National University of Singapore
CS2106 Operating System
Midterm Summary Notes

Dong Shaocong A0148008J

March 16, 2018

1 Basic Idea

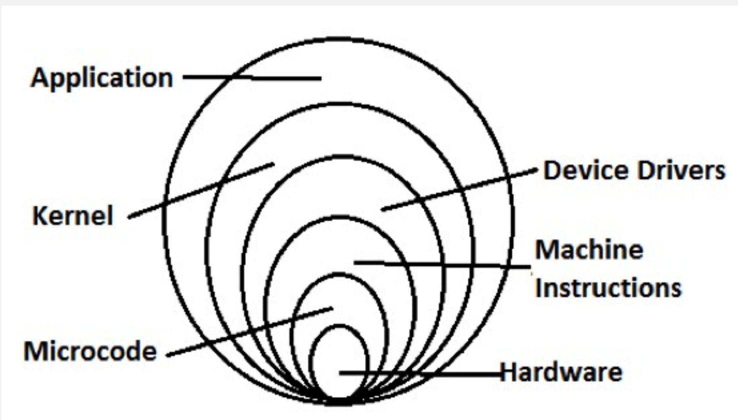
Definition 1.1. Operating System is a suite (i.e. a collection) of specialised software that:

- Gives you access to the hardware devices like disk drives, printers, keyboards and monitors.
- Controls and allocate system resources like memory and processor time.
- Gives you the tools to customise your and tune your system.

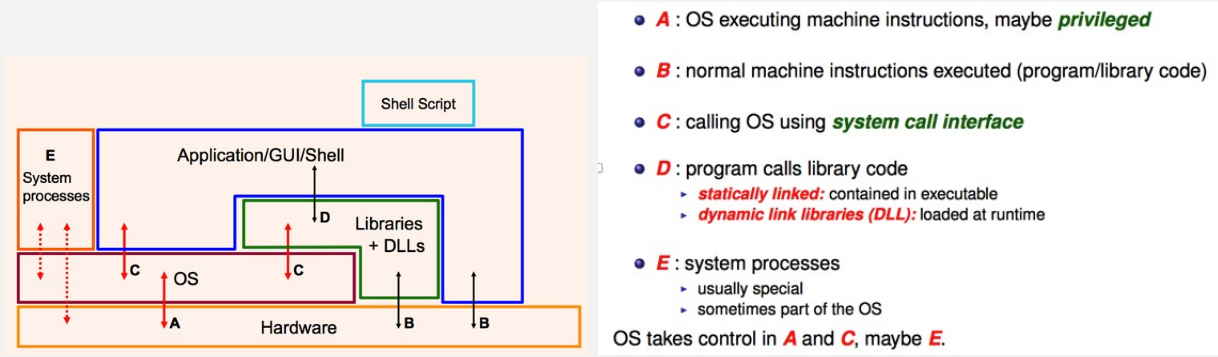
Example 1.1. LINUX, OS X (or MAC OS, a variant of UNIX), Windows 8

What are Operating System? It usually consists of several parts. (Onion Model)

- Bootloader First program run by the system on start-up. Loads remainder of the OS kernel.
  - On Wintel systems this is found in the Master Boot Record (MBR) on the hard disk.
- Kernel The part of the OS that runs almost continuously.
- System Programs Programs provided by the OS to allow:
  - Access to programs.
  - Configuration of the OS.
  - System maintenance, etc.



Abstraction Layer & Opearating System Structure



Definition 1.2. Boostrapping

- The **OS is not present in memory** when a system is cold started.
  - When a system is first started up, memory is completely empty.
- We start first with a **bootloader** to get an operating system into memory.
  - Tiny program in the first (few) sector(s) of the hard-disk.
  - The first sector is generally called the boot sector or master boot record for this reason.
  - Job is to load up the main part of the operating system and start it up.

Definition 1.3. Core CPU units that can execute processes, because we have much more number of processes than the number of cores, we have to do **context switching** to share a core very quickly between different processes.

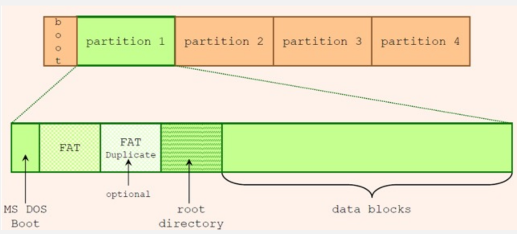
- Entire sharing must be transparent.
- Processes can be suspended and resumed arbitrarily.

