

OPERATING SYSTEMS II
CSC3002F

Processes

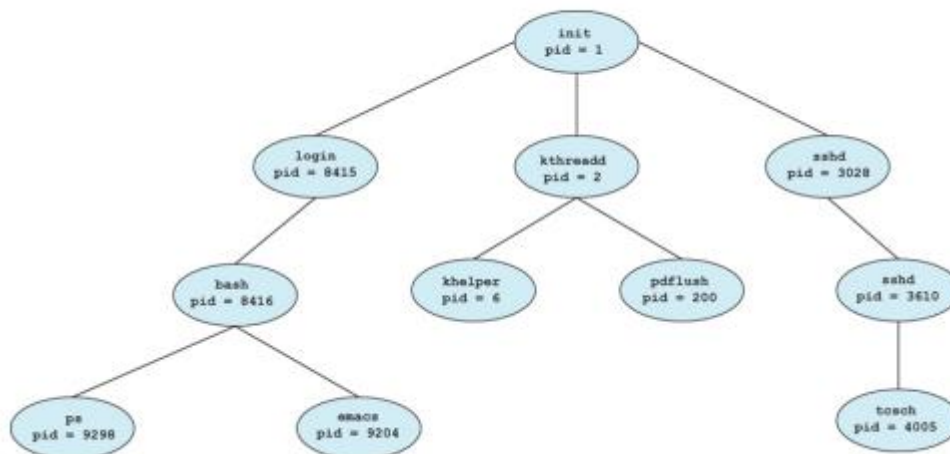
- **Program: static piece of compiled code**
 - By itself, does nothing
 - Only becomes process when run by OS
- Process: program in execution
- OS provides environment for program execution and services to those executing programs
- OS distinguishes between two different kinds of processes:
 - **OS processes** – run at more privileged level, provides services that OS exposes to other kind of process
 - **User based process** – the kind of process that we run
 - This process may need to access kernel services which are themselves processes
- OS has (OS processes + user processes) executing concurrently
- Have large number of processes running in system, all doing work, in various stages
 - Need to ensure that they're executing correctly
 - Want to avoid inappropriately access shared memory etc
- Three main phases for processes:
 - (1) **Creation** – initialisation, what code they run, etc
 - (2) **Scheduling** – which process should run next
 - Needed because have limited # of processing cores
 - OS creates illusion of infinite # of processors – and all processes can run concurrently (impossible)
 - This happens when have worst case of one core
 - OS scheduler deals with all this
 - (3) **Termination**
- OS also enables *inter-process communication* (ipc)
- **Process: unit of work in a time-sharing/multi-tasking OS**
 - Also called a job
 - All modern OSES are time-sharing/multitasking OSES – appear to be running multiple tasks at the same time
 - Need to use time-sharing mechanism to create this illusion
- A user creates a process by double-clicking or running a program from the command line
 - Two copies of a program running are two separate processes – OS is smart enough to not duplicate program code, and other shared states between them
- A program is a static entity (the code)
 - The specification, compiled code that process has to run
- A process is a dynamic entity – a program in memory executing or ready to execute
 - Need to consider memory implications i.to usage

Process vs Program

- OS provides this process to run program – has lower access lvl than kernel
 - **Kernel is core of OS – provides services to user programs**
 - Distinguish between things running in kernel vs user mode

- User can only run processes in user mode – lower access processes which can't modify certain things e.g. an area of disk/memory
 - Such services would be provided by kernel through a kernel API
 - User process would use that to get access to protected services
- A process is the *execution of an application program with restricted rights*
 - The process is the abstraction for protected execution provided by the operating system kernel
 - Kernel provides restricted environment for program to run in
- A process needs permission from the operating system kernel before accessing the memory of any other process, before reading or writing to the disk, before changing hardware settings, and so forth
- Linux – use `ps -A` cmd
 - Many options for `ps`
 - PID is process ID – unique
 - Child process can be spawned by parent process – thus have PID
 - Processes can also be execution environment for other code

Process Tree



- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
 - Every process has a parent – exception is process created by kernel when the PC first boots
 - In diagram, given name *init*
- Identified and managed via a *process identifier (PID)*

Linux Parent Process

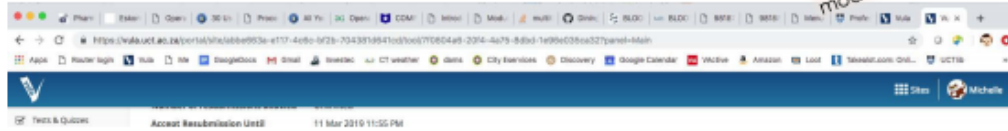
- Linux cmds:
 - `ps -A -forest` OR `pstree`
- The `init` or `systemd` or `launchd` process (depending on distro) is the parent of all processes on the system
 - First program that is executed on boot
 - Manages all other processes on the system

- Started by the kernel itself, so in principle it does not have a parent process
- Always has process ID of 1

Aside: Multiprocess Architecture: Web Browsers

Multiprocess architecture: Chrome browser

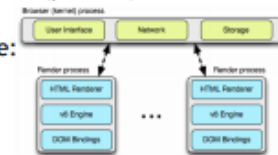
Chrome uses the same type of process isolation found in modern operating systems.



Tabbed browsing raises the issue that if any tab crashes, the entire browser (process) and all other tabs may crash too.

Google's Chrome browser addresses this through a multi-process architecture:

- There are three types of process in Chrome:
 - one browser process,
 - renderer processes (for rendering webpages) each running in a sandbox*
 - plug-in process for each plug-in.



Advantage is that websites run in isolation from each other, a crash of a website only affects the renderer.

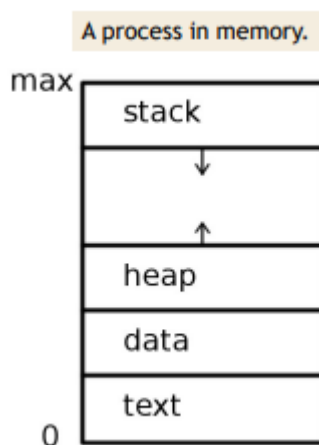
*sandbox provides increased security through restricted access to disk and network IO

11

- Web browsers such as Chrome, Internet Explorer, Firefox, and Safari, play a role similar to an OS
- Browsers load and display web pages, but many pages embed scripting programs that the browser must execute
 - These scripts can be buggy or malicious; hackers have used them to take over vast numbers of home computers

Process Creation

Process Context



- Surrounding info that's required to ensure process can execute correctly
 - This context includes necessary data to manage the orderly transfer of processes between different states – can be waiting, sleeping etc.

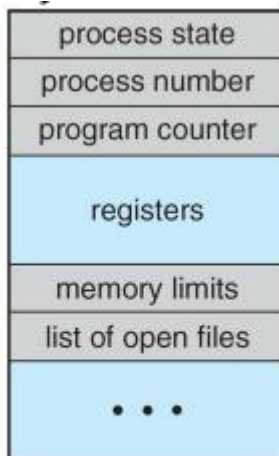
- A **process is a program together with its execution context** which encompasses:
 - The state of the processor (processor context), which is the value of its program counter and all of its registers
 - Recall that program counter is where next instruction is stored in memory
 - This part of execution context is necessary for a process to be stopped and then restarted at a later point
 - Process' memory map, which identifies the various regions of memory that have been allocated to the process
 - Can involve paging, etc
- A process in memory comprises:
 - *Text* (compiled code) – static program; has no meaning unless executing (then considered a process)
 - *Current state* (incl. program counter and registers)
 - Have memory regions that process stores info as it executes
 - Static region contains global data – fixed over process' lifetime
 - Corresponds to static keyword in languages
 - *Stack* (function parameters, return addresses, temporary data/local data)
 - The stack holds the state of local variables during procedure calls.
 - Limited resource – stores function call mechanism in memory
 - When call function – essentially push series of parameters onto stack that represent arguments to that function and return address (so function can return to correct place and continue executing code)
 - Function can call another function – creates another stack frame, which does the same
 - When create local variable inside a function – that variable is created on stack as well
 - Stack can run out with deep recursion
 - On diagram – arrows pointing towards each other should never converge
 - If they do, have **crit failure of memory resource** and will have some kind of error e.g. segmentation fault / other kernel lvl signal to show there's an issue with memory usage
 - *Data* (initialized static and global variables)
 - *Heap* (memory dynamically added during runtime – e.g. using malloc or new)
 - The heap holds any dynamically allocated data structures the program might need
 - Dynamic store – e.g. all calls to new in Java, c++ ex require space from heap
 - Used, then freed up and returned to the system
 - Happens automatically in java

Process Lists/ Table

- To keep track of processes, the operating system maintains a **process table (or list)**
 - List/ double linked list data structure
- Each entry pertains to each process which is running/ can run in that system
 - Process isn't always running – can be sleeping, waiting for i/o
 - Need to record these states so OS can operate appropriately e.g schedule next available thread/process onto the next core

- Large, complete data struct – stores all info that kernel requires to manage processes effectively
 - Scheduler makes heavy use of PCB
- **Each entry in the process table is a Process Control Block (PCB)**
 - Corresponds to a particular process
 - Contains fields with information that the kernel needs to know about the process
- A process is identified by a unique process ID (PID)
- Some OSes (UNIX-derived systems) have a notion of a *process group*
 - A way to group processes together so that related processes can be signalled
 - E.g. send terminal signal to process group – all processes in that group will be terminated
 - Collection of processes are doing some common task
 - Every process is a member of some process group

Process Control Block



On an OS that supports threads, the PCB is expanded to include the concept of a thread.

- Info associated with each process (also task control block)
 - Process state – running, waiting, etc
 - Knowing process state is critical to ensuring that it's managed correctly
 - Program counter – location of instruction to next execute
 - Necessary manage how processes resume once they've stopped and when they're running, prog counter points to next instruction to be executed
 - CPU registers – contents of all process centric registers, number and type is architecture dependent
 - Contains necessary info from CPU when a process is halted
 - Crit to correct running of sys
 - CPU scheduling information – priorities, scheduling queue pointers
 - E.g. a certain process must run frequently because it's important
 - Memory-management info - memory allocated to the process
 - Mapping, paging, etc
 - Accounting information – CPU used, clock time elapsed since start, time limits
 - How process is allowed to operate

- For well-managed processes – need to place limits on them (can't hog core)
- I/O status information – I/O devices allocated to process, list of open files
 - Series of queues attached to each IO device
 - Info that pertains to files that the process is manipulating
- Aside – modern OSes operate at finer granularity and use threads (essentially lightweight processes)

Process Representation

- Linux:
 - In Linux, the PCB is implemented as the data structure `task_struct`
 - Contain info about a process, such as process id, process state, file information, and so on
- OSes are written in C – lower lvl language
 - C is well-suited to resource management at a low lvl
- **The process block is the most complicated data structure in the operating system**
 - E.g. `task_struct` contains 83 field variables; 60 are primitive types, 3 are composite data structures, and 20 are pointers to composite data structures

Process Creation Unix

```
int main(void)
{
    pid_t pid = fork();

    printf("Hello from process PID = %d\n",pid);

    if (pid == -1) {
        fprintf(stderr,"fork failed");
        return 1;
    }
    else if (pid == 0) {
        printf("Bye from the child process! PID= %d\n",pid);
        return 0;
    }
    else { /* parent waits for child to end */
        int status;
        pid_t childpid;
        childpid = wait(&status);
        printf("child %d completed with status %d\n",
            childpid, status);
    }
    printf("Bye from the parent process! PID= %d\n",pid);
    return 0;
}
```

19

- UNIX has a system call to allow a process to spawn another process
 - fork() system call creates new (child) process
 - Executes independently of parent (but parent can wait())
 - E.g. can force the parent to wait for child to terminate
 - *Execution continues in BOTH processes after fork() statement*
 - fork() can be dangerous in the wrong hands 😊
- Once process uses a fork to create a child – for those processes, there are a series of options that need to be set – determines how they run past that point
 - Essentially, they then execute concurrently – leads to all the concurrency issues
- C code (above)
 - Call fork cmd – creates a child process with identifier stored in pid
 - Now have parent process and child – both execute at the instruction immediately after fork cmd
 - Both print out statement with process id
 - Put parent into wait state – then stops operating
 - Scheduler removes it from core and it goes onto wait queue
 - Once child is terminated, the parent process is woken up and continues

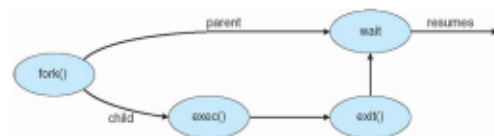
Process Creation

- **When creating a child process – need to determine how resources will be allocated to that child**
- Resource sharing options for processes
 - Parent and children share all resources
 - Children share subset of parent's resources (e.g. Linux)
 - Parent and child share no resources (e.g. Windows)
- **Also have different options for how child and parent will execute after the fork() cmd – i.e. child process creation**
- Execution options
 - Parent and children execute concurrently – as two independent processes that can terminate separately or *parent can wait for child*

- Also can overlay an additional program in child process and have it execute that instead – child executes its own program
 - Parent waits until children terminate
- Address space
 - Child duplicate of parent - behaviour of fork system call in UNIX
 - Have the same window in the address space – allows for easy sharing of info between processes
 - But often also want child to do something else
 - Child has a program loaded into it - behaviour of exec system call in UNIX
 - Replaces parent program with a new program – creates a new address space and destroys whatever was there before and now operate completely independently
 - Running different programs
 - A common use case

UNIX Example

Process creation in Unix



exec() system call used after a *fork* to replace the process' memory space with a new program

```

int main(void)
{
    pid_t pid = fork();
    printf("Hello from process PID = %d\n", pid);

    if (pid < 0) {
        fprintf(stderr, "fork failed");
        return 1; /* error code */
    }
    else if (pid == 0) { /* in child */
        execlp("/bin/ls", "ls", NULL);
    }
    else {
        wait(NULL);
    }
    printf("Bye from the parent process! PID= %d\n", pid);
    return 0; /* successful completion */
}
  
```

21

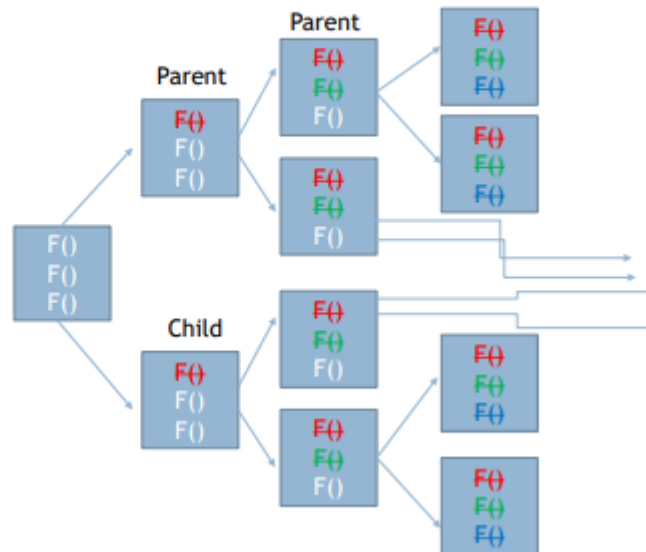
- If process ID is negative => fork has failed
 - If 0, know we're in the child process, can then write logic for that
- Write execlp – variant of exec cmd
 - Allows for input of cmd line parameters passed into a given executable
- Run directory listing program
- Program then terminates
- Inside parent, further down
 - Execute wait cmd – waits until child terminates
 - Since there's only one child – unambiguous
- Issue exit cmd – signal to OS to terminate process and recoup all its resources
 - Can either explicitly ask process to exit or can simply return
 - With return, have return code and exit called on your behalf – use return # as exit code

Another Fork()ing example

- `fork()` will always cause a process to spawn a new, duplicate process
- Execution continues IN BOTH process, after the `fork()` command
- How many processes will be created by the small program below?

