PROCESS MANAGEMENT

CONTENT
• OS Model
• Process Concept
• Process States
• Data Structure (PCB)
• Operations for Processes
• Interrupt
• Threads

A MODEL FOR THE OS
• System resources multiplexed among a number of active programs (activities)
• Can view the system as a set of queues representing different resources:
  • CPU queue, memory queue, I/O queue
  • Activities moving from one queue to another;
  • activities may be executed in parallel (truly parallel if more than 1 processor) or time sharing
• How to keep track of these activities?
• Have to identify and organize these relatively independent activities and let the O.S. executes them in turn
• Some kind of data structure is required to represent activities

NETWORK QUEUEING MODEL

CONCEPT OF A PROCESS
• Program: a passive entity
• Process:
  • a program in execution, i.e. an active entity containing a program counter specifying the next instruction to execute
  • 2 processes may be executing the same piece of code but following different paths
    • includes a process stack which contains temporary data
      • e.g. subroutine parameters, return addresses, temporary variables
    • includes a data section
• Why more than one process?
  • Divide and conquer
  • Fully utilize CPU time
  • Parallelism, cut down elapse time
  • Priority control
• In multiprocessing environment, need to:
  • have some means to synchronize processes, especially cooperative processes (e.g. buffers, locks)
  • keep track various states of a process

PROCESS STATES
• Running
  • Using CPU to execute codes, may or may not be interrupted
• Ready
  • Get all the resources required except CPU
  • Waiting for the scheduler to select for execution
• Blocked
  • Lack resource other than CPU, e.g. memory, peripheral device, I/O
  • or waiting for other process or a particular event, e.g. interrupt, signal, time
• Other situations
  • Deadlock, waiting to exit/terminate
• Scheduler is a module in OS (or a process itself) which schedules the CPU to execute different processes
• There is a ready queue for all processes in the ready state and the scheduler selects a particular process to run according to a
The process state changes along predefined paths:

- **Dispatch**
  - Assigns the process at the top of the ready queue to the CPU
  - Process: Ready --> Running

- **TimerRunOut**
  - To avoid any one process monopolizing the system, a specific time interval is allowed for each process
  - Process: Running --> Ready

- **I/O wait or Event wait while running (Blocked)**
  - Process: Running --> Blocked

- **I/O completion or Event completion (Wakeup)**
  - Process: Blocked --> Ready

**STATE TRANSITION DIAGRAM**

The process state changes along predefined paths:

- **Dispatch**
  - Assigns the process at the top of the ready queue to the CPU
  - Process: Ready --> Running

- **TimerRunOut**
  - To avoid any one process monopolizing the system, a specific time interval is allowed for each process
  - Process: Running --> Ready

- **I/O wait or Event wait while running (Blocked)**
  - Process: Running --> Blocked

- **I/O completion or Event completion (Wakeup)**
  - Process: Blocked --> Ready

**DATA STRUCTURE**

Information needed to characterize a process is in the **PROCESS CONTROL BLOCK**:

- Process identity
- State
- Priority
- Status flags to activate/resume processes
- Program counter, registers
- Memory management information
  - E.g. base & limit registers, page table, etc.
- List of requested resources, allocated resources, outstanding resources
- Number and type of outstanding I/Os
- Accounting information:
  - How many CPU time accumulated so far
  - How many I/O operations performed
- As process of different states are linked into different lists, these information should be referred from nodes of various queues

**EXAMPLE PCB DATA STRUCTURE IN OSP**

```c
typedef struct pcb_node PCB;
extern struct pcb_node {
    int pcb_id; /* PCB id */
    int size; /* process size in bytes; assigned by SIMCORE */
    int creation_time; /* assigned by SIMCORE */
    int last_dispatch; /* last time the process was dispatched */
    int last_cpuburst; /* length of the previous cpu burst */
    int accumulated_cpu; /* accumulated CPU time */
    PAGE_TBL *page_tbl; /* page table associated with the PCB */
    STATUS status; /* status of process */
    EVENT *event; /* event upon which process may be suspended */
    int priority; /* user-defined priority; used for scheduling */
    PCB *next; /* next pcb in whatever queue */
    PCB *prev; /* previous pcb in whatever queue */
    int *hook; /* can hook up anything here */
};
```