Definition 1.4. Context switching

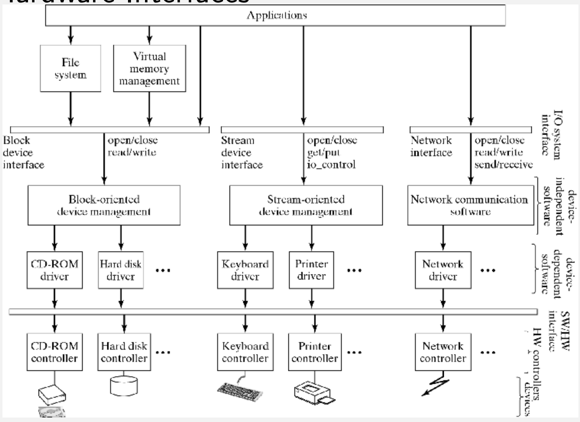
1. Save the context of the process to be suspended.
2. Restore the context of the process to be (re)started.
3. Issues of scheduling to decide which process to run.

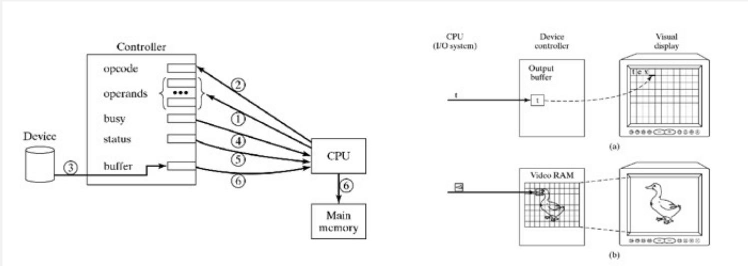
Definition 1.5. File system A set of data structures on disk and within the OS kernel memory to organise persistent data.

How OS file system works?



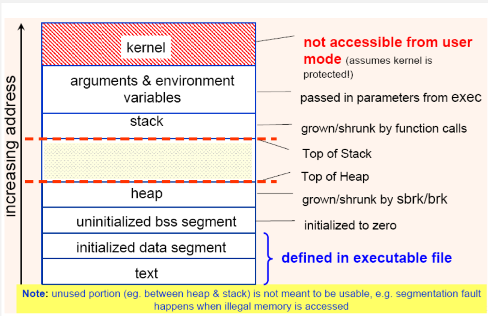
Hardware Interfaces





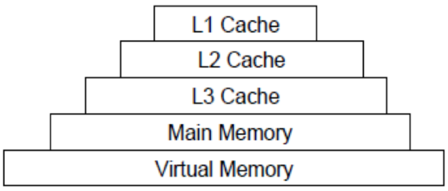
Definition 1.6. Memory static/dynamic (new, delete, malloc, free). Memory to store instructions Memory to store data.

Memory Management



Definition 1.7. Virtual Memory management

- For cost/speed reasons memory is organized in a hierarchy:



- The lowest level is called "virtual memory" and is the slowest but cheapest memory.
  - Actually made using hard-disk space!
  - Allows us to fit much more instructions and data than memory allows!

Definition 1.8. OS security

- Data (files): Encryption techniques, Access control lists
- Resources: Access to the hardware (biometric, passwords, etc), Memory access, File access, etc.

Writing an OS (BSD Unix)

Machine independent

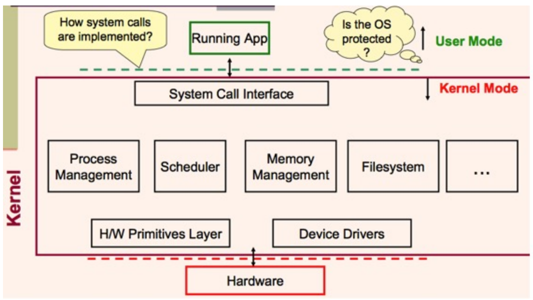
- 162 KLOC
- 80% of kernel
- headers, init, generic interfaces, virtual memory, filesystem, networking+protocols, terminal handling

Machine dependent

- 39 KLOC
- 20% of kernel
- 3 KLOC in asm
- machine dependent headers, device drivers, VM

Definition 1.9. Kernel

- Monolithic Kernel (Linux, MS Windows)
  - All major parts of the OS-devices drivers, file systems, IPC, etc, running in ”kernel space” (an elevated execution mode where certain privileged operations are allowed).
  - Bits and pieces of the kernel can be loaded and unloaded at runtime (e.g. using ”modprobe” in Linux)



- MicroKernel (Mac OS)
  - Only the ”main” part of the kernel is in ”kernel space” (Contains the important stuff like the scheduler, process management, memory management, etc.)
  - The other parts of the kernel operate in ”user space” as system services: The file systems, USB device drivers, Other device drivers.