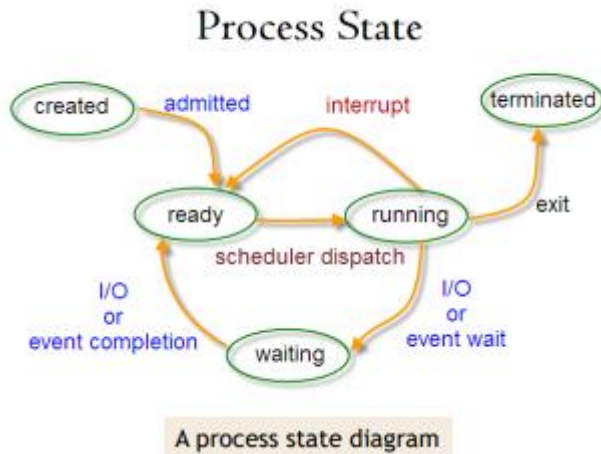
```
int main(void)
{
    fork(); /* 1 */
    fork(); /* 2 */
    fork(); /* 3 */

    return 1;
}
```



- Fork is always used to create a child process from another process – can cascade these
- Execution continues IN BOTH process, after the `fork()` command
 - Running the same code in parallel (unless `exec` used)
- Can check process ID
 - If zero, we know we're in a child process with some parent that used `fork()` to create it
- Q: How many processes will be created by the small program below?
- A: 8!
 - Doubling each time we `fork()` – why?
 - At first lvl – program executes, hits first `fork` cmd, creates a new child process
 - Now have parent and child
 - One `fork` cmd has occurred – so crossed out
 - Execution resumes in both processes after that
 - Another `fork` – both go ahead and execute
 - Now have two separate processes, executing `fork`
 - Each of these now create another process
 - Parent spawns a new child – 3rd column
 - Another `fork` crossed out
 - Now have parent and child, and then original child and its child
 - All four of these have `fork` cmd to execute
 - That happens – creates new subchild for each of those
 - Now have 8 processes running

Process State



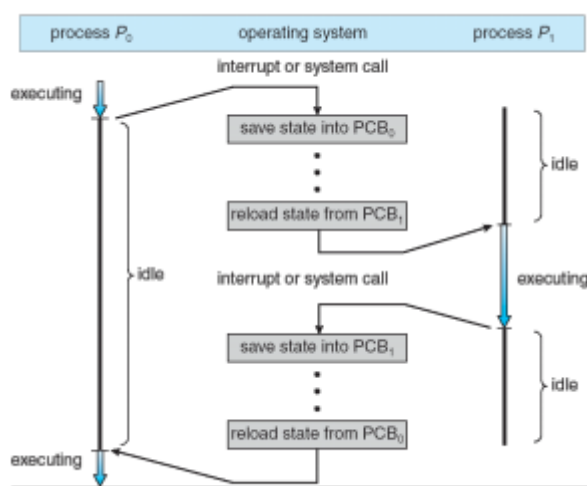
- **Only one process can be running on any processor at any instant**
- A process changes state as it executes:
 - **New/created** - the process is being created
 - Takes time for process to be created, then admitted to ready state
 - **Running** - instructions are being executed
 - Process has been scheduled by dispatcher
 - Can exit and move into terminated state
 - **Waiting** - the process is waiting for some event to occur
 - Process transitions when system event is raised (that it can't deal with itself) or it's waiting on i/o transaction
 - Typically order of magnitudes slower than CPU work – thus makes no sense to have process sitting on core, doing nothing
 - Thus, why process is moved from running to waiting state – kernel moves process off core
 - Lingers in waiting state until event is serviced or i/o is complete, then moves back into ready state (where it can be dispatched again by scheduler to run on some core)
 - **Ready** - the process is waiting to be assigned to a processor
 - Available for scheduling by dispatcher
 - Interrupt changes process from running state to ready – process is put onto the ready queue (can be rescheduled back onto the core it came from, or another core)
 - **Terminated** - the process has finished execution
- Process state diagram represents all stages a process can be in – represented as nodes
 - Link/edges linking various states together – shows under which conditions they transition from one state to another
- **OS has to manage process through its entire lifetime – has to be aware of transitions and have methods in place to recoup resources, manage prioritising of processes, generate interrupts, service interrupts, etc**

- Process can be moved off core by an interrupt – most common interrupt type is when process exceeds its execution quantum
 - To give illusion of large array of processes running concurrently, need to time-slice the amount of time that they can be run
 - Only allow process to run from start to finish if it's short – otherwise after a certain amount of time, we interrupt its running on the core

Context Switching

- Entire execution environment – all the state it needs to run
- **When a process is running, the system is said to be in the context of that process.**
- Context switching can also be used for interrupts and system calls
- **When the kernel decides to execute another process, it performs a context switch, causing the system to execute in a different process' context**
 - For a context switch, the kernel has to save enough information so that it can switch back to the earlier process and continue executing exactly where it left off
 - **When a process executes a system call and has to change from user to kernel mode, the kernel also has to save enough information so that it can later return to user mode and continue executing the program**
 - System calls run in kernel mode because they have a privileged level – they're accessing low lvl resources
 - Existing state has to be saved while system call is managed
 - Need to restore process to its state prior to save occurring
- 75,000-100,000 context switches per second not uncommon!
 - Context switching needs to happen very fast

CPU Context Switching



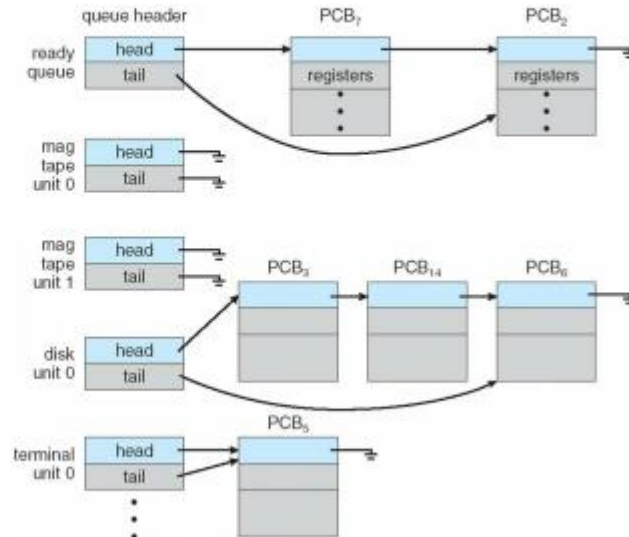
- Diagram shows process that has been interrupted – another process will be switched in by scheduler
 - Need to keep cores occupied at all times otherwise sys degrades
- Context switch steps:
 - Save current process to PCB – save current state
 - Decide which process to run next

- Load of new process from PCB – load pcb of new process that has been swapped in, and continue executing
- **Context switch should be fast, because it is overhead- the system does no useful work while switching processes**
- Diagram – process P0 on lhs, busy executing
 - Solid line indicates where process is doing nothing because a sys call has happened
 - E.g. i/o operation
 - Save pcb for process – takes an amount of time
 - Other process P1 is idle – thus its wasted time
 - Want to minimise this time – context switch needs to be fast
 - Reload state of process that want to swop in – P1
 - Executes and will service interrupt / manage sys call
 - Once complete, gives up control of call that its been running on
 - Loads/saves its state so that it can be resumed later on if necessary
 - Reload original process P0
- Have a lot of idle time – want to minimise this
- The more complex the OS and the PCB → the longer the context switch
 - Speed at which it happens depends on OS complexity, peculiarities of pcb (how info rich it is)
- Time taken is dependent on hardware support (helps manage context switches):
 - Hw designers try to support routine context -switch actions like saving/restoring all CPU registers by one pair of machine instructions
 - Some hw provides multiple sets of registers per CPU → multiple contexts loaded at once

Process Scheduling Queues

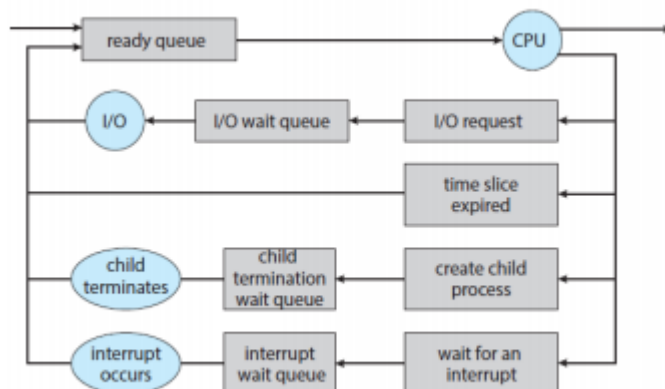
- **Scheduling is used to determine which process next gets access to CPU** – OS uses collection of queues to manage this (non-exhaustive list below)
 - **Job queue** – set of all processes in the system
 - Can have thousands of entries – each need to be “scheduleable” and able to run
 - All processes that can run but may not be able to because they don’t reside in main memory
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - Just need to be context-switched onto core – can then execute
 - **Device queues** – set of processes waiting for a particular I/O device
 - Can attach different priorities to these queues, etc
- A process scheduler selects among available processes for next execution on CPU
- *Processes migrate among the various queues* – OS can move processes between these queues, depending on sys constraints
- Queues generally implemented as a doubly-linked list of PCBs

Ready Queue And Various I/O Device Queues



- Ready queue has pcbs which are ready to be executed
- Can have queues attached to i/o devices
 - Have jobs waiting for slow service from these devices
- Have disk queue – have various processes waiting to read data from disk
- Also, terminal queue waiting for keyboard i/o
- Above are subset of possible queues
 - Queues available depends on OS config

Process Scheduling



- Queueing diagram represents queues, resources and flow between them
- Circles are resources, arrows indicate flow
- New processes start in ready queue until dispatched
- Ready queue – sent here when process enters the sys
 - Available to be dispatched onto CPU
- Once process runs on CPU i.e. moves from ready queue down to CPU

- Exits to the right when terminating
- Or can experience a number of different conditions – moves it off the CPU and onto a different queue
- IO request comes in – process migrates (because of that request) onto io wait queue
 - Sits on queue until io resource available – once it is, and can service io request/transaction, process moves onto ready queue
- Timeslice expires, process has to move off CPU so something else can run
 - Moves directly back into ready queue
- Process could also create child – has to wait until child terminates (depending on nature of parent-child relationship)
 - Goes onto child-termination wait queue
 - Once child terminates, can move onto ready queue again and be rescheduled
- If interrupt occurs when process running (last condition)
 - Moves into interrupt wait queue
 - Once interrupt is serviced, moves onto ready queue

Scheduler

- **Short-term scheduler (or CPU scheduler)** – selects process to be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Invoked frequently (milliseconds) – must be fast (cores needs to have something running on it at all times)
 - Needs to made decisions rapidly – algos used to make these decisions need to be simple and sophisticated enough that they can be executed quickly
- **Long-term scheduler (or job scheduler)** – selects processes to be brought into the ready queue
 - Invoked infrequently (seconds, minutes) – may be slow
 - Controls the degree of multitasking – i.e. the number of concurrent processes in memory (can run, but aren't necessarily running at a particular point in time)
 - Brings in processes with lower priority – these processes often aren't in main memory
 - Moves them into memory – now available for short term scheduler
- Processes can be described as either:
 - **I/O-bound** – spend more time doing I/O than computations, many short CPU bursts
 - Slow, ends up waiting a lot
 - Long-term scheduler concerned with this
 - **CPU-bound** – spends more time doing computations; few very long CPU bursts
- **Problem is striking a balance between how much CPU time these processes get – where the scheduler comes in**
 - Long-term scheduler strives for good process mix – collection of CPU-bound processes doesn't hog all the computational cores (otherwise io won't happen)
 - But also, don't want io processes lingering on CPU – they'll just need to be swopped off again
 - IO-bound thus need a fair chance at CPU, but must not be unnecessarily swopped in – will need to do a context swop

Aside: Multitasking in Mobile Systems

Multitasking in mobile systems

Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended

From iOS 4, limited multitasking for user apps. iOS provides for:

- A **single foreground process**- controlled via user interface
- Multiple background processes- in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Advantages: better battery life, memory use and security.



Android runs **foreground and background**, with fewer limits

- Background process uses a service to perform tasks
- Service can keep running, even if background process is suspended
- Service has no user interface, small memory use.

Advantages: more flexibility, customizable.



9

Aside: iOS multitasking

Traditional multitasking means being able to run more than one program at the same time. Multitasking on the iPhone differs: the iPhone allows a few kinds of apps to run in the background while other apps work in the foreground.

Most commonly, iPhone apps are paused when you're not using them and then quickly come back to life when you select them.

When you click the home button to leave an app and return to the home screen, the app you just left essentially freezes where you were and what you were doing. The next time you return to that app, you pick up where you left off instead of starting over each time. This isn't really multitasking.

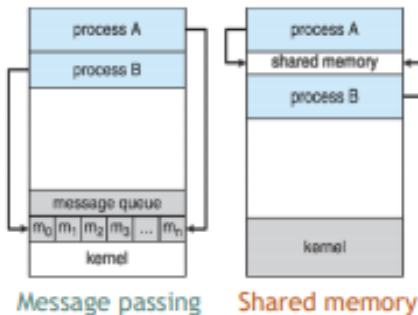
iOS apps can exist in any of five states of execution. These are:

- **Not Running:** The app has been terminated or has not been launched.
- **Inactive:** The app is in the foreground but not receiving events (for example, the user has locked the device with the app active).
- **Active:** The normal state of "in use" for an app.
- **Background:** The app is no longer on-screen but is still executing code.
- **Suspended:** The app is still resident in memory but is not executing code.

apps that are suspended do not use battery life, memory, or use other system resources.

References: <https://www.lifewire.com/use-multitasking-on-iphone-2000763>

Inter-process Communication



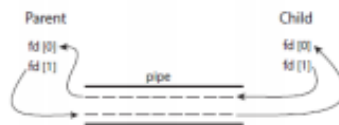
- **Processes within a system may be independent or cooperating**
 - Fork cmd is an independent way of creating a process
- Want processes to co-op – can achieve more
 - These processes need a way to co-op – exchange info, control each other via synchronizing primitives
 - Affected through inter-process communication
- Cooperating process can affect, or be affected by, other processes, including sharing data
- Reasons for supporting cooperating processes:
 - **Info sharing** – typically shared memory
 - **Computation speedup** (parallelization)
 - Take computation and divide up into parallel tasks – want tasks to be able to talk to one another and exchange info
 - Decomposition of a problem into many smaller, sub problems
 - **Modularity** – break up large pieces of functionality and wrap them in a process
 - Need those processes to communicate
- Cooperating processes need interprocess communication (IPC)
- **Two models of IPC**
 - **Shared memory** – can be faster (fewer system calls)
 - Processes share a part of memory, where they can exchange info – leads to many issues
 - **Message passing** – better for distributed systems generally
 - Processes send msgs to each other – end up with collection of queues
 - One process will send to queue – another will read from that queue

IPC Models: Shared Memory

- OS usually tries to prevent one process from accessing another process' memory
- **With shared memory, two or more processes agree to remove this restriction and exchange information by reading and writing data in shared areas**
 - Now we have to manage synchronizing and ensuring correct computation happens
- e.g. POSIX shared memory In UNIX

IPC Models: Message Passing

- **Message system – processes communicate with each other without resorting to shared variables, using mechanisms for processes to communicate and to synchronize their actions**
 - Have send and receive msg facility which all processes can access
 - One process will send msg – arrives in msg queue
 - Receiving process will basically receive a msg from a queue (i.e. pull a msg off)
- IPC facility provides two operations:
 - `send(message)`
 - `receive(message)`
 - Msg size fixed or variable
- **Examples:**
 - Windows advanced local procedure call (ALPC) facility
 - Allows methods on a local system to behave akin to RPCs
 - Pipes (named or unnamed) – UNIX (also Windows)
 - Ordinary pipes – io device that child and parent use to communicate (without using shared memory)
 - Create a child with fork, then create pipe which allows parent and child to write info back and forth
 - Named pipes – persistent data structures that exist on disk
 - Processes can read to/write from – requires more setup
 - Client-server communication
 - Sockets
 - Remote Procedure Calls (RPCs) – have processes on different machine that communicate with each other (using RPCs, pass msgs)



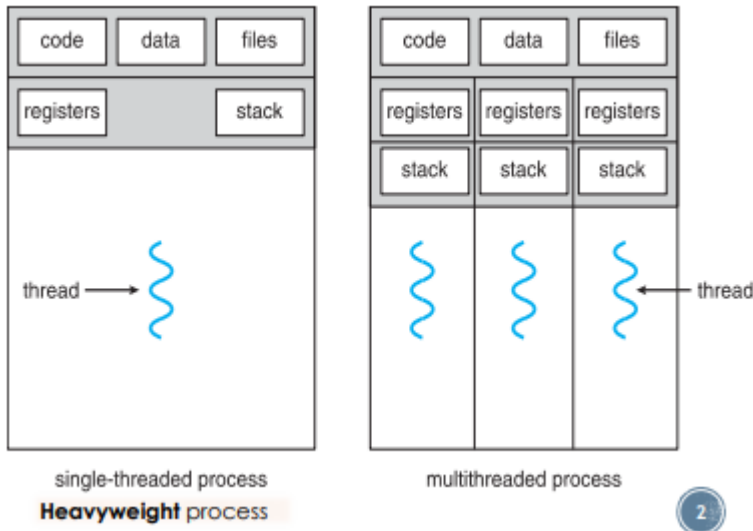
Process Termination

- **Process terminates when it executes last statement**
 - Once process has finished all its work – **can either put explicit call to exit, or use return**
 - Generates call to exit with appropriate return code
 - Asks the OS to delete it using the `exit()` system call
- On exit, process may return a status value (typically an integer) to a waiting parent process
 - Parents wait for children with a `wait()` system call – moves onto ready queue until that call is serviced
- **On exit, all process resources are deallocated and reclaimed by the OS**
 - Including physical and virtual memory, open files, and I/O buffer, pcb and process ID
- Termination can occur in other circumstances
- A process can terminate another process via a system call

