

National University of Singapore

CS2106 Operating System

Midterm Summary Notes

Dong Shaocong A0148008J

March 22, 2018

1 Basic Idea

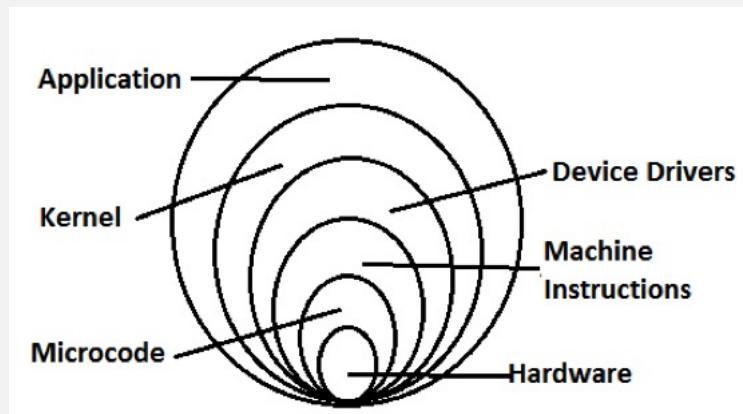
Definition 1.1. **Operating System** is a suite (i.e. a collection) of specialised software that:

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

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

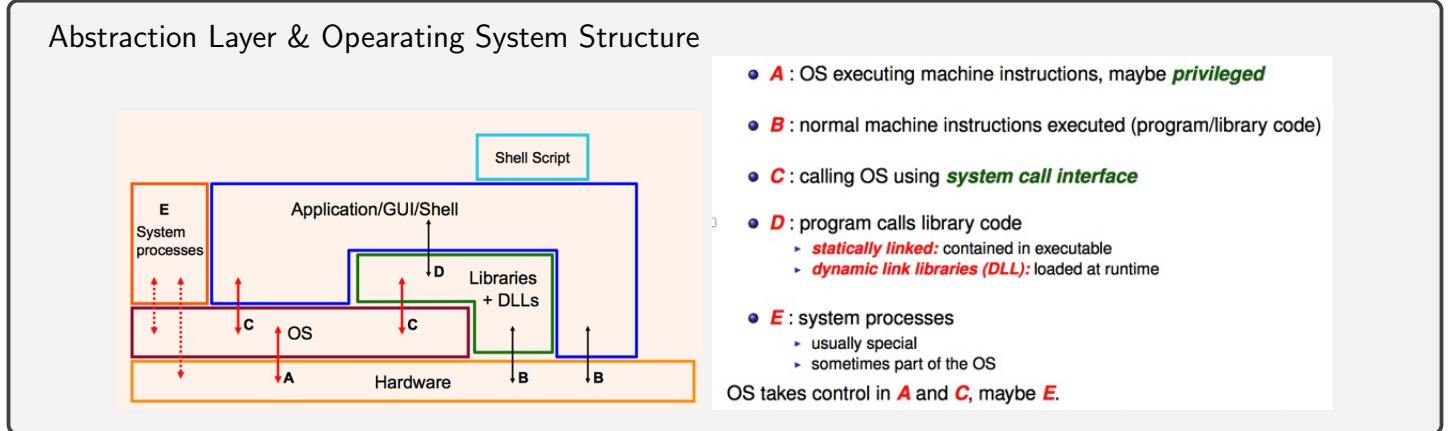
What are Operating System? It usually consists of several parts. (Onion Model)

- 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.



Definition 1.2. Micro-coding CPU designers implement a set of basic operations directly in hardware, then create a "microcode" a language that uses these operations to create complex machine instructions.

(Reason: some instructions are too complex to do properly in hardware (e.g. the Intel string and block operations).)



Definition 1.3. Bootstrapping

- 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.

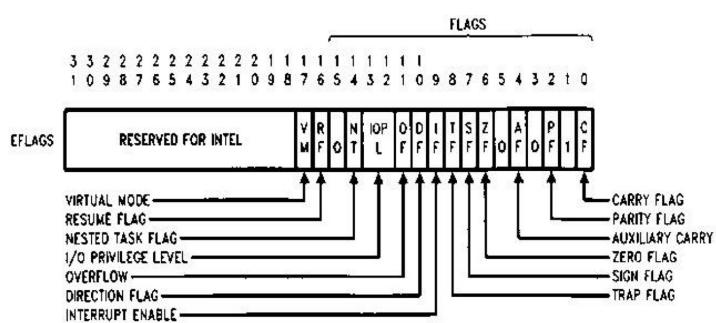
Definition 1.4. Core CPU units that can execute processes, because we have much more number of processes than the number of cores, we have to do **context switching** to share a core very quickly between different processes.

