Operating Systems

Steven Hand

12 lectures for CST Ia

*Easter Term 2000*

Part II: Operating System Functions

(Handout 1 of 2)
What is an Operating System?

- A program which controls the execution of all other programs (applications).

- Acts as an intermediary between the user(s) and the computer.

- Objectives:
  - convenience,
  - efficiency,
  - extensibility.

- Similar to a government ...
An Abstract View

- The Operating System (OS):
  - controls all execution.
  - multiplexes resources between applications.
  - abstracts away from complexity.

- Typically also have some libraries and some tools provided with OS.

- Are these part of the OS? Is IE4 a tool?
  - no-one can agree …

- For us, the OS ≈ the kernel.
In The Beginning ...

- 1949: First stored-program machine (EDSAC)
- to ~ 1955: “Open Shop”.
  - large machines with vacuum tubes.
  - I/O by paper tape / punch cards.
  - user = programmer = operator.
- To reduce cost, hire an operator:
  - programmers write programs and submit tape/cards to operator.
  - operator feeds cards, collects output from printer.
- Management like it.
- Programmers hate it.
- Operators hate it.
⇒ need something better.
Batch Systems

• Introduction of tape drives allow *batching* of jobs:
  – programmers put jobs on cards as before.
  – all cards read onto a tape.
  – operator carries input tape to computer.
  – results written to output tape.
  – output tape taken to printer.

• Computer now has a *resident monitor*:
  – Initially control is in monitor.
  – Monitor reads job and transfer control.
  – At end of job, control transfers back to monitor.

• Even better: *spooling systems*.
  – use interrupt driven I/O.
  – use magnetic disk to cache input tape.
  – fire operator.

• Monitor now *schedules* jobs …
Multi-Programming

- Use memory to cache jobs from disk ⇒ more than one job active simultaneously.

- Two stage scheduling:
  1. select jobs to load: *job scheduling*.
  2. select resident job to run: *CPU scheduling*.

- Users want more interaction ⇒ *time-sharing*:

- e.g. CTSS, TSO, Unix, VMS, Windows NT ...
Today and Tomorrow

• Single user systems: cheap and cheerful.
  – personal computers.
  – no other users ⇒ ignore protection.
  – e.g. DOS, Windows, Win 95/98, …

• RT Systems: power is nothing without control.
  – soft-real time: mp3 player.

• Parallel Processing: the need for speed.
  – SMP: 2–8 processors in a box.
  – MIMD: super-computing.

• Distributed computing: global processing?
  – Java: the network is the computer.
  – CORBA: the computer is the network.
Monolithic Operating Systems

- Oldest kind of OS structure ("modern" examples are DOS, original MacOS)
- Problem: applications can e.g.
  - trash OS software.
  - trash another application.
  - hoard CPU time.
  - abuse I/O devices.
  - etc ...
- No good for fault containment (or multi-user).
- Need a better solution ...
Dual-Mode Operation

- Want to stop buggy (or malicious) program from doing bad things.

⇒ provide *hardware* support to differentiate between (at least) two modes of operation.

1. *User Mode*: when executing on behalf of a user (i.e. application programs).
2. *Kernel Mode*: when executing on behalf of the operating system.

- Hardware contains a mode-bit, e.g. 0 means kernel, 1 means user.

- Certain machine instructions only possible in kernel mode …
Protecting I/O & Memory

- First try: make I/O instructions privileged.
  - applications can’t mask interrupts.
  - applications can’t control I/O devices.

- But:
  1. Application can rewrite interrupt vectors.
  2. Some devices accessed via memory

- Hence need to protect memory also...

- e.g. define a base and a limit for each program.

- Accesses outside allowed range are protected.
Protection Hardware

- Hardware checks every memory reference.

- Access out of range ⇒ vector into operating system (just as for an interrupt).

- Only allow update of base and limit registers in kernel mode.

- Typically disable memory protection in kernel mode (although a bad idea).

- Other hardware protection schemes possible …
Protecting the CPU

- Need to ensure that the OS stays in control.
  \[ \Rightarrow \text{use a } \textit{timer}. \]

- Usually use a \textit{countdown} timer, e.g.
  1. Set timer to initial value (e.g. 0xFFFF).
  2. Every \textit{tick} (e.g. 1\mu s), timer decrements value.
  3. When value hits zero, interrupt.

- (Modern timers have programmable tick rate.)