External View of an OS

- The kernel itself is not very useful. (Provides key functionality, but need a way to access all this functionality.)
- We need other components:
  - System libraries (e.g. stdio, unistd, etc.)
  - System services (creat, read, write, ioctl, sbrk, etc.)
  - OS Configuration (task manager, setup, etc.)
  - System programs (Xcode, vim, etc.)
  - Shells (bash, X-Win, Windows GUI, etc.)
  - Admin tools (User management, disk optimization, etc.)
  - User applications (Word, Chrome, etc).

Definition 1.10. System Calls calls made to the Application Program Interface or API of the OS.

- UNIX and similar OS mostly follow the POSIX standard. (Based on C. Programs become more portable.) *POSIX: portable operating system interface for UNIX, minimal set of system calls for application portability between variants of UNIX.*
- Windows follows the WinAPI standard. (Windows 7 and earlier provide Win32/Win64, based on C. Windows 8 provide Win32/Win64 (based on C) and WinRT (based on C++).)

Example 1.2. User mode + Kernel mode

- Programs (process) run in user mode.
- During system calls, running kernel code in kernel mode.
- After system call, back to user mode.

> How to switch mode? Use privilege mode to switching instructions:
>
> - syscall instruction
> - software interrupt - instruction which raises specific interrupt from software.

Example 1.3. LINUX system call

- User mode: (outside kernel)
  - C function wrapper (eg. **getpid()**) for every system call in C library.
  - assembler code to setup the system call no, arguments
  - trap to kernel
- Kernel mode: (inside kernel)
  - dispatch to correct routine
  - check arguments for errors (eg. invalid argument, invalid address, security violation)
  - do requested service
  - return from kernel trap to user mode
- User mode: (Outside kernel)
  - returns to C wrapper - check for error return values

2 Process Management

Definition 2.1. Program consists of: Machine instructions (and possibly source code) and Data. A program exists as a file on the disk. (e.g. command.exe, MSword.exe)

Definition 2.2. Process consists of Machine instructions (and possibly source code), Data and Context. It exists as instructions and data in memory, **may** be executing on the CPU.

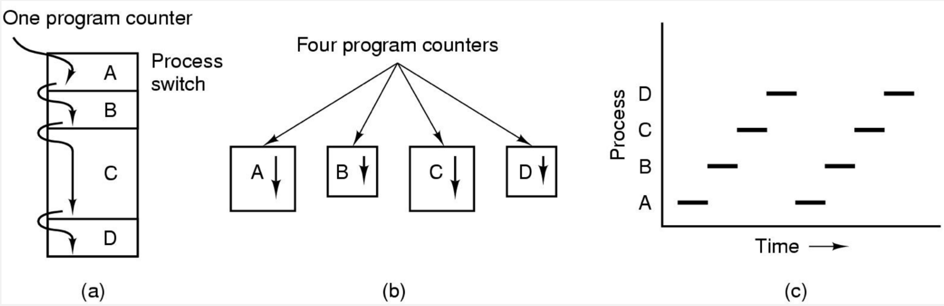
> Program vs. Process
> A single program can produce multiple processes. (e.g. chrome.exe is a single program, but every tab in Chrome is a new process!)

Definition 2.3. Execution Modes

- Programs usually run sequentially. (Each instruction is executed one after the other.)
- Having multiple cores or CPUs allow parallel ("concurrent") execution. (Streams of instructions with no dependencies are allowed to execute together.)
- A multitasking OS allows several programs to run "concurrently". (Interleaving, or time-slicing)

Remark. we mostly assume number of processes $\geq$ number of CPU otherwise can have idle tasks. So each core must still switch between processes even for multi-cores, and we will assume a single processor with a single core.