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

Definition 1.5. 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 1.6. special register Machine Status Word (MSW) or Status Register (SREG)



We can see that it contains flags that tell us the results of a previous arithmetic operation. E.g. Zero (ZF) tells us if a subtract resulted in a 0, Sign (SF) tells us if it resulted in a negative number. The Carry

flag (CF) tells us if an addition resulted in a carry, the Overflow flag (OF) tells us if an overflow resulted (which means the results could be invalid), etc.

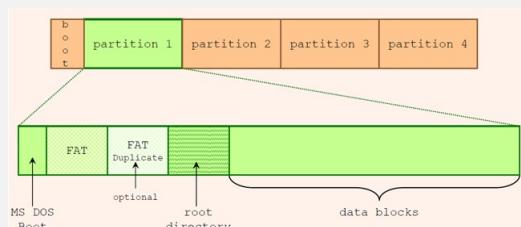
The ZF and SF are necessary for branch instructions. E.g. for a branch on less than (BLT), the ALU performs a subtraction, and the branch is taken if SF is set. Similarly for a BEQ, the branch is taken if ZF is set, etc.

The MSW also contains configuration flags, like the Interrupt Enable (IF) flag that enables or disables maskable interrupts (special signals that I/O hardware can use to get the CPU's attention essentially this flag tells the CPU whether to entertain or ignore such requests).

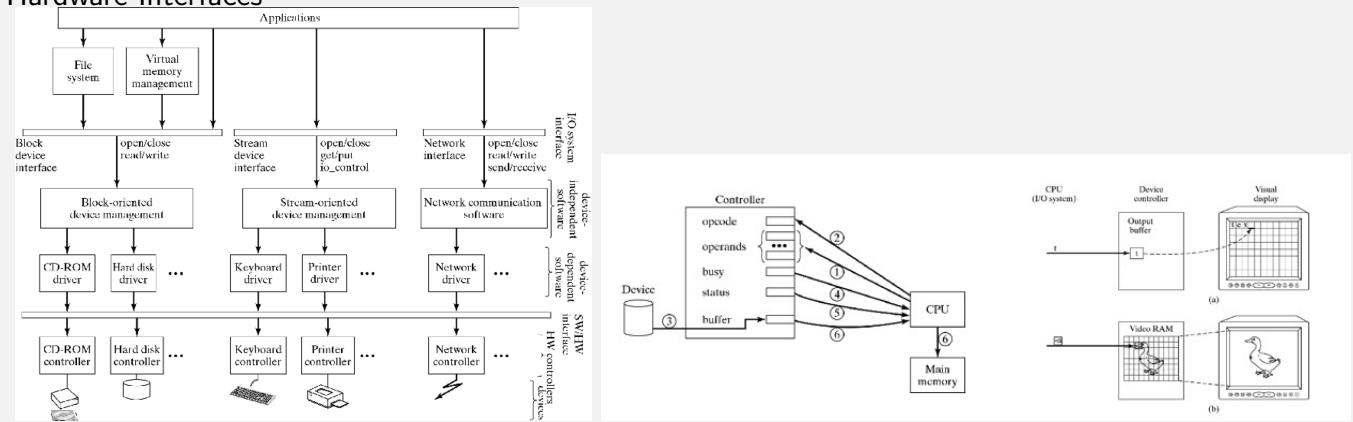
For correctness, MSW has to be stored during context save and restored during context restore.

Definition 1.7. File system A set of data structures on disk and within the OS kernel memory to organise persistent data.

How OS file system works?

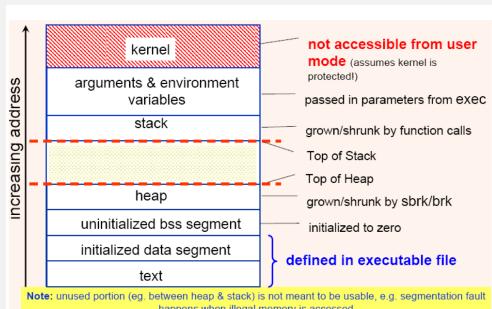


Hardware Interfaces

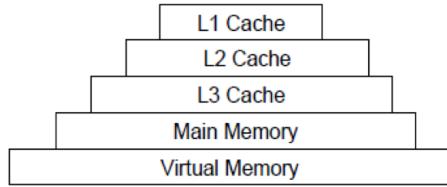


Definition 1.8. Memory static/dynamic (new, delete, malloc, free). Memory to store instructions Memory to store data.

