CPU/Process Management: Objectives & Challenges

- Scheduling
  - Determine the order in which processes get to use the CPU
  - Prevent starvation, ensure fairness, be efficient
- Allow processes to communicate efficiently and securely
- Synchronization
  - When things need to be done in a certain order
- Prevent deadlocks
  - A situation where no-one can make progress
  - More later

Data Structures for CPU Management

- Already seen: PCBs, one per process
- Several linked lists
  - List of ready (Runnable) processes
  - List of waiting (Blocked) processes
    - Often multiple lists, one per event
  - List of running processes??
- Que: Why linked lists? Why not better data structures?
Re-entrant Kernels

- Kernel control path
- Multiple kernel control paths may be active simultaneously
- Reason: Efficiency
- How to achieve this?

Interleaved Interrupt Handling

[Diagram showing user and kernel modes with user interrupts and exceptions.]
Data Structures for CPU Management (contd.)

- Kernel-level stacks for processes

Linked List Example from Linux

```c
struct list_head {
    struct list_head *next, *prev;
};

struct runqueue {
    [...]
    struct list_head migration_queue;
    [...]
};
```
Linked List in Linux (2.4 onwards)

- A linked list (what we normally encounter)

- A linked list in Linux

- Why?

Process Operations: Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes

- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources

- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (contd.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork` system call creates new process
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program
C Program Forking Separate Process

```c
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(1);
    } else if (pid == 0) { /* child process */
        execlp("/bin/ls", ":", NULL);
    } else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

A tree of processes on a typical Solaris

![Diagram of process tree](image)
Process Operations: Process Termination

- Process executes last statement and asks the operating system to delete it *(exit)*
  - Output data from child to parent (via *wait*).
  - Process’ resources are deallocated by operating system.
- Parent may terminate execution of children processes *(abort)*
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - If parent is exiting.
    - Some operating system do not allow child to continue if its parent terminates.
      - All children terminated - *cascading termination*.

CPU Scheduling
CPU Scheduling

• OS implements a CPU scheduling algorithm that decides which process to run when (plus itself!)
• Needs to be online
• Set of processes changes dynamically
• Needs to be fast, efficient
• May need to implement specified policies
  – E.g., Run certain processes at a higher priority

Timers

• CPU is a *temporal* resource
  – Other temporal resources? Non-temporal?
• OS needs to be aware of the passage of time
• Several clocks might be present
  – E.g., Intel: Real time clock, Time Stamp Counter, Programmable Interval Timer
  – RTC: Battery operated
  – TSC: All Intel CPUs have a CLK input pin, which receives the clock signal of an external oscillator
    • 64-bit TSC register, can be read using the `rdtsc` assembly instruction
    • Incremented at each clock signal
    • E.g., 400 MHz clock => TSC incremented every 2.5 nanosec
  – PIT: Like an alarm, the OS sets the frequency at which it interrupts
    • Linux programs PIT to send interrupt on IRQ0 every *tick* time units
    • Linux/Intel: ~10 msec, Linux/Alpha: ~1 msec
    • jiffies: a variable in Linux that stores # ticks since startup
Choosing the Right Scheduling Algorithm/Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Fairness

When is the scheduler invoked?

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling only under 1 and 4: nonpreemptive scheduling
- All other scheduling is preemptive
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- Dispatch latency – time it takes for the dispatcher to stop one process and start another running

Example from Linux 2.6.x

```c
// from Linux 2.6.x

asm volatile volatile(SAVE_CONTEXT

asm asm volatile volatile(SAVE_CONTEXT

```

```c
void __sched_schedule(void)
{
    [ . . . ]
    prepare_arch_switch(rq, next);
    prev = context_switch(rq, prev, next);
    barrier();
    finish_task_switch(prev);
}

task_t* context_switch(runqueue_t* rq, task_t* prev, task_t* next) {
    struct mm_struct* mm = next->mm;
    struct mm_struct* oldmm = prev->active_mm;
    /* Here we just switch the register state and the stack. */
    switch_to(prev, next, last);
    return prev;
}
```
First-Come, First-Served Scheduling (FCFS)

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: P₁, P₂, P₃
  The Gantt Chart for the schedule is:

  0  24  27  30
  P₁  P₂  P₃

- Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
- Average waiting time: \((0 + 24 + 27)/3 = 17\)

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order
P₂, P₃, P₁
- The Gantt chart for the schedule is:

  0  3  6  30
  P₂  P₃  P₁

- Waiting time for P₁ = 6; P₂ = 0, P₃ = 3
- Average waiting time: \((6 + 0 + 3)/3 = 3\)
- Much better than previous case
- Convoy effect short process behind long process