Operating Systems Functions

Steven Hand

8 lectures for CST Ib and Diploma

Lent Term 2000

Handout 1

Recommended Reading

- Bacon J M
  Concurrent Systems (2nd Ed)
  Addison Wesley 1997

- Silberschatz A, Peterson J and Galvin P
  Operating Systems Concepts (5th Ed)
  Addison Wesley 1998

- Tannenbaum A S
  Modern Operating Systems
  Prentice Hall 1992

- Leffler S J
  The Design and Implementation of the 4.3BSD UNIX Operating System.
  Addison Wesley 1989

- Solomon D
  Inside Windows NT (2nd Ed)
  Microsoft Press 1998

- Singhal M and Shivaratris, N
  Advanced Concepts in Operating Systems
  McGraw-Hill 1994

- OS links (via course web page)
  `http://www.cl.cam.ac.uk/Teaching/1999/OSFuncs/`

Course Outline

1. Introduction and Review.
   OS functions & structures. Multiprocessor schemes. Processes and threads.

2. CPU Scheduling.
   Static/dynamic priority schemes. RT scheduling (RM, EDF, etc.). SRT scheduling.

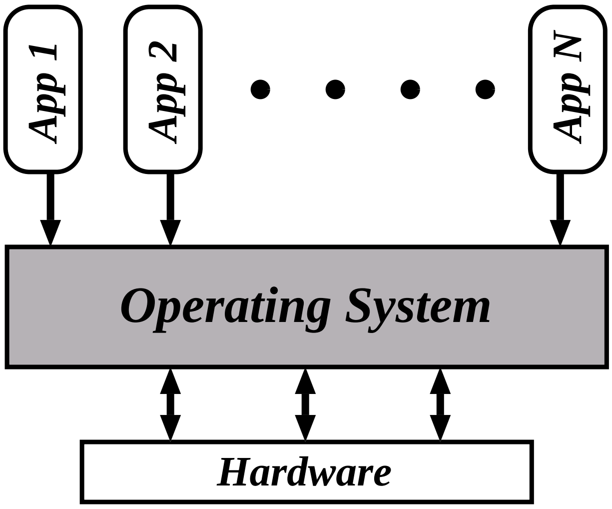
3,4. Memory Management.
   Review: segmented/paged memory. Translation schemes. Demand paging & replacement strategies. Case studies. Other VM techniques.

5,6. Storage Systems.
   Basic I/O revisited. Disks & disk scheduling. Caching and buffering. Case studies. Filing systems (FAT, FFS/EXT2, NTFS).

7. Protection.
   Subjects and objects. Authentication schemes. Capability systems.

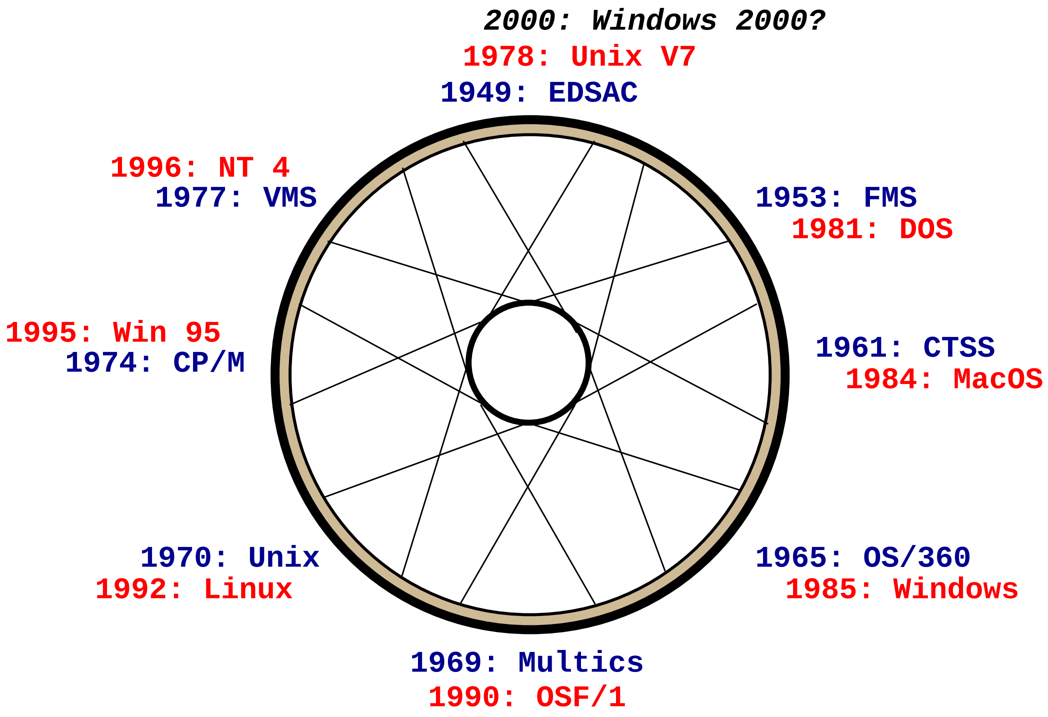
8. Extensibility.
   Motivation. Low-level, OS-level and user-level techniques (and examples).