Memory Management



Definition 1.9. 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 1.10. OS security

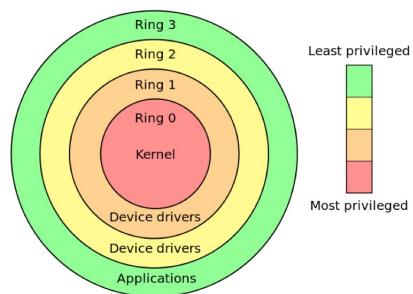
- Data (files): Encryption techniques, Access control lists
- Resources: Access to the hardware (biometric, passwords, etc), Memory access, File access, etc.

Writing an OS (BSD Unix)

<p>Machine independent</p> <ul style="list-style-type: none"> • 162 KLOC • 80% of kernel • headers, init, generic interfaces, virtual memory, filesystem, networking+protocols, terminal handling <p>Machine dependent</p> <ul style="list-style-type: none"> • 39 KLOC • 20% of kernel • 3 KLOC in asm • machine dependent headers, device drivers, VM 		
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--	--

Example 1.2. Privilege Levels used in Intel CPUs

This diagram is useful to understand privilege rings:

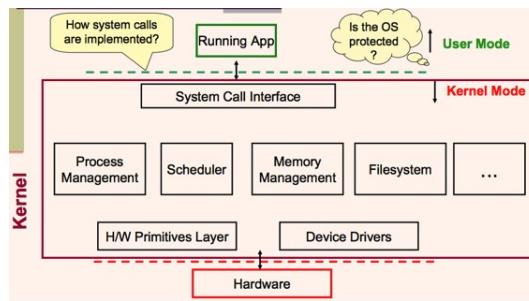


processes running in lower (outer) rings have more restrictive access to the machine than processes in the higher (inner) rings, to prevent a user, for example, from erasing the entire system drive.

why the highest priority task can still be interrupted by the lowest priority interrupt, and how this affects ISR design. (Hint: Interrupts are implemented in hardware in the CPU itself). This is because task priorities are seen only by the OS. As far as the CPU is concerned, it is just another program that is running, with instructions being fetched and executed. On the other hand interrupt lines are checked at the end of each instruction execution cycle, so interrupts are always serviced regardless of their priority level (and of the priority level of the running task).

Definition 1.11. 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 1.12. 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 1.3. 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 1.4. 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

Example 1.5. UNIX signal: SIGTERM or SIGKILL

- SIGTERM (triggered using `kill <process id>`):
 - The OS receives the SIGTERM request and passes it to the process.
 - The process receives this signal and can clean up and release resources it is using.
 - If the process has child processes, it will terminate the child processes also using a SIGTERM.
 - The process exits gracefully.
- SIGKILL (triggered using `kill -9 <process id>`):
 - Process is terminated immediately by init (the UNIX master process). SIGKILL is not passed to the process, and the process does not have any chance to do cleaning up.
 - SIGKILL can create zombie processes particularly if the killed process has children.

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.