- Hence OS gets to run periodically and do its stuff.

- Need to ensure only OS can load timer, and that interrupt cannot be masked.
  - use same scheme as for other devices.

- Same scheme can be used to implement time-sharing.
Kernel-Based Operating Systems

- Applications can’t do I/O due to protection
  \[\Rightarrow\] operating system does it on their behalf.

- Need secure way for application to invoke operating system:
  \[\Rightarrow\] require a special (unprivileged) instruction to allow transition from user to kernel mode.

- Generally called a software interrupt since operates similarly to (hardware) interrupt …

- Set of OS services accessible via software interrupt mechanism called system calls.
Microkernel Operating Systems

- Alternative structure:
  - Push some OS services into servers.
  - Servers may be privileged (i.e. operate in kernel mode).

- Increases both *modularity* and *extensibility*.

- Still access kernel via system calls, but need new way to access servers:
  \[ \Rightarrow \] interprocess communication (IPC) schemes.
Kernels versus Microkernels

- Lots of IPC adds overhead
  ⇒ microkernels usually perform less well.

- Microkernel implementation sometimes tricky:
  need to worry about synchronisation.

- Microkernels often end up with redundant copies
  of OS data structures.

⇒ today most common operating systems blur the
  distinction between kernel and microkernel.

- e.g. Linux is “kernel”, but has kernel modules and
  certain servers.

- e.g. Windows NT was originally microkernel (3.5),
  but now (4.0) pushed lots back into kernel for
  performance.

- Still not clear what the best OS structure is, or
  how much it really matters ...
Operating System Functions

- Regardless of structure, OS needs to securely *multiplex resources*, i.e.
  1. protect applications from each other, yet
  2. share physical resources between them.

- Also usually want to *abstract* away from grungy hardware, i.e. OS provides a *virtual machine*:
  - share CPU (in time) and provide a virtual processor,
  - allocate and protect memory and provide a virtual address space,
  - present (relatively) hardware independent virtual devices.
  - divide up storage space by using filing systems.

- Remainder of this part of the course will look at each of the above areas in turn …
Process Concept

- From user’s point of view, the operating system is there to execute programs:
  - on batch system, refer to jobs
  - on interactive system, refer to processes
  - (we’ll use both terms fairly interchangeably)

- Process ≠ Program:
  - A program is static, while a process is dynamic
  - In fact, a process $\triangleq$ “a program in execution”

- (Note: “program” here is pretty low level, i.e. native machine code or executable)

- Process includes:
  1. program counter
  2. stack
  3. data section

- Processes execute on virtual processors
Process States

- As a process executes, it changes state:
  - **New**: the process is being created
  - **Running**: instructions are being executed
  - **Ready**: the process is waiting for the CPU (and is prepared to run at any time)
  - **Blocked**: the process is waiting for some event to occur (and cannot run until it does)
  - **Exit**: the process has finished execution.

- The operating system is responsible for maintaining the state of each process.
Process Control Block

| Process Number (or Process ID) | Current Process State | CPU Scheduling Information | Program Counter | Other CPU Registers | Memory Management Information | Other Information (e.g. list of open files, name of executable, identity of owner, CPU time used so far, devices owned) | Refs to previous and next PCBs |

OS maintains information about every process in a data structure called a process control block (PCB):

- Unique process identifier
- Process state (Running, Ready, etc.)
- CPU scheduling & accounting information
- Program counter & CPU Registers
- Memory management information
- ...
Context Switching

- *process context* = machine environment during the time the process is actively using the CPU.

- i.e. context includes program counter, general purpose registers, processor status register, . . .

- To switch between processes, the OS must:
  a) save the context of the currently executing process (if any), and
  b) restore the context of that being resumed.

- Time taken depends on h/w support.
Scheduling Queues

- **Job Queue**: batch processes awaiting admission.
- **Ready Queue**: set of all processes residing in main memory, ready and waiting to execute.
- **Wait Queue(s)**: set of processes waiting for an I/O device (or for other processes)
- **Long-term & short-term schedulers**:
  - *Job scheduler* selects which processes should be brought into the ready queue.
  - *CPU scheduler* selects which process should be executed next and allocates CPU.
Process Creation

- Nearly all systems are *hierarchical*: parent processes create children 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.

- Address space:
  - Child duplicate of parent.
  - Child has a program loaded into it.

