Operating System
- Program that acts as an intermediary between a computer user and the computer hardware
- Must run in kernel mode

No OS	Minimal overhead, not portable, inefficient use of computer
Batch	Execute job sequentially, inefficient use of CPU
Time-sharing	Users use terminals to schedule jobs
Personal	Dedicated to user

Benefits:
1. Abstraction: same API to access all types of same category of hardware
2. Resource allocator: manages all resources like CPU, memory, I/O and arbitrate potentially conflicting requests for efficient and fair resource use
3. Control program: controls execution of programs, preventing errors and improper use

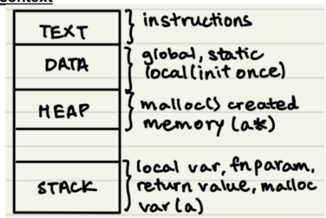
Kernel Segmentation fault => when a program is accessing memory that is not allowed
- Deals with hardware issues, provides system call interface, and special code for interrupt handlers and device drivers

Type	Description	Pros	Cons
Monolithic	Single program; does everything for OS to handle user <> hardware	Well understood and good performance (since switching btwn kernel and user mode takes a lot of clock cycles)	Highly coupled components, complex structure
Microkernel	Only implements essential functionality like IPC; higher-level services run outside of kernel and use IPC	Robust and extendible; better isolation and protection	Lower performance

IPC is interprocess communication

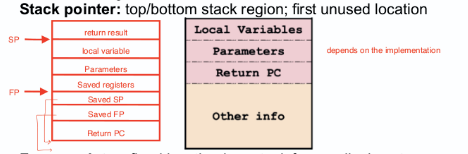
Hypervisor (Virtual Machine Monitor): creates and manages virtual machines
Type 1: bare metal (i.e. hypervisor is lightweight OS to create virtual machines) interact directly with the CPU hardware and provide support to multiple operating systems
Type 2: run on host OS -> slower (remaps calls from the guest OS to the host OS)

Memory context



Stack: information for function invocation in stack frame
- Deallocation timing; allow "jump" back in recursive stack

Stack frame: return address of the caller, parameters for the function, storage for local variables, etc.
Stack pointer: top/bottom stack region; first unused location



Frame pointer: fixed location in a stack frame; displace to access elements
Function call convention: non-universal ways to setup stack frame
CS2106 convention:

Caller:	Push $fp and $sp to stack
	Copy $sp to $fp
	Reserve space on stack for parameters by moving $sp
	Write parameters to stack using offsets of $fp
	JAL to callee
Callee:	Push $ra to stack (callee saves the return address)
	Push current register values used in function
	Use $fp to access parameters
	Compute results
	Write result to stack (where $fp is)
	Restore registers saved
	Get $ra from stack
	Return to caller using JR $ra
Caller:	Get result from stack
	Restore $sp and $fp

Register spilling: move register value to memory temporarily when no registers available $sp points to the return address (after restoration)

Dynamically allocated memory
- Memory allocated during execution time using malloc()
- Size unknown during compilation time, no definite deallocation timing

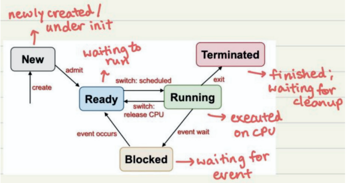
Heap memory: store information for dynamically allocated values
- Trickier to manage because of variable size and allocation/deallocation timings

Process
- Dynamic abstraction & information for executing program

Cores <> Processes: m CPUs has at most m processes
init: root process for all processes (PID = 1)
- Created by kernel at boot up time
- Like Supervisor in Elixir

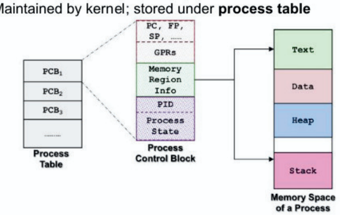
With 1 CPU: n <= 1 process in running state; n conceptually 1 transition at a time. With m CPUs: n <= m process in running state; n possibly parallel transitions

