Multi-core computing
Multi-processor Scheduling

Rajat Moona
moona@iitk.ac.in
Introduction

- Design issues in system with multiple processors
  - Tightly coupled or loosely coupled.
  - Message passing, shared memory or both.
- Several new issues are introduced into the design of scheduling functions.
- Loosely coupled systems are easier to handle.
  - Not an OS issue but an application issue
  - Several message passing systems are available (such as MPI)
- We will examine these issues and the details of scheduling algorithms for tightly coupled multi-processor systems.
Issues with Multi-processor computations.

- Granularity of computation
  - Fine grain
  - Coarse grain
    - Various shades of coarseness.

- Design issues
  - Assignment of processes to processors
  - Multiprogramming on individual processors
  - Actual dispatching of a process
Granularity

• The main purpose of having multiple processors is to realize ????.
• Applications exhibit parallelism at various levels with varying degree of granularity.
• **Fine grain** parallelism: Inherent parallelism within a single instruction stream.
  • High data dependency ➔ high frequency of synchronizations required between processes.
  • Dynamic Scheduling/Superscalar architectures exploit this kind of parallelism.
Fine Grain Parallelism

• Given a sequence of instructions the issue logic will look at the dependencies between multiple instructions in a window.
  • Dependencies: Data or Control dependency.
  • Dependencies: False or true.
  • Consider add R1, R2 followed by add R3, R4.
  • Consider div R1, R2 followed by div R3, R4
  • Consider load X followed by load Y

• Independent instructions can be executed in parallel if structural resources are available.
Granularity

- **Medium grain parallelism**: Parallelism of an application can be implemented by multiple threads in a single process.
  - Usually programmers have to “use” threads in the design.
  - Threads are scheduled by user (e.g. pthreads) or by OS (kernel threading).

- **Coarse grain parallelism**: Parallelism in a system by virtue of several concurrent processes
  - Need to synchronize using semaphore or other synchronization objects.
  - Example: Server side threads for web servers, FTP servers etc.
Granularity

• **Very coarse grain**: When synchronization needs are not high among parallel processes.
  • Processes can even be distributed across network.
    • Example: CORBA standard for distributed system.

• **Independent parallelism**: Multiple unrelated processes.
  • All independent processes can run in parallel.
Design Issues

• Where does the OS run?
• Mapping processes to processors.
  • Static or dynamic
• Use of multiprogramming on individual processors.
• Actual dispatching of a process.
Where does the OS run?

- **Master/slave assignment**: Kernel functions always run on a particular processor. Other processors execute user processes.
  - Advantage: Resource conflict resolution simplified since single processor has control.
  - Single point of failure (Master).
- **Peer assignment**: OS executes on all processors. Each processor does its own scheduling from the pool of available processes. Most OSes implement this in an SMP environment.
Mapping Processes to Processors

- Largely an application issue.
- Applications may be written for a particular configuration of the machine.
  - Application create threads of computations and channel of communication between these threads assuming a particular machine configuration.
- Applications may be generic in nature
  - Create \( n \) threads of computations. These threads can execute on machines with 1, \(<n\), \( n \) or \( >n \) processors.
  - OS may assign any number of processors (1 to \( n \)) to this application.
Multiprogramming at each processor

- In a multi-processor system, CPU utilization is not all that important
  - As long as some computation is being carried out, it is fine.
- Application efficiency are more important
  - Turn around time
  - application-related performance metrics.
- Assigning all threads to different processors may not always yield high performance.
Multiprogramming

- A multi-threaded application may require all its threads be assigned to different processors for good performance.
  - If there are $n$ threads and $m$ processors ($m > n$), $m-n$ processors may not have anything to execute.
- If threads are dependent on each other and require heavy synchronization, assigning all to the same processor may be beneficial.
- Process allocation can be Static or dynamic.
Process dispatching

• After assignment, deciding who is selected from among the pool of waiting processes
  • process dispatching.
• Single processor multiprogramming strategies may be counter-productive here.
• Priorities and process history may not be sufficient.
Process scheduling

- Single queue of processes or if multiple priority is used, multiple priority queues, all feeding into a common pool of processors.
- Multi-server queuing model: multiple-queue/single queue, multiple server system.
  - Inference: Specific scheduling policy does not have much effect as the number of processors increase.
- Conclusion: Use FCFS with priority levels.
Thread scheduling

- An application can be implemented as a set of threads that cooperate and execute concurrently in the same address space.
- **Load sharing**: pool of threads, pool of processors.
- **Gang scheduling**: Bunch of related threads scheduled together.
- **Dedicated processor assignment**: Each program gets as many processors as there are parallel threads.
- **Dynamic scheduling**: More like demand scheduling.
Contents

• Scheduling
  • Traditional
  • Real time
  • Multi-processor
• Synchronization and synchronization objects
• Inter-process communication
• Security
Multi-core computing

CPU Scheduling
Scheduling

- To hide the effects of I/O Bursts and achieve higher CPU utilization
- To give a fair chance to all processes
- Short term scheduling
  - CPU scheduling
- Long term scheduling
  - Process admission policies
- Medium term scheduling
  - Swap management
CPU Scheduling