A Generic Operating System



- What is the OS?
  - The “master control program”.
  - A virtual machine.
  - Everything shipped by a vendor.
  - The management . . .
- Objectives:
  - convenience
  - efficiency
  - extensibility
- All about trade-offs . . .

Historical Perspective



2000: Windows 2000?

1978: Unix V7

1949: EDSAC

1953: FMS

1981: DOS

1961: CTSS

1984: MacOS

1965: OS/360

1985: Windows

1969: Multics

1990: OSF/1

1970: Unix

1992: Linux

1995: Win 95

1974: CP/M

1996: NT 4

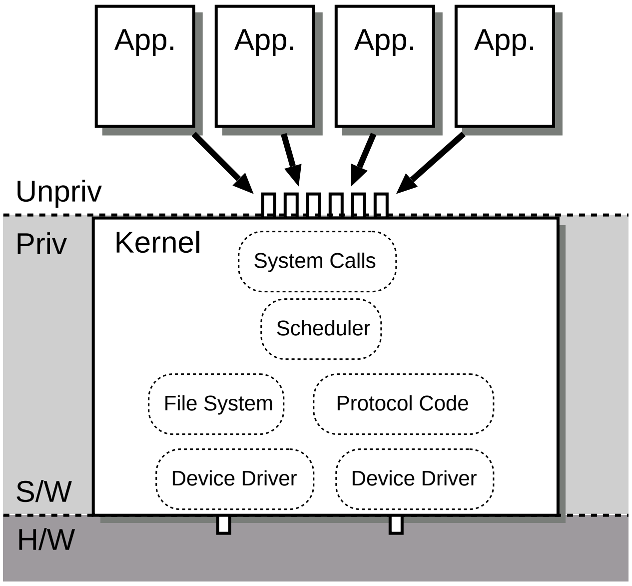
1977: VMS

- 1949: "Open Shop" — team of people design, build, operate & maintain computer.
- 1953: Batch Processing — "resident monitor" schedules jobs and (later) CPU.
- 1961: Time-Sharing — fine-grained multiplexing; job submission (and output) via terminals.
- 1981: Personal Computing — focus on single user; easy to forget earlier lessons.

Hardware Protection

- We want to ensure that a buggy (or malicious) application cannot:
  - compromise the operating system.
  - compromise other applications.
  - deny others service (e.g. abuse resources)
- To solve this efficiently and flexibly, need hardware support e.g. dual-mode operation.
- Then:
  - add memory protection hardware $\Rightarrow$ applications confined to subset of memory;
  - make I/O instructions privileged $\Rightarrow$ applications cannot directly access devices;
  - use a *timer* to force execution interruption $\Rightarrow$ OS cannot be starved of CPU.
- Dual-mode operation leads naturally to a two-tiered OS structure ...

Kernel-Based Operating Systems



- Applications can't do I/O due to protection

  ⇒ operating system does it on their behalf.

- Need secure way for application to invoke operating system:

  ⇒ 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*.

System Call Implementation

Most processors have an instruction such as:

- Software Interrupt (SWI, INT)
- System Call (SYSCALL)
- TRAP

which forces the processor to defined state, i.e.

- save current (user) state
- enter supervisor mode
- jump to defined address

This provides (usually) a single point of entry to the kernel where can check, e.g.

- if sensible arguments have been passed in,
- if process has the relevant access rights.

Entering supervisor mode typically allows the issuing of instructions not possible in user mode:

- access to memory protection hardware
- access to I/O instructions or I/O address space
- setting interrupt level (disabling interrupts)

Syscall Implementation
- User Space -

```
#include <syscall.h>

int ThreadCreate(Asid asid, ThreadDesc *desc,
                 vir_bytes arg);

... <in syscall.h> ...

#define SC_NULL                   1000

#define SC_SAS_KERNEL             1001
#define SC_GET_ENV                1002
#define SC_GET_STATISTICS         1003
#define SC_GET_SYSTYPE            1004

#define SC_THREAD_CREATE          1009
#define SC_THREAD_EXIT            1011
#define SC_THREAD_ID              1012
#define SC_BLOCK                  1014

... etc...
```

Syscall Implementation (ARM)
- User Space -

```
#include "syscall.h"

#define SYSCALL(routine, number)    \
.global routine;                    \
routine: ;                          \
        mov r12, \# number - 1000 ; \
        swi number ;                \
        movs r15, r14


SYSCALL(_ThreadCreate,  SC_THREAD_CREATE)

SYSCALL(_ThreadExit,    SC_THREAD_EXIT)

SYSCALL(_ThreadId,      SC_THREAD_ID)

SYSCALL(_Block,         SC_BLOCK)

... etc ...
```