**OPERATIONS FOR**

- Primitives for process management is similar in different...
CREATE / SPAWN / FORK

• A process must be created by other processes
• Obtain new PCB from pool of system free memory
• Set up appropriate fields according to definition of PCB
• Insert into ready list
• Data usually can be defined (supplied as parameters in system call):
  • Priority
  • Level of privilege
  • Memory requirement

• Parent child relationship
  • Parent creates child processes and child process creates grand children
  • Form a family tree
  • Parent process can have some control over the children
  • Some systems do not implement parent child relationship - detach process

• Possible situations after process creation
  • Parent process continues execution concurrently with child
  • Parent waits till all children terminate
  • Parent process share all variables in common with child
  • Child share only subset of their parent's variables

• When child terminates, parent will be notified
• When parent terminates, child may or may not continue depending on design

EXAMPLE PSEUDO CODE

{(Parent process)
   ........
   (Parent creates child)
   process_ID := fork();
   if process_ID != 0 { (Parent and child execute test)
      then
         begin
            (Parent's code continues here)
            ....
            done_ID := wait(status) {Parent waits for child's termination}
            end
      else
         begin
            (Child's code continues here)
            ....
            exit(status) {Child terminates execution}
            end;
   (Parent continues here after child's termination)
   ....
OPERATION OF FORK SYSTEM CALL IN UNIX

PROCESS TERMINATION
- Process finishes its execution and asks OS to delete it
- Termination can be caused by other process (usually parent)
  - Cascading termination
  - When the parent terminates, all its children will also be terminated

EXIT / TERMINATE / DELETE
- Process terminates, all resources are released, files are closed, memory image removed, finally PCB itself is released back to system buffer
- If terminated process is a child process, then the parent process will be notified with the reason of the termination of the child process

KILL / ABORT
- Forceful termination of a process, usually used to removed hanged process
- OS usually provides some register & memory dump, in form of a file or a message to make subsequent investigation possible

PROCESS SYNCHRONIZATION
- FORK splits two sequences of codes into 2 processes
- JOIN/WAIT merges the two sequences of code, used by parent to synchronize with the child

SUSPEND / RESUME
- A process can be suspended indefinitely, placed in suspended state but remain in system
- Process may suspend itself, or others if it is privileged
- OS responds by inserting the PCB of the suspended process into the suspended list
- A process must be resumed by other process and OS inserts resumed PCB into ready list
- Some OS keep track of the depth of suspension and maintain a suspend count, increment when suspended and decrement when resumed, PCB will be moved back to ready list only when suspend count is zero
- SUSPEND/RESUME is the most basic primitives for interprocess synchronization

DELAY
- Wait voluntarily for a period of time (note difference from SUSPEND)
- Time may be a number of system clock ticks or standard time
units, usually OS handles in a form of interrupt

- Usage:
  - Time out handling (e.g. terminal)
  - Periodic running of a certain job
  - e.g. database backup, log file processing
  - usually scheduled by a daemon in UNIX

**HOUSE KEEPING OPERATIONS**
- Get Attributes
- Change Priority, etc.

**EXAMPLE USAGE OF PROCESS**
- UNIX initialization sequence
- Background & foreground process
- Communication handler

**CAN I INTERRUPT PLEASE?**
- Different situations happen in a system at the same time:
  - processes requesting system services
  - I/O devices performing I/O
  - hardware initiate exceptions

  All these situations can be abnormal or normal
  - for abnormal situations, actions must be taken immediately, e.g. hardware error
  - for normal situations, processing have to be scheduled according to priority

  To do this:
  - these abnormal situation has to be detected and identified
  - two possible ways:
    - periodically checking the occurrence of all possible abnormal situations
    - when situation arises, CPU is interrupted, stop whatever it is doing

**INTERRUPT TYPES**
- Different types of classification:
  - INTERNAL
    - Internal to CPU, during executing instruction
  - EXTERNAL
    - External to CPU, abnormal environmental situations
      - e.g. I/O completion, hardware fault, power failure, clock, inter CPU interrupt
  - SOFTWARE GENERATED
    - System calls
  - HARDWARE GENERATED
    - All other hardware related interrupts