- CPU scheduler selects a process from “Ready to run” queue
  - Scheduling decision
- Allocates CPU to this process.
  - Dispatch
    - Context switching
- Queue is maintained by PCB pointers
- Pre-emptive scheduling
  - When a running process may be removed and put back in the “ready to run” queue.
When to have CPU scheduling

- Process moves from running state to waiting state
  - Due to an I/O request
  - Due to a call to wait
- Upon Process Termination
- Expiration of time quota (preemptive)
- Upon any other time when OS is called
  - Interrupts, System Calls
Scheduling Criteria

• CPU Utilization *(System centric)*
  • Keep CPU as much busy as possible
• Throughput *(System centric)*
  • Number of processes completed/time
• Turnaround time *(Process centric)*
  • Real time taken to complete a process
• Waiting time *(Process centric)*
  • How much time a process is in ready queue
• Response time *(Process centric)*
  • Factor for an interactive process
• Deadline *(Real time behavior)*
  • Time guarantee to schedule a task.
Scheduling Algorithms

- **First Come First Serve (FCFS)**
  - Simplest implementation
  - No preemption

- **Shortest Job First (SJF)**
  - Optimal scheduling algorithm
  - Minimum Average Waiting Time
  - Difficulty: To know the time that it will take
  - Batch systems: A good choice.
Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
<th>Wait Time</th>
<th>Turnaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>20</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>P3</td>
<td>5</td>
<td>1</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

Gantt Chart (FCFS)

- Avg. Waiting time: 7.66
- Avg. Turnaround time: 16.66
- Avg. Burst: 9
### Example (Contd.)

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
<th>Wait Time</th>
<th>Turnaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>20</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>P3</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Gantt Chart (SJF)**

- Avg. Waiting time: 1.33
- Avg. Turnaround time: 10.33
- Avg. Burst: 9
SJF

• Though optimal, not practical.
  • No way to find the CPU burst.

• Scheduling techniques try to approximate SJF
  • Guess/predict the next burst ($\tau_n$: Predicted $n$th burst. $t_n$: Actual $n$th burst)
    \[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n \quad 0 \leq \alpha \leq 1 \]
\[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \quad 0 \leq \alpha \leq 1 \]

\[ \tau_{n+1} = t_n \quad \text{when } \alpha = 1 \]

\[ \tau_{n+1} = \tau_n \quad \text{when } \alpha = 0 \]

\[ \tau_{n+1} = \alpha t_n + (1 - \alpha).\alpha.t_{n-1} + \ldots + (1 - \alpha)^j.\alpha.t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0 \]

- Burst guess is obtained by *exponential* averaging.
- \( \tau_0 \) is a system wide scheduling constant.
- \( \alpha \) is a weight factor.
Example of “Guessing”

<table>
<thead>
<tr>
<th>CPU Burst</th>
<th>6</th>
<th>4</th>
<th>7</th>
<th>5</th>
<th>9</th>
<th>12</th>
<th>12</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guess ($\tau_0=10, \alpha=0.5$)</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

![Chart showing CPU burst times and guesses with annotations $\tau$ and $t$]
Preemptive SJF

• When a new task arrives, the SJF is evaluated again and rescheduling can take place.
  • *aka* Shortest-remaining-time-first (SRTF) scheduling
## Example (SRTF)

### Process Table

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>P3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

### Wait Time and Turnaround Time

<table>
<thead>
<tr>
<th>Wait Time</th>
<th>Turnaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Gantt Chart (SRTF)

- **P2** arrives at 4
- **P3** arrives at 5
- **P3** finishes at 9

### Performance Metrics

- Avg. Waiting time: 2
- Avg. Turnaround time: 11.66
- Avg. Burst: 9.66
Adding Priority (Priority scheduling)

- Simplest form is to add priority to a process.
  - Select the highest priority job first
  - Apply SJF/SRTF/FIFO/… scheduling only within a group of processes with the same priority
Priority scheduling + SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU Burst</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

![Diagram showing process scheduling](image-url)
Issues with Priority

• **Priority definition.**
  • How to define a priority?
    • System policies: Charge, job quantum, I/O
    • External or internal definition.

• **Pre-emptive vs. non-preemptive**
  • When a process arrives, should we remove the running process? (*e.g.* SRTF + Priority)

• **Starvation**
  • When high priority jobs come at a frequency that a low priority job is blocked.
  • Solution: add “aging”.
Aging with Priority

- Each time a process is scheduled,
  - Increase the priority of each pending task by 1. (could even be periodic)
- Priority: defined at admission time only.
- In pre-emptive scheduling.
  - When a job is pre-empted, reset the priority.
- Guaranteed “No starvation”.
Adding Time Quantum: Round Robin Scheduling

• FCFS + Preemption
  • Each time a process is scheduled, a time quantum is given to this.
  • When time quota expires, process is moved to ready queue.
    • Job is entered at the end of the queue and scheduled from the beginning of the queue.
      • Round robin scheduling
RR Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
</tr>
<tr>
<td>P2</td>
<td>20</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
</tr>
</tbody>
</table>

Time Quantum: 3 Units

Smaller Quantum: Large number of context switching. High response.
Larger Quantum: Slow response but fewer context switches
Multilevel Queue Scheduling