- e.g TerminateProcess() in Windows, abort() in Unix/Linux (abort exits without doing proper resource management)
- Usually, a terminate system call can be invoked only by a parent process
 - Otherwise, a user, or a misbehaving app could arbitrarily kill another user's processes
 - Because a parent needs to know the identities of its children to terminate them, the identity of the newly created process is passed to the parent on creation
- A parent may terminate the children processes because:
 - **Child has exceeded allocated resources**
 - **Task assigned to child is no longer required**
 - **The parent is exiting and the OS does not allow a child to continue if its parent terminates**
- Once a child process exits and lets OS know by issuing its exit cmd – OS recoups all resources for that process
 - However, at that point it hasn't completely recovered the pcb – so there's info in the pcb that lingers until parent calls wait, or otherwise acks that process has completed
- After exiting, *a child process still has an entry in a process list*, so waiting parent can check that it is complete and read its status code
 - **If no parent waiting (did not yet invoke wait()) process is a zombie – resources freed up but entry still in pcb (parent needs to query that status)**
 - Once parent calls wait, process unravels
 - **If parent terminated without invoking wait(), process is an orphan – pcb is still active, process lingers on system**
 - On UNIX systems, orphaned processes are generally inherited by process with id 1, which then proceeds to kill them
 - Parents need to check status of child process – wait() return child's status (i.e. what was the termination code for the child)
 - That info is stored in pcb for the process – all other info can be recovered, but some needs to stay there so that the correct behaviour occurs when wait cmd is serviced
- Once process exits and resources recovered, parent issues wait cmd and allows it to be serviced, and then returns status code to parent
 - Parent can then go on and issue its own exit and terminate

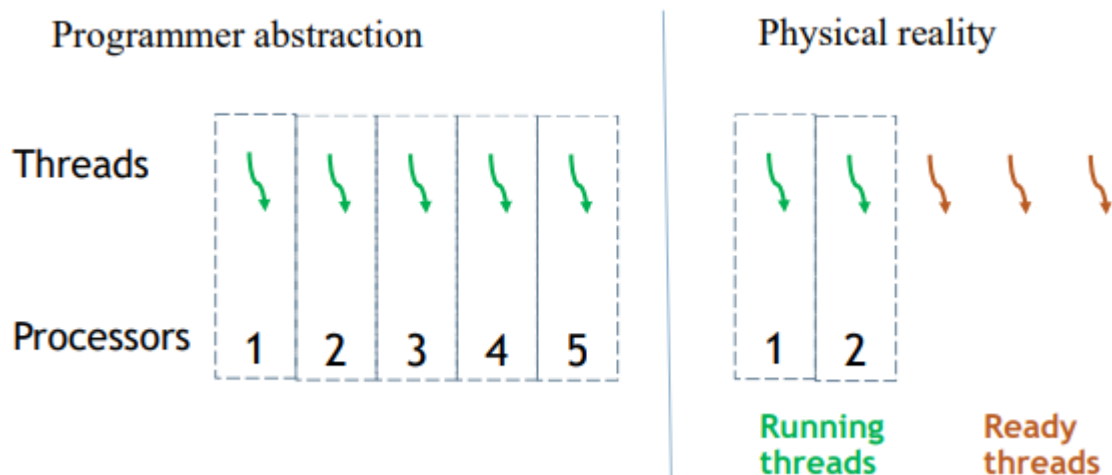
Multithreading

Threads



- Most modern applications have multiple “threads of control”, or threads, in order to process more than one task at a time
- **Thread is a basic unit of CPU utilization**
- **Thread comprises thread id, program counter, register set and a stack**
- Threads within a process share code, data and files + signals etc, with creating parent process
 - Other things are unique to each thread e.g. registers, stack
- **Can only have one thread running on any one time on a core** – hence why multiple cores/CPUS is advantageous
 - With multiple cores, can have multiple threads running concurrently

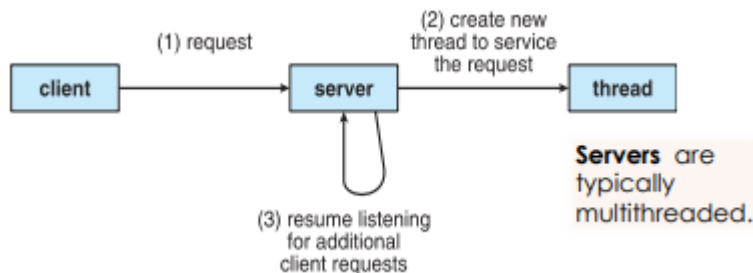
On Processors



- It is the job of the OS to provide the illusion of a nearly infinite number of virtual processors
 - Achieved by timeslicing, moving threads off the cores, etc

Multithreading Benefits

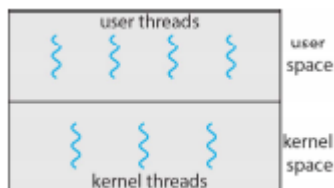
- **Responsiveness** – may allow continued execution if part of process is blocked (e.g. an io operation gets serviced), especially important for user interfaces and servers
 - If only have one thread of execution, for io, entire process halts
 - With multiple threads, these “subprograms” can continue running on different cores while io request is serviced
- **Resource sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching has a lower overhead than context switching
 - Process is a heavyweight thread – need to create a process to act as container for the threads
- **Scalability** – process can take advantage of multiprocessor/ multicore architectures: more concurrency



Kernel threads

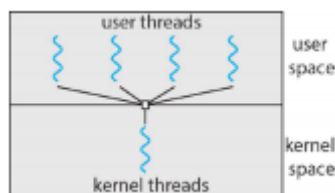
- The OS kernel is multithreaded
- In Unix/Linux `ps -ef` will display the kernel threads
- The kernel thread `kthreadd` (pid = 2) is the parent of all other kernel threads

User and Kernel Threads



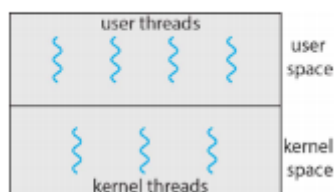
- **Support for threads provided at two different levels:**
 - **User threads** – supported above the kernel and managed without kernel support, primarily by user-level threads library
 - Exist in user space – collection of processes that user has spawned
 - Regular threads, created with no additional privilege
 - **Kernel threads** – supported and managed directly by the OS
 - Supported by nearly all modern OS (Windows, Linux, and Mac OS X)
 - Threads are doing low lvl, kernel operations e.g. monitoring ports, managing disk io requests, etc

- *This is what is scheduled by scheduler to the various cores*
 - Have escalated level of privilege
- Need mapping between user and kernel threads
- **Mapping between user and kernel threads can be:**
 - **Many-to-One** – all user threads (created by a user level thread library) are mapped onto a single thread
 - Create illusion that system is multithreaded
 - Pure user-level thread implementation
 - If one thread blocks, all block
 - Few systems currently use this model, because multiple threads do not run in parallel on multicore systems
 - No collection of threads running in the kernel, so can't have true parallel support for threads (thus can't run concurrently)



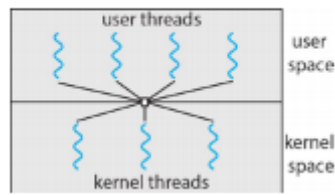
- **One-to-One** – every user thread is mapped onto corresponding kernel thread and is dispatched onto different cores
 - Creating a user-level thread creates a kernel thread
 - # threads per process typically restricted
 - Users can create a large number of threads – have to decide how to schedule those
 - *OS can impose constraints as to how many threads one can create to ensure system doesn't get overwhelmed*

One-to-One



- **Many-to-Many** – take large collection of user threads and map them to a smaller collection of kernel threads
 - E.g. Windows, Linux
 - *Many user-level threads to be mapped to as smaller or equal number of kernel threads*
 - Allows the OS to create a sufficient number of kernel threads
 - Mapping chosen to optimise various criteria around resource utilisation
 - OS tries to ensure that it services as many threads as possible using resource optimally
 - E.g. threading waiting on io will be swooped out

Many-to-Many



Thread Libraries

- **A thread library provides programmer with API for creating and managing threads**
- May be:
 - Entirely in user space, no kernel support OR
 - E.g. many-to-one, don't use kernel directly at all – create and manage threads as lightweight user processes
 - Mapping needs to then happen
 - Kernel-level library supported by the OS
 - Directly support creation of kernel threads – different levels of privilege
- Three primary thread libraries:
 - POSIX Pthreads – user-level, kernel-level
 - Windows threads – kernel lvl
 - Operate with elevated access
 - **Java threads – JVM uses thread library on host system,**
 - e.g. Windows threads on Windows and Pthreads on Unix/Linux

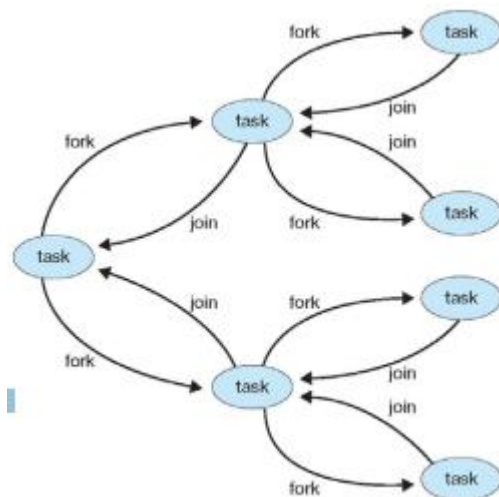
Pthreads

```
#include <pthread.h>
#include <stdio.h>
/* this function is run by the second thread */
void *inc_x(void *x_void_ptr)
{
    /* increment x to 100 */
    int *x_ptr = (int *)x_void_ptr;
    while(++(*x_ptr) < 100);
    printf("x increment finished\n");
    /* the function must return something - NULL will do */
    return NULL;
}

int main()
{
    int x = 0, y = 0;
    /* show the initial values of x and y */
    printf("x: %d, y: %d\n", x, y);
    /* this variable is our reference to the second thread */
    pthread_t inc_x_thread;
    /* create a second thread which executes inc_x(&x) */
    if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }
    /* increment y to 100 in the first thread */
    while(++y < 100);
    printf("y increment finished\n");
    /* wait for the second thread to finish */
    if(pthread_join(inc_x_thread, NULL)) {
        fprintf(stderr, "Error joining thread\n");
        return 2;
    }
    /* show the results */
    printf("x: %d, y: %d\n", x, y);
    return 0;
}
```

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
 - May be provided either as user-level or kernel level
 - Specification, not implementation
 - API specifies behaviour of the thread library, implementation is up to development of the library
- Common in UNIX operating systems
- Diagram:
 - Top function is method run by thread
 - Takes variable as an arg, and increments it
 - Passing reference to variable – change persists to parent code
 - Begin in main process (thread of execution)
 - Print out their values
 - Create reference to new thread
 - Then in pthread_create, thread creation happens
 - Thread now active
 - Main program does the same operation – increments y
 - Thread rejoins with parent before terminating

Java Threads



- Managed by the JVM, specification does not say how threads are mapped onto kernel threads
 - Provides consistent API for the user to create threads
 - Typically implemented using the threads model provided by underlying OS
 - Two techniques for creating threads in a Java program:
 - (1) Create a new class that is derived from the Thread class and to override its run() method
 - (2) Define a class that implements the Runnable interface
 - When a class implements Runnable, it must define a run() method
 - The code implementing the run() method then runs as a separate thread

Implicit Threading

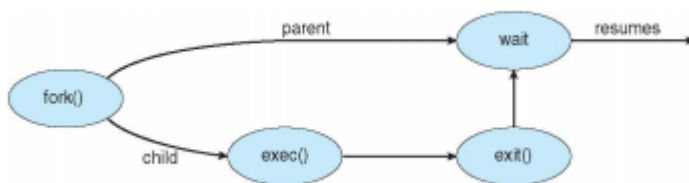
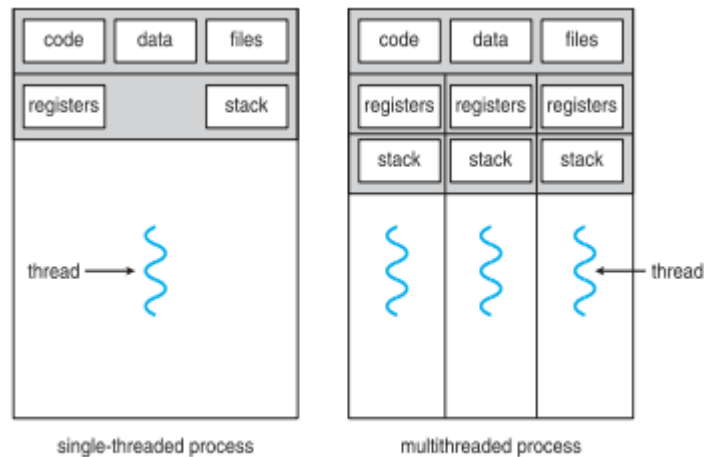
```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

- In this model, creation and management of threads is done by compiler and threading library, to take burden off programmer
 - Rather than directly creating threads, create tasks
 - These *tasks are taken by a framework, mapped onto threads* and then distributed to different cores
- E.g. Java fork join, OpenMP (high lvl programming construct)
 - OpenMP is embedded in C, c++ - use hash pragmas (instructions to pre-processor, compiler to parallelise the code given sum constraints)

Threading Issues

Semantics



- Semantics of fork() and exec() system calls – fork() system call is used to create a separate, duplicate process
 - Does fork() duplicate only the calling thread (thread that fork was issued in), or all threads (in the process)?
 - Some UNIX systems have two versions of fork which operates at two different levels
 - Duplicating all is resource intensive
 - When issued in process, fork() duplicates entire process – those processes then carry on executing
 - exec() usually works as normal – replace the running process including all threads
 - Takes new process and overlays a new memory space and some executable to run in

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
 - Process has to respond
 - E.g. kill signal to terminate process
- May be *synchronous* (e.g. division by 0) or *asynchronous* (e.g. terminating a process or expiration of a timer)
 - Synchronous need to be operated on immediately/asap – managed by thread which triggered that signal
 - E.g. thread has division by zero or illegal memory access
 - Thread receives signal and must act on it

- Asynchronous – emerge from outside a thread
 - Instruction to terminate a process, an interrupt to stop a particular thread from executing (so another can be swapped in)
- For single-threaded, signal delivered to process
- Where should a signal be delivered for a multi-threaded program?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process (and execute specific responses to that signal)
- The choice of which paradigm to use depends on program and what support is available in the OS
- Most versions of Unix allow threads to specify which signals it will accept and which not
 - However, if there's a particular signal they can all respond to, signals delivered only to first accepting thread only
 - Signal then considered to be processed

Thread Cancellation

- Terminating a thread before it has finished – thread to be cancelled is target thread
- Two general approaches:
 - Asynchronous cancellation terminates the target thread immediately
 - Deferred cancellation allows the target thread to periodically check if it should be cancelled, allows the thread to terminate in an orderly fashion
- Deferred cancellation is *generally preferred* to asynchronous termination
- Asynchronous cancellation is problematic *where resources allocated to a cancelled thread are either not released or are in an unsafe state*

Synchronisation

- **Sync** – ensuring that certain set of relationships hold over a sequence of events
 - Want to ensure, by using the correct sync primitives, that **the correct order of events always plays out and ordering of events is guaranteed and consistent**
- Here, synchronization refers to relationships among events—
 - any number of events and
 - any kind of relationship (before, during, after)
- Synchronization constraints are requirements pertaining to the order of events
- In OSes, often need to satisfy synchronization constraints without the benefit of a clock, either because there is no universal clock, or because we do not know to a fine enough resolution when events occur
- Sync primitives don't depend on time

Reminder: Threading Model Summary



Multiple *threads*

Single shared *memory*

- *objects* live in memory

Unpredictable asynchronous delays

- arbitrary speed for each thread

Interleaving

- arbitrary order for threads

Asynchrony means sudden, unpredictable delays

- Cache misses (*short*)
- Page faults (*long*)
- Scheduling quantum used up (*really long*)

Basic Sync Patterns

▪ Signaling

- a1 before b1

Thread A
statement a1

Thread B
statement b1

▪ Rendezvous

- a1 before b2 and b1 before a2.

Thread A
statement a1
statement a2

Thread B
statement b1
statement b2

- Signalling – requirement that if have multiple threads, one particular statement executes in one thread before another
 - E.g. Thread A, statement a1 must execute before statement b1, thread B

- **Rendezvous – requirement is that a certain order of execution is maintained for statements in different threads**
 - E.g. Thread A, statement a1 executes before statement b2, thread B AND that b1, thread B must execute before statement a2, thread A

Concurrency and Sync Problems

- **Biggest issue: concurrent access to shared data is prone to data races and other issues of data integrity**
 - Concurrent access to shared data may result in data inconsistency
- **Maintaining data consistency requires the correct sync mechanisms to ensure the orderly execution of cooperating processes**
- The synchronization mechanism is usually provided by both hardware and the OS
 - The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems
 - Need primitives themselves to be atomic

Synchronisation Mechanisms

- Solutions to sync problems
 - Techniques range from hardware to software-based API's
- **All based on protecting critical regions through use of locks**
 - Two processes cannot have a lock simultaneously

Classic Synchronisation Problems #1: Critical Section Problem (mutual exclusion)

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

- **Each process has critical section segment of code**
 - Process may be changing common variables, updating table, writing file, etc
 - Part of code that involves updating some shared state or using some shared resource
 - E.g. writing to shared memory
 - Need mechanism to control access to this section
- Simplest solution is using a mutex – i.e. a *mutual exclusive*
 - **Need to ensure there's mutually exclusive access to that section of code**
- Critical section is mutually exclusive – when one process in critical section, no other may be in its critical section
 - Guarantees there won't be data races and other conditions occurring
- Critical section problem is how to design a protocol to solve this
 - Want to design a consistent protocol that allows for peak performance
- Theoretical perspective:
 - Diagram shows loop – process iteratively going through a computation
 - Part of the computation involves critical section (want to protect)
 - Part that doesn't need to be protected is the remainder section
 - Also have the entry and exit section – entry is where acquire means to control access to critical section
 - Exit section is where you give up control

Requirements for Solutions to Critical Section problem

- Solution must satisfy:

- **Mutual Exclusion** – if process P_i is in its critical section, no other processes can be in their critical sections
- **Progress** – if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
 - Basically, any process that wishes to enter its crit section should be allowed to compete and should be allowed to make that decision in a reasonable amount of time
- **Bounded Waiting** – there exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the n processes
 - Once process requests access to crit section – should receive access within a reasonable amount of time

Solving Critical Section Problem with Locks

For critical section problem:

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
```