The Process Model



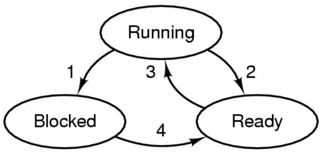
- Figure (b) shows what appears to be happening in a single processor system running multiple processes:
  - There are 4 processes each with its own program counter (PC) and registers.
  - All 4 processes run independently of each other at the same time.
- Figure (a) shows what actually happens.
  - There is only a single PC and a single set of registers.
  - When one process ends, there is a "context switch" or "process switch":
    - PC, all registers and other process data for Process A is copied to memory.
    - PC, register and process data for Process B is loaded and B starts executing, etc.
- Figure (c) illustrates how processes A to D share CPU time.

Definition 2.4. Process States there are three possible states for a process

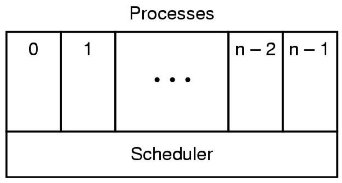
- Running
  - The process is actually being executed on the CPU.
- Ready
  - The process is ready to run but not currently running.
  - A "scheduling algorithm" is used to pick the next process for running.
- Blocked.
  - The process is waiting for "something" to happen so it is not ready to run yet. e.g. include waiting for inputs from another process.

Definition 2.5. Process Context (values change as a process runs)

- CPU register values.
- Stack pointers.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available



- CPU Status Word Register
  - This maintains information about whether the previous instruction resulted in an overflow or a ”zero”, whether interrupts are enabled, etc.
  - This is needed for branch instructions assembly equivalents of ”if” statements.

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

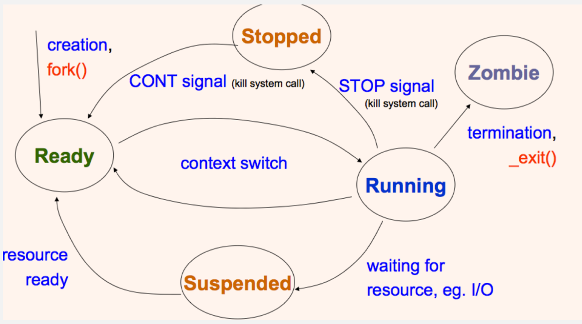
Example 2.1. Context Switching in FreeRTOS Atmega Port FreeRTOS relies on regular interrupts from Timer 0 to switch between tasks. When the interrupt triggers:

1. PC is placed onto Task As stack.
2. The ISR calls portSAVECONTEXT, resulting in Task As context being pushed onto the stack.
3. pxCurrentTCB will also hold SPH/SPL after the context save.
   - This must be saved by the kernel.
   - The kernel stores a copy of the stack pointer for each task.
4. The kernel then selects Task B to run, and copies its SPH/SPL values into pxCurrentTCB and calls portRESTORE_CONTEXT.
5. The rest of portRESTORE_CONTEXT is executed, causing Task Bs data to be loaded into R31-R0 and SREG. Now Task B can resume like as though nothing happened
6. Only Task Bs PC remains on the stack. Now the ISR exits, causing this value to be popped off onto the AVRs PC.
   - PC points to the next instruction to be executed.
   - End result: Task B resumes execution, with all its data and SREG intact!

How can context switching be triggered?
It can be triggered by a timer; currently running process waiting for input; currently running task blocking on a synchronisation mechanism; currently running task wants to sleep for a fixed period; higher priority task becoming READY; . . .

Definition 2.6. Process Control Block maintains information about that process: Process ID (PID), Stack Pointer, Open files, Pending signals, CPU usage, . . .

Process Life Cycle



Definition 2.7. Creating a new process - fork()

- Fork system call creates a new process by duplicating the current image into a new process, *child process*
- same code (executable image) is executed
- Child differs only in process id (PID) and parent (PPID), fork return value
- Data in child is a COPY of the parent (i.e. not shared)
- In PARENT process after fork:
  - PC is at return from fork system call
  - fork return value: new child PID
- In CHILD process after fork:
  - PC is at return from fork system call
  - fork return value: 0
  - Shares open file & signal handlers with parent, current working directory
  - Independent copy of: memory, arguments, environment variables (note: cloning example)
- fork return result is -1 if the fork failed.

Definition 2.8. The Master Process

- Every process has parent: where does it stop?
- Special initial process - init process created in kernel at the end UNIX boot process, traditionally having PID=1.
- Forking creates process tree, init is the root process.
- init watches for processes and response where needed, e.g. terminal login.
- init also manages system run levels (e.g. shutdown, power failure, single-user mode), etc. Example of a system-like process running in kernel mode.