Process scheduling



Zombie: child process terminates but parent does not call wait()
Orphan: parent process terminates before child process
- init process becomes pseudo-parent (handles termination)

Process Control Block (PCB)
- Entire execution context for a process
- Maintained by kernel; stored under **process table**



System calls simplifies our calls, do the formatting for us
- Must change from user mode -> kernel mode

Function Wrapper	Function Adapter
Syscall with library version with same name and parameters	User friendly syscall with less params/more flexible param

Mechanism: In C/C++ -> system call can be invoked almost directly
1. User program invokes library call
2. Library call places system call number in a designated location like a register (usually in assembly code)
3. Library call executes special instruction to switch from user to kernel mode (TRAP), saving the CPU state
4. Appropriate system call handler is determined by system call number (index), handled by dispatcher
5. System call handler executed
6. System call handler ends, restoring CPU state and returning to library call, switch back to user mode
7. Library call return to user program

Exception & Interruption occur due to program execution
Exception: machine level instructions (checked exceptions)
- Synchronous and must be executed by an exception handler

Interrupt: external events, hardware related (unchecked exception)
- Asynchronous and suspends program execution, must be executed by interrupt handler

Handler: could do nothing
1. Save register/CPU state
2. Perform handler routine
3. Restore register/CPU
4. Return from disruption

argc = number of command line arguments (including the program name itself)

fork() init process -> created by kernel at boot up time -> PID = 1 -> root process of all processes

int fork();

- Returns PID of child process (in parent) OR 0 (in child)
- Creates new process, duplicates the current executable machine; executes the remaining code below the fork()
- Memory in child is copied from parent (copy-on-write)

Implementation:
1. Create address space of child process (virtual address space)
2. Allocate p' = new PID
3. Create kernel process data structures (process table entry)
4. Copy kernel environment of parent process (priority)
5. Initialize child process context: PID = p', PPID = parent PID, 0 CPU time
6. Copy memory regions from parent (expensive, optimised with copy-on-write)
7. Acquires shared resources (open files, pwd)
8. Initialize hardware context for child process (copy registers from parent process)
9. Ready to run (add to scheduler queue)

Copy on Write: optimization for memory copying operation where memory locations that are written to are duplicated; noop if reading only

exec()

int execl(const char *path, const *char arg0, …, const *char argN, NULL);

- Replaces code but not process information (same PID)
- fork() + exec() on child process to separately run tasks

exit()

void exit(int status);

- Ends execution of process
- Status 0 success, !0 error (status is returned to the parent process)
- Most system resources released like file descriptors
  - Left with PID & status, PCB
- Implicitly done when returning from the main() function

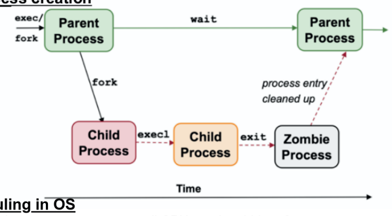
wait()

int wait(int *status);

- Parent waits for immediate child process (only) to terminate
- Returns PID of terminated child process
- Status is passed by address (wait(&status)); NULL if unused
- Blocking behavior on parent until one child terminates
- Clears remainder child sys resources, kills zombie processes
- execl() does not affect wait()
- If not children, terminate

wait() creates zombie processes, remainder can only be cleaned up when wait() happens

Process creation



Scheduling in OS
- If ready to run processes > avail CPU → who shld run?
- **Scheduler:** Part of OS that makes scheduling decision
- **Scheduling algo:**
  - is the algo used by scheduler
  - Based on process environments
- **Process behaviour:** Requirement for CPU time

Concurrent Execution
- Multiple processes progress in execution at the same time
  - Virtual (illusion) or physical (multiple CPUs)

Time slicing: interleaving instructions for multiple processes
- Context switching between processes incur overhead