- Processes may be grouped.
  - System processes, interactive processes, batch processes etc.
- Ready queue may be made one for each group.
- Within a queue the scheduling can be one of the earlier defined scheduling (typically RR is used)
- The queues have associated priorities
Multilevel queue scheduling

- **Q1** G1 processes (Max Priority)
  - Process scheduled only when Q1 is empty

- **Q2** G2 processes (Next Priority level)
  - Process scheduled only when Q1 & Q2 are empty

- **Q3** G3 processes (Next Priority level)

- **Q4** G4 processes (Lowest Priority)
Issues with Multilevel queue

- Processes show a varying character
  - Interactive or non-interactive
  - Response requirement: high to low
- Processes in low level of priority may starve
- Solution: Add movement of processes from one queue to another
  - Multilevel Feedback Queue Scheduling
  - Movement can be due to
    - Change in character of the process
    - Aging
Scheduling in Real time OS

• Real time jobs have an additional criterion
  • Deadline, or time guarantee to execute a process

• Earliest Deadline First (EDF) Scheduling
  • Schedule a task that has earliest deadline
## EDF examples

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2</th>
<th>P1</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>8</td>
<td>14</td>
</tr>
</tbody>
</table>
EDF examples

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>16</td>
<td>6</td>
</tr>
</tbody>
</table>

EDF may miss a deadline even when a scheduling is possible without missing the deadline.
Possible feasible schedule

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>16</td>
<td>6</td>
</tr>
</tbody>
</table>

No process misses deadline
Least Slack First scheduling

- Slack: Burst + Deadline – Current time
- Least slack first is optimal scheduling for meeting the deadline.
- In real time systems, the task are well behaved
  - Easier to guess the burst accurately.
  - Or, the tasks announce their burst time.
LSF schedule

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Burst</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>16</td>
<td>6</td>
</tr>
</tbody>
</table>

**Slack**
P1: 20
P2: 22

**Slack**
P2: 17
RT Scheduling

- If task declare their CPU burst
  - A task that finishes within the time is a good task.
  - A task that leaves a lot of slack is an issue and can influence the admission policy.
    - Denial of service??
- Solution: “Charge” tasks based on their declared burst.
Characteristics of Real-Time (RT) systems

- Determinism
- Responsiveness
- User control
- Reliability
Deterministic Response

- External event and timings dictate the request of service.
- OS’s response depends on
  - speed at which it can respond to interrupts
  - whether the system has sufficient capacity to handle requests.
- Can we put an upper bound on time for OS response?
  - Factor of the OS design.
- In non-RT this delay averages around 50 to 500 ms,
  - Also is usually non-deterministic.
- In an RT, delay is usually guaranteed to have an upper-bound (usually small: few $\mu$s to 1-2ms.)
Responsiveness

- The time for servicing the interrupt once it has been acknowledged.
- Comprises:
  - Time to transfer control, (and context switch) and execute the ISR
- Depends upon
  - Interrupt latency of the hardware (usually very small)
  - Priority of interrupts.
  - Priority of tasks.
  - response time = \( f(\text{responsiveness, determinism}) \)
User Control

- **User control**: User has broader control on process characteristics specs in an RTOS
  - Priority
  - Deadlines: Hard or soft.
- **Memory Management**: paging or **swapping**
- **Name the processes to be resident in memory**
- **Scheduling policies**
**Reliability**

- **Reliability**: A processor failure in a non-RT may result in reduced level of service. But in an RT it may be catastrophic: life and death, financial loss, equipment damage.
- Fail-soft operation: Ability of the system to fail in such a way preserve as much capability and data as possible.
  - In the event of a failure, immediate detection and correction is important.
  - Notify user processes to rollback.
  - Apply compensation.
- Check-pointing and rollback states.
Requirements of RT

- Fast context switch
- Minimal functionality (small size)
- Ability to respond to interrupts quickly (Special interrupts handlers)
- Multitasking with signals and alarms
- Special storage to accumulate data fast
- Preemptive scheduling
Requirements of RT (contd.)

- Priority levels
- Minimizing “interrupt disabled” state
- Short-term scheduler (“omni-potent”)
- Time monitor
- Goal: Complete all hard real-time tasks by deadline. Complete as many soft real-time tasks as possible by their deadline.
Multi-processor scheduling

- Load sharing
  - Each processor runs one process.
  - A single ready queue is maintained (in a shared memory).
  - Each processor makes scheduling decisions independently.

- Speed up of an application
  - Processor assignment (within threads of the process): Dynamic or fixed
  - Scheduling: Gang scheduling or independent scheduling
Multi-core computing

Synchronization
Problem

• Multiple concurrent processes or threads using shared memory to communicate.
  • Concurrent access to the same data may lead to inconsistencies.
• An innocent looking code may not work when concurrency is involved.
• Remember *Producer-Consumer* code.
Anatomy of Downloading Window

- Picture Update during copy
- Status Update during copy
- Look for button during copy

Information Update

Opening: setup_cww6.exe from ftp.varietygames.com
Download to: Temporary Folder
Transfer rate: 57.1 KB/Sec

Close this dialog box when download completes

Open
Open Folder
Cancel

66% of setup_cww6.exe Completed
The Producer-Consumer Problem

Produce(item_t item) {
    while (count == bufsz);
    buffer[in] = item;
    in = (in + 1) % bufsz;
    count = count + 1;
}

Consume(item_t *item) {
    while (count == 0);
    *item = buffer[out];
    out = (out + 1) % bufsz;
    count = count - 1;
}

Shared Variables

buffer, count
What is Wrong?