Definition 2.9. Start/Stop a Process

- kill() system call sends signal to process
- Special process signals:
  - stopping process (SIGSTOP)
  - killing process (SIGKILL)
  - restart stopped process (SIGCONT)

Terminating a process

- system call: `void _exit(int status)`
- _exit system call used for immediate voluntary termination of process (never returns!).
- Closes all open file descriptors; children processes are inherited by init process;
- parent sent `SIGCHLD` signal (see later section on signals & IPC)
- status returned to parent using wait()
- Usually status is used to indicate errors, eg. convention is
  - _exit(0) for success, 0 means no error
  - _exit(1) for error, positive number for error number

- Process finished execution
- can release **most** system resources used by process are released on exit
- **BUT** some basic process resources not releasable: PID & status needed when `wait()` is called. Also process accounting info, e.g. cpu time. Note: may mean that process table entry still being used

Notes: See `wait()`, zombies

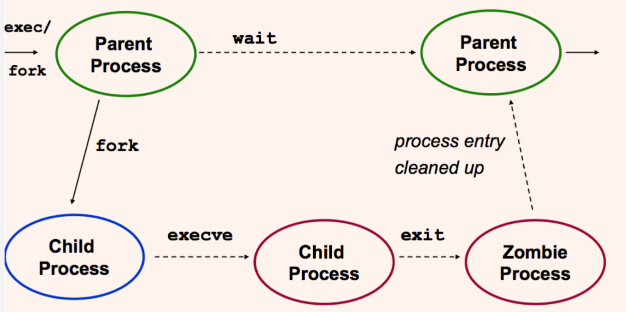
Definition 2.10. Normal Program Termination - void exit(int status) from standard C library function

- Usuall don't use _exit() but exit(), which cleans up: open streams from C stdio library (e.g., fopen, printf) are flushed and closed
- calls some exit handlers
- finally calls _exit(status) after all standard C cleanup done.

Remark. returning from main() implicitly calls exit. exec didn't actually call main directly but a startup routine. Open files also get flushed automatically.

Waiting for Child Processes to Terminate - process interaction

- system call: `pid_t wait(int *status)`
- Parent can wait until some child processes terminates (calls `exit`)
- Note: also some other conditions
- `status` gives child exit status
- cleans up remainder of child system resources – ones not removed when exit!
- Other versions of wait: (some are non blocking!) `waitpid(), wait3(), wait4()`



Note: example uses one ordering of execution, others are possible!

Zombie Process

- process is "dead" / terminated
- Recall: `wait()` means that process termination is not complete when it exits
- process goes to **zombie state**: remainder of process data structure **cleaned up** when `wait()` happens (if it happens!)
- **can't delete** process since dont know if wait from parent needs exited process info! (so it's a consequence of having a wait() operation defined!)
- **cannot kill** zombie since already exited!

2 cases:

- Parent dies before child, the `init` process becomes "pseudo" parent of child processes. Child dying sends signal to `init` process, which organizes to call `wait()` to cleanup process
- Child dies before parent but parent didn't call `wait`. Becomes a zombie process. Can fillup process table requiring reboot. Unix SVR4 can specify no zombie creation on termination
- modern Unixes have mechanisms to avoid zombies

3 Process Scheduling

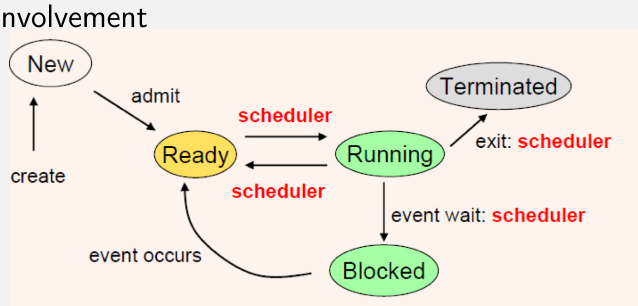
Definition 3.1. Computer jobs Computatings + reading/writing memory + input/output.

- CPU bound

- Most of the time spent on processing on CPU
- Graphics-intensive applications are considered to be "CPU" bound.
- Multitasking opportunities come from having to wait for processing results.

- I/O bound
  - Most of the time is spent on communicating with I/O devices
  - Multitasking opportunities come from having to wait for data from I/O devices.