Process Environments

Batch processing: No user interaction, no need to be interactive	**Turnaround time:** finish – arrival **Waiting time:** time spent in queue **Throughput:** number of tasks finished per unit time **CPU utilization:** % time where CPU is not idling
Interactive: Active users interacting with system, must be responsive	**Response time:** first CPU use – arrival time **Predictability:** variation in response time (consistency in response time)
Real-time processing: Deadline to meet, usually periodic process	

Criteria for all processing envs
- **Fairness**
  - Shld get fair share of CPU time (per process/ per user)
  - NO STARVATION
- **Utilization**
  - All parts of computing system must be utilized

Types of Scheduling
1. Non-preemptive (cooperative): process keeps running until blocked or gives up CPU voluntarily
2. Preemptive: process given fixed time quantum to run
   - Can be blocked or give up early
   - Forced to suspend once time quantum up
   - Ensures good response time as scheduler runs periodically
   - Periodic timer interrupt that cannot be intercepted; invokes scheduler (Timer interrupt handler invokes scheduler)
   - **Interval of Timer Interrupt (ITI):** OS scheduler invoked every timer interrupt (usually 1-10ms)
   - **Time quantum:** execution duration given to a process; in multiples of ITI; can be variable (usually 5-100ms)

Scheduling Steps
1. Scheduler is triggered (OS takes over)
2. If context switch is needed, context of current running process is saved and placed on blocked queue/ready queue
3. Pick a suitable process P to run based on scheduling algorithm
4. Setup context for P
5. Let P run

Pipes blocking
If process Q is trying to read and pipe is empty, it will **block** process Q until the pipe has something
If process P is trying to write and the pip is full, it will **block** process P until the pipe has space

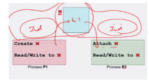
Scheduling Algorithms

FCFS	*Batch processing, non-preemptive* Tasks stored in queue based on arrival time & tasks run till done or blocked Blocked tasks are removed from queue and when I/O completes, added back to queue Assuming nobody monopolises the market Guarantees no starvation as every task runs eventually Average waiting time can be improved and convoy effect present where long CPU bound task blocks I/O and vice versa Shortest average response time if jobs arrive in order of increasing lengths or all jobs have same completion time (same length)
SJF Shortest Job First	*Batch processing, non-preemptive* Select task with smallest total CPU time (needs this info in advance/guess) error percentage = (Abs(Actual – Predict) / Actual * 100%) Prediction algorithm from previous CPU-bound phases using exponential average $Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$ Minimizes average waiting time given a fixed set of tasks Starvation is possible due to bias towards short jobs
SRT Shortest Remaining Time First	*Batch processing, preemptive* Variation of SJF using remaining time can be starved if streams of shorter jobs coming in New jobs with shorter remaining time pre-empts currently running jobs; good for short jobs with late arrival
RR	*Interactive, preemptive* Like FCFS but preemptive so when time quantum lapses, task forced to give up CPU and placed to end of queue (after I/O) Response time (guaranteed) to be upper bounded by (n - 1)q with n tasks and time quantum q Larger time quantum yields better CPU utilization but longer waiting time, shorter time quantum yields larger context switching overhead but shorter waiting time no starvation Behaves like FCFS if job lengths are shorter than time quantum Performs worse than FCFS if job lengths are all the same and greater than time quantum (higher average turnaround time) or when there are many jobs and job lengths exceed time quantum (reduced throughput due to increased overhead)
Priority Scheduling	*Interactive, preemptive/non-preemptive* Assigns priority to processes and run highest priority first Preemptive: Higher priority process can preempt running process with lower priority Non-preemptive: Late coming waits for next round of scheduling Low priority process can starve if high priority keeps hogging CPU (worse for preemptive); resolved by decreasing priority or current process after each time quantum or giving current process a time quantum **Priority inversion:** lower priority preempts higher priority because resource is locked so higher priority is blocked
MLFQ	*Interactive, preemptive* Multiple queues with different priority levels, minimizes response time for I/O bound processes and turnaround time for CPU bound processes Highest priority queue must be empty before next queue is used Rules: 1. Priority(A) > Priority(B) -> A runs 2. Priority(A) == Priority(B) -> A and B runs in RR 3. New job -> highest priority 4. Job fully used time quantum -> reduce priority 5. Job gives up before time quantum -> retain priority **Change of heart:** process with lengthy CPU phase right before I/O phase sinks to the lowest priority and gets starved; periodically move all tasks to highest priority to fix **Gaming the system:** process repeatedly gives up CPU before time quantum lapses remains in high priority and starves other processes; accumulate total CPU use time across all quantums
Lottery Scheduling Solve the issue for controlling how much time allocated to each process	*Interactive, preemptive* Give out "lottery tickets" to processes for system resources, randomly choose lottery ticket, winner gains resource; lottery tickets can be distributed to children **Provide gd lvl of control** Process holding X% of tickets has X% chance to win and use X% of resource **Responsive** Responsive as new processes can participate