- Only one process can acquire the lock at a time
- When the process has the lock, it can enter its crit section and perform its computation
 - No other process can enter its critical section at that time because they don't have the lock
 - Have to wait until other process releases the lock
- Once the lock is released, one of the other processes that was forced to wait can acquire the lock and execute its crit section

Hardware Solutions to Synchronisation

test-and-set()

```
int test_and_set (int *target)
{
    int rv = *target;
    *target = TRUE;
    return rv;
}
```

- Hw provides special atomic hw instructions to implement locks
- **test-and-set():** combines the actions of testing variable and setting it to a specific value into a single machine instruction which cannot be interrupted
- Example with lock implementation:

```
int test_and_set (int *target)
{
    int *rv = target;
    *target = TRUE;
    return rv;
}
```

```
do {
    while (test_and_set(&lock)); /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

- test-and-set on LHS
- RHS shows critical section is executed – once finished, the lock is released

compare-and-swap()

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

- Hw provides special atomic hw instructions to implement locks
- **compare-and-swap():** swaps the value of a variable only if it is equal to a given expected value
- Example with lock implementation:

```
int compare_and_swap(int *value, int
expected, int new_value)
{
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
}
```

- Lock initially set to 0/false – if value matches expected value, can grab the lock

- Lock assumes values 1 in that case
 - Return original value of lock, which is 0
- Once lock grabbed, can execute crit section
 - Any other process trying to grab lock will fail
- Once crit section completed, set lock to zero – allows other program waiting to enter its critical section
- Note: these approaches do not satisfy “bounded waiting” – a solution in terms of CAS is provided in textbook
 - Guarantees at most n-1 waiting turns for n threads

Atomic Variables

```
void increment(atomic int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != compare and swap(v, temp, temp+1));
}
```

- The compare-and-swap() hardware not usually used directly to provide mutual exclusion
 - **Used to construct synchronization primitives such as atomic variables**
- Updates to atomic variables are guaranteed to be atomic – no data races on a single variable
- E.g. might have variable x and want to atomically increment/decrement it some languages have direct support for this through hw
 - Guarantee with atomic increment on variable, that nothing can interrupt it
- Operation sequence for increment involves several registers – those can be interrupted by interleaving of instructions from different processes
 - **Making an operation atomic allows a variable to be safely incremented without having to worry about operations being interleaved or inconsistent data (due to data races)**
- Diagram shows example of atomic increment
- Hardware solutions are complicated and generally inaccessible to application programmers, so OS designers build software tools to solve critical section problem

Software Solutions to Synchronisation

Mutex Lock

```
acquire() {
    while (!available);
    /* busy wait */
    available = false;
}
```

```
release() {
    available = true;
}
```

For critical section problem:

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```

- Simplest is the **mutex lock, which has an associated Boolean variable “available” to indicate if the lock is available or not**
 - Two operations to access - acquire() and release()

- **Calls to acquire() and release() are atomic**- usually implemented via hardware atomic instructions
 - Can't be interrupted
- **Solution requires busy waiting – lock is therefore called a spinlock**
 - Wasteful – CPU core runs tight while loop, constantly checking the state of this variable
 - Not doing any productive work
 - Busy waiting seen as bad (from performance pov) – BUT if crit sections are short, then amount of time spent in crit section will be small
 - Thus, amount of time spent waiting for lock to be freed up is small
 - With multiple CPUs with multiple cores, could have process spinning on spinlock on one core, other cores occupied by other process
 - But issues can occur when only have single core
- Diagram – while in crit section, available is false
 - Once done, set available to true
 - These operations are atomic

Semaphores

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S = S - 1;
}
```

```
signal(S) {
    S = S + 1;
}
```

For critical section problem:

Create a semaphore "synch" initialized to 1

```
wait(synch)
critical section
signal(synch);
```

- More robust and sophisticated synchronization tool (than mutex locks)
 - **Generalisation of mutex locks** – but allow for more sophisticated processing
- Semaphores are way of signalling what's going on to system
- Semaphore S – integer variable accessed only via two atomic operations:
 - wait() and signal()
 - Originally called P() and V()
 - In Java, called acquire() and release()
- Diagram
 - Wait(s) – while s < 0, spin around and do nothing
 - If S positive, exit loop and decrement semaphore value
 - Signal(s) – takes semaphore value and decrements it by 1
 - *Implementing mutex lock with semaphores?*
 - **Create semaphore with int value 1, and call wait() on it**
 - **When initialised, process comes in, executes wait method**
 - Decrements semaphore value – becomes 0, is able to enter its crit section
 - Will raise signal afterwards to *increase semaphore value to 1*
 - While it's in its crit section, S = 0, so no other process can enter its critical section
 - If they call wait, they will be kept in busy waiting state – looping around

- Once original process signals that its finished, only one other process is then able to grab the semaphore, decrement the value and enter its crit section (blocking other processes)
- When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are value test, increment and decrement
 - Decrement – a thread can only decrement the semaphore if it is > 0
 - Otherwise the thread blocks itself and cannot continue until another thread increments the semaphore
 - Increment – when a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked
- In the original definition you cannot read the current value of the semaphore
 - BUT Java will let you do this

Types of Semaphores

- Binary semaphore** – integer value can range only between 0 and 1
 - Same as a mutex lock
 - Generally initialised to 1
- Counting semaphore** – integer value can range over an unrestricted domain
 - Used to control access to a restricted number of processes
 - Some value $n > 1$
 - Place resource limit on the # of processes that can simultaneously enter their critical section
 - Multiplex

Example

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S = S - 1;
}
```

```
signal(S) {
    S = S + 1;
}
```

e.g. To allow at **most two** processes to execute in a critical section :

Create a semaphore “synch” initialized to 2

```
wait(synch)
critical section
signal(synch);
```

- If the value of the semaphore is positive, then it represents the number of threads that can decrement without blocking
- If the value is zero, it means there are no threads waiting, but if a thread tries to decrement, it will block
- NOTE – if it is negative (permitted under the non-busy-waiting implementation) then the value represents the number of threads that have blocked and are waiting

Semaphore Implementation

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

- Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time (atomic) – usually done with hardware locks

- Also, want to avoid busy waiting - although this is not a problem if critical section, and hence waiting time, is short, also busy waiting avoids a context switch
 - Modify wait() so that process blocks (suspends) and is put in waiting queue
 - CPU scheduler chooses another process to execute
 - Each semaphore now has two data items:
 - Semaphore value (of type integer)
 - List of processes in waiting queue
- Requires *two additional operations*:
 - sleep()
 - Required by wait() – suspend invoking process (the processes must also be added to waiting queue)
 - wakeup(P)
 - Required by signal() – remove a process P from the waiting queue, call wakeup(P)
 - Scheduler can then dispatch it again (it is now ‘runnable’)

Basic Sync Patterns with Semaphores

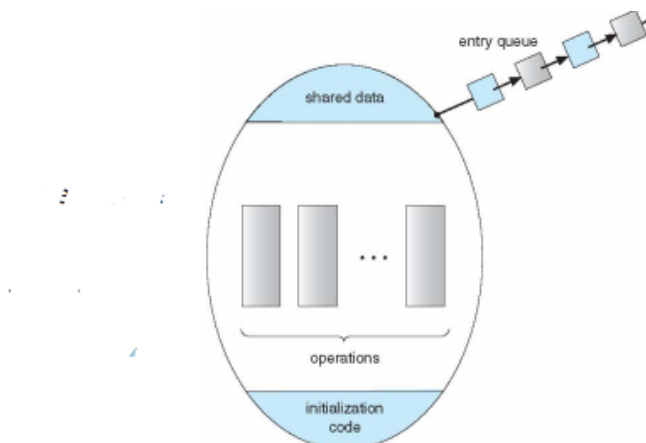
<ul style="list-style-type: none"> Signaling <ul style="list-style-type: none"> a1 before b1 	Thread A statement a1	Thread B statement b1
<ul style="list-style-type: none"> Rendezvous <ul style="list-style-type: none"> a1 before b2 and b1 before a2. 	Thread A statement a1 statement a2	Thread B statement b1 statement b2
<ul style="list-style-type: none"> Signaling <ul style="list-style-type: none"> a1 before b1 	Thread A statement a1 sem.signal ()	Thread B sem.wait () statement b1
<ul style="list-style-type: none"> Rendezvous <ul style="list-style-type: none"> a1 before b2 and b1 before a2. 	Thread A statement a1 aArrived.signal () bArrived.wait () statement a2	Thread B statement b1 bArrived.signal () aArrived.wait () statement b2

- Signalling:
 - Thread B – have semaphore initialised e.g. to 0
 - Statement a1 has to execute first since semaphore can only be passed once it's value is >1, so its value can be decremented down to zero
 - Can only happen once signal() increments sem to 1
 - Once incremented, b1 can execute
- Rendezvous:

- aArrived and bArrived initialised to 0
- a1 and b1 has no constraints – can execute in interleaved fashion
- a1 executes, then b1 (or other way around)
- Then one of these will run its signal first on a particular semaphore
- E.g. aArrived does it first – semaphore goes up to 1
 - Then tries to execute wait – has to wait because value of bArrived is still 0
 - Assume bArrived executes at some point – semaphore value increases to one
 - bArrived wait() can execute now
 - Then a2 will be executed
 - Similarly, correct sequence of operations ensure that thread B's statements will execute correctly

Monitors

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    procedure Pn (...) { ... }
    initialization code (...) { ... }
}
```



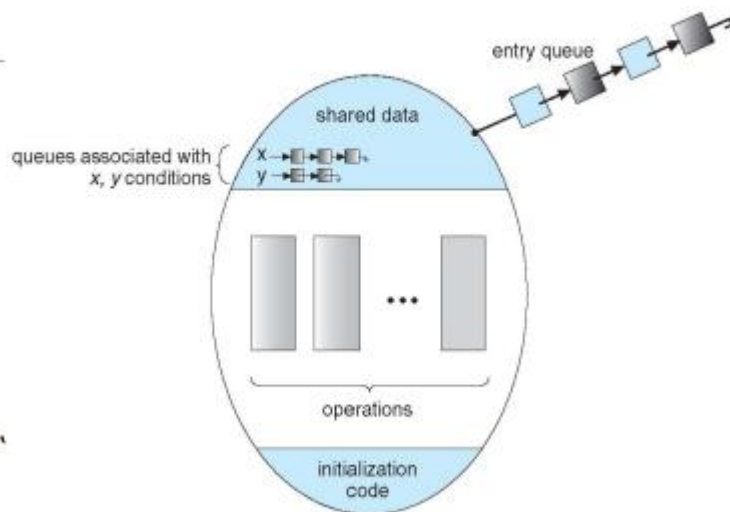
A monitor is essentially a class in which all data is private, and where only one method within any given monitor object may be active at the same time.

Monitor methods may only access the shared data within the monitor and any data passed to them as parameters. i.e. they cannot access any data external to the monitor.

- Alternative to semaphores
- High-level abstraction that provides a convenient and effective mechanism for process synchronization
 - Using semaphores is prone to error – can put wait() and signal() in incorrect place or swap them around, refer to wrong semaphore with wait() or signal() etc
 - Leads to hard-to-detect semaphore bugs
 - Can alternatively wrap sync behaviour in abstract data type – monitor

- Abstract data type – internal variables only accessible by code within the procedure
- **Monitor is collection of data and series of methods that are guaranteed exclusive access to that data**
 - Guarantee that only one of those methods can access those methods at any one time
 - Java has monitor data type
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes
 - Need some way to signal between different methods that some condition has changed and now another method is allowed to start operating on monitor
 - Achieved via condition variables

Condition Variables



- Have condition x
- Two operations are allowed on a condition variable:
 - (1) `x.wait()`
 - A process that invokes the operation is suspended until `x.signal()`
 - (2) `x.signal()`
 - Resumes one of processes (if any) that invoked `x.wait()`
 - If nothing waiting on the variable, then it has no effect on the variable

Basic synchronization patterns II

- Mutual exclusion

(already discussed)

- Multiplex

critical section with n threads at a time

how do we do this with semaphores?