Syscall Implementation
- Kernel -

```
File syscall.c (kernel)

typedef int (*IFP)();

IFP syscalls[256] = {
   null,                        /*  0: Null */
   sas_kernel,                  /*  1: SASKernel */
   environ_get,                 /*  2: GetEnv */
   GetStatistics,               /*  3: GetStatistics */
   get_systype,                 /*  4: GetSystype */
   bad_sys,                     /*  5: */
   bad_sys,                     /*  6: */
   bad_sys,                     /*  7: */
   bad_sys,                     /*  8: */
   threadCreate,                /*  9: ThreadCreate */
   bad_sys,                     /* 10: ThreadFork (obsolete) */
   threadExit,                  /* 11: ThreadExit */
... etc ..
```

Syscall Implementation (ARM)
- Kernel -

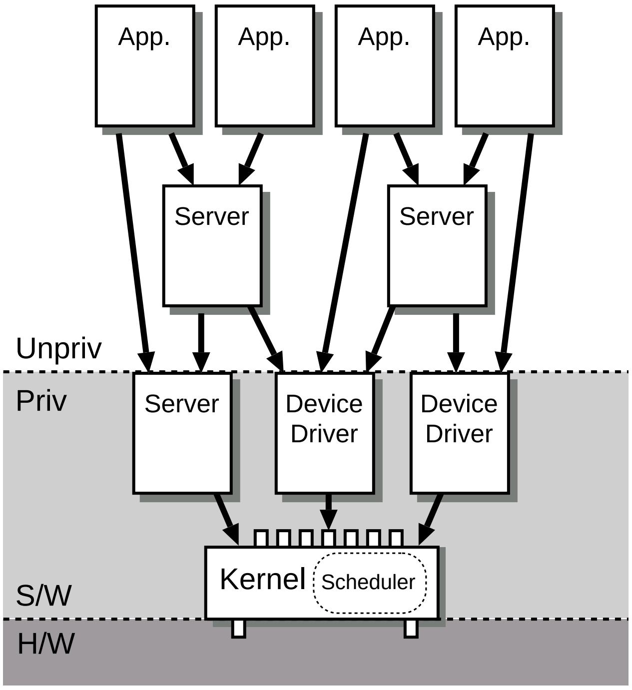
```
   @ ************************
   @ Supervisor Call Dispatch
   @ ************************

   @ NB: A SWI also causes interrupts to be disabled!

_do_swi:
   cmp     r12, #0
   blt     do_user_sem
   stmfd   r13!, {r14}
   ldr     r14, syscallptr          @ r14 <- table base
   and     r12, r12, #0xff          @ Bounds check syscall #
   ldr     r12, [r14, r12, lsl #2] @ Load relevant entry
   mov     r14, r15
   adds    r15, r12, #3             @ Branch to routine +
                                    @ enable ints, svr mode.
   ldr     r1, _cur_thread
   ldr     r1, [r1, #76]            @ Check if thread now
   cmp     r1, #1                   @ marked as dying.
   ldmnefd r13!, {r15}^             @ If not, return.
   b       _sleepy                  @ Else, terminate it.

syscallptr:
    .word _syscalls
```

Microkernel Operating Systems



- Kernel schemes perceived as inflexible ⇒
  - 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:

  ⇒ 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.

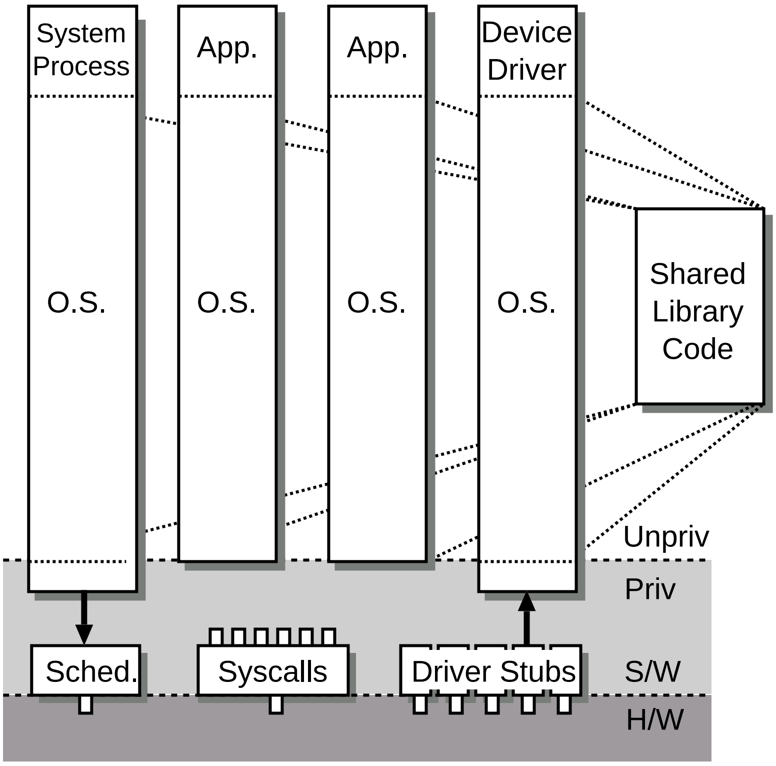
⇒ today most common operating systems blur the distinction between kernel and microkernel.