Interprocess communication
- Mem space is independent → need IPC mechanisms

1. Shared memory

MASTER
CREATE: shmget(..., SIZE, ...) → int shmid
ATTACH: (int*) shmat(shmid,..., ...) → int *shm
Set control bit: shm[0] = 0, sleep while its 0
DEATTACH: shmdt (*char shm)
DELETE: shmctl (shmid, ..., 0)



OS only involved in creation and additioin not R/W → easy to use ops as array ops

Advantages
- Efficient
- Ease of use

Disadvantages
- Synchronisation
- Harder Implementation

SLAVE attach, do its thing, set control bit, deattach
ATTACH (provided shmid): (int*) shmat(shmid,..., ...) → int *shm
Indicate finished: shm[0] = 1
DEATTACH: shmdt (*char shm)

2. Message passing
Direct communication: sender/receiver of message explicitly name the other party
- One link per pair of communicating processes
- Identify of other party must be known

Indirect communication: messages are sent to/received from message storage (mailbox/port)
- One mailbox shared among multiple processes

Synchronization behaviors for send() and receive(): blocking primitives (synchronous) or non-blocking primitives (asynchronous)
Advantages: easier to implement on different processing (portable) environment and easier to synchronize
Disadvantages: requires OS intervention so inefficient and harder to use due to limits on message size/format
- The Msg have to be stored in kernel memory space
- Every send/receive operations need to go through OS (i.e. a system call)

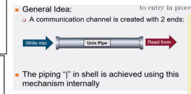
3. Unix pipes
- Share input/output between processes (producer/consumer)

Process communication channels: stdin, stderr, stdout
stderr cant be redir to a file

FOR SHELL PIPES (use piping mechanism stdout(a) → stdtin(B)

`open (char *path, int flag)`
- opens a file specified by pathname, if doesn't exist optionally be created (depending on flag)
- returns file descriptor (small, non-negative integer) which acts as index to entry in process' table of open file descriptors, else -1 & errno set

- General Idea:
  - A communication channel is created with 2 ends:



- The piping "|" in shell is achieved using this mechanism internally

```
pipe( pipeFd );
if ((pid = fork()) > 0) { /* parent */
  close( pipeFd[READ_END] );
  write( pipeFd[WRITE_END], str, strlen(str)+1 );
  close( pipeFd[WRITE_END] );
} else {                    /* child */
  close( pipeFd[WRITE_END] );
  len = read( pipeFd[READ_END], buf, sizeof buf );
  printf("Proc %d read: %s\n", pid, buf);
  close( pipeFd[READ_END] );
```

The pipe is an **anonymous first in first out file**

Pipe in C: circular bounded byte buffer with implicit synchronization (writer waits when buffer is full, reader waits when buffer is empty)
Unidirectional (half-duplex): one write end, one read end
Bidirectional (full-duplex): any end for read/write
System calls:

int pipe(int fd[]) close(int) -> side to close write(int, data, length) -> side to write to int len = read(int, buffer, length) -> read from
0 for success, !0 for errors
fd[0] is reading, fd[1] is writing

- Must close the end not in use, otherwise, other processes might accidentally write/read to it

Redirecting communication channels to pipes: use dup() and dup2() => dup() except assigns to <known> file descriptor

if the parent doesn't close the reading end. The parent will not get the sig pipe signal.
If the child doesn't close the writing end, the child will not get the end of call.