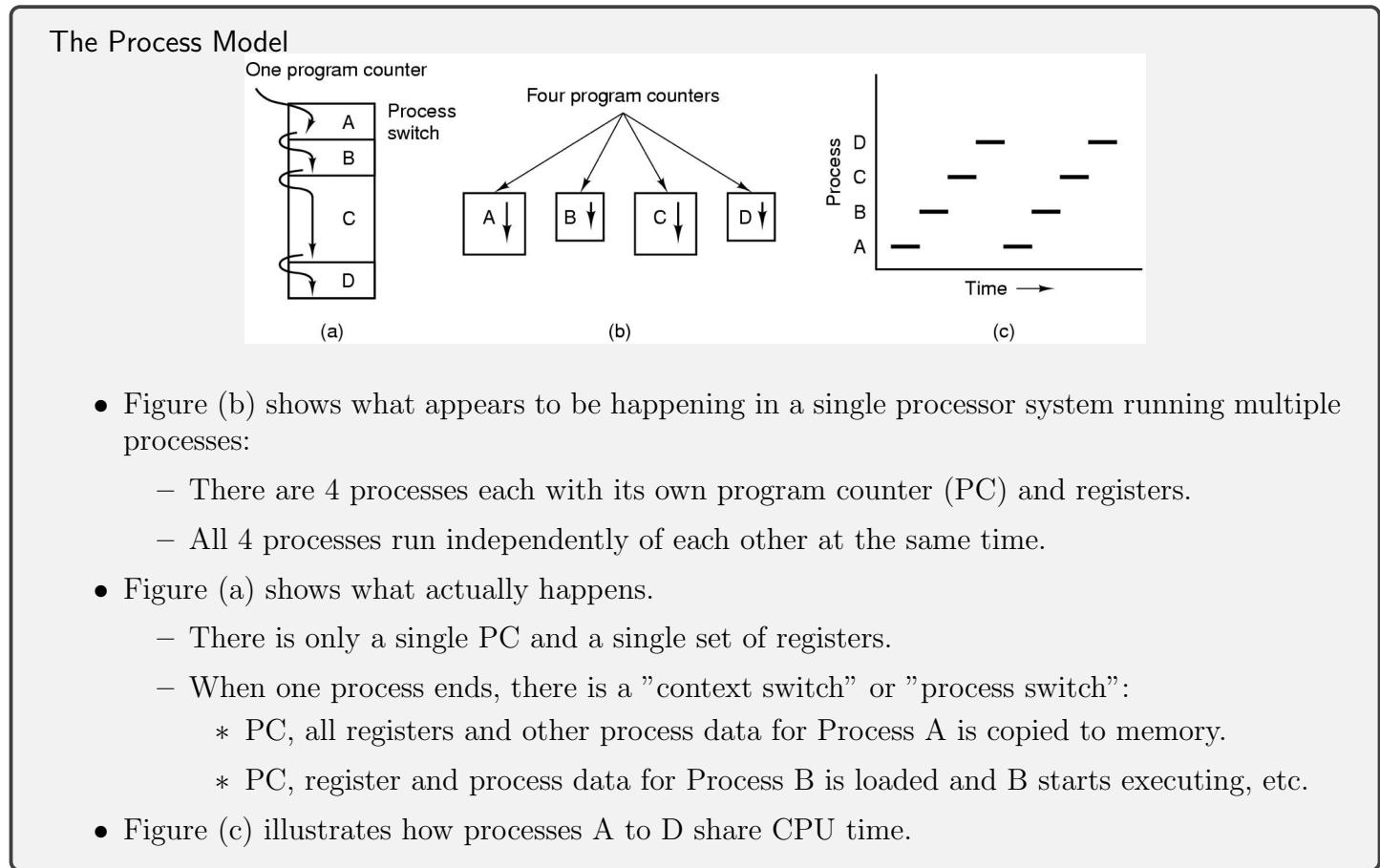
Program vs. Process

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.



Definition 2.4. Process States

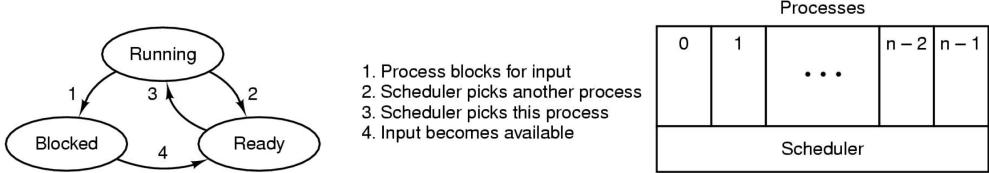
there are three possible states for a process

- 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.
- 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	SREG
0x3F (0x5)	I	T	H	S	V	N	Z	C	
ReadWrite	R/W								
Initial Value	0	0	0	0	0	0	0	0	

What other pieces of information does the OS need to save about a process?

- **File handles / Open File Table:** These are data structures that maintain information about files that are opened by the process, like the location that a process is inside the file, access rights to the file, file open modes, etc.
- **Pending signals:** A signal is an OS message to the process.
- **Process Running State:** Whether the process is suspending, ready, running, terminated, etc.
- **Accounting Information:** How much CPU time the process has used, how much disk space, network activity, etc.
- **Process ID:** Unique number identifying the process. Etc.

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 AVR's PC.
 - PC points to the next instruction to be executed.

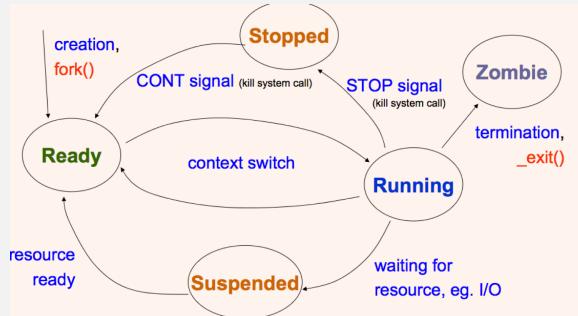
- End result: Task B resumes execution, with all its data and SREG intact!

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.
- `for(int i=0; i<10; i++) fork();`, this for general case n, there are 2^n processes created including the original process.