- Variable count is shared.
- Both processes read and modify this variable.
- The assembly code for these two statements may be something like following:

```
//count=count+1
P1: load R1,count
P2: add R1,1
P3: store count,R1

//count=count-1
C1: load R1,count
C2: sub R1,1
C3: store count,R1
```

- Two processes run concurrently (Single or multiple CPUs)
A Possible Scenario

• The producer and consumer processes may be scheduled in any order and may be preempted.

• Consider the following sequence of statements
  • P1, \textit{CSwitch}, C1, C2, C3, \textit{CSwitch}, P2, P3.
A Possible Scenario

- The producer and consumer processes may be scheduled in any order and may be preempted.
- Consider the following sequence of statements
  - P1, CSwitch, C1, C2, C3, CSwitch, P2, P3.
A Possible Scenario

- The producer and consumer processes may be scheduled in any order and may be preempted.
- Consider the following sequence of statements
  - P1, \textit{CSwitch}, C1, C2, C3, \textit{CSwitch}, P2, P3.

\begin{tabular}{|c|c|c|c|}
  \hline
  & 5 & 5 & 4 \\
  count & R1 in P’s context & R1 in C’s context & C2: sub R1,1 \\
  \hline
\end{tabular}
A Possible Scenario

- The producer and consumer processes may be scheduled in any order and may be preempted.
- Consider the following sequence of statements:
  - P1, \texttt{CSwitch}, C1, C2, C3, \texttt{CSwitch}, P2, P3.

\begin{tabular}{|c|c|c|}
\hline
4 & 5 & 4 \\
\hline
count & R1 in P’s context & R1 in C’s context \\
\hline
\end{tabular}

C3: store count, R1
A Possible Scenario

• The producer and consumer processes may be scheduled in any order and may be preempted.
• Consider the following sequence of statements
  • P1, CSwitch, C1, C2, C3, CSwitch, P2, P3.
A Possible Scenario

• The producer and consumer processes may be scheduled in any order and may be preempted.
• Consider the following sequence of statements
  • P1, _CSwitch_, C1, C2, C3, _CSwitch_, P2, P3.
Race Conditions

• Situation when several concurrent processes operate on the same variable and the result of the computation depends upon the order in which they execute.

• In the preceding example, the value of count could be 4, 5 or 6.
  • C1, P1, P2, P3, C2, C3 \rightarrow final value 4
  • C1, C2, C3, P1, P2, P3 \rightarrow final value 5
  • P1, C1, C2, C3, P2, P3 \rightarrow final value 6
Critical Section Problem

• We model the problem using the notion of Critical Sections.
  • Critical sections are with respect to the shared data.
• There are $n$ processes, each sharing some common resource.
  • Each wants to modify the common resource.
• For each process, define the region of code where it accesses a shared piece of data as a critical section.
• In a correct system of multiple cooperating processes,
  • Only one process must be inside its critical section.
Problem Abstraction

• Processes execute code similar to the following.

while (1) {
    Non critical code
    Entry code
    Critical Section Code
    Exit Code
    Non Critical Code
}


Solution to Critical Section Problem

• Requirements
  • Mutual Exclusion
  • Progress
  • Bounded Wait

• We can make no assumptions on
  • Processor speed
  • Relative speeds of processes
  • Time to execute any critical/remainder section
  • Time to execute any entry/exit code
Mutual Exclusion

• Statement of obvious
  • if a process $P_i$ is in critical section at some point in time, no other process should be permitted to enter its critical section.

• This is clearly the most basic requirement for the solution.
Progress

• Given: There is no process in the critical section
  • And one or more processes want to enter the critical sections
  • Then one of them must be permitted to enter the critical section.
• One those processes waiting to enter the critical section must take part in the arbitration.
  • This arbitration must be done in a finite amount of time.
Bounded Wait

• A scheme of fairness and ensuring no starvation.
• There is an upper bound on the number of times that other processes are allowed to enter their critical sections between a process making its request and the request getting granted.
Solution issues

- Preemptive kernel
  - A process may be preempted when in kernel mode
- Non-preemptive kernel
  - A process may not be preempted when in kernel mode
- Preemptive kernels are difficult
  - Especially for SMP machines.
- Threads on multi-processors face independent OS.
- Preemptive kernels are essential
  - In embedded systems with RT guarantees
- Windows 2000, XP are non-preemptive
- Linux kernel became preemptive since Linux 2.6
Two process critical section solution

- Processes are P0 and P1.
- Processes share a common variable \( \text{turn} \) (=0 or 1).
- If \( \text{turn} = i \), \( P_i \) is permitted to enter the critical section.

```c
while (1) {
    While (turn != i);
    Critical section
    turn = 1-i;
    Remainder section
}
```

