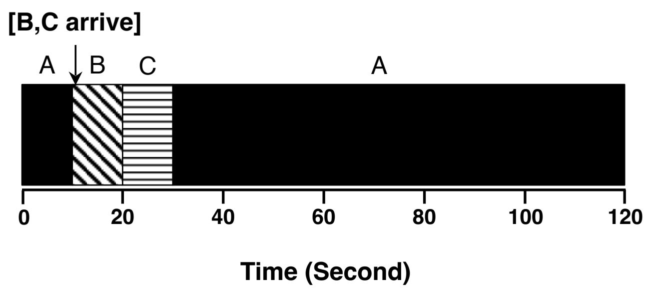
- Let's relax assumption that jobs all arrive at the same time
  - what could happen?
- Example:
  - A arrives at t=0, runs for 100 seconds
  - B, C arrive at t=10, run for 10 seconds
  - still non-preemptive



Average turnaround time = $\frac{100 + (110 - 10) + (120 - 10)}{3}$ = **103.3** seconds

Scheduling *STCF*

- Add preemption to SJF
  - Shortest Time-to-Completion First (STCF)
  - or Preemptive Shortest Job First (PSJF)
- New job arrives in system:
  - compare *remaining* time on all jobs
  - choose the shortest



$$\text{Average turnaround time} = \frac{120 + (20 - 10) + (30 - 10)}{3} = \mathbf{50} \text{ seconds}$$

Scheduling *response time*

- New metric: response time
  - time from job entering system, to start of first run

$$T_{response} = T_{first\ run} - T_{arrival}$$

  - or Preemptive Shortest Job First (PSJF)
- STCF etc. are not very good for response time
  - Why care?
  - How can we build a scheduler that is sensitive to response time?

Scheduling *round robin*

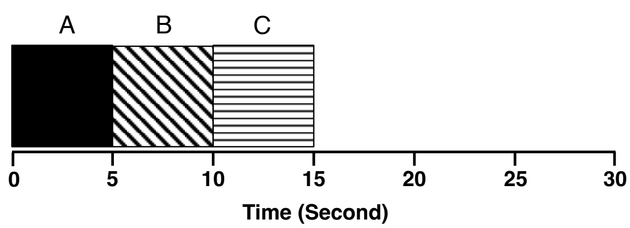
- Scheduling *time-slices*
  - run job for one time slice and then switch to next job
    - old job goes to end of *ready queue*
  - time slice also called *scheduling quantum*
  - mptive Shortest Job First (PSJF)

- Relies on timer interrupts to regain control
  - *Quantum* is a multiple of the timer-interrupt period

- STCF etc. are not very good for response time
  - Fair?
    - yes
  - Turnaround time?
    - not really

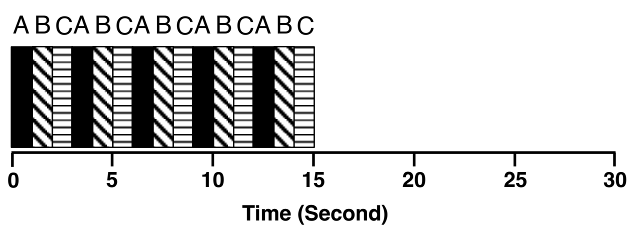
Scheduling *round robin*

- A, B, C arrive at the same time, each to run 5 seconds



SJF (Bad for Response Time)

$$T_{avg\ response} = \frac{0+5+10}{3} = 5 \text{ sec}$$



RR with a time-slice of 1sec (Good for Response Time)

$$T_{avg\ response} = \frac{0+1+2}{3} = 1 \text{ sec}$$

Scheduling *time slices*

It's a tradeoff….

- shorter time-slices
  - better response time
  - context switching overhead goes up

- Longer time-slices
  - Cost of switching amortized over more time
  - Worse response time

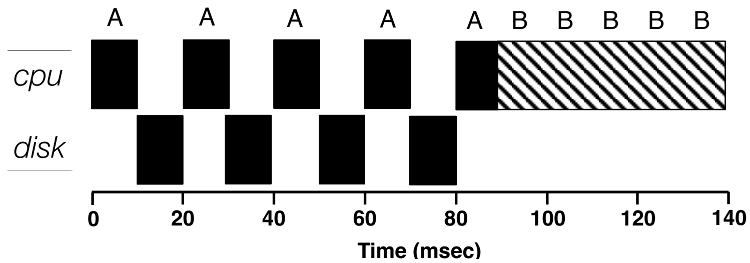
Scheduling *incorporating I/O*

Let's allow processes to perform *synchronous* (*blocking*) I/O

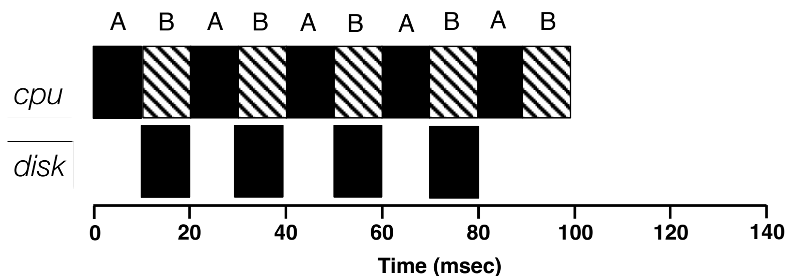
- When a job initiates an I/O request:
  - Job is blocked waiting for I/O completion
  - Scheduler chooses another job to run
- When I/O completes:
  - Interrupt is raised
  - OS moves process to ready state/queue

- Example:
  - A and B need 50 ms of CPU time each
  - A repeatedly runs for 10ms, then issues I/O request
    - assume 10 ms to satisfy requests
  - B needs 50ms CPU time, performs no I/O

Scheduling *incorporating I/O*



Poor Use of Resources



Overlap Allows Better Use of Resources

STCF and
treat each time slice
as distinct job