Process states - scheduler involvement



Definition 3.2. Types of Multitaskers

- Batch Processing (Not actually multitasking since only one process runs at a time to completion.)
- Co-operative Multitasking (Currently running processes cannot be suspended by the scheduler; Processes must volunteer to give up CPU time.)
- Pre-emptive Multitasking (Currently running processes can be forcefully suspended by the scheduler.)
- Real-Time Multitasking (Processes have fixed deadlines that must be met.)
  - Hard Real Time Systems: Disaster strikes! System fails, possibly catastrophically!
  - Soft Real Time Systems: Mostly just an inconvenience. Performance of system is degraded

Remark. Policies are determined by the kind of multitasking environment.

Definition 3.3. Scheduling Policies - enforce a priority ordering over processes

- **Fixed Priority** for all kinds of multitaskers
  - Each task is assigned a priority by the programmer. (usually priority number 0 has the highest priority.)
  - Tasks are queued according to priority number.
  - Batch, Co-operative: Task with highest priority is picked to be run next.
  - Pre-emptive, Real-Time: When a higher priority task becomes ready, current task is suspended and higher priority task is run.
- Policies for **Batch Processing**
  - First-come First Served (FCFS)
    * Arriving jobs are stored in a queue.
    * Jobs are removed in turn and run.
    * Particularly suited for batch systems.
    * Extension for interactive systems: Jobs removed for running are put back into the back of the queue. This is also known as round-robin scheduling.

    - Starvation free as long as earlier jobs are bounded.
  - Shortest Job First (SJF)
    - Processes are ordered by total CPU time used.
    - Jobs that run for less time will run first.
    - Reduces average waiting time if number of processes is fixed.
    - Potential for starvation.
- Policies for **Co-operative Multitasking**
  - Round Robin with Voluntary Scheduling (VS)
  - **Voluntary Scheduling**: Processes call a special yield function. This invokes the scheduler. Causes the process to be suspended and another process started up.
- Policies for **Pre-emptive multitasking**
  - Round Robin with Timer (RR)
    - Each process is given a fixed time slot c_i.
    - After time c_i, scheduler is invoked and next task is selected on a round-robin basis.
  - Shortest Remaining Time (SRT)
    - Pre-emptive form of SJF.
    - Processes are ordered according to remaining CPU time left.
- Policies for **Real-Time Multitaskers**
  - Rate Monotonic Scheduling (RMS)
    - Processes are prioritized according to P_i, Shortest period = highest priority.
    - **Critical Instance Analysis** is used to test that all processes meet their deadlines
  - Earliest Deadline First Scheduling (EDF)
    - Processes are prioritized according to who is closest to their respect deadlines.
    - All processes are guaranteed to meet their deadlines as long as: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$
    - There is no context switching for processes, each process will run till the end if gets started.

Remark. Real Time Scheduling must guarantee that processes complete within time limits.

- Time limits are called deadlines
- Processes are assumed to be periodic with period P_i for process i.
- Processes are assumed to use a fixed amount of CPU time C_i each time.
- Deadline D_i is assumed to be the same as period P_i. (if a process runs at time T_i, it must finish running by $D_i = T_i + P_i$, If it doesnt, process has missed its deadline.) T_i is the process or task arriving time.

Definition 3.4. Critical Instance Analysis for RMS

1. Sort T by period of each task, if T is not already sorted. (We will assume that T_1 has the shortest period, T_2 has the 2^{nd} shortest, etc.)
2. For each task $T_i \in T$, recursively compute $S_{i0}, S_{i1}, \ldots$ where:

$$S_{i,0} = \sum_{j=1}^{i} C_j ; S_{i,(x+1)} = C_i + \sum_{j=1}^{i-1} C_j \times \lceil \frac{S_{i,x}}{P_j} \rceil$$

3. Stop when $S_{i,(x+1)} = S_{i,x}$. Call this $S_{i,(x+1)}$ the final value $S_{i,F}$ (termination value).
4. If $S_{i,F} < D_i (= T_i + P_i)$, then the task i is schedulable and will not miss its deadlines.

Example 3.1. Managing Multiple Policies Multiple policies can be implemented on the same machine using multiple queues:

- Each queue can have its own policy.
- This scheme is used in Linux, as we will see shortly.