- Barrier



- Reusable Barrier

Classic Problems of Synchronisation

Producer-Consumer Problem

- In multithreaded programs there is often a division of labour between threads
- In one common pattern, some threads are producers and some are consumers
 - Producers create items of some kind and add them to a data structure;
 - Consumers remove the items and process them
- E.g. Event-driven programs
 - Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer
 - Concurrently, consumer threads (“event handlers”) take events out of the buffer and process them
 - Event handlers
- Constraints to be enforced by synchronization:
 - While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer
 - If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item
 - In bounded buffer variant: producer cannot produce if buffer of BUFFER_SIZE is full

Pseudocode

Producer

```
while (true) {  
    /* produce item in next_produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumer

```
while (true) {  
  
    while (counter == 0) ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    counter--;  
    /* consume item in next=_consumed */  
}
```

buffer is implemented as a circular array, with two logical pointers, in and out: in points to next free position in buffer, out points to first full position.

May not function correctly when executed concurrently.

Producer-Consumer

- Pseudocode shows there's nothing to provide mutex access to parts where buffer needs to be updated, as well as updating the counter
 - These parts need to be protected to prevent inappropriate access which would lead to data inconsistencies

With Semaphores

- Shared variables:
 - `int n;`
 - Size of the buffer
 - Semaphore `mutex = 1` – binary semaphore
 - Ensures mutex access to crit section
 - Semaphore `full = 0`
 - Counting semaphore
 - Provides the # fully occupied buffer slots that consumer can pull data from
 - Semaphore `empty = n`
 - Counting semaphores
 - Provides # of empty slots for producer to write into

Semaphores Pseudocode

Producer

```
do {  
    ...  
    /* produce item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
  
    ...  
    /* add next_produced to buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

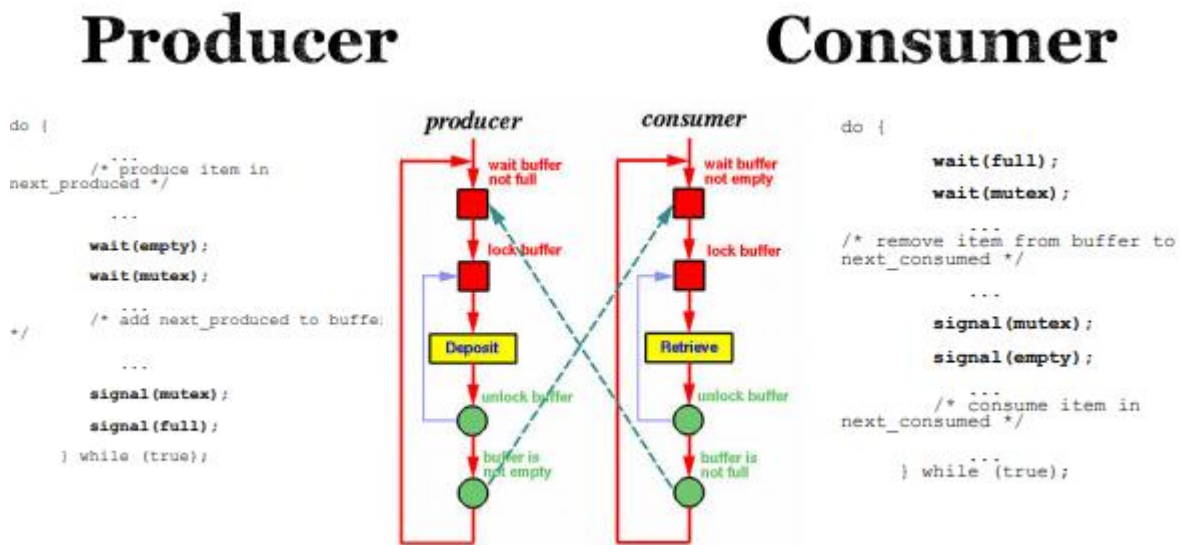
Consumer

```
do {  
    wait(full);  
    wait(mutex);  
  
    ...  
    /* remove item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
  
    ...  
    /* consume item in next_consumed */  
    ...  
} while (true);
```

- Only one process can have mutex at a time – once mutex grabbed, either p or c has it
 - Can then do updates
- Producer waits(empty)
 - Number of slots available for it to write on
 - Can immediately pass through because all buffer slots are available
 - Waits until it can grab mutex, once it has
 - It updates, puts item in buffer
 - Then signals an increase in full
- Consumer waits(full) i.e. there is something there for it to consume
 - If item in the buffer, it can then pass by wait and decrement full to 0
 - Grabs mutex to do its update
 - Signals that there's one more slot available to write info into

- Producer now has more space to insert things in

Diagram



- Producer waits until buffer isn't full – tries to get mutex lock
 - Once it grabs it, it can safely add to buffer and do its updates
 - Can then release the mutex

Aside: Producer Consumer with Monitors

```
public class Buffer {
    private int[] rack;
    private int in, out;
    private int RACK_SIZE;
    private int counter;
```

```
    Buffer(int rackSize) {
        this.rack = new int[rackSize]; //
        this.in = 0;
        this.out = 0;
        this.RACK_SIZE = rackSize;
        this.counter = 0;
    }

    public synchronized void addWidget(int widget_id) throws InterruptedException {
        if (counter >= RACK_SIZE) {this.wait();}
        rack[in] = widget_id; // add widget to buffer
        in = (in + 1) % RACK_SIZE; //circular buffer
        counter++;
        System.out.println("added widget #" + widget_id + " to buffer.");
        this.notifyAll();
    }

    public synchronized int removeWidget() throws InterruptedException {
        if (counter == 0) {this.wait();}
        int next = rack[out];
        out = (out + 1) % RACK_SIZE; //circular buffer
        counter--;
        System.out.println("removed widget #" + next + " from buffer.");
        this.notifyAll();
        return next;
    }
}
```

Producer-Consumer with monitors

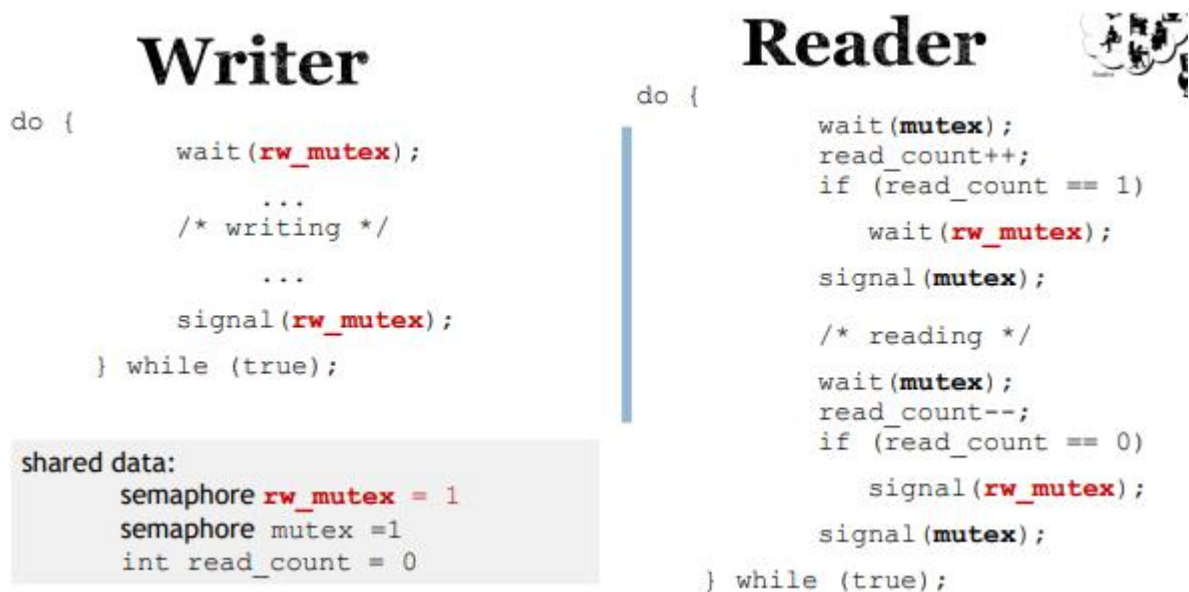
4

- Force each method in monitor class to be synced
 - So only one method can be in the monitor at any one time

Reader-Writers Problem

- Any situation where a data structure, database, or file system is read and modified by concurrent threads, some of which read and others write
 - Can't be reading when records are being modified
 - When writing into, no other thread should be writing at the same time
- As with Producer-Consumer, asymmetric solution (different code for readers and writers)
- Synchronization constraints:
 - Any number of readers can be in the critical section simultaneously
 - Use counting semaphore
 - Writers must have exclusive access to the critical section
- "Categorical mutual exclusion" – have two different categories of thread and we treat them differently
- Several variants of how readers and writers are considered – all involve priority
 - First variation – no reader kept waiting unless writer has permission to use shared object
 - Favours readers – unless writer in crit section, should get readers in asap
 - Once in, all readers can pile in – writer has to wait until all readers are finished
 - Second variation – once writer is ready, it performs the write ASAP

Solution with Semaphores



- Read_count refers to # of readings reading crit section
- Rw_mutex – can be grabbed by writer/reader
 - Allows for exclusive access to that crit section
- Both are binary mutexes
- Reader mutex
 - Protects crit section for reader only – once read_count == 1, grabs rw mutex
 - Then releases reader mutex (NOT rw mutex)
 - Another thread can come in, grab mutex and put read_count up further, etc
 - Waiting process can be starved i.e. producer
 - Once reader acquires rw mutex, can commence reading, then update state data
 - Once crit section exited, frees up the mutex

- Not the best solution

Issues with Synchronisation

- Liveness – processes must make progress, not wait indefinitely
 - Must guarantee this when using sync primitives
- Starvation – indefinite blocking
 - e.g. process may never be removed from the semaphore queue in which it is suspended
- Deadlock – two or more processes are waiting infinitely long for an event that can never occur because can only be caused by one of the waiting processes

E.g. Let S and Q be two semaphores initialized to 1

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>...</code>	<code>...</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Dining Philosophers

Philosophers spend their lives alternating between thinking and eating

- Don't interact with their neighbors,
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - Need both to eat, then release both when done
- In the case of 5 philosophers
 - Shared data
 - Bowl of rice (data set)
 - Semaphore **chopstick [5]** initialized to 1
- Create an array of 5 binary semaphores – once chopstick grabbed can't be grabbed by another philosopher
 - Once finished, the philosopher releases it again

```
while True :
    think ()
    get_chopsticks ()
    eat ()
    put_chopsticks ()
```

With Semaphores

solution with **semaphores** Philosopher

```
do {  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
  
    // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    // think  
} while (true);
```

What is the problem with this algorithm?

- Have semaphores – call wait until we get the left and right chopstick
 - Then eat – once done, call signal() to release system resources
 - Put down left then right chopstick
 - Mod operation is because the table is circular (last philosopher grabs the first chopstick) – using circular buffer

Deadlock

```
while True :  
    think ()  
    get_chopsticks ()  
    eat ()  
    put_chopsticks ()
```

- Standard solutions prone to deadlock
 - Can make a dining-philosophers specific solution – then won't be generalisable
 - Instead want to develop set of methodologies/ tools that can work with any deadlocking system – to avoid or recover from them if necessary
- Several solutions:
 - Allow at most 4 philosophers to be sitting simultaneously at the table (with same number of chopsticks)
 - (deadlock then impossible)
 - Allow a philosopher to pick up the chopsticks only if both are available (picking up must be done in a critical section)
 - Makes acquiring atomic
 - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick

- Even-numbered philosopher picks up first the right chopstick and then the left chopstick
 - Preferred solution
- Also issue of starvation...

Deadlock

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>...</code>	<code>...</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

- Can occur in Java code – locks/semaphores acquired in opposite order, 'accidentally'
- Deadlock is possible, but may not occur
- Diagram - example illustrates a problem with handling deadlocks:
 - Difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances

System Model

- Represents OS
- System comprises:
 - A finite number of resources
 - CPU cycles, files, and I/O devices
 - Also sync tools such as mutex locks and semaphores
 - Distributed among a number of competing threads
 - Want threads to have fair access to these resources
- Under this simplified system model, a thread may use a resource in only the following sequence:
 - Request – if request can't be immediately honoured, thread must wait
 - Use – once acquired, often use exclusively
 - Release – so that other threads competing for resource can access it
- A set of threads is in deadlock when every thread in the set is waiting for an event that can be caused only by another thread in the set
 - OS system kernel – have many threads that are potentially "scheduleable" and are competing for resources
- Sync tools are the most common sources of deadlock

Safety vs Liveness

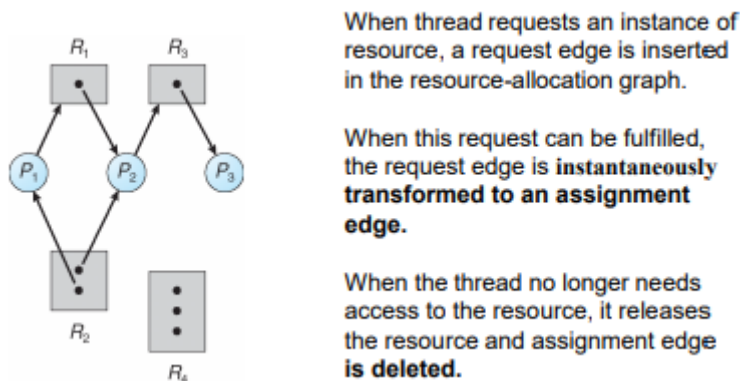
- There are two types of correctness properties for concurrent programs:
 - Safety properties – the property must always be true
 - E.g. acquiring a semaphore to get mutex access to crit section; using mutexes/semaphores to ensure specific order of operation
 - Liveness properties – the property must eventually become true
 - Weaker constraint
 - Liveness failure – most exemplified by deadlock
 - Livelock – have set of processes unable to make progress at all

- Spin around doing no useful work
- E.g. acquire and release lock – doing something, but it's not productive
- Solutions to safety issues often raise the possibility of liveness issues

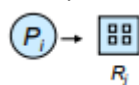
Deadlock Characterisation

- Four necessary conditions must be satisfied simultaneously for deadlock to occur:
 - Mutual exclusion- only one process at a time can use a resource
 - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
 - No pre-emption – a resource can be released only voluntarily by the process holding it, after that process has completed its task
 - Circular wait – there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0

Resource Allocation Graph

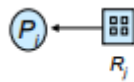


- A set of vertices V and a set of edges E
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - i.e. all threads
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
 - Resource nodes – different # of resource nodes vs number of threads/process in system
- Edge of two types:
 - Request edge – directed edge $P_i \rightarrow R_j$
 - Have process P_i making a request from the resource node i.e. a resource held by the node
 - Have different resource categories – could be requesting mutex lock or semaphore, file system handle, etc



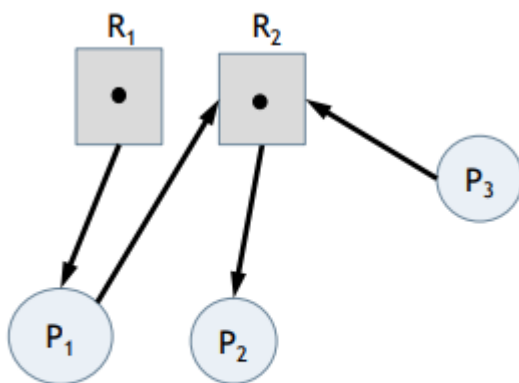
- Assignment edge – directed edge $R_j \rightarrow P_i$

- Edges from resource to process



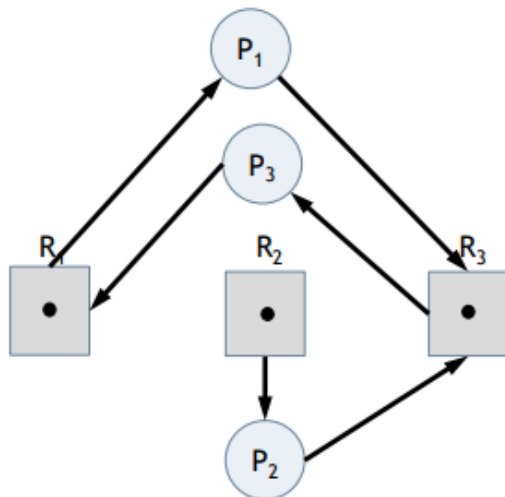
- Diagram:
 - Have P1 requesting resource from resource pool R1, but R1 is allocated to P2
 - Similarly, for P2, R3 and P3
 - P1 and P2 requesting from R2
 - Snapshot of resource request and allocation in a system
- If graph contains no cycles => no deadlock
- If graph contains a cycle:
 - If *only one* instance per resource type, then deadlock exists
 - If several instances per resource type, then possibility of deadlock
 - This is the case with the diagram above
 - Possible to break cycle and work out process of sequence execution such that process can finish its task, free up its resources and honour resource request edges from other processes

Checkpoint



Is there a deadlock here?

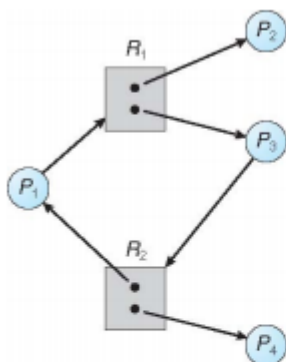
- No, there is no cycle with a closed loop



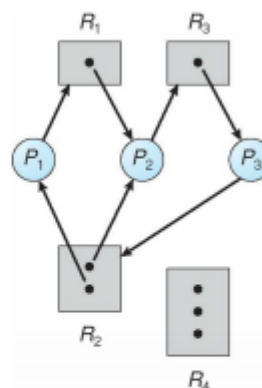
Is there a deadlock here?

- Yes – P1 to R3 to P3 to R1 to P1
 - All nodes participating in cycle and only one resource available for each resource node in that cycle

Resource Allocation Graphs With a Cycle



A) Is there a deadlock?



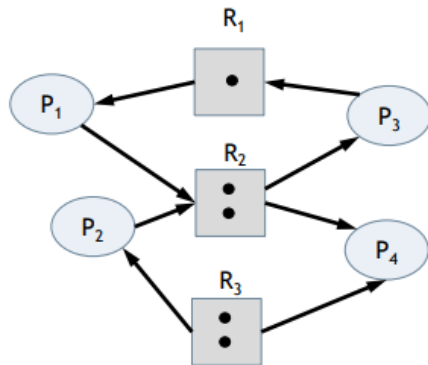
B) Is there a deadlock?

For those situations which are deadlock, provide the cycle of threads and resources. Where there is no deadlock, provide an order in which threads may complete execution.