- E.g. Unix:
  - `fork()` system call creates a new process
  - All resources shared (child is a clone).
  - `execve()` system call used to replace the process’ memory space with a new program.

- NT/2000: `CreateProcess()` system call includes name of program to be executed.
Process Termination

• Process executes last statement and asks the operating system to delete it (exit):
  – Output data from child to parent (wait)
  – Process’ resources are deallocated by the OS.

• Process performs an illegal operation, e.g.
  – makes an attempt to access memory to which it is not authorised,
  – attempts to execute a privileged instruction

• Parent may terminate execution of child processes (abort, kill), e.g. because
  – Child has exceeded allocated resources
  – Task assigned to child is no longer required
  – Parent is exiting (“cascading termination”)
  – (many operating systems do not allow a child to continue if its parent terminates)

• E.g. Unix has wait(), exit() and kill()

• E.g. NT/2000 has ExitProcess() for self and TerminateProcess() for others.
Process Blocking

- In general a process blocks on an event, e.g.
  - an I/O device completes an operation,
  - another process sends a message

- Assume OS provides some kind of general-purpose blocking primitive, e.g. await().

- Need care handling concurrency issues, e.g.

  ```
  if(no key being pressed) {
    await(keypress);
    print("Key has been pressed!\n");
  }
  // handle keyboard input
  ```

  What happens if a key is pressed at the first '{'?

- (This is a big area: lots more detail next year.)

- In this course we'll assume problems of this sort do not arise.
**CPU-I/O Burst Cycle**

- CPU-I/O Burst Cycle: process execution consists of a *cycle* of CPU execution and I/O wait.

- Processes can be described as either:
  1. I/O-bound: a process which spends more time doing I/O that than computation; has many short CPU bursts.
  2. CPU-bound: a process which spends more time doing computations; has few very long CPU bursts.

- Observe most processes execute for at most a few milliseconds before blocking

  ⇒ need multiprogramming to obtain decent overall CPU utilization.
**CPU Scheduler**

Recall: CPU scheduler selects one of the ready processes and allocates the CPU to it.

- Can choose a new process to run when:
  1. a running process blocks (running → blocked)
  2. a timer expires (running → ready)
  3. a waiting process unblocks (blocked → ready)
  4. a process terminates (running → exit)

- If only make scheduling decision under 1, 4 ⇒ have a *non-preemptive* scheduler:
  
  - ✔ simple to implement
  - ✗ open to denial of service
    - e.g. Windows 3.11.

- Otherwise the scheduler is *preemptive*.
  
  - ✔ solves DoS problem
  - ✗ introduces concurrency problems ...

26
Idle system

What do we do if there is no ready process?

• halt processor (until interrupt arrives)
  ✔ saves power (and heat!)
  ✘ might take too long.

• busy wait in scheduler
  ✔ quick response time
  ✘ ugly, useless

• invent idle process, always available to run
  ✔ gives uniform structure
  ✔ could use it to run checks
  ✘ uses some memory
  ✘ can slow interrupt response
Scheduling Criteria

A variety of metrics may be used:

1. CPU utilization: the fraction of the time the CPU is being used (and not for idle process!)

2. Throughput: # of processes that complete their execution per time unit.

3. Turnaround time: amount of time to execute a particular process.

4. Waiting time: amount of time a process has been waiting in the ready queue.

5. Response time: amount of time it takes from when a request was submitted until the first response is produced (in time-sharing systems)

Sensible scheduling strategies might be:

- Maximize throughput or CPU utilization

- Minimize average turnaround time, waiting time or response time.

Also need to worry about fairness and liveness.
First-Come, First-Served (FCFS) Scheduling

- Depends on order processes arrive, e.g.

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>25</td>
</tr>
<tr>
<td>$P_2$</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>9</td>
</tr>
</tbody>
</table>

- If processes arrive in the order $P_1$, $P_2$, $P_3$:

  - Waiting time for $P_1=0$; $P_2=25$; $P_3=29$;
  - Average waiting time: $(0 + 25 + 29)/3 = 18$.

- If processes arrive in the order $P_3$, $P_2$, $P_1$:

  - Waiting time for $P_1=13$; $P_2=8$; $P_3=0$;
  - Average waiting time: $(13 + 8 + 0)/3 = 7$.
  - i.e. over twice as good!

- First case poor due to *convoy effect*.  