Example 3.2. Scheduling in Linux

- Processes in Linux are dynamic:
  - New processes can be created with `fork()`
  - Existing processes can exit.
- Priorities are also dynamic:
  - Users and superusers can change priorities using "nice" values.
  - `nice n 19 tar cvzf archive.tgz *` (Allows tar to run with a priority lowered by 19 to reduce CPU load. Normal users can only $0 \leq n \leq 19$. Superusers can specify $-20 \leq n \leq 19$. Negative nice increases priority.)
- Linux maintains **three types of processes**:
  - **Real-time FIFO**: RT-FIFO processes cannot be pre-empted except by a higher priority RT-FIFO process.
  - **Real-time Round-Robin**: Like RT-FIFO but processes are pre-empted after a time slice.
  - Linux only has "soft real-time" scheduling. (Priority levels 0 to 99) Cannot guarantee deadlines, unlike RMS and EDF.
  - Non-real time processes (Priority levels 100 to 139)
- Linux maintains 280 queues in two sets of 140: An active set, an expired set.
- The scheduler is called at a rate of 1000 Hz. (e.g. time tick is 1 ms, called a "jiffy".) RT-FIFO processes are **always** run if any are available. Otherwise:
  - Scheduler picks highest priority process in active set to run.
  - When its time quantum is expired, it is moved to the expired set. Next highest priority process is picked.
  - When active set is empty, active and expired pointers are swapped. Active set becomes expired set and vice versa.
  - Scheme ensures no starvation of lowest priority processes.

> What happens if a process becomes blocked? (e.g. on I/O)
>
> - CPU time used so far is recorded. Process is moved to a queue of blocked processes.
> - When process becomes runnable again, it continues running until its time quantum is expired.
> - It is then moved to the expired set.
> - When a process becomes blocked its priority is often upgraded (see later).

- Time quantums for RR processes: Varies by priority. For example: Priority level 100 800 ms, Priority level 139 5 ms, System load
- How process priorities are calculated: Priority = base + f(nice) + g(cpu usage estimate)
  - $f(.)$ = priority adjustment from nice value.
  - $g(.)$ = Decay function. Processes that have already consumed a lot of CPU time are downgraded.
  - Other heuristics are used: Age of process, More priority for processes waiting for I/O - I/O boost, Bias towards foreground tasks.
- **I/O Boost**
  - Tasks doing read() has been waiting for a long time. May need quick response when ready.
  - Blocked/waiting processes have not run much.
  - Applies also to interactive processes blocked on keyboard/mouse input.

> How long does this boost last?
>
> - Temporary boost for sporadic I/O
> - Permanent boost for the chronically I/O bound?
> - E.g. Linux gives -5 boost for interactive processes.
> - Implementation: We can boost time quantum, boost priority, do both.

4 Inter-Process Communication

Definition 4.1. Race Condition occur when two or more processes attempt to access shared storage. This causes the final outcome to depend on who runs first. "Shared storage" can mean:

- Global variables.
- Memory locations.
- Hardware registers.- This refers to configuration registers rather than CPU registers.
- Files.

Definition 4.2. Critical Sections mutual exclusion - mutex
a RUNNING process is always in one of two possible "states":

- It is performing local computation. This does not involve global storage, hence to race condition is possible.
- It is reading/updating global variables. This can lead to race conditions. (it is within its "critical section")

Theorem 4.1. FOUR rules to prevent race conditions

1. No two processes can simultaneously be in their critical section.
2. No assumptions may be made about speeds or number of CPUs.
   - Note: We can relax this assumption for most embedded systems since they have single CPUs.
   - May apply to systems using multicore micro-controllers.
3. No process outside of its critical section can block other processes.
4. No process should wait forever to enter its critical section.

Example 4.1. Mutual Exclusion Implementation

- **Disabling Interrupts**
  - disabling interrupts will prevent other processes from starting up and entering their critical sections.
  - Carelessly disabling interrupts can cause the entire system to grind to a halt.
  - This only works on single-processor, single core systems. Violates Rule 2.
- Using **Lock Variables**
  - A single global variable lock is initially 1.
  - Process A reads this variable and sets it to 0, and enters its critical section.
  - Process B reads lock and sees its a 0. It doesnt enter critical section and waits until lock is 1.
  - Process A finishes and sets lock to 1, allowing B to enter
  - PROBLEM: Theres a race condition on lock itself.
  - **Test and Set Lock** (TSL)
    * CPU locks the address and data buses, and reads "lock" from memory. The locked address and data buses will block accesses from all other CPUs. ("atomic". This means that NOTHING can interrupt execution of this instruction. This is guaranteed in hardware.)
    * The current value is written into register "reg".
    * A "1" (or sometimes "0") value is written to "lock".
    * CPU unlocks the address and data buses.
    * ALTERNATIVE: the XCHG instruction, used on Intel machines. Swaps contents of "lock" and "reg" instead of just writing "1" to lock.