**INTERRUPT PROCESSING**
- basically
  - a conditional "procedure call" but with mode switch involving 2 major steps: Interrupt **Preprocessing** and Interrupt **Postprocessing**

- but
  - even in normal operation, numerous interrupts are flooding CPU (similar to real time systems)
  - some must be handled immediately, or at least capture what is happening otherwise lose information
  - during processing of an interrupt, other interrupts may occur

- therefore
  - interrupt handlers must be written in a very efficient way
  - ability to handle non deterministic occurrence
  - mechanisms to prioritize interrupts occurring at the same time

**STEPS IN HANDLING INTERRUPT**
- Sensing interrupt
- Type determination & branch
- Execute Interrupt Servicing Routine (ISR)
- Return from interrupt
Upon Sensing Interrupt

- CPU temporarily give up current activity which can be suspended at any point and has to be resumed later
- to do this, context of the interrupted activity must be preserved:
  - Program Status Word, PSW
  - Program counter
  - Registers going to be modified
- saving of information can be performed by hardware or software
  - hardware saves all registers, program counter, status flags...etc.
  - hardware saves only PC & status flags, let software (ISR) saves registers
  - tradeoff: hardware is fast in speed, software is flexible
- saving of information is context save and switching the CPU to another routine is context switch
- before context save is completed, the interrupt system must be turned off, i.e. disable interrupt and CPU won’t response to any interrupt otherwise context of running process will be lost

Type Determination & Branch

- OS has to know what kind of interrupt has occurred so that an appropriate ISR (Interrupt Servicing Routine) can be called, 2 main methods:
  - Polling
    - for all types of interrupts, control are transferred by hardware to a fixed location in memory
    - CPU executes polling sequence to test who is interrupting
    - e.g. multiple I/O device environment, status testing
  - Problems:
    - only applicable in I/O environment, since some interrupts cannot be polled
    - waste CPU cycles to find out who is interrupting
    - polling sequence affects the priority if 2 interrupts occur at the same time
  - Vectoring

Execution in ISR

- ISR gets control from this point onward, usually enable interrupt system immediately
- may acknowledge interrupting device to stop continuous interruption

Return From Interrupt

- upon finish processing, all saved information must be restored either by software or a special instruction, e.g. RTI; this is context restore followed by another context switch
- jump to CPU scheduler, selects the next process to run
  - preemptive: depends on scheduling criteria, may not be the originally running process
  - non-preemptive: always the interrupted process

Diagram Showing the Steps

Example Interrupts in OSP

- the interrupting controller (hardware) provides information as part of interrupt request
- either as a unique identifier or address of the ISR
- hardware uses this information to transfer control:
  - loading the ISR address to program counter and starts execution, or
  - use the identifier to index into a table of ISR addresses (VECTOR TABLE) then jump to corresponding address
  - use of ISR address directly is inflexible
EXAMPLE INTERRUPTS IN OSP

The Module Structure of OSP

SOFTWARE INTERRUPTS

- pageint: page fault
- devint: device interrupt

SOFTWARE INTERRUPTS

- iosvc: I/O read, write
- procsvc: process control

SOFTWARE INTERRUPTS

- startsvc: start
- killsvc: kill
- termsvc: terminate
- sigsvc: signal
- waitsvc: wait

SOFTWARE INTERRUPTS

- Interrupts may occur during ISR execution
- Context of ISR saved and nested interrupt processing
- Interrupts are assigned levels of priority, and can be checked by :
  - Hardware knows the order and honour interrupts accordingly or by
  - Software :
    - use interrupt mask register which can be set by ISR
    - each bit represent an interrupt type
    - when an interrupt occurs corresponding bit is set in interrupt register
    - AND operation is performed for the 2 registers, interrupt allowed if resulting register is not zero
    - usually all lower priority and priority of the ISR itself will be disabled

SOFTWARE INTERRUPTS

- Interrupt type 4 occurs
- Interrupt type 2 occurs

SOFTWARE INTERRUPTS

- ISR set mask register

SOFTWARE INTERRUPTS

- Return from nested interrupt must go to next outer level of ISR until all pending interrupts are processed