- **Mutual Exclusion:** ✓
- **Progress:** ×
- **Bounded Wait:** ×
Solution to Critical Section Problem

- Consider two processes P0 and P1.
- Shared variables (for solution to CSP)
  - int turn; boolean flag[2];
  - flag[i] is true when P_i is ready to enter its critical section.

```java
while (1) {
    flag[i] = TRUE; turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = FALSE;
    Remainder section
}
```

- Mutual Exclusion: ✔
- Progress: ✔
- Bounded Wait: ✔
Synchronization support in OS/ISA

• Synchronization code can be written using locks
  while (1) {
    Non critical code
    *Acquire Lock*
    Critical Section Code
    *Release Lock*
  }

• Implementation of Lock require support from the ISA
  • TestAndSet instruction, Swap instruction.
Support in ISA

• Recall: All instructions in a processor execute uninterrupted.
  • Within a single processor, instructions are atomic.
• Pentium ISA provide xchg instruction. (Swap instruction)
  
  xchg(register r, memory_address a) {
    t = r; r = *a; *a = t;
  }

  • One Read and One write in memory and register each.
Implementing locks using swap

- **Acquire Lock:**
  
  Register $AX = TRUE$;
  
  While $(AX=TRUE)$ xchg($AX, &lock$);

- **Release Lock:**
  
  - $Lock = false$;
Other Supports from ISA

- Some processors support TestAndSet instruction.
  ```c
  boolean TestAndSet(boolean *mem) {
    boolean ret = *mem;
    *mem = TRUE;
    return ret;
  }
  ```

- Acquire Lock:
  ```c
  while TestAndSet(&lock) ;
  ```

- Release Lock:
  ```c
  lock = false;
  ```
Support from the OS

• If OS is non-preemptive
  • System calls can be provided for
    • Acquire Lock and Release Lock.
  • Process can not be preempted while acquiring and releasing locks.

• If OS is preemptive.
  • System calls are tricky to support but not impossible.
Synchronization support in OS/ISA

• Synchronization code can be written using locks
  
  while (1) {
    Non critical code
    **Acquire Lock**
    Critical Section Code
    **Release Lock**
  }

• Implementation of Lock require support from the ISA
  • TestAndSet instruction, Swap instruction.

• OS may provide system calls to
  • Acquire lock or release lock.
  • For preemptive OS kernels, hard to implement these calls
Solutions for Multi-processes

boolean waiting[n], lock;
waiting[i] = TRUE;
key = TRUE;
while (waiting[i] && key) key = TestAndSet(&lock);
waiting[i] = FALSE;
// Critical Section
j = (i+1)%n;
while ((j != i) && waiting[j]==FALSE) j = (j+1)%n;
if (j==i) lock=FALSE; else waiting[j] = FALSE;
// Remainder Section
Multiprocessor issues

- Multiple processors share a single bus.
  - Arbitration is for bus cycles
  - Atomicity across processors is at the granularity of bus cycle.
- TestAndSet or Swap instructions require
  - At least one read and one write cycle
- Bus arbitration logic has to be instructed to give bus for two cycles.
  - Lock instruction prefix in Pentium
    - For example `lock xchg %ax, mem16`
- Lock instruction causes an arbitration sequence to be done for the entire instruction.
  - Atomic instruction execution across multiple processors
Semaphores

- A data structure abstraction for lock
  class semaphore {
    private: int s;
    public:
      void wait(void) {
        while (s < 0) ;
        s--; 
      }
      void signal(void) {
        s++;
      }
  }

  wait and signal are Atomic Methods.
  Also known as P and V.
  Also known as down and up.
Mutual Exclusion using Semaphore

```c
semaphore mutex;

: mutex.wait();
// Critical Section
mutex.signal();
// Remainder Section
:
```
Semaphore Types

• Binary or counting
• Binary: Semaphore value can be T or F only
• Binary semaphores provide mutual exclusion
  • Sometimes known as mutex locks
• Counting semaphores
  • Can be used when the given resource has finite number of instances ($n$).
  • Initialize semaphore.value to $n$. 
Synchronization among processes

• Two Processes P1 and P2.
• How do we make sure that P1 executes S1 first before P2 executes S2?

semaphore synch;

P1:
P2:
S1; synch.wait();
synch.signal(); S2
OS Implementations

• Busy wait is not tolerated in an OS!!
• Bounded wait is to be ensured.
• Solution:
  • Use sleep and wakeup to move processes from running to waiting state and waiting to ready state.
  • Maintain a queue of processes waiting and wake up processes in that order.
class semaphore {
private:
   int value;
   list<PCB> wait_list;
public:
   semaphore() {
      value = 0;
      wait_list = list<PCB>();
   }
   semaphore(int a) {
      value = a;
      wait_list = list<PCB>();
   }
   void wait(void) {
      value--;
      if (value < 0) {
         wait_list.push_front(current);
         sleep();
      }
   }
   void signal(void) {
      PCB p;
      value++;
      if (value <=0) {
         p = *(wait_list.end());
         wait_list.pop_back();
         wakeup(p);
      }
   }
};