- 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.

- Hence kernel for performance, but microkernel for extensibility.

Vertically Structured Operating Systems



- Consider interface people really see, e.g.
  - set of programming libraries / objects.
  - a command line interpreter / window system.
- Separate concepts of protection and abstraction ⇒ get extensibility, accountability & performance.
- Examples: Nemesis, Exokernel, Cache Kernel.

Multiprocessor Operating Systems

- Multiprocessor OSs may be roughly classed as either *symmetric* or *asymmetric*.
- Symmetric Operating Systems:
  - identical system image on each processor ⇒ convenient abstraction.
  - all resources directly shared ⇒ high synchronisation cost.
  - typical scheme on SMP (e.g. linux, NT).
- Asymmetric Operating Systems:
  - partition functionality among processors.
  - better scalability (and fault tolerance?)
  - partitioning can be static or dynamic.
  - common on NUMA (e.g. Hive, Hurricane).
- Also get hybrid schemes, e.g. Disco.

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.
- And want to do above *efficiently* and *robustly*.

Virtual processors

Why virtual processors?

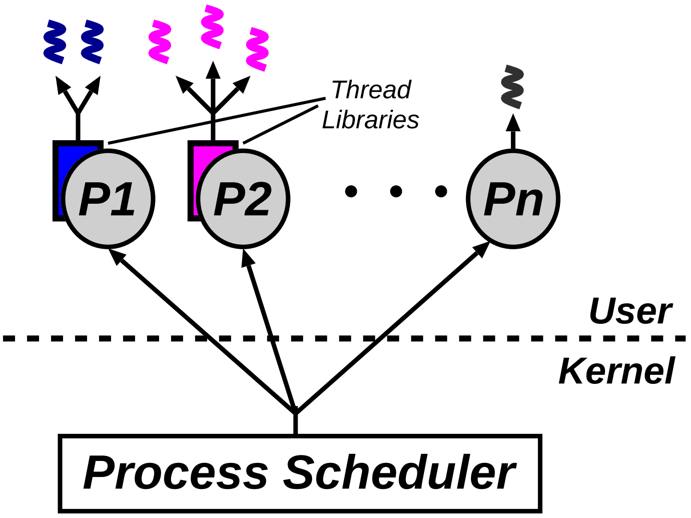
- to provide the illusion that a computer is doing more than one thing at a time;
- to increase system throughput (i.e. run a thread when another is blocked on I/O);
- to encapsulate an execution context;
- to provide a simple programming paradigm.

In modern systems virtual processors are implemented via *processes* and *threads*:

- A process (or task) is a unit of resource ownership — a process is allocated a virtual address space, and control of some resources.
- A thread (or lightweight process) is a unit of dispatching — a thread has an execution state and a set of scheduling parameters.
- In general, have 1 process $\leftrightarrow n$ threads, $n \geq 1$

We may implement threads at *user-level*, at *kernel-level*, or use a *hybrid scheme*.

User-Level Threads



- Kernel unaware of threads’ existence.

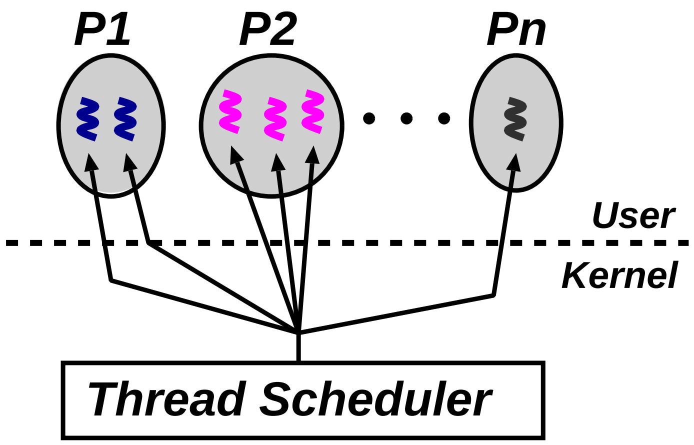
- Thread management done by application using a *thread library*.

- Pros: lightweight creation/termination; fast ctxt switch (no kernel trap); application-specific scheduling; OS independence.

- Cons: non-preemption; blocking system calls; multiple processors.