29
SJF Scheduling

Intuition from FCFS leads us to *shortest job first* (SJF) scheduling.

- Associate with each process the length of its next CPU burst.

- Use these lengths to schedule the process with the shortest time.

- (FCFS can be used to break ties.)

For example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1=0; P_2=6; P_3=3; P_4=7$;

- Average waiting time: $(0 + 6 + 2 + 7)/4 = 3.75$.

SJF is optimal in that it gives the minimum average waiting time for a given set of processes.
SRTF Scheduling

- SRTF = Shortest Remaining-Time First.

- Just a preemptive version of SJF.

- i.e. if a new process arrives with a CPU burst length less than the *remaining time* of the current executing process, preempt.

For example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

![Process Execution Timeline]

- Waiting time for $P_1 = 9$; $P_2 = 1$; $P_3 = 0$; $P_4 = 2$;

- Average waiting time: $(9 + 1 + 0 + 2)/4 = 3$.

What are the problems here?
Predicting Burst Lengths

- For both SJF and SRTF require the next “burst length” for each process ⇒ need to estimate it.

- Can be done by using the length of previous CPU bursts, using exponential averaging:

  1. \( t_n \) = actual length of \( n^{th} \) CPU burst.
  2. \( \tau_{n+1} \) = predicted value for next CPU burst.
  3. For \( \alpha, 0 \leq \alpha \leq 1 \) define:

\[
\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n
\]

- If we expand the formula we get:

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

- Choose value of \( \alpha \) according to our belief about the system, e.g. if we believe history irrelevant, choose \( \alpha \approx 1 \) and then get \( \tau_{n+1} \approx t_n \).

- In general an exponential averaging scheme is a good predictor if the variance is small.
Round Robin Scheduling

Define a small fixed unit of time called a *quantum* (or *time-slice*), typically 10-100 milliseconds. Then:

- Process at the front of the ready queue is allocated the CPU for (up to) one quantum.

- When the time has elapsed, the process is preempted and appended to the ready queue.

Round robin has some nice properties:

- Fair: if there are $n$ processes in the ready queue and the time quantum is $q$, then each process gets $1/n^{th}$ of the CPU.

- Live: no process waits more than $(n - 1)q$ time units before receiving a CPU allocation.

- Typically get higher average turnaround time than SRTF, but better average *response time*.

But tricky choosing correct size quantum:

- $q$ too large $\Rightarrow$ FCFS/FIFO

- $q$ too small $\Rightarrow$ context switch overhead too high.
Static Priority Scheduling

- A priority value (an integer) is associated with each process.

- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - preemptive
  - non-preemptive

- e.g. SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time.

- Problem: how to resolve ties?
  - round robin with time-slicing
  - allocate quantum to each process in turn.
  - Problem: biased towards CPU intensive jobs.
    * per-process quantum based on usage?
    * ignore?

- Problem: starvation ...
Dynamic Priority Scheduling

- Use same scheduling algorithm, but allow priorities to change over time.

- e.g. simple aging:
  - processes have a (static) base priority and a dynamic effective priority.
  - if process starved for $k$ seconds, increment effective priority.
  - once process runs, reset effective priority.

- e.g. computed priority:
  - First used in Dijkstra’s THE
  - time slots: $\ldots, t, t+1, \ldots$
  - in each time slot $t$, measure the CPU usage of process $j$: $w^j$
  - priority for process $j$ in slot $t+1$:
    $$p^j_{t+1} = f(w^j_t, p^j_t, w^j_{t-1}, p^j_{t-1}, \ldots)$$
    - e.g. $p^j_{t+1} = p^j_t/2 + kw^j_t$
  - penalises CPU bound $\rightarrow$ supports I/O bound.

- today such computation considered acceptable $\ldots$
Multilevel Queue

- Ready queue partitioned into separate queues, e.g.
  - foreground (interactive),
  - background (batch)

- Each queue has its own scheduling algorithm, e.g.
  - foreground: RR,
  - background: FCFS

- Scheduling must also be done between the queues:
  - Fixed priority scheduling; i.e., serve all from foreground and then from background.
  - Time slice: each queue gets a certain amount of CPU time which it can divide between its processes, e.g. 80% to foreground via RR, 20% to background in FCFS.

- Also get multilevel feedback queue:
  - as above, but processes can move between the various queues.
  - can be used to implement dynamic priority schemes, among others.