Wait also known as \( P \)
Signal also known as \( V \)
Linux API for Semaphores

- Semget
  - To get an array of semaphores
- Semctl
  - Semaphore controls. For example initial value
- Semop
  - Semaphore Operations (lock, signal etc.)
Pthread Mutex Locks

- **Data types**: `pthread_mutex_t`
- **Creation**: `pthread_mutex_init`
- **Lock**: `pthread_mutex_lock`
- **Unlock**: `pthread_mutex_unlock`

```c
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```
Win32 Mutex Locks and Semaphores

- Declaration
  - HANDLE mutex; (or HANDLE semaphore);

- Creation
  - CreateMutex: To create a mutex lock
  - CreateSemaphore: To create a semaphore
    - semaphore = CreateSemaphore(NULL, 1, 5, NULL);

- Wait for mutex or semaphore
  - WaitForSingleObject(HANDLE, WAITTYPE)
    - WaitForSingleObject(Sem, INFINITE);

- Releasing
  - ReleaseMutex(mutex)
  - ReleaseSemaphore(sem, 1, NULL)
Multi-core computing

Inter-process communication
Processes in a group

- A process can be independent
  - Is not directly affected by other processes.
  - Does not affect other processes.
  - Example: /bin/ls and the shell
    - Are they related?

- Processes may be cooperating
  - Information Sharing
  - Speed up of execution
  - Modularity and convenience
Cooperating processes (example)

- Not really an example of “processes” but “threads”.
- The issues are the similar though.
Cooperating Processes

• Require
  • Inter process communication
    • Shared memory between processes
    • Message passing
      • Sender makes a call to the OS to send a message
      • Receiver makes a call to read message from the OS
  • Producer consumer relationship
    • A process produces data to be consumed by other process.
Shared Memory System
Message Passing (Send)
Message Passing (Receive)

- **Receive can be blocking**
  - A process makes a system call to receive a message.
    - If message is not available, the process is made to sleep (wait) and woken up when message is received.

- **Receive can be non-blocking**
  - Process makes a system call to receive a message.
    - Return value from the system call determines whether a message is ready or not.
Cooperating Processes

- Inter process communication
  - Shared memory between processes
  - Message passing
- Producer consumer relationship
**IPC: Shared memory**

- **Shared buffer between processes**
  ```c
  #define BUF_SZ 1024
  typedef struct {
    ...
  } BUF_Data;
  struct {
    BUF_Data items[BUF_SZ];
    int inptr, outptr; /* Global variables */
  } buffer; /* Must be shared between
  /* two processes */
  ```
Producer and consumer code

```c
void produce(BUF_Data item) {
    while (((buffer.inptr+1)%BUF_SZ == buffer.outptr) ;
    buffer.items[buffer.inptr] = item;
    buffer.inptr = (buffer.inptr +1)%BUF_SZ;
}