- e.g. linux `pthreads`

Kernel-Level Threads



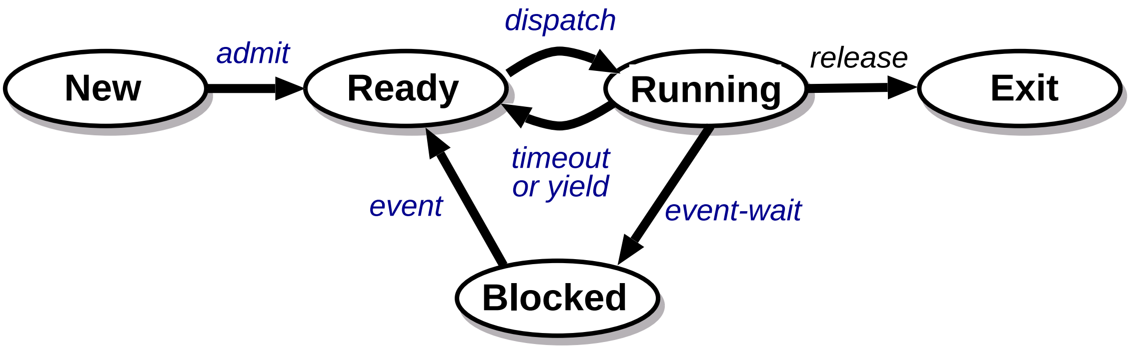
- All thread management done by kernel.
- No thread library (but augmented API).
- Sched two-level, or direct.
- Pros: can utilise multiple processors; blocking system calls just block thread; preemption easy.
- Cons: higher overhead for thread mgt and context switching; less flexible.
- e.g. Windows NT.

Hybrid Schemes

- Three-level scheduling (Solaris 2):
  - 1 kernel thread $\leftrightarrow$ 1 LWP $\leftrightarrow$ n user threads
  - Use ULTs for lightweight operation.
  - Use LWPs to get multiprocessor benefit.
- First class threads (Psyche):
  - Kernel processes implement virtual processor.
  - User-level threads package does *most* but not all thread management.
  - Shared data for user-kernel communication.
  - Kernel *upcalls* threads package on thread block, timer expiration, etc.
- Scheduler activations:
  - Assigned by kernel to processor.
  - Kernel provides space for context, and does context save (but not restore).
  - On CPU allocation or any event, upcall user-level threads package.
  - On block, create new scheduler activation (i.e. keep #scheduler activations constant).
  - In critical sections, kernel does restore.

CPU Scheduling

For now assume a five-state model:



The Operating System must:

- decide if a new thread should be admitted
- wake up blocked threads when appropriate.
- clean up after threads terminate.
- choose amongst runnable thread ⇒ *schedule*

Typical scheduling objectives:

- Maximise CPU utilisation.
- Maximise throughput.
- Minimise average response time.

Also want to minimise overhead (space + time).

VP Data Structures

For each process have a *process control block* (PCB):

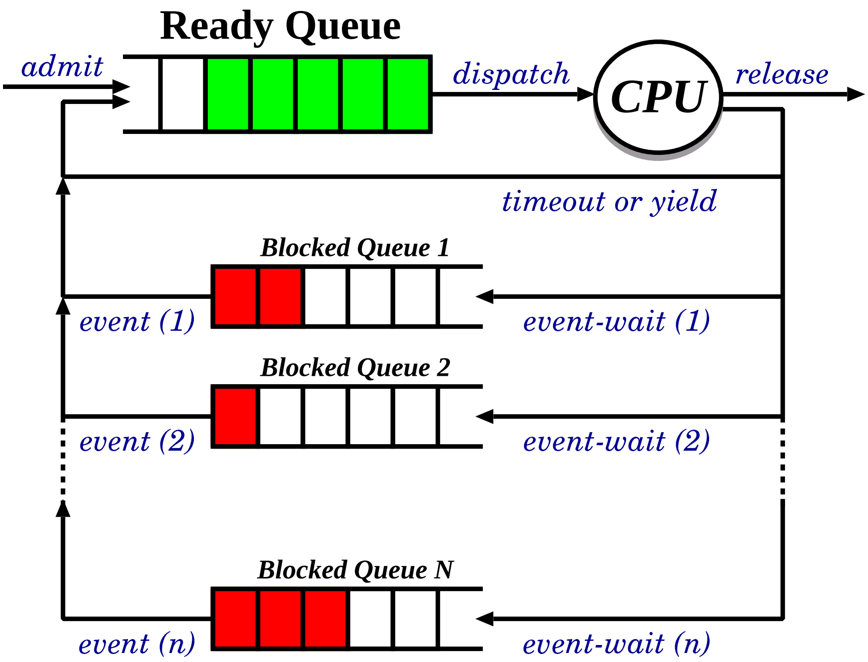
- Identification (e.g. PID, UID, GID)
- Memory management information.
- Accounting information.
- (Refs to) one or more TCBs ...

For each thread have a *thread control block* (TCB):

- Thread state.
- Context slot (perhaps in h/w).
- Refs to user (and kernel?) stack.
- Scheduling parameters (e.g. priority).

The *scheduler* is responsible for managing TCBs.