- (a) Deadlock possible – but have more than one resource per node
 - Deadlock potential but not guaranteed
 - If P4 executed first, frees up resource to go back to resource pool R2
 - P3 can then acquire resource – becomes resource assignment edge from R3 to P3
 - Could then free up resource in R1 and R2 – P4 and P3 now finished, P1 can grab resource
 - Multiple solutions to avoid deadlock
- (b) Multiple cycles – all involve resource nodes with more than one resource element
 - Deadlock – no way to satisfy resource requests to break deadlock condition
- If have cycle and resource nodes have single resource, definitely deadlock

- But with multiple resource elements per resource node/vertex, then need to look at problem more carefully and see if there's a sequence of operations which can be performed which will allow for resources to be gradually freed up

Checkpoint: Resource Allocation Graphs



- Some cycles – P1 and P3
 - Potential deadlock cycle – but there is not deadlock
 - Can find execution path which can break the cycle by flipping an edge
 - Allow P4 to execute and will return resource to R3 and R2
 - Because R2 available, can then use resource in P1- becomes assignment edge (cycle broken)
 - P1 has resource from R1 and now R2, so it can complete – frees up resources back to R2 and R1

Handling Deadlock

- Pretend that deadlocks never occur in the system (ignore the problem)
 - E.g. most OS's, including Windows and Linux
 - Up to kernel and application developers to write programs that handle deadlocks
- Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state
 - Deadlock prevention – ensure that at least one of the necessary conditions cannot hold
 - All 4 conditions have to hold for deadlock to occur – if can get around one of them, then can prevent deadlock
 - Deadlock avoidance – OS requires advance information on which resources a thread will request
 - Additional info given to system when trying to acquire resources
 - System runs some algos, does some computations to detect if honouring that resource request is going to lead to deadlock in future
 - Could then deny request, put process on wait queue and try again later
 - Deadlock recovery – allow the system to enter a deadlocked state, detect it, and recover
 - E.g. dtbs – commit, rollback if there's a problem and then rollback to previous state

- Can roll-back to prior non-deadlocked state

OSes and Deadlock

- Why do OSes not handle deadlock?
 - Undetected deadlock is bad
 - Will cause the system's performance to deteriorate, because resources are held by threads that cannot run.... and more and more threads, as they make requests for resources, will enter a deadlocked state.
 - Eventually, the system will stop functioning and will need to be restarted manually
- So, why don't OS's handle deadlock?
 - Needs too much resources and has performance implications for a conventional OS
 - i.e., resource intensive
 - Rather assume it's a rare event and leaves it up to the programmers

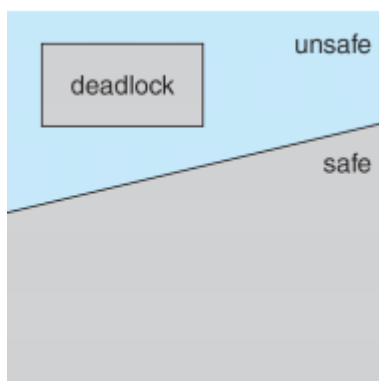
Deadlock Prevention

- Four necessary conditions must be satisfied simultaneously for deadlock to occur
 - Which of these can we remove?
 - To prevent deadlock from occurring
- (1) Mutual exclusion – only one process at a time can use a resource
 - Safety is an absolute requirement – removal cannot be used for prevention
- (2) Hold and wait – a process holding at least one resource is waiting to acquire additional resources held by other processes
 - Have beginnings of a circular dependency – have resource, but trying to grab another one
 - In grabbing another, doesn't necessarily free the one you're holding
 - Could say – can only request a resource, it's not allowed to hold any other resources
 - Or it needs to request all resources before it begins execution
 - Or allow it to request resources only when no other is allocated to it
 - All of these are impractical, bad solutions with performance hits
 - Removal impractical – low resource utilization
 - Starvation possible
- (3) No pre-emption – a resource can be released only voluntarily by the process holding it, after that process has completed its task
 - Removal impractical for most systems – performance implication for constantly interrupting threads in order to release their resources
 - Sync primitives used to avoid/control deadlock can't be interrupted
 - i.e. atomic operations protecting crit sections
- (4) Circular wait – there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0
 - Create total ordering of resource types i.e. sequential ordering of resources
 - Processes request resources – walk through sequence from start to end, can find a way of avoiding it
 - Use resource graphs

Deadlock Avoidance

- Ensure that the system will never enter a deadlock state
 - Essentially force all threads in system to declare their resource requests upfront
- Requires additional a priori information available on possible resource requests
 - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
 - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
 - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State Concept



- System is in safe state if there exists a sequence of ALL the processes in the system, such that for each P_i the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
 - Sequence that is determined must allow us, for any particular entry in the sequence, must be able to satisfy the requests of that thread using only the free, available resources at that time AND the resources held at that time by a lower index
 - E.g. have 5 threads in system, and we're working with thread 3 T3
 - When looking at T3's request to see if it can be honoured, we must look at all available resources, as well as those held by T1 and T2
 - If this property holds for all of the P_i in the sequence that was determined, then can't enter circular wait state and thus can't deadlock
- That is:
 - If P_i resource needs cannot be met now, then P_i can wait until all processes P_j ($j < i$) have finished executing
 - When they have finished executing, they release all their resources and then P_i can obtain the needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its required resources, and so on
 - When predecessor threads finish – they return allocated resources to pool and thus allows next process in sequence to grab those and execute
 - Execution may be deferred for a short period BUT still be able to execute – preferable to deadlock (have no control over resource requests)

- There are some performance hits – but can work out sequence of thread execution (such that conditions hold), then can guarantee circular wait can't occur
- If a system is in safe state => no deadlocks
 - i.e. obeys above properties -
- If a system is in unsafe state => possibility of deadlock
 - i.e. can't find sequence of thread execution (and resource deallocation when thread terminates)
 - Deadlock *not* guaranteed but possible under certain conditions

Example

Consider a system with 12 resources and 3 threads: T_0 , T_1 , and T_2

at t_0 in **safe state**:

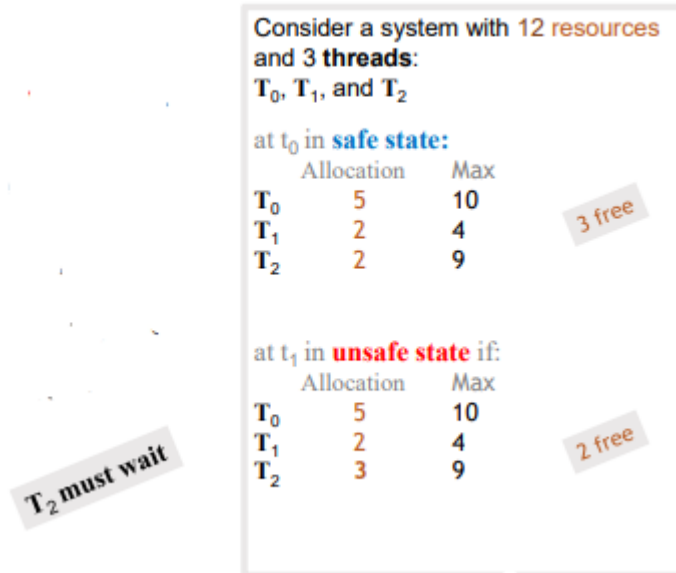
	Allocation	Max
T_0	5	10
T_1	2	4
T_2	2	9

at t_1 in **unsafe state** if:

	Allocation	Max
T_0	5	10
T_1	2	4
T_2	3	9

- Have system consisting of 12 resources divided up amongst 3 threads T_0, T_1, T_2
- Taking snapshots at two different times – want to determine if that config of threads is in a safe state/not
- At time 0:
 - T_0 – 5 resources
 - T_1 – 2
 - T_2 – 2
 - 9 resources out of 12 – 3 resources free
 - In a safe state
 - T_0 – can still grab 5 resources (only 3 avail)
 - Can't safely execute T_0 first
 - Can execute after T_1 – T_0 grabs 2, now have 5 resources free
 - Grabs 5 resources so it can execute
 - T_1 – has 2, can grab 2
 - Can execute T_1
 - T_2 – has 2 but could claim 7 more
 - Executes after T_0 – have 8 resources in system
 - $7 < 8$ so can execute
- At time 1 – shows that i.t.o total # of allocated resources, have 10 (out of 12)
 - Only 2 free
 - Unsafe state
 - Can't honour T_0 or T_2 – have to execute T_1

- If T1 executes and returns 2 resources, now only have 4 resources in resource pool
 - Can't do anything else – T2 needs 6 and T0 needs 5
- Ensure that a system will never enter an unsafe state

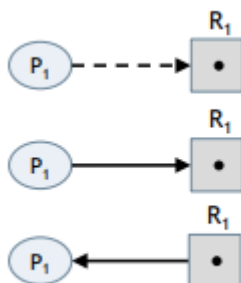



- Whenever a thread requests a resource that is currently available, the system decides whether the resource can be allocated immediately or the thread must wait
 - The request is granted only if the allocation leaves the system in a safe state
 - If a thread requests a resource currently available, may have to wait
 - Resource utilization may be lower than it would otherwise be

Deadlock Avoidance Algorithms

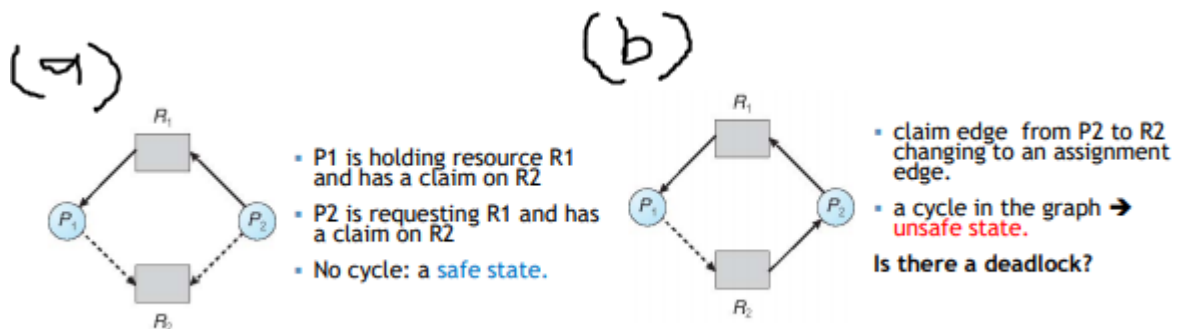
- For:
 - Single instance of a resource type – use a variant of the resource-allocation graph
 - Multiple instances of a resource type – use the Banker's Algorithm

Resource-Allocation-Graph Algorithm



- For single instance of a resource type 
- Each process must a priori claim maximum resource use

- Use a variant of the resource -allocation graph with claim edges
 - Claim edge $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
 - Represented by a dashed line
 - Claim edge converts to request edge when a process requests a resource
 - Request edge converted to an assignment edge when the resource is allocated to the process
 - When a resource is released by a process, assignment edge reconverts to a claim edge
 - May want to claim that resource again at a later point in execution
 - A cycle in the graph implies that the system is in unsafe state
- Suppose that process P_i requests a resource R_j
 - The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
 - Otherwise, the process must wait
 - Check for safety with a cycle-detection algorithm
 - Order of n^2 operations, where n is the number of threads in the system
 - Expensive



- (a) No cycle in this state – look at thread requesting resource
 - Only grant resource if converting request edge to assignment edge doesn't form a cycle
 - No deadlock
- (b) P_2 granted the resource on R_2 – now have a cycle
 - Unsafe state
 - No deadlock because P_1 has not yet requested R_2 – P_2 could complete, free up R_2 (no longer a request edge – becomes claim edge)
 - P_1 could then release R_1 and then P_2 could then claim and complete its execution
 - Deadlock potential – but not guaranteed
- By forcing things to avoid unsafe states, impose performance penalties because threads end up waiting more because can't grant certain resource requests
 - Preferable to allowing deadlock

Bankers Algorithm

Data structures (n - # threads, m - # resource types)

- **Available:** Vector of length m .
 - If available $[j] = k$, k instances of resource type R_j are available
 - **Max:** $n \times m$ matrix.
 - If $Max[i, j] = k$, P_i may request at most k instances of resource type R_j
 - **Allocation:** $n \times m$ matrix.
 - If $Allocation[i, j] = k$, P_i currently allocated k instances of R_j
 - **Need:** $n \times m$ matrix.
 - If $Need[i, j] = k$, P_i may need at most k more instances of R_j to complete its task.
 - $Need[i, j] = Max[i, j] - Allocation[i, j]$
-
- Multiple instances of a resource type
 - Less efficient than resource-allocation graph
 - Each process must a priori claim maximum use
 - When a process requests a resource, it may have to wait
 - At every step, there is two phases:
 - (1) Check if we're in a safe state
 - (2) If there's a new resource request – can we grant it without moving into an unsafe state?
 - If both there are guaranteed, then deadlock won't occur
 - When a process gets all its resources it must return them in a finite amount of time
 - Have to define a series of data structures (vectors and matrices) that represent info about the current state of the system and the future request state of the various threads
 - Available – length m (# of resource types)
 - How many resources are available to be grabbed by different threads to use?
 - Unallocated
 - Max – rows n (per thread) x columns m (per resource type)
 - Each row corresponds to a different thread
 - Number of instances of a particular resource type that thread may request
 - i.e. over all time, thread i can't request more than k of resource j
 - Allocation – $n \times m$
 - Thread i (row) is currently allocated k instances of R_j
 - Each row will be for a particular thread
 - Need – how much does a thread need of each resource to complete its task

Safety Calculation

To calculate whether system is in a safe state

1. **Work** and **Finish** are vectors of length m and n , respectively. Initialize:
Work = Available
Finish[i] = false for $i = 0, 1, \dots, n - 1$
2. Find an i such that both:
(a) **Finish[i] = false**
(b) **Need_i ≤ Work**
If no such i exists, go to step 4
3. **Work = Work + Allocation_i**
Finish[i] = true
go to step 2
4. If **Finish[i] == true** for all i , then the system is in a safe state.
Otherwise, in an unsafe state.

may require on order $m \times n^2$ operations to determine whether a state is safe.

- Determines if we're in a safe state/not
- Work set to available set of resources i.e. amount of resources system has available
 - Can add to/ remove from as the algo executes
 - M elements per resource type
- Finish – Boolean vector; per thread
 - Whether/not thread has completed its work i.e. has executed
 - Set to false for every thread
- Just want to search through list of threads and find an ordering of threads such that they can be executed in that order, whilst ensuring always being in a safe state
 - Scan list of available threads, looking at their resource needs, based on amount of resources available
 - Find a thread that hasn't completed, whose needs are less than the current resources available in the system
 - If can find that thread, then commit those resources to it
 - Go to step 3 – add the amount of resources needed for that thread to continue executing to current running total of all available system resources (because it has executed and completed)
 - Those resources are returned to the system and flag that thread as completed
 - Then search for additional threads to commit resources to
 - If can't find such a thread – know there's a problem
 - Go to step 4 and finish[i] will be false for some i – know system is in an unsafe state
- Ideally want to find complete sequence that visits all threads in some order, such that when step 4 is reached, finish is true for all threads in the sequence
 - Thus, have executable sequence that guarantees no deadlock
- Computationally expensive
 - May require on order $m \times n^2$ operations to determine whether a state is safe

Example

Example: safety calculation

Consider a system with 3 resource types - A (10 instances), B (5), and C (7)
- and 5 threads: T_0, \dots, T_4

1. $Work = Available$
 $Finish[i] = false$ for $i = 0, 1, \dots, n-1$
 2. Find an i such that both:
 (a) $Finish[i] = false$
 (b) $Need_i \leq Work$
 If no such i exists, go to step 4
 3. $Work = Work + Allocation_i$
 $Finish[i] = true$
 go to step 2
 4. If $Finish[i] == true$ for all i , then safe state. Otherwise, in an unsafe state.