BUF_Data consume(void) {
    BUF_Data item;
    while (buffer.outptr == buffer.inptr) ;
    item = buffer.items[buffer.outptr];
    buffer.outptr = (buffer.outptr +1)%BUF_SZ;
    return (item);
}
```
Producer and consumer code

BUF_Data consume(void) {
    while (buffer.outptr == buffer.inptr) {
        buffer.items[buffer.outptr];
        buffer.outptr = (buffer.outptr + 1)%BUF_SZ;
        return (buffer.items[buffer.outptr-1]%BUF_SZ);
    }

    • What is wrong with this code?
Message Passing

- Communication channel
  - How to name the channel between two processes?
- Direct Communication
  - The sender process (P) must know the receiver process (Q) and vice versa
    - P: send(Q, msg) ♦ Q: receive(P, msg);
  - Some versions may have receive(ANY, msg);
- Indirect Communication
  - Mailboxes must be named rather than the processes.
    - Sender: Create_mailbox(Name, Properties);
    - Sender: Send(Name, msg);
    - Sender: Destroy_mailbox(Name);
    - Receiver: Open_mailbox(Name);
    - Receiver: Receive(mailbox, msg);
Synchronization with messages

- **Send:**
  - Blocking send
    - Sender process is blocked till receiver process receives the message
  - Non-blocking send
    - Sender process resumes after send.

- **Receive**
  - Blocking receive
    - Receiver process waits until a message is received
  - Non-blocking receive
    - Receiver process does not wait if the message is not ready. Return status indicates if message is ready or not.
Buffering Model

• **Buffer capacity:**
  • **Zero capacity.**
    • Link can not have any waiting message.
    • Sender must block till receiver is ready to receive
  • **Bounded capacity.**
    • Finite storage of waiting messages.
    • Sender blocks when link is full.
  • **Unbounded capacity.**
    • Sender never blocks.
Multiprocessor techniques

• Depends upon the hardware services.
  • Shared memory machines usually provide shared memory IPC.
  • Distributed memory machines usually provide message passing.
• On networked machines: Message passing.
• On multi-core machines: Shared memory and threading.
Multi-core computing

Security
First Job

• A quote from Network Security book by Charlie Kaufman, Radia Perlman, Mike Speciner.

Si spy net work, big fedjaw iog link kyxogy

Please decipher it for me.

Hint: Dedication.
Decipher

Si spy net work, big fedjaw iog link kyxogy
Replace S by T \(\rightarrow\) Ti tpy net work, big fedjaw iog link kyxogy
Replace i by o \(\rightarrow\) To tpy net work, bog fedjaw oog lonk kyxogy
Replace p by h \(\rightarrow\) To thy net work, bog fedjaw oog lonk kyxogy
Replace y by e \(\rightarrow\) To the net work, bog fedjaw oog lonk kyxogy
Replace n by b \(\rightarrow\) To the bet work, bog fedjaw oog lobk kyxogy
Replace e by a \(\rightarrow\) To the bat work, bog fadjaw oog lobk kyxogy
Replace t by d \(\rightarrow\) To the bad work, bog fadjaw oog lobk kyxogy
Replace k by s \(\rightarrow\) To the bad wors, bog fadjaw ooglobs syxoge
Replace w by g \(\rightarrow\) To the bad gors, bog fadjag ooglobs syxoge
Replace o by u \(\rightarrow\) To the bad gurs, bog fadjag ouglobs syxuge
Replace g by r \(\rightarrow\) To the bad gurs, bor fadjag ourlobs syxure
r by y, b by f, … \(\rightarrow\) To the bad guys, for making our jobs secure
Security Infrastructure

Goals
Confidentiality
Integrity
Availability

Security Infrastructure

Policies

Mechanisms

Enforcement Media

Resources

Information
Goals: Confidentiality

- Access to information or resources only to authorized users/entities/persons.

- Mechanism:
  - Access Control List (ACL) or Access control specifications
  - Data encryption

- Must have a mechanism to authenticate
  - Password, Cryptographic techniques

- Some time necessary to even conceal the fact that the information exist.
  - Access control on directories, Steganography
Goals: Integrity

• “Can I trust the information?”
• Data integrity to ensure that the data is genuine and is not modified
• Some time need to ensure that data originated from the right place
  • Origin integrity
• Mechanism
  • Cryptographic techniques
  • Hashing techniques
  • Checksums and use of alternate channels to send them.
Goals: Availability

• A most common attack is “denial of service” attack.

• Attacker does not get the access but can prevent other authorized users getting access as well.
Policies

• “What is permitted”
  • For example “only course students can have read access to these lecture notes”

• Policies are usually defined by the administrator or owner of the resource.
Mechanisms

• Mechanisms are techniques/methods to enforce a policy
• For example a “attributes” associated with a file can be changed by the owners
• Mechanism need not even be technical
  • A lost ID card application must be approved by the Dean’s office before a new one is issued.
• In computer related security, typically procedural mechanisms are used.
Enforcement Media

- The channel through which the information or a resource access is granted.
  - OS for example.
- Sometimes the media may not be trustworthy (for example the network)
  - In the security policies and mechanism this aspect has to be taken care of.
Threat and attack

- Threat is a potential violation of security
- Attack is actual violation.
  - Leakage of information
  - Modification of message while in transit
  - Loss of information
  - Proxy
  - Active and passive attacks
Information Security

• Mechanism to ensure that none of the threats, applicable to a scenario, apply.

• Techniques
  • Authorization
    • “Should you be doing that?”
  • Authentication
    • “Who are you?”
  • Cryptography
Threat perception and cost of security

- Securing a system has three components
  - Prevention, intrusion detection and recovery
- Each system has its own cost.
- Security techniques may not be easy to use
- Cost of securing a system must match the threat perception and value of information.
Cryptography: κρυπτο γραφή (hidden writing)

• Mangling of information in a way that unauthorized parties not able to de-mangle.
• Applications include integrity checking and authentication.
• Plaintext or cleartext: The message in its original form.
• Ciphertext: The mangled information.
Cryptography

Plaintext → encryption → ciphertext → decryption → Plaintext

Safe from unauthorized access
Cryptography

- Cryptographers: (good guys)
  - Invent clever algorithms
- Cryptanalysts: (bad guys)