Scheduler Data Structures



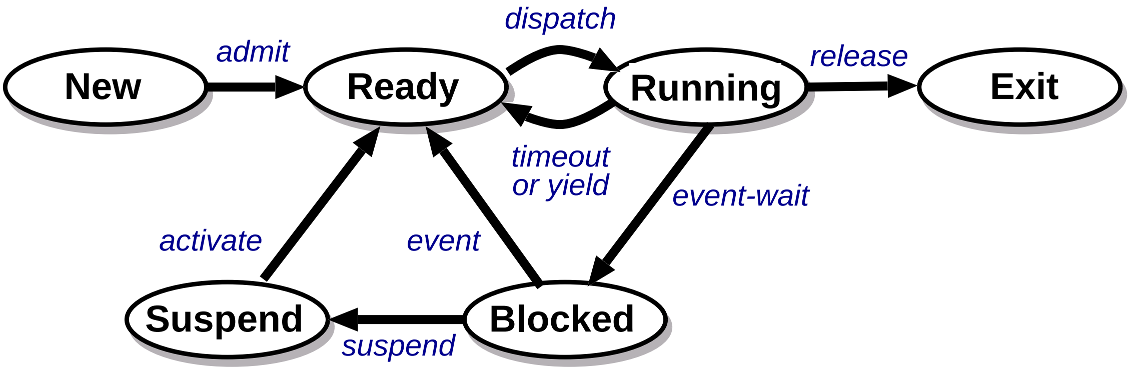
Inside scheduler maintain TCBs according to state:

- Runnable ⇒ "`current_thread`"
- Ready ⇒ on ready queue
- Blocked ⇒ on a blocked queue

Sometimes will have:

- Multiple current threads.
- Multiple ready queues.

The Need for Swapping



- Many OSs constructed using the basic principles described above
- However there is good justification for extending the model:
  - I/O devices are much slower than CPU
- Solution: swap a blocked process out to disk
- Add processes on disk to a *suspend* queue
- Q: how much overhead from additional I/O?
- Q: how to select process to suspend/activate?

When do we schedule?

Can choose a new thread to run when:

1. a running thread blocks (`running` → `blocked`)
2. a timer expires (`running` → `ready`)
3. a waiting thread unblocks (`blocked` → `ready`)
4. a thread terminates (`running` → `exit`)

If only make scheduling decision under 1, 4 ⇒ have a *non-preemptive* scheduler:

✔ simple to implement

✘ open to denial of service

✘ poor priority concept

✘ doesn't extend cleanly to MP

Most modern systems use *preemptive* scheduling:

✔ solves above problems

✘ introduces concurrency problems . . .

Static Priority Scheduling

- All threads are not equal ⇒ associate a *priority* with each, e.g.

  0. interrupt handlers (highest)
  1. device handlers
  2. pager and swapper
  3. other OS daemons
  4. interactive jobs
  5. batch jobs (lowest)

- Scheduling decision simple: just select runnable thread with highest priority.

- Problem: how to resolve ties?
  - round robin with time-slicing
  - allocate quantum to each thread in turn.
  - Problem: biased towards CPU intensive jobs.
    - per-thread 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:
  - threads have a (static) *base priority* and a dynamic *effective priority*.
  - if thread starved for k seconds, increment effective priority.
  - once thread runs, reset effective priority.
- e.g. computed priority:
  - First used in Dijkstra's THE
  - time slots: ... , t, $t + 1$, ...
  - in each time slot t, measure the CPU usage of thread j: u^j
  - priority for thread j in slot $t + 1$: $p^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$
  - e.g. $p^j_{t+1} = p^j_t/2 + ku^j_t$
  - penalises CPU bound → supports I/O bound.
- today such computation considered acceptable ...

Example: 4.3BSD Unix

- Priorities 0–127; user processes $\geq$ `PUSER` $= 50$.
- Round robin within priorities, quantum 100ms.
- Priorities are based on usage and "nice" value:

  $$P_j(i) = Base_j + \frac{CPU_j(i-1)}{nticks} + 2 \times nice_j$$

  gives the priority of process j at the beginning of interval i, where $nice_j \in [-20, 20]$ is a (partially) user controllable parameter.
- i.e. penalizes (recently) CPU bound processes in favour of I/O bound ones.
- $CPU_j(i)$ is incremented every tick in which process j is executing, and decayed each second using:

  $$CPU_j(i) = \frac{2 \times load_j}{(2 \times load_j) + 1} CPU_j(i-1) + nice_j$$
- $load_j(i)$ is the sampled average length of the run queue in which process j resides, over the last minute of operation
- so if e.g. load is 1 $\Rightarrow \sim 90\%$ of 1 seconds CPU usage "forgotten" within 5 seconds.