SOFTWARE INTERRUPTS

- inter

SOFTWARE INTERRUPTS

- timeint: timer interrupt
PRE & POST WHEN IN AN ISR:

PROCESSING

- do actual processing as quickly as possible or
- capture all necessary information (in memory buffer), then schedule a process to handle it
- actual processing or scheduled process to handle interrupt is Post-processing; all other steps are Pre-processing
- execute return
- under some situations, control must be back to previous process/execution
  - e.g. nested ISR
  - running in kernel, modifying critical system data

PROCESS SWITCH

- After an interrupt, if CPU is switched to another process, i.e. Process Switch, more overhead is involved:
  - update PCB of the current process (Change state to Ready, Blocked, etc.)
  - move PCB to the corresponding queue
  - update PCB of the selected process
  - establish address space and memory image of new process
  - update of memory management data structure in PCB
- Frequent process switch is therefore inefficient
- To avoid this, use another mechanism, Light Weight Process or

MOTIVATION FOR USING THREADS

- Consider a typical client-server situation e.g. a DBMS
- Server implementation can be:
  - single process, serving one client at a time
  - multiple processes, serving many clients especially with multiple CPUs, but process switch is expensive (heavy weight process)
  - multiple threads (multi-threading)

THREADS OR LIGHT WEIGHT PROCESS

- A thread is an execution instance within a process sharing the same address space with other threads
- A thread consists of
  - a program counter
  - a register set
  - a stack
  - a state
  - may be some child threads
- Threads are very similar to processes
  - has own program counter and stack
  - can create child threads and can block waiting for system calls to complete
- Threads are different
  - owned by a single process
  - threads are not independent
  - created to perform tasks cooperatively & efficiently
- Threads within a process use the same address space owned by the process, therefore they:
  - share global variables
  - share code section and data section
  - can access every virtual address
  - one thread can read, write, or even wipe out another thread’s stack
  - no protection between threads
  - share the same set of open files, child processes, timers, signals, OS resources, etc.
- Extensive sharing makes switching among peer threads inexpensive, only require context switch, but no memory
management related work needs to be done

<table>
<thead>
<tr>
<th>COMPARISON</th>
<th>Per thread item</th>
<th>Per process item</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Program counter</td>
<td>Program counter</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>Register set</td>
<td>Register set</td>
</tr>
<tr>
<td></td>
<td>Child threads</td>
<td>Child processes</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td>Address space</td>
<td>Address space</td>
</tr>
<tr>
<td></td>
<td>Global variables</td>
<td>Global variables</td>
</tr>
<tr>
<td></td>
<td>Open files</td>
<td>Open files</td>
</tr>
<tr>
<td></td>
<td>Timers</td>
<td>Timers</td>
</tr>
<tr>
<td></td>
<td>Signals</td>
<td>Signals</td>
</tr>
<tr>
<td></td>
<td>Semaphores</td>
<td>Semaphores</td>
</tr>
<tr>
<td></td>
<td>Accounting info</td>
<td>Accounting info</td>
</tr>
</tbody>
</table>

• Disadv.:
  - kernel has no knowledge of threads (only schedules processes)
  - if a thread is waiting for system call, the entire process has to wait until the call returns
  - this is against the original design goal of preventing a blocked thread from affecting the others

KERNEL LEVEL THREADS

• Threads supported and scheduled by kernel
  - For each process, the kernel has a table with one entry per thread (similar to PCB) holding registers, state, priority, etc.
  - When a thread is blocked, the kernel can run either another thread of the same process, or a thread from another process
• Disadv.:
  - switching is more time consuming compared to User Level threads (done via an interrupt)

USER LEVEL THREADS

• Implemented as a set of user-level libraries (instead of system calls)
• Threads switching does not call the OS and cause interrupt to the kernel, switching can be done quickly
• When a thread is blocked, another thread from the same process will be run until the kernel takes the CPU away
• Adv.:
  - can be implemented on system that its kernel does not support threads
  - allow each process to have its own customized scheduling algorithm

A user-level threads package vs a threads package managed by the kernel.