Work 3 3 2

Finish

0 false

1 false

2 false

3 false

4 false

at t_0 :									
	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	4	3	3	3	2
T_1	2	0	0	1	2	2			
T_2	3	0	2	6	0	0			
T_3	2	1	1	0	1	1			
T_4	0	0	2	4	3	1			

- i.e. T_0 allocated 0 As, 1 Bs and 0 Cs (and so forth)
- Sum across column for A, B and C
 - Have 7 As allocated – with 10 max in system \Rightarrow 3 As available to be used by threads in their resource allocation requests
 - Calculate Bs = 3, Cs = 2
 - So have work = 3 3 2
- Is there an ordering of threads such that can look at its current need and see if its less than current available work
 - Need matrix = Max – Allocation
 - Examine each row as it corresponds to different thread
- T_0 does not match criteria- move to T_1
 - T_1 does match criteria as $1\ 2\ 2 < 3\ 3\ 2$
 - Assume T_1 executed – free its resources back up
 - So, 2 0 0 added onto running total of available system resources i.e. work
 - Flag T_1 to be true
- Available resources now 5 3 2
 - Look at list – scan from top
 - T_0 = no, T_1 processed, T_2 = no – land at T_3 which meets criteria
 - $5\ 3\ 2 + 2\ 1\ 1 = 7\ 4\ 3$
 - Flag T_3 true
- Available resources now 7 4 3
 - Look at list – scan from top
 - T_0 now meets criteria
 - Add 0 1 0 to 7 4 3 = 7 5 3
 - Flag T_0 true
- Available resources now 7 5 3
 - T_2 fits criteria
 - Add 3 0 2 to 7 5 3 = 10 5 5
- Available resources now 10 5 5

- Only have T4 left – can satisfy that
- 4 3 1 < 10 5 5
- Add 0 0 2 + 10 5 5 = 10 5 7 (all system resources)
- The system is in a safe state as the sequence < T1, T3, T0, T2, T4 > satisfies safety criterion

Resource-Process Request for Process P_i

Let $Request_i[...]$ be the request vector for process P_i .

$Request_i[j] = k$. Process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
 2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
 3. Pretend to allocate requested resources to P_i by modifying the state as follows:
 - $Available = Available - Request_i$
 - $Allocation_i = Allocation_i + Request_i$
 - $Need_i = Need_i - Request_i$
 - If safe \Rightarrow the resources are allocated to P_i
 - If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored
- $Request_i[j] = k$ – particular thread/process wants k instances of resource type j
 - E.g. 3 0 2 \Rightarrow 3 As, 0 Bs and 2 Cs
 - Check if can meet that – if can, just apply request (i.e. mod data structures as if request valid)
 - Run safety detection algo to determine if we're left in a safe state
 - If safe – allocate resources to thread
 - Otherwise make thread wait

Example

Example: T₁ Request (1,0,2)

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait
3. Pretend to allocate requested resources to P_i :
 - $Available = Available - Request_i$
 - $Allocation_i = Allocation_i + Request_i$
 - $Need_i = Need_i - Request_i$
 - If safe \Rightarrow the resources are allocated to P_i
 - If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Check that $Request \leq Available$ ((1,0,2) \leq (3,3,2) \Rightarrow true)
 $Need = Max - Allocation$

at t_0 :

	Allocation	Max	Available
	A B C	A B C	A B C
T ₀	0 1 0	7 5 3	3 3 2
T ₁	2 0 0	3 2 2	
T ₂	3 0 2	9 0 2	
T ₃	2 1 1	2 2 2	
T ₄	0 0 2	4 3 3	



at t_1 :

	Allocation	Need	Available
	A B C	A B C	A B C
T ₀	0 1 0	7 4 3	3 3 2
T ₁	2 0 0	1 2 2	
T ₂	3 0 2	6 0 0	
T ₃	2 1 1	0 1 1	
T ₄	0 0 2	4 3 1	

The system is in a **safe state** since the sequence < T₁, T₃, T₄, T₂, T₀ > satisfies safety criterion

- Have previous state with resources available, max and needs
- T1 request (1,0,2)
 - Is request < need i.e. 1 0 2 < 1 2 2? Yes!

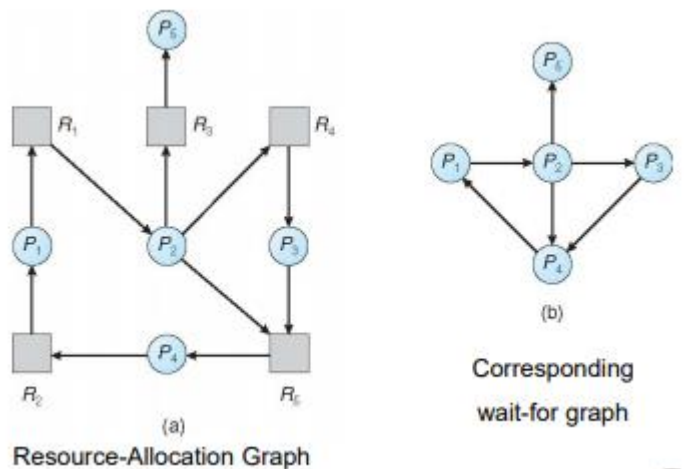
- Is request < available resources i.e. $1\ 0\ 2 < 3\ 3\ 2$? Yes!
- Update structures to see if going to be in safe state
 - Available = available – request i.e. $3\ 3\ 2 - 1\ 0\ 2 = 2\ 3\ 0$
 - Allocation = allocation + request i.e. $2\ 0\ 0 + 1\ 0\ 2 = 3\ 0\ 2$
 - Need = Need – request i.e. $1\ 2\ 2 - 1\ 0\ 2 = 0\ 2\ 0$
 - Run safety algo
 - The system is in a safe state since the sequence < T1, T3, T4, T0, T2> satisfies safety criteria.
 - Request is therefore granted
- Request (3,3,0) by T4
 - Need T4 is $3\ 3\ 0 < 4\ 3\ 1$ so request < need
 - BUT $3\ 3\ 0 > 2\ 3\ 0$ so cannot grant this
- Request (0,2,0) by T0
 - Need T0 is $0\ 2\ 0 < 7\ 4\ 3$ so request < need
 - $0\ 2\ 0 < 2\ 3\ 0$ so request < available resources
 - So mod vectors
 - Available = $2\ 3\ 0 - 0\ 2\ 0 = 2\ 1\ 0$
 - Allocation = $0\ 1\ 0 + 0\ 2\ 0 = 0\ 3\ 0$
 - Need = $7\ 5\ 3 - 0\ 3\ 0 = 7\ 2\ 3$
 - Run safety check – result is unsafe state
 - Restore old resource info – go back to previous request
 - Force T0 to wait

Deadlock Recovery

- Allow system to enter deadlocked state and then recover
- Requires:
 - Deadlock detection
 - Detection algorithms for
 - Single instance of a resource type
 - Multiple instances of a resource type
 - Recovery scheme – so system can recover and continue to do useful work

Deadlock detection

Single instance of each resource type



- Maintain a wait -for graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- System needs to maintain the wait-for graph and periodically invoke an algorithm to search for a cycle in the graph
 - Cycle \Rightarrow deadlock
 - n^2 operations – n is the number of vertices in the graph
- System might elect to run deadlock detection algo periodically e.g. when CPU utilisation drops below a particular threshold
 - Calling the algo too frequently is costly (because of cycle detection)
 - Resource implication for detecting that deadlock has occurred

Multiple Instances of each resource type

- Use algorithm similar to Banker's algorithm
- Requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state
 - M – resources
 - N – threads

Aside: Java Thread Dumps

- Java does not provide explicit support for deadlock detection
- A thread dump can be used to analyse a running program to determine if there is a deadlock
 - Snapshot of the states of all threads in a Java application
 - Show locking information, including which locks a blocked thread is waiting to acquire
- To generate a thread dump of a running application, from the command line enter: `jstack -l`

Recovery from Deadlock

Process Termination

- Options:
 - Abort all deadlocked processes – may leave process(es) in an inconsistent state
 - E.g. stopping a file i/o before its done
 - Will also need a mechanism to recover resources
 - E.g. if you've acquired mutex and want to abort a process– need to ensure that the lock is again made available to system
 - Have to restart computations from scratch
 - Lost productivity
 - Abort one process at a time until the deadlock cycle is eliminated
 - In which order should we choose to abort?
 - Have to look at process priority, how many processes a thread is hanging onto, etc

Resource Pre-emption

- Forcibly remove resources from threads that're deadlocked, in order to break the deadlock
 - Again, there are implications
- Selecting a victim – minimize cost
 - Choose a thread to "steal resources from" which minimises impact on the rest of the system
- Rollback – return to some safe state, restart process for that state
 - Take process and remove its resources – in doing so, have interfered with its computation
 - Thus, need to backtrack to a point and restart thread where it can continue with computations and try and reacquire that resource again
 - Not a trivial task
- Starvation – same process may always be picked as victim, include number of rollback in cost factor
 - Due to victim selection criteria, could end up picking the same victim – thread will be unable to acquire any resources
 - May need to keep track of # of rollbacks – additional cost
 - If rollback too many times, then won't steal resource from it

Proceed without Resource

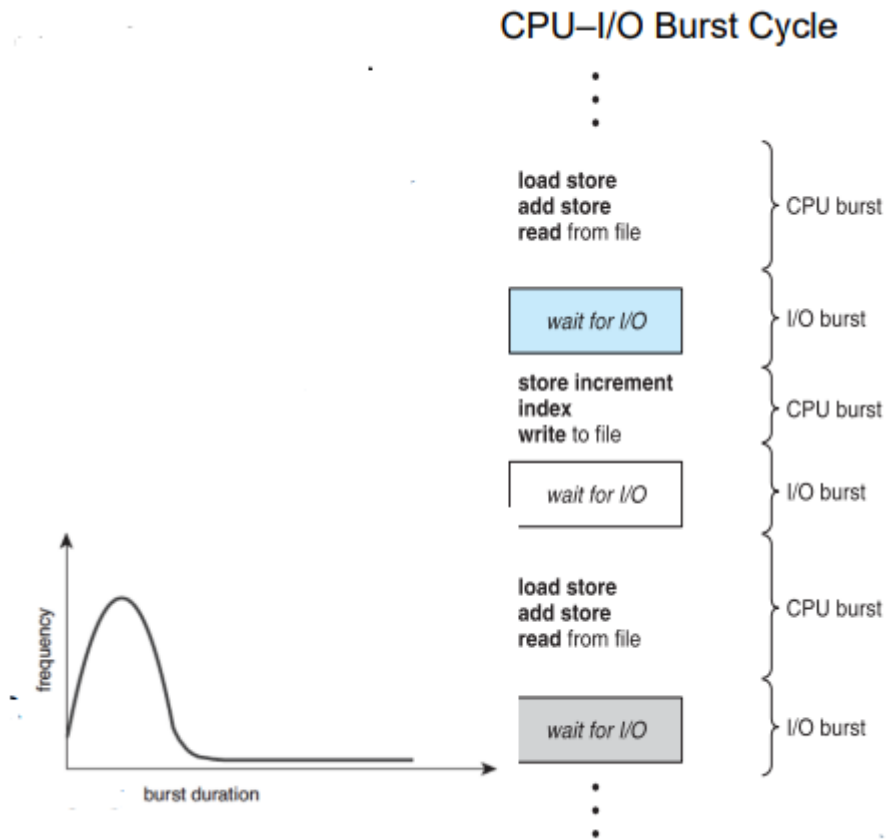
- E.g. Amazon web site – how it deals with web requests when its server is overloaded
 - If inventory server not available for a specific length of time, front end web server will give up, complete order and queue a background check, sending apology to customer if item not available.
 - This is preferable to hanging indefinitely

Transactions

- Transactions are used widely in file systems and databases
 - Transaction either complete (and transaction commits) or do not happen at all (aborts)
- Transactional systems are good for deadlock recovery:
 - Thread roll back

- Roll back or undo threads actions to a clean state (picking one or more victim threads)
- Thread restarting
 - Victim thread is restarted
- Would have to be built on top of OS in practice – thus difficult to get working at user thread level
 - Would also need to maintain data strcs to maintain this transactional nature

CPU Scheduling



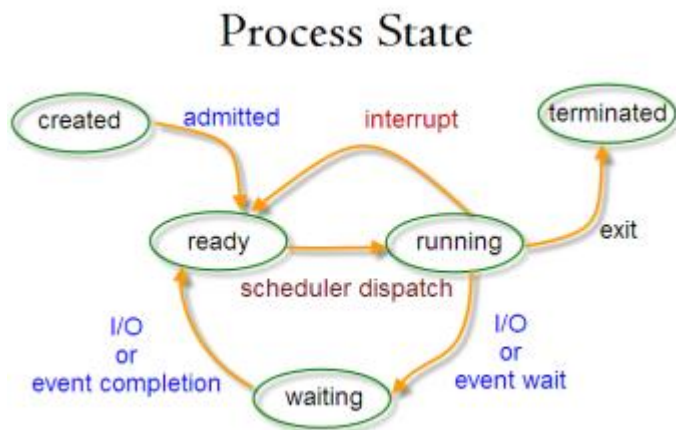
- Crit OS function – underlies the OSes responsiveness
- Modern OS's schedule kernel-level threads — not processes
 - But terms "process scheduling" and "thread scheduling" used interchangeably
 - Scheduling allows another process run on CPU when process waiting for I/O
- Scheduling allows OS to hide latency – latency arises because threads have to wait for system operations
 - E.g. thread calls sys function – has to wait until function resolved before it can continue executing
 - Bad for performance if CPU core sits idle whilst tasks are happening
- In multicore environment – want to ensure that whenever CPU released – there is a thread that can be scheduled to use CPU core effectively (i.e. no slack)
- Processes typically have many short CPU bursts and few long CPU bursts
 - I/O-bound - many short CPU bursts
 - Intermittent CPU bursts
 - Bulk of its type for a process-activity type is spent waiting for IO to be completed
 - CPU-bound - few long CPU bursts
 - CPU burst is concentrated burst of CPU activity – e.g. performing computations, updating registers, etc

- This distribution (i.e. proportion of each of these cycles) is important when implementing a CPU-scheduling algorithm
 - Continual interleaving of the CPU and IO burst cycles
- A priori there is no global scheduling algo that works well for all possible scenarios – need to look at characteristics of your sys, the computation and IO characteristics and choose scheduler based on that
- Graph above shows a well-balanced system
 - CPU bursts are quite short – then have a long tail (small number of long CPU bursts)

CPU Scheduler

- Selects one of the processes in the ready queue to be executed
- Ready queue can be implemented as:
 - FIFO queue
 - A priority queue
 - A tree
 - An unordered linked list
 - Choose which can based on the complexity of underlying system or how entries need to be inserted into queue
- The records in the queue are process control blocks (PCBs)
 - PCB contains necessary info to execute processes – process state, peripheral data, etc
- How does the operating system choose which thread to run next?

Pre-emptive and Non-Pre-emptive scheduling



- CPU scheduling decisions can take place when a process switches from:
 - (1) Running state to waiting state
 - (2) Running state to ready state
 - (3) Waiting state to ready state
 - (4) Running state to terminated
- Scheduler decides how to do these transitions and what process to swop in once something has been swopped out
 - Trying to max CPU utilisation – don't want CPU to be idle

- E.g. when going from running state to waiting state – want another thread to be scheduled in
- Non-pre-emptive/cooperative scheduling if done in situations 1 and 4 only
 - Once it has CPU, process keeps it until terminated or waiting
 - If only use this kind of scheduling = huge performance hit
 - Have processes that will hog CPU
- Otherwise pre-emptive scheduling
 - Can result in race conditions
- Virtually all modern OS's (e.g. Windows, Mac OS X, Linux, and UNIX) use pre-emptive scheduling algorithms

Scheduling Criteria

- Criteria depends on specifics of underlying system
- Have a series of tradeoffs – want to ID way of picking certain metrics and comparing them against each other for different utilisation patterns in OS
 - Use that to try to decide which scheduler makes sense for a particular use case
- Range of different scheduling algorithms with different properties
- Many potential criteria for comparison, such as:
 - Maximise CPU utilization – keep the CPU as busy as possible
 - Looking at the single core case
 - Obviously, the best utilisation depends on the device e.g. don't want 90% utilisation for a mobile device (drains battery too fast)
 - Maximise throughput – number of processes that complete their execution per time unit
 - Minimise turnaround time = interval from the time of submission of a process to the time of completion
 - i.e. total amount of time a process has been waiting in the ready queue
 - Minimise total amount of time a process has been waiting in the ready queue
 - Minimise response time = time to start responding
 - Amount of time it takes from when a request was submitted until the first process response is produced, not output (for time-sharing environment)

Scheduling Algorithms

- Concerned with the problem of deciding which of the processes in the ready queue is allocated the CPU's core
- Focus first on uniprocessor/ single core systems
- Five algos:
 - (1) First-come, First-serve (FCFS)
 - (2) Shortest-Job-First Scheduling (SJF)
 - (3) Round-Robin Scheduling (RR)
 - (4) Priority Scheduling
 - (5) Multilevel Queue Scheduling

First-come, First-serve Scheduling

- Managed with FIFO queue
 - Arriving process PCB added to end of the queue

- Process at head of queue scheduled next on CPU
- Advantages:
 - Simple
 - “Fair”
- Disadvantages – average waiting time/ average response time can be long
 - E.g. a long job can be scheduled onto CPU and hog it
 - Less equitable access to CPU

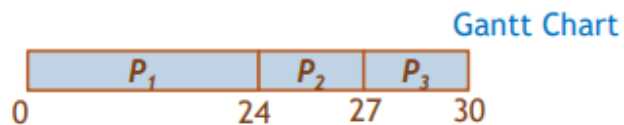
Consider:

Process	Burst time
P_1	24
P_2	3
P_3	3

Convoy effect - short process queued behind long process

- Consider one CPU-bound and many I/O-bound processes

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



Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

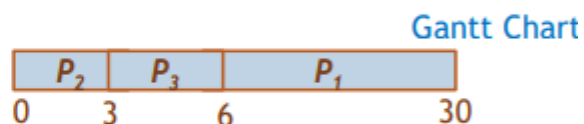
Average Waiting time : $(0 + 24 + 27)/3 = 17$

- Gantt Chart denotes how long process takes to execute and their order
- Series of process arrive at time 0 – each process has a burst time measured in a unit (milliseconds for examples)
- P_1, P_2, P_3 arrive at the same time – using FIFO scheduler, P_1 first placed on CPU and executes
 - Runs for 24 ms – then P_2 (goes to 27 ms), and then finally P_3 at 30 ms
- Waiting times for processes can be computed and calculate the average
 - Avg waiting time = 17 ms – an example of convoy effect
 - Convoy effect: short process queued behind long process
 -
 - NB this is a possible exam question

Consider:

Process	Burst time
P_1	24
P_2	3
P_3	3

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



Waiting time for $P_2 = 0$; $P_3 = 3$, $P_1 = 6$;

Average Waiting time : $(0 + 3 + 6)/3 = 3$

- With a different ordering, as seen directly above, can see that have an avg waiting time of 3 ms
 - Much lower than 17ms from previous example
- Is FIFO sensible? Yes! It can be!
 - When all tasks that need to be accomplished will each take roughly the same amount of time, then it doesn't matter
- Aside:

Memcached – FIFO can work?