Example: Windows NT 4.0

- Hybrid static/dynamic priority scheduling:
  - Priorities 16–31: “real time” (static priority).
  - Priorities 1–15: “variable” (dynamic) priority.
- Default quantum 2 ticks ($\sim$20ms) on Workstation, 12 ticks ($\sim$120ms) on Server.
- Threads have *base* and *current* ($\geq$ base) priorities.
  - On return from I/O, current priority is *boosted* by driver-specific amount.
  - Subsequently, current priority decays by 1 after each completed quantum.
  - Also get boost for GUI threads awaiting input: current priority boosted to 14 for one quantum (but quantum also doubled)
  - Yes, this is true.
- On Workstation also get *quantum stretching*:
  - “ ... performance boost for the foreground application” (window with focus)
  - fg thread gets double or triple quantum.
- Later we’ll see another horrible scheduler hack ...

Multiprocessor Scheduling (1)

- Objectives:
  - Ensure all CPUs are kept busy.
  - Allow application-level parallelism.
- Problems:
  - Preemption within critical sections:
    - thread $\mathcal{A}$ preempted while holding spinlock.
    - ⇒ other threads can waste many CPU cycles.
    - Similar situation with producer/consumer threads (i.e. wasted schedule).
  - Cache Pollution:
    - If thread from different application runs on a given CPU, lots of compulsory misses.
    - Generally, scheduling a thread on a new processor is expensive.
  - Frequent context switching:
    - if number of threads greatly exceeds the number of processors, get poor performance.

Multiprocessor Scheduling (2)

Consider basic ways in which one could adapt uniprocessor scheduling techniques:

- Central Queue:
  - ✓ simple extension of uniprocessor case.
  - ✓ load-balancing performed automatically.
  - ✗ n-way mutual exclusion on queue.
  - ✗ inefficient use of caches.
  - ✗ no support for application-level parallelism.
- Dedicated Assignment:
  - ✓ contention reduced to thread creation/exit.
  - ✓ better cache locality.
  - ✗ lose strict priority semantics.
  - ✗ can lead to load imbalance.

Are there better ways?

Multiprocessor Scheduling (3)

- Processor Affinity:
  - modification of central queue.
  - threads have *affinity* for a certain processor ⇒ can reduce cache problems.
  - but: load balance problem again.
  - make dynamic? (cache affinity?)
- 'Take' Scheduling:
  - pseudo-dedicated assignment: idle CPU "takes" task from most loaded.
  - can be implemented cheaply.
  - nice trade-off: load high ⇒ no migration.
- Coscheduling / Gang Scheduling:
  - Simultaneously schedule "related" threads.
  - ⇒ can reduce wasted context switches.
  - Q: how to choose members of gang?
  - Q: what about cache performance?

Example: Mach

- Basic model: dynamic priority with central queue.
- Processors grouped into disjoint *processor sets*:
  - Each processor set has 32 shared ready queues (one for each priority level).
  - Each processor has own local ready queue: absolute priority over global threads.
- Contention-free sharing of
- Quantum inversely proportional to load.
- Applications provide *hints* to improve scheduling:
  1. Discouragement hints: used to reduce penalty for spinlocks, etc.
  2. Handoff hints: improve producer/consumer synchronisation.
- Simple gang scheduling used for allocation.

Real-Time Systems

- Produce correct results **and** meet predefined deadlines.
- "Correctness" of output related to time delay it requires to be produced, e.g.
  - nuclear reactor safety system
  - JIT manufacturing
  - video on demand
- Typically distinguish hard (HRT) and soft real-time (SRT):

  HRT — output value = 100% before the deadline, 0 (or less) after the deadline.

  SRT — output value = 100% before the deadline, (100 - kt)% if t seconds late.
- Building such systems is all about *predictability*.
- It is *not* about speed.

Real-Time Scheduling

- Basic model:
  - consider set of tasks T_i, each of which requires s_i units of CPU time before a (real-time) deadline of d_i.
  - often extended to cope with *periodic* tasks: require s_i units every p_i units.
- Best-effort techniques give no predictability
  - in general priority specifies *what* to schedule but not *when* or *how much*.
  - i.e. CPU allocation for thread t_i, priority p_i depends on all other threads at t_j s.t. $p_j \geq p_i$.
  - with dynamic priority adjustment becomes even more difficult.

⇒ need something different.

Static Offline Scheduling

Advantages:

- Low run-time overhead.
- Deterministic behavior.
- System-wide optimization.
- Resolve dependencies early.
- Can prove system properties.

Disadvantages:

- Inflexibility.
- Low utilisation.
- Potentially large schedule.
- Computationally intensive.

In general, offline scheduling only used when determinism is the overriding factor, e.g. MARS.

Static Priority Algorithms

Most common is Rate Monotonic (RM)

- Assign static priorities to tasks at off-line (or at ‘connection setup’), high-frequency tasks receiving high priorities.
- the tasks processed with no further rearrangement of priorities required ($\Rightarrow$ reduces scheduling overhead).
- optimal, static, priority-driven alg. for preemptive, periodic jobs: i.e. no other static algorithm can schedule a task set that RM cannot schedule.
- Admission control: the schedule calculated by RM is always feasible if the total utilisation of the processor is less than $ln2$
- for many task sets RM produces a feasible schedule for higher utilisation (up to $\sim 88\%$); if periods harmonic, can get 100%.
- Predictable operation during transient overload.