  - Attempt to break algorithms
- *If lots of smart people have failed to solve a problem, then it probably won’t be solved, at least in near future.*
- Cryptography systems depend upon computationally difficult problems which become simple when a secret (key) is known.
Is Algorithm Secret?

- Some believe that keeping the algorithm secret enhances its security.
- Some believe that publishing the algorithm will enhance the security.
- Difficult to keep the algorithm secret.
- Common practice: Commercial algorithms are public while military applications keep it secret.
Some Simple Ciphers

- **Caesar Cipher**
  - Rotation of alphabet (substitution cipher)
  - Caesar used a fixed rotation of 3. (Computer ↔ Frpsxwhu)
  - Variant is when this rotation is variable.

- **Mono-alphabetic Substitution Cipher**
  - Si spy net work, big fedjaw iog link kyxogy
Breaking a cipher scheme

- Various kinds of attacks are possible.
  - Cipher-text only.
    - The attacker has access to cipher-text only but not the plaintext.
  - Known plaintext.
    - The attacker has access to few cipher-text and corresponding plaintext pairs.
  - Chosen plaintext.
    - The attacker can run his plaintext to get the corresponding cipher-text.
  - Chosen Cipher-text.
    - Same as chosen plaintext (but on the decryption algorithm)
Secret key cryptography

- Also known as Symmetric Key Cryptography or conventional cryptography.
- The following is relevant
  - Plaintext
  - Encryption Algorithm
  - Secret Key
  - Decryption Algorithm
  - Cipher-text
- Requirements
  - Strong encryption algorithm (attackers may have access to the algorithm and a few cipher-text-plaintext pairs)
  - Sender and receiver must have access to the secret key.
Secret Key Cryptography

Sender

Encryption Algorithm

Secure Channel

Key Source

Receiver

Decryption Algorithm

Cryptanalyst/Attacker

Key Source

Encryption Algorithm

Secure Channel

Decryption Algorithm

Cryptanalyst/Attacker

Key Source
Asymmetric Key Cryptography

Sender

Encryption Algorithm

$c$

$c$

$p$

$p'$

$p$

$p'$

Pr

Pu

Pu

Known to the world

Key Gen

Cryptanalyst/Attacker

Decryption Algorithm

$p'$

Pr'
Encrypting a large message

• If the message is more than the size of the block, the message can be broken in multiple blocks.
• Let the message be known as concatenation of \( p_1, p_2, p_3, \ldots, p_n \).
• There are the following modes of operation
  • Electronic Code Book (ECB)
  • Cipher Block Chaining (CBC)
  • Cipher Feedback Mode (CFB)
  • Output Feedback Mode (OFB)
  • Counter Mode (CTR)
ECB Mode of operation

Enc(k)

Dec(k)

Enc(k)

Dec(k)

Enc(k)

Dec(k)

Enc(k)

Dec(k)

Enc(k)

Dec(k)

Enc(k)

Dec(k)

Enc(k)

Dec(k)
Threats on ECB operational mode

- If the message contain two identical blocks, the corresponding cipher blocks are also identical.
  - Eavesdropper gets some information.
- Consider data base rows being sent from one end to another
  - Columns: Name, Position, Salary
  - Each is 64 byte wide (block size: 64 bytes)
- Now the eavesdropper can find out
  - Number of people at a particular position
  - Number of people at the same salary.
Threats on ECB

• An employee can even alter the message to change his own salary.

• Serious flaws:
  • Some one looking at the cipher text can gain information from the repeated blocks.
  • Some can even alter or rearrange the cipher text to his advantage.

• ECB is rarely used to encrypt messages.
CBC Mode of Encryption

\[ \text{IV} \xrightarrow{\oplus} \text{Enc}(k) \xrightarrow{\oplus} p_1 \xrightarrow{\oplus} \text{Enc}(k) \xrightarrow{\oplus} p_2 \xrightarrow{\oplus} \text{Enc}(k) \xrightarrow{\oplus} p_3 \xrightarrow{\oplus} \text{Enc}(k) \xrightarrow{\oplus} p_n \xrightarrow{\oplus} \text{Enc}(k) \xrightarrow{\oplus} c_n \]
CBC Mode of Decryption

- The receiver and sender must know key, and IV.
- Or key, and the method to compute IV.
Threats on CBC operation

- Modification of cipher text blocks.
- Changing $c_i$ has predictable effect on $p_{i+1}$.
  - However, it also changes $p_i$ in an unpredictable manner
- If the receiver is known to ignore $p_i$ then such attack is possible.
  - Possible safeguard is to attach a checksum (such as CRC) to the message before encryption.
PKI

• PKI Operations:
  • Encryption, Decryption of short messages
  • Digital signatures
  • Authentication

• RSA: Uses modular exponents to make it computationally infeasible to recover message without key.
  • Provided the message is carefully crafted. Keys are carefully selected.
Security Mechanism

- **Confidentiality**
  - Will an attacker make any sense out of a picked packet?
- **Integrity**
  - Is the message unaltered?
- **Owner Integrity**
  - Is the message really from that person?
- **Authentication**
  - Is it you?
Security Mechanism: Confidentiality

- Use of symmetric cipher.
- Encryption and decryption operations are needed.
- A Shared key is needed
  - Can be sent using PKI or any other reliable channel.
Security Mechanism: Integrity

- Map the message to smaller number of bits
  - Using one-way functions.
    - Message digest functions (such as MD5, SHA etc.)
    - Can be cryptographic functions as well.
- Send the mapped information on an alternate channel.
  - Confidentiality may be used.
- Or, send it along with the message
  - Confidentiality is a must.
- Integrity keys are different than the confidentiality keys.
  - Symmetric ciphers are used.
Authentication: Symmetric cipher based

- **Challenge response**
  - Challenger sends a random number to the subject.
- **Subject gives a response to the challenge**
  - Response derived using cryptography. For example, the encryption of the challenge using a shared key.
Owner Integrity: PKI

- Digital Signature
  - Can only be generated using the private key.
  - Can be verified using the public key.
- Since private key is one person,
  - Only the owner can generate it.
  - A document may be hashed and the hash may be signed digitally by the owner of the private key.
- Any one can verify the sign. Must have access to the public key of the signer.
Authentication: PKI

- **Challenge-Response**
  - Challenger can ask the subject to sign a random number
  - Challenger has access to the “certified” public key of the subject.
  - Only subject can sign it correctly since it must have the access to the private key.
  - Challenger can verify using public key.