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

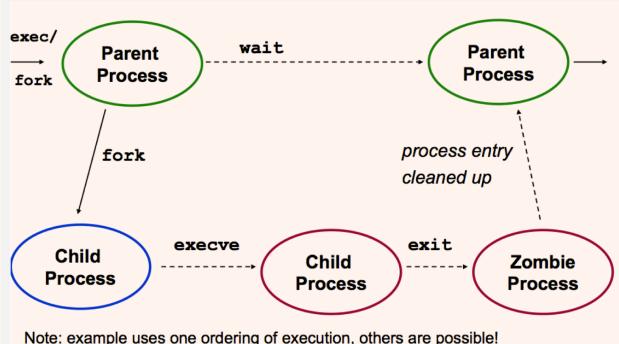
Definition 2.10. Normal Program Termination - `void exit(int status)` from standard C library function

- 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 don't 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:

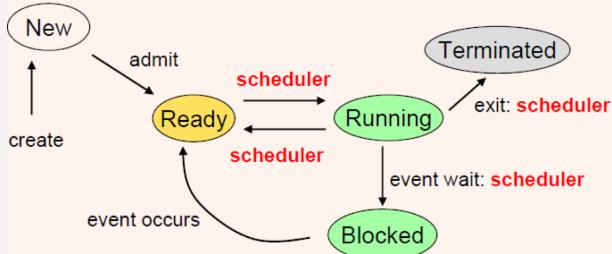
- Parent dies before child, the `init` process becomes "pseudo" parent of child processes. Child dying sends signal to `init` process, which organizes to call `wait()` to cleanup process
- Child dies before parent but parent didn't call `wait`. Becomes a zombie process. Can fillup process table requiring 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 Computations + 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. Context switching is controlled entirely by processes themselves. Co-operative multitasking is simpler and less prone to concurrency issues, but should any process go into an infinite loop, it could potentially freeze up the system.)
- Pre-emptive Multitasking (Currently running processes can be forcefully suspended by the scheduler. Timer-triggered multitasking.)
- 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.

Method 3.1. 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.
 - * Any process that has finished its run is blocked until the next time it is due to run again.
 - * The process can be preempted in the middle of its run.
 - * It simply cycles through the processes without caring about their deadlines, (unlike RMS and EDF which take into account deadlines), so we cannot guarantee that processes meet their deadlines in general.
- 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
- * We can also write down the table to see and the pattern will repeat for every $LCM(P_1, P_2, \dots, P_n)$
- 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 doesn't, process has missed its deadline.) T_i is the process or task arriving time.

Method 3.2. 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 2nd shortest, etc.)
2. For each task $T_i \in T$, recursively compute S_{i0}, S_{i1}, \dots where:

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

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.

Remark. Comparison between RMS and fixed priority policy on critical instance analysis

In **critical instance analysis**, the only things that are important are:

1. The priority remains fixed in some way throughout the execution life of the processes. In RMS the priority remains fixed because we assume that P_i is fixed. Here the priority is fixed by definition.
2. The period P_i of each process is fixed so that each process is pre-empted by a higher priority processed in predictable ways.

If these two conditions are met, then it doesn't matter whether we are using a fixed priority policy or an RMS policy; If $S_{i,F} < D_i$ for all i, it means that for every process, the worst case execution time including all possible preemptions is still less than the deadline for each process, and thus all processes are guaranteed to meet their deadlines.

This is important because it means that CIA can also be used for fixed priority systems as long as the two conditions hold.

Method 3.3. 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 3.1. 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 quanta 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: $\text{Priority} = \text{base} + f(\text{nice}) + g(\text{cpu usage estimate})$
 - $f(\cdot)$ = priority adjustment from nice value.
 - $g(\cdot)$ = 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.

Example 4.1. Race condition for two threads

There maybe four possible cases for tow threads running concurrently:

- Thread 1 runs to completion, $x = 6$, Thread 2 runs to completion, $x = 12$.
- Thread 2 runs to completion, $x = 10$, Thread 1 runs to completion, $x = 11$.
- Thread 1 loads x and is pre-empted, Thread 2 loads x and writes back $5 \times 2 = 10$, Thread 1 increments x to 6 and writes it back.
- Thread 2 loads x and is pre-empted, Thread 1 loads x , updates it to 6, writes back, Thread 2 gets $5 \times 2 = 10$

Definition 4.2. Correctness in multithreaded programs depends on the intended sequence of execution.

why we have wrong value?

It is either because the threads are executed in the wrong sequence, or because one thread gets pre-empted before it can save its results and the next thread gets a stale value. The first thread then overwrites the next threads results.

Definition 4.3. 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 no 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.

Can local variables created in C be affected by race condition?

Local variables are created on the process stack when a function is called. They're popped off and lost when the function exits. They can never cause race conditions because only the thread or process calling that function can access to the variables. Since no other process or thread has access, race conditions are not possible.