Dynamic Priority Algorithms

Most popular is Earliest Deadline First (EDF):

- Scheduling pretty simple:
  - keep queue of tasks ordered by deadline
  - dispatch the one at the head of the queue.
- EDF is an optimal, dynamic algorithm:
  - It may reschedule periodic tasks in each period
  - If a task set can be scheduled by any priority assignment, it can be scheduled by EDF
- Admission control: EDF produces a feasible schedule whenever processor utilisation is $\leq$ 100%.
- Problem: scheduling overhead can be large.
- Problem: if system overloaded, all bets are off.

Priority Inversion

- All priority-based schemes can potentially suffer from *priority inversion*:

- e.g. consider low, medium and high priority processes called P_l, P_m and P_h respectively.

  1. First P_l admitted, and locks a semaphore $\mathcal{S}$.
  2. Then other two processes enter.
  3. P_h runs since highest priority, tries to lock $\mathcal{S}$ and blocks.
  4. Then P_m gets to run, thus preventing P_l from releasing $\mathcal{S}$, and hence P_h from running.

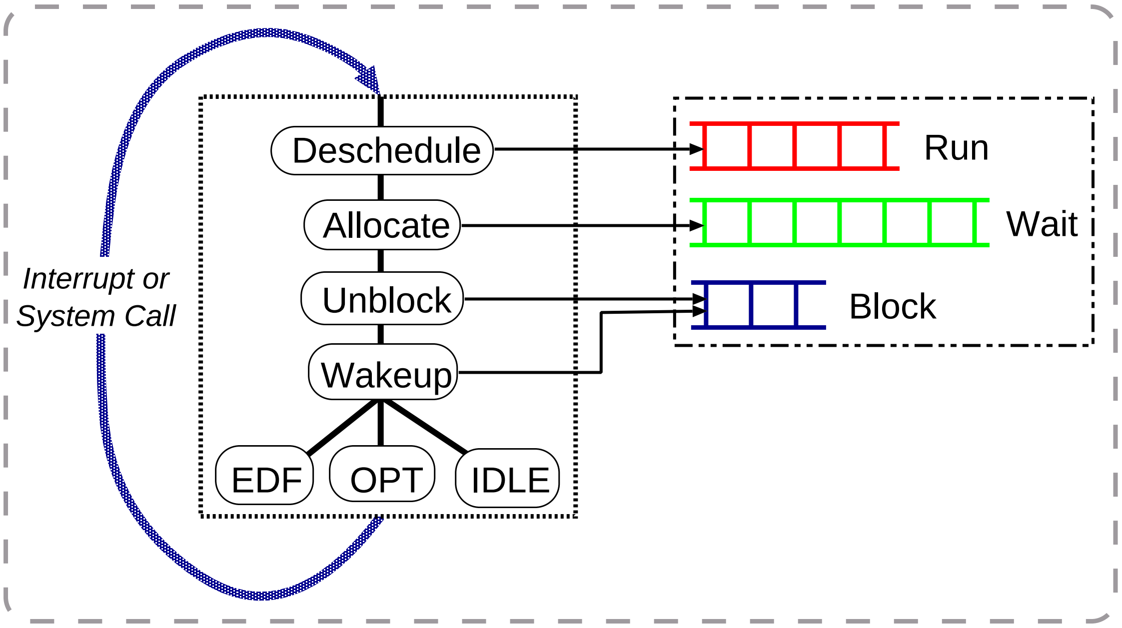
- Usual solution is *priority inheritence*:
  - associate with every semaphore $\mathcal{S}$ the priority P of the highest priority process waiting for it.
  - then temporarily boost priority of *holder* of semaphore up to P.
  - can use handoff scheduling to implement.

- NT “solution”: priority boost for CPU starvation
  - checks if $\exists$ ready thread not run $\geq$ 300 ticks.
  - if so, doubles quantum & boosts priority to 15

Multimedia Scheduling

- Increasing interest in multimedia applications (e.g. video conferencing, mp3 player, 3D games).
- Challenges OS since require presentation (or processing) of data in a timely manner.
- OS needs to provide sufficient *control* so that apps behave well under contention.
- Main technique: exploit SRT scheduling.
- Effective since:
  - The value of multimedia data depends on the timeliness with which it is presented or processed.
  - ⇒ Real-time scheduling allows applications to receive sufficient and timely resource allocation to handle their needs even when the system is under heavy load.
  - Multimedia data streams are often somewhat tolerant of information loss.
  - ⇒ informing applications and providing *soft* guarantees on resources are sufficient.
- Still ongoing research area ...

Example: Atropos (Nemesis)



- use a variant of EDF: QoS maps to (p,s,x)
- expose CPU via activations
- admission control in system domain
- actual scheduling is easy (∼200 lines C)