- “Although you may think that FIFO is too simple to be useful, there are some important cases where it is exactly the right choice for the workload. One such example is memcached. Many web services, such as Facebook, store their user data in a database. The database provides flexible and consistent lookups, such as, which friends need to be notified of a particular update to a user's Facebook wall. In order to improve performance, Facebook and other systems put a cache called memcached in front of the database, so that if a user posts two items to her Facebook wall, the system only needs to lookup the friend list once. The system first checks whether the information is cached, and if so uses that copy. Because almost all requests are for small amounts of data, memcached replies to requests in FIFO order. This minimizes overhead, as there is no need to time slice between requests. For this workload where tasks are roughly equal in size, FIFO is simple, minimizes average response time, and even maximizes throughput.”

— Operating Systems: Principles and Practice (Volume 2 of 4) by Thomas Anderson, Michael Dahlin

Shortest-Job-First Scheduling

- Under this approach, next job to be chosen by scheduler is the one that has the least remaining time
 - Not concerned with length of the running process – only at burst time
- Actually shortest-next-CPU-burst algorithm
- Associate with each process the length of its next CPU burst – when a process arrives at the queue
 - Use these lengths to schedule the process with the shortest time from those currently in the queue
- SJF is provably optimal for response time:
 - Minimum average Waiting time for a given set of processes
 - Consider a case where have a short burst-time process scheduled after a long one
 - If computing avg waiting time as metric – if move the short one before the long time, then avg waiting time decreases for the entire system
- Example:
 - All processes arrive at time 0 – p4 waiting time = 0ms, p3 = 3ms, etc
 - 7ms as avg waiting time
 - FCFS is longer

Consider:

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Gantt Chart



Average Waiting time : $(0 + 3 + 9 + 16)/4 = 7$

Compare FCFS: 10.25 ms

- Accurate implementation is impossible.
- Approximate implementation is difficult:
 - Hard to predict the length of the next CPU burst
 - Thus, have to model – need some prediction scheme to guess at the next burst time and use that as burst length for incoming job
- Assume that the next CPU burst will be similar to previous
 - Predict length of next CPU burst as exponential average of previous – moving avg which exponentially waits closer samples
 - Samples closer to current point in time will receive a heavier weight – further away get a much lower weight
 - Pick the process with the shortest predicted CPU burst – schedule on that basis, apply SJF algo to that

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

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

α is weighting,
usually set to $\frac{1}{2}$

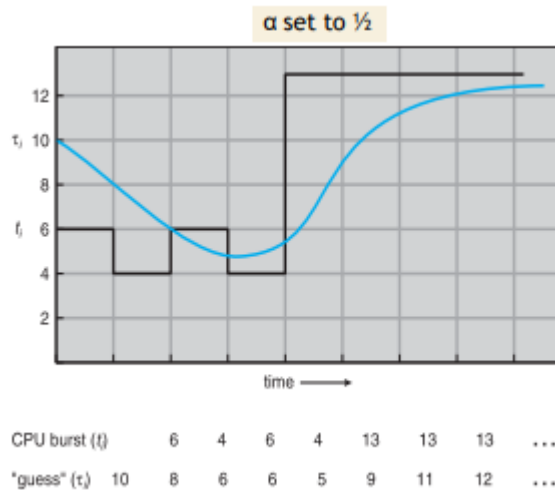
history

initial τ_0 can be defined as a constant or an overall system average.

- Formalised:
 - α weight between 0-1 – weighting factor that determines relative importance of contributing terms
 - α will sum to 1
 - t is system burst time – don't know upfront, use hindsight (deducible once the burst has happened)
 - τ is predicted burst value at a particular time step
 - τ_{n+1} – value used in SJF algo at next time step

- Predicted next CPU burst length is linear weighted sum of the previous actual burst length + some other weighted term that encapsulates entire history up until that point
- Basically, a recurrence relation
- As one iterates, τ_n will accumulate more info about the past
- Need to set τ_0 – e.g. a constant

• Example:



If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ + (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

If $\alpha = 0$, $\tau_{n+1} = \tau_n$

- Recent history does not count

If $\alpha = 1$, $\tau_{n+1} = \alpha t_n$

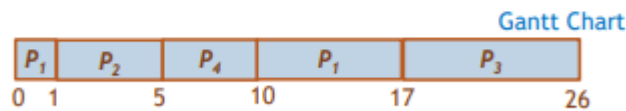
- Only the actual last CPU burst counts

- Black line is the actual burst time, blue represents the value computed by exponential averaging scheme – lags when updated with new values, until the sequence of burst times become similar (then converges)
- Establish impact of weighting using recurrence relations τ_{n+1} – expand it
 - By plugging in values and iteratively expanding them out – get an equation
 - Terms very far in past have a high exponential factor (e.g. τ_0) – and these factors are less than one
 - If take value between 0-1 and multiply it by itself repeatedly, it gets smaller and smaller quickly
 - Thus, impact of very distant terms compared to current, is very very low
- Extreme of α value – setting α to 0 will make formula reduce to next predicted burst (ie entire history), but ignores most recent burst info that you get from system
 - Use if system is unstable and burst times are not converging
- If α set to 1 – next predicted burst is the same as previous actual burst
 - Ignoring history completely
- So, with guess $\tau_j = 10$ and cpu burst $\tau_i = 6$, then $5+3=8$ (diagram)
 - And so on
 - Eventually, guesses being made start converging on the actual burst time – once system has settled down

Pre-emptive or Non-pre-emptive

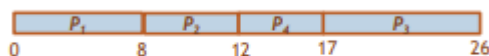
Consider:

Process	Arrival time	Burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Average Waiting time : $((10-1)+(1-1)+(17-2)+(5-3))/4 = 6.5 \text{ ms}$

- SJF is either pre-emptive or non-pre-emptive
- If the predicted CPU burst of a newly arrived process is shorter than what is left of the currently executing process, a pre-emptive SJF algorithm will pre-empt the currently executing process (boot off the running process and swop in the newly arrived process) , whereas a non-pre-emptive algorithm will allow the currently running process to finish its CPU burst
- Pre-emptive example:
 - Have process 1-4, each arriving at different times with different past times
 - P1 arrives and immediately gets scheduled and runs – 1ms later, p2 arrives
 - P2 has burst time of 4ms, and p1 has 7ms left to execute
 - So, move p1 off CPU, put it back into queue and put p2 onto CPU
 - P2 runs for 4ms
 - P3 arrives – p2 has 3 ms left, while p3 has 9, so it continues
 - P4 arrives – at time 3, p2 has 2ms left, and that is less than p4's 5ms
 - P2 then completes at 5ms
 - Then take processes that remain i.t.o their burst length – schedule p4, p1 and p3
 - Avg waiting time – have to account for when each process arrived and how long they had to wait
- Non-pre-emptive example:



- P1 arrives and runs for 8ms – but all the other processes arrive during that 8ms
- Once p1 finishes, choose the one with the next shortest burst time – p2, then p3 and finally p3
- Avg waiting time is thus 7.75ms (worse then pre-emptive but better than FCFS)

Potential Issues

- If have long task, it will be switched on and off the CPU quite frequently, especially if it's mixed in with a bunch of smaller tasks
 - This requires a context switch each time – adds to the general system overhead
- Potential to have starvation if have a bunch of shorter jobs that pre-empt a longer job

- Depending on job length, may never get to run, or very little time – will take much longer to finish

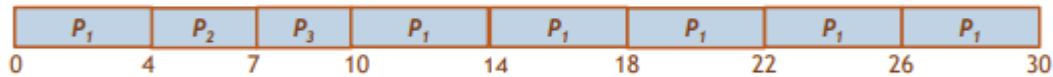
Round-Robin Scheduling (RR)

- Each process gets a small slice of CPU time (time quantum, q) – q is generally 10 to 100 ms
 - Ensures that starvation can't occur – even long processes are guaranteed to run on the CPU (even if it's just for a short amount of time), without being pre-empted by shorter tasks
 - All tasks are pre-empted if they go behind the time quantum, but even longer tasks will be able to run quite frequently
- After this, process is pre-empted and added to end of the ready queue
 - Ready queue treated as a circular queue – processes enter the queue, process at the front is taken off, allowed to run for 1 time quantum
 - If it hasn't finished, it's pre-empted and pushed onto the back of the queue
 - Next item then runs, and cycle repeats
 - New processes are added to 'end' of queue
 - Scheduling is repeated until all processes have been completed
- Timer interrupts every quantum to schedule next process
- Performance depends heavily on size of quantum – choice has a severe impact on performance of round robin scheduling scheme
 - large \Rightarrow FCFS (FIFO)
 - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high
 - Smaller queue has more characteristics of a SJF approach i.e. how it regularly schedules processes with shorter remaining burst times
 - Must still be mindful of quantum choice – everytime there's an interrupt, need to handle a process/thread context switch
 - These switches still have a cost
 - A quantum that is too small will be context switching all the time – adds considerably to amount of times process required to execute
 - Context switch $< 10 \mu s$
- For N processes in the ready queue and time quantum q :
 - Each process gets $1/N$ of the CPU time in chunks of at most q time units at once
 - Waiting time for a process no more than $(N-1)*q$ time units – guaranteed bound on waiting time, so can't have any process starving
- Example

Consider: (all processes arrive at $t=0$)

time quantum $q = 4$

Process	Burst time
P_1	24
P_2	3
P_3	3



Average Waiting time : $((0 + (10 - 4)) + 4 + 7) / 3 = 5.66$ ms

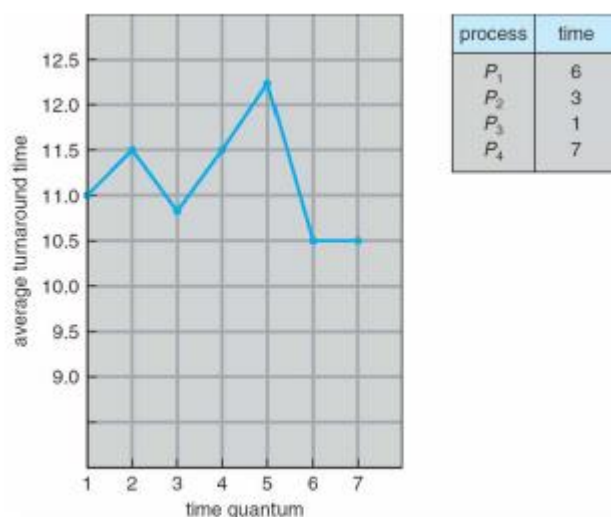
Compare non-preemptive SJF: $(0 + 3 + 6) / 3 = 3$ ms

Compare FCFS: $(0 + 24 + 27) / 3 = 17$ ms (convoy effect)

The average Waiting time under the RR policy is often long.

Also higher average Turnaround time than SJF, but better Response time.

- Short processes have a burst time shorter than the time quantum – when scheduled, they'll be able to run to completion
- P_1 will run for time quantum, pre-empted (placed on the back of the queue)
 - Then p_2 scheduled – and so on
- When only p_1 left – can be scheduled to be exclusively run on the queue or it could be interrupted every quantum, chosen to run on the queue (and so on)
 - Depends on scheduler
- Waiting time for round-robin approach isn't necessarily better than others – because for longer programs, they have to wait a long time to achieve completion
- Better response time because only need to run a process until it gets to a point where it can start returning data/producing a response that's useful for the user
- A bigger quantum size can have unexpected results on turnaround time



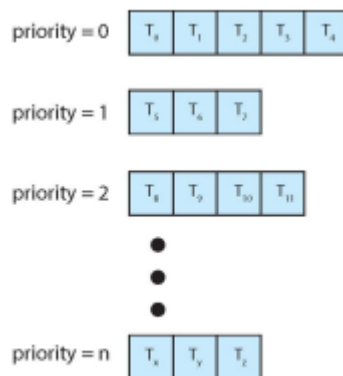
- Average Turnaround time of a set of processes does not necessarily improve as the time-quantum size increases
 - Average turnaround time can usually be improved if most processes finish their next CPU burst in a single time quantum.

- A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum
 - Can use statistical data and then modify time quantum dynamically depending on computational load

Priority Scheduling

- A priority number (integer) is associated with each process
- CPU allocated to the process with the highest priority
 - Preemptive – higher priority process can pre-empt the CPU and start running
 - Non-preemptive
- SJF is actually priority scheduling, where priority is the inverse of predicted next CPU burst time.
- Usually smallest # \equiv highest priority
- Priorities can be assigned internally – e.g. . I/O burst or externally (importance of process etc.)
 - Problem: Starvation (indefinite blocking) – low priority processes may never execute.
 - Solution: Aging – as time progresses increase the priority of the process
- Can combine priority scheduling and RR
 - System executes the highest priority process; processes with the same priority will be run using round-robin
 -

Multilevel Queue Scheduling



- For priority scheduling, $O(n)$ search may be necessary to determine the highest-priority process.
 - Easier to have separate queues for each distinct priority, priority scheduling schedules the process in the highest - priority queue.
 - A process is then permanently in a given queue.
- Can also partition according to type of process, eg. background /foreground
 - i.e. some queues associated to some kinds of processes
 - Foreground – need more rapid response time
 - Background – run more slowly, and less frequently (lower priority)
- Each queue has its own scheduling algorithm:
 - Foreground – RR
 - Background – FCFS

- Biggest issue: need to be able to extract tasks across all the queues to run on the collection of cores
 - Can service the high priority queues, and then move down – i.e. only service the next pq when the current one is empty
 - Problem – those current queues may never empty, leads to starvation
 - Could thus allocate a fraction of the CPU budget to each of the queues
- Scheduling must be done between the queues:
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes;
 - i.e., 80% to foreground in RR
 - 20% to background in FCFS
- A process can move between the various queues – depending on various criteria
 - Aging can be implemented this way
 - Multilevel-feedback-queue scheduler defined by the following parameters:
 - Number of queues
 - Scheduling algorithms for each queue
 - Method used to determine when to upgrade a process
 - Method used to determine when to demote a process
 - Method used to determine which queue a process will enter when that process needs service

Scheduling Algorithm Evaluation

- Analytical modelling: deterministic evaluation.
 - Can always get the same result based on a given result and series of input #s
 - Give exact numbers for comparison – models are the algos, take a collection of data that corresponds to burst time in the system, run the model and see what it predicts over all the scheduling algos
 - Simple and fast, but limited to fixed scenario
- Queueing Models: queueing-network analysis.
 - distribution of CPU and I/O bursts measured and then approximated or estimated (probabilistic).
 - Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on.
 - E.g. Little's formula
- Simulations: program a model of the computer system.
 - Use random-number generator to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions.
 - As simulation executes, generates statistics of algorithm performance.
- Actual implementation
- Terms:
 - n = average queue length
 - W = average waiting time in queue
 - λ = average arrival rate into queue

- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:
 - $n = \lambda \times W$
 - Valid for any scheduling algorithm and arrival distribution