What can cause Co-operative multitaskers produce incorrect results?

- The sequence of process execution might be wrong. The scheduler might choose to run process A before B, when A depends on a result in B.
- - A process may inadvertently give up control of the CPU before completing calculations, causing a dependent process to run and get the wrong results. An example of this is when a process decides to call `printf`, which will trigger an OS call that might trigger a context switch without the programmers knowledge.

How mutual exclusion guarantees that the two processes produce only correct values of n?

Mutex guarantees that each process reads, updates and writes n before the other process runs. (Race condition for two not atomic operations)

Definition 4.4. Atomicity means that the steps taken in an instruction, or the steps taken in a group of instructions, are executed as one complete unit without possibility of interruption.

- Single-processor operating systems may disable interrupts to guarantee atomicity because task/process/thread switching does not happen without interrupts.
- User programs may not be as thoroughly tested, and allowing user programs to disable interrupts could lead to the entire system being crippled, as no further task switches would be possible.

Example 4.2. 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 doesn't enter critical section and waits until **lock** is 1.
- Process A finishes and sets **lock** to 1, allowing B to enter
- PROBLEM: There's 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() → DOWN(sema) → critical_section() → 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 (on the left is the correct version)

<pre> #define N 100 typedef int semaphore; semaphore mutex = 1; semaphore empty = N; semaphore full = 0; void producer(void) { int item; while (TRUE) { item = produce_item(); /* generate something to put in buffer */ down(&empty); down(&mutex); insert_item(item); up(&mutex); up(&full); } } void consumer(void) { int item; while (TRUE) { down(&full); down(&mutex); item = remove_item(); up(&mutex); up(&empty); consume_item(item); } } </pre>	<pre> /* number of slots in the buffer */ /* semaphores are a special kind of int */ /* controls access to critical region */ /* counts empty buffer slots */ /* counts full buffer slots */ void producer(void) { int item; while (TRUE) { item = produce_item(); /* generate something to put in buffer */ down(&empty); down(&mutex); insert_item(item); up(&mutex); up(&full); } } void consumer(void) { int item; while (TRUE) { down(&full); down(&mutex); item = remove_item(); up(&mutex); up(&empty); consume_item(item); } } </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- 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!**

Reusable/Consumable Resources

- * Reusable Resources - usually causes deadlocks
 - Examples: memory, devices, files, tables
 - Number of units is **constant**
 - Unit is either free or allocated; **no sharing** (no simultaneous using)
 - Process **requests, acquires, releases units**
- * Consumable Resources
 - Examples: messages, signals
 - Number of units **varies** at runtime
 - Process **releases** (create) units (without acquire)
 - Other process **requests** and **acquires** (consumes)
 - Deadlock when A and B are waiting for each other's message/ signal ...

- Make Peterson's solution work

```
#define FALSE 0
#define TRUE 1
#define N 2
/* number of processes */

int turn;
int interested[N];
/* whose turn is it? */
/* all values initially 0 (FALSE) */

void enter_region(int process);
/* process is 0 or 1 */
/* number of the other process */

other = 1 - process; /* the opposite of process */
interested[process] = TRUE; /* show that you are interested */
turn = process; /* set flag */
while (turn == process && interested[other] == TRUE) /* null statement */;

}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

- For Peterson's solution, it differs from the lock variable implementation in that there are two different lock variables, one for each process
 - * Each processes only WRITES to the lock variable, which is an atomic operation. The main weakness of the Lock Variable solution is that it involved read-update-write operations on a single shared variable, which does not happen in this case.
 - * Before entering the critical section, both processes check to see whether THE OTHER process intends to enter. If so it waits.
 - * Setting interested to TRUE takes place BEFORE this check, so in the worst case, both set interested to true at the same time, leading to deadlock. **Rescue:** by using turn, which can be either 0 or 1, we prevent both process stuck and busy waiting, one of them will execute the critical section and then leave and make interested = FALSE.

Theorem 4.2. Dealing with deadlocks

1. **Detection and Recovery:** Allow deadlock to happen and eliminate it
2. **Avoidance (dynamic):** Runtime checks disallow allocations that might lead to deadlocks
3. **Prevention (static):** Restrict type of request and acquisition to make deadlock impossible

Theorem 4.3. Conditions for Deadlock

1. Mutual exclusion: Resources not sharable
2. Hold and wait: Process must be **holding one** resource while **requesting another**

3. Circular wait: At least 2 processes must be blocked on each other

Method 4.1. Deadlock Prevention

1. Eliminate mutual exclusion

- Not possible in most cases
- Spooling makes I/O devices sharable

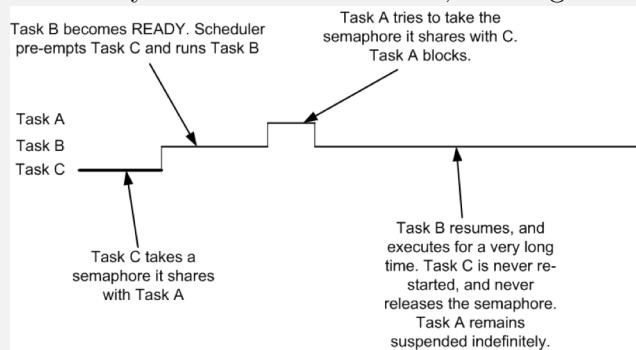
2. Eliminate hold-and-wait

- Request all resources at once
- Release all resources before a new request
- Release all resources if current request blocks

3. Eliminate circular wait

- Order all resources
- Process must request in ascending order

Problems with Semaphores: Priority Inversion priority(Process C) < priority(Process B) < priority(Process A), Process B effectively blocks out Process A, although Process A has higher priority!



Definition 4.5. Monitor, similar to a class or abstract-data type in C++ or JAVA:

- Collection of procedures, variables and data structures grouped together in a package. Access to variables and data possible only through methods defined in the monitor.
- However, only one process can be active in a monitor at any point in time. I.e. if any other process tries to call a method within the monitor, it will block until the other process has exited the monitor.
- Implementation: (mutexes or binary semaphores)
 - When a process calls a monitor method, the method first checks to see if any other process is already using it.
 - If so, the calling process blocks until the other process has exited the monitor.
 - The mutex/semaphore operations are inserted by the compiler itself rather than by the user, reducing the likelihood of errors.

Definition 4.6. Condition Variable - mechanisms for coordination

1. One process WAITS on a condition variable and blocks.
2. Another process SIGNALS on the same condition variable, unblocking the WAITing process.