> Busy-wait approaches like Peterson and TSL/XCHG have a problem called **deadlock**. Consider two processes H and L, and a scheduler rule that says that H is always run when it is READY. Suppose L is currently in the critical region.
>
> 1. H becomes ready, and L is pre-empted.
> 2. H tries to obtain a lock, but cannot because L is in the critical region.
> 3. H loops forever, and CPU control never gets handed to L.
> 4. As a result L never releases the lock.

- **Sleep/Wake**
  - When a process finds that a lock has been set (i.e. another process in the critical section), it calls "sleep" and is put into the blocked state.
  - When the other process exits the critical section and clears the lock, it can call "wake" which moves the blocked process into the READY queue for eventual execution.
  - producer-consumer problem: Deadlock occurs when:
    1. Consumer checks "count" and finds it is 0.
    2. Consumer gets pre-empted and producer starts up.
    3. Producer adds an item, increments count to "1", then sends a WAKE to the consumer. (Since consumer is not technically sleeping yet, the WAKE is lost.)

4. Consumer starts up, and since count is 0, goes to SLEEP.
5. Producer starts up, fills buffer until it is full and SLEEPs.
6. Since consumer is also SLEEPing, no one wakes the producer. Deadlock.

- **Semaphores**, a special lock variable that counts the number of wake-ups saved for future use.
  - A value of 0 indicates that no wake-ups have been saved.
  - Two ATOMIC operations on semaphores:
    * DOWN, TAKE, PEND or P: If the semaphore has a value of > 0, it is decremented and the DOWN operation returns. If the semaphore is 0, the DOWN operation blocks.
    * UP, POST, GIVE or V: If there are any processes blocking on a DOWN, one is selected and woken up. Otherwise UP increments the semaphore and returns.
  - When a semaphores counting ability is not needed, we can use a simplified version called a mutex. (1 = Unlocked. 0 = Locked.)
  - non_critical_section() $\rightarrow$ DOWN(sema) $\rightarrow$ critical_section() $\rightarrow$ UP(sema)
  - We can also implement mutexes with TSL or XCHG. 0 = Unlocked, 1 = Locked

```
mutex_lock:
      TSL REGISTER,MUTEX             | copy mutex to register and set mutex to 1
      CMP REGISTER,#0                | was mutex zero?
      JZE ok                         | if it was zero, mutex was unlocked, so return
      CALL thread_yield              | mutex is busy; schedule another thread
      JMP mutex_lock                 | try again later
ok:   RET | return to caller; critical region entered


mutex_unlock:
      MOVE MUTEX,#0                  | store a 0 in mutex
      RET | return to caller
```

Problems with Semaphores Deadlock

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                    /* TRUE is the constant 1 */
          item = produce_item( );      /* generate something to put in buffer */
          down(&empty);                /* decrement empty count */
          down(&mutex);                /* enter critical region */
          insert_item(item);           /* put new item in buffer */
          up(&mutex);                  /* leave critical region */
          up(&full);                   /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                    /* infinite loop */
          down(&full);                 /* decrement full count */
          down(&mutex);                /* enter critical region */
          item = remove_item( );       /* take item from buffer */
          up(&mutex);                  /* leave critical region */
          up(&empty);                  /* increment count of empty slots */
          consume_item(item);          /* do something with the item */
     }
}
```

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                    /* TRUE is the constant 1 */
          item = produce_item( );      /* generate something to put in buffer */
          down(&mutex);                /* decrement empty count */
          down(&empty);                /* enter critical region */
          insert_item(item);           /* put new item in buffer */
          up(&mutex);                  /* leave critical region */
          up(&full);                   /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                    /* infinite loop */
          down(&mutex);                /* decrement full count */
          down(&full);                 /* enter critical region */
          item = remove_item( );       /* take item from buffer */
          up(&mutex);                  /* leave critical region */
          up(&empty);                  /* increment count of empty slots */
          consume_item(item);          /* do something with the item */
     }
}
```

  - Producer successfully DOWNs the mutex.
  - Producer DOWNs empty. However the queue is full so this blocks.

- Consumer DOWNs mutex and blocks.
    * Consumer now never reaches the UP for empty and therefore cannot unblock the producer.
    * The producer in turn never reaches the UP for mutex and cannot unblock the consumer. Deadlock!