```

monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;

```

```

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
end;

```

Similar to Sleep/Wake, however, the mutual exclusion from the monitor prevents the SIGNAL from being lost!

Monitors and Condition Variables Problems - Violation of mutual exclusion

- When a process encounters a WAIT, it is blocked and another process is allowed to enter the monitor.
- When there's a SIGNAL, the sleeping process is woken up.
- We will potentially now have two processes in the monitor at the same time:
 - The process doing the SIGNAL (the signaler).
 - The process that just woke up because of the SIGNAL (the signaled).

Ways to resolve

- We require that the signaller exits immediately after calling SIGNAL.
- We suspend the signaller immediately and resume the signaled process.
- We suspend the signaled process until the signaller exits, and resume the signaled process only after that.

Remark. Comparison between semaphore and condition variable

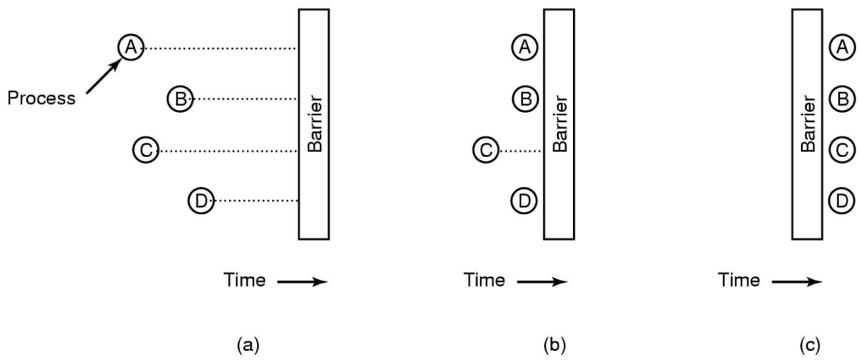
Semaphore

- If Process A UPs a semaphore with no pending DOWN, the UP is saved.
- The next DOWN operation will not block because it will match immediately with a preceding UP.

Condition variable

- If Process A SIGNALs a condition variable with no pending WAIT, the SIGNAL is simply lost.
- This is similar to the SLEEP/WAKE problem earlier on.

Definition 4.7. Barrier is a special form of synchronization mechanism that works with groups of processes rather than single processes.

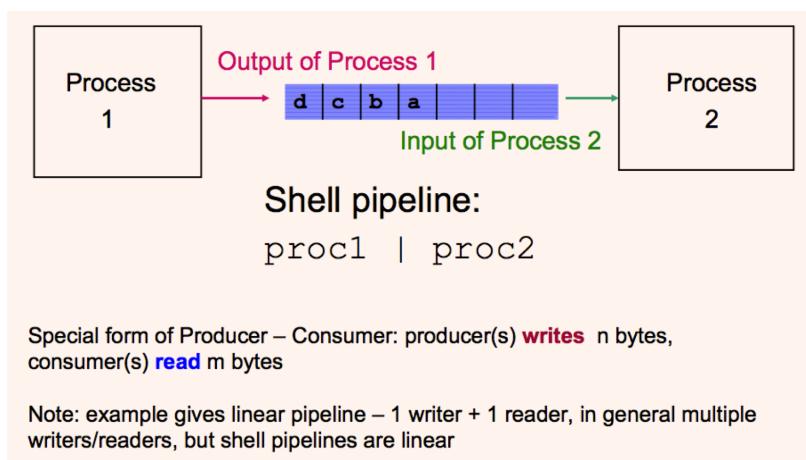


The idea of a barrier is that all processes must reach the barrier (signifying the end of one phase of computation) before any of them are allowed to proceed.

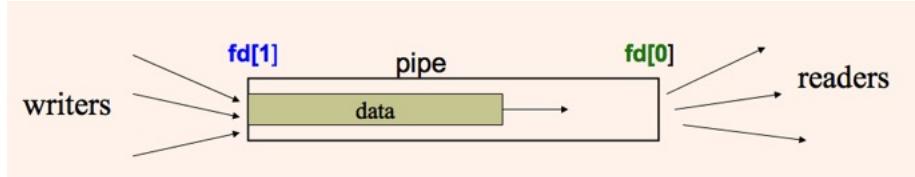
- Process D reaches the end of the current phase and calls a BARRIER primitive in the OS. It gets blocked.
- Similarly processes A and B reach the end of the current phase, calls the same BARRIER primitive and is blocked.
- Finally process C reaches the end of its computation, calls the BARRIER primitive, causing all processes to be unblocked at the same time.

Definition 4.8. UNIX Pipe

- A pipe provides synchronisation
 - A process reading a pipe will block until there is data.
 - Data send is asynchronous but blocks when buffer is full.
- A pipe provides byte-level message transfer between processes.
- Traditional method of communication between processes in UNIX: Shell pipelines
 - `grep buffer *.c | sort -u | wc`
 - Output from program to left of bar provides input to program on right.
 - Meaning: search for occurrences of string "buffer" in C files (grep process), sort those lines making them unique (sort process), and count how many occurrences (wc process)



- Creating pipes: `int pipe(int fd_array[])` returns 2 file descriptor for the (anonymous) pipe
 - Data is sent/received using normal write/read system calls
 - write on `fd[1]`, read on `fd[0]`, e.g., `fd[1]` is the write end, `fd[0]` is the read end.



- Some versions of Unix support duplex pipes, can readwrite on either end – with corresponding write/read on opposite end; other Unixes only have one way pipes
- **File Descriptor:** a reference to a file when making system call, come from opening a file with `open()` system call but also other system calls which deal with files such as `pipe()`

Remark. • Closing unused pipe descriptor is good practice (not necessary to close all unused pipe descriptors).

- Closing the write end of the pipe, allows reader to determine when there is no more data: when all pipe ends closed, read gives EOF.
- Data read is the minimum of what is available and requested size. (e.g., buffer size is 100 but the string written was 10 chars)
- Pipes only used for unstructured byte streams.

Definition 4.9. FIFO files (Named pipes)

- **Anonymous pipe:** can only use between related processes, e.g., parent + children.
- FIFO files are named pipes, pipes with a filename.
- Exist independent of process: any processes can use FIFO
- FIFO is a special file since it's really a pipe, cannot seek, no data is written to filesystem, every open FIFO corresponds to 1 pipe object.
- Can be created with `mkfifo` shell command or `mknod()` system call.

Example 4.3. Shell named pipe

1. `mkfifo pipe; ls -l > pipe`
2. `cat pipe` (in another shell login)

Example 4.4. Programming named pipe

Listing 1: writer.c

```
int fd ;
char * myfifo = "/tmp/myfifo" ;

mknod( myfifo , 0666);

fd = open( myfifo , O_WRONLY) ;
write(fd , "Hi" , sizeof("Hi")) ;
close(fd) ;

/* remove the FIFO */
unlink( myfifo );
```

Listing 2: reader.c

```
int fd ;
char * myfifo = "/tmp/myfifo" ;
char buf[MAX BUF] ;

/* open , read , and display */
fd = open( myfifo , O_RDONLY) ;
read(fd , buf , MAX BUF) ;
printf(" Received : %s \n" , buf) ;
close(fd) ;
```

Note: When only one program is run it will hang. `writer.c` blocks at the "write" statement, which means it will not call `unlink` to delete the named pipe until the reader has read the pipe.