- Sets new file descriptor to refer to old as well (i.e. old and new are the same now)

dup2(fd[0], STDIN_FILENO)	Redirect input to read from read end of pipe
dup2(fd[1], STDOUT_FILENO)	Redirect standard output to write to write end of pipe
dup2(file, STDOUT_FILENO)	Redirect standard output to write to contents of file

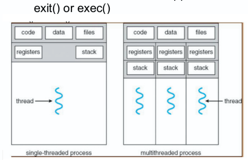
4. Unix signals
- Asynchronous notification regarding an event sent to a process/thread
  - E.g. kill, stop, continue (memory error, arithmetic error...)
- Recipient of the signal must handle the signal via default handlers or user supplied handlers (signal(type, handler))

Usage: `if (signal (SIGSEGV, myOwnHandler ) == SIG_ERR)`

Threading
- Units of work within a process
- Shares memory context (text, data, heap) and OS context (process ID, files, etc.) (doesn't share hardware context)
- Uniquely identified by thread ID, registers, and stack

(within the same process)
Thread switching: change hardware context like registers and stack (actually just changing FP and SP registers)
Benefits:
1. Economy: multiple threads in the same process requires less resources to manage (compared to multiple processes)
2. Resource sharing: threads share most of the resources of a process (No need for additional mechanism for passing information around)
3. Responsiveness (Context switching btwn threads much lighter and faster than btwn processes)
4. Scalability: take advantage of multiple CPUs

Problems: (re-entrancy problem)
1. System call concurrency, need to ensure that parallel system calls have no race condition
2. Process behavior: what happens when a single thread calls exit() or exec()



User Thread

a runtime system

- Thread implemented as user library (process handles thread related operations)
- Kernel not aware of user threads
- User thread with no kernel thread still runs on a single thread

Pros:

1. Can have multithreaded program on any OS
2. Thread operations are just library calls — no system calls required -> much faster
3. More configurable and flexible — can configure per process basis

Cons:

1. OS not aware of threads and scheduling is performed at process level
   - If one user thread blocks, the entire process is blocked so all threads are blocked
   - Cannot exploit multiple CPUs

Kernel Thread

- Thread is implemented in the OS handled as system calls
- Kernel schedule by threads, not processes; may make use of threads for its own execution — kernel itself can be multi-threaded

Pros:

1. Kernel schedules on thread levels
   - Multiple threads can run for the same process and be non-blocking

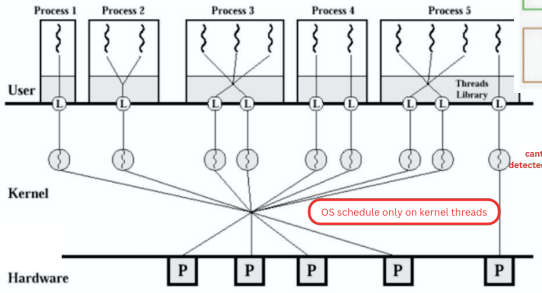
Cons:

1. Thread operations are system calls that are slower and more resource intensive
2. Less flexible since kernel threads used by all multithreading programs

Hybrid Thread

- User threads bind to a kernel thread

Offers great flexibility -> can limit concurrency of any process/ / user by limiting the number of threads you give them



Simultaneous Multi-threading

- Supply multiple sets of registers to allow threads to run natively and in parallel on the same core

pthread — void* -> arbitrary pointer

POSIX thread

-> an example of thread API

- Implemented as either user or kernel threads
- **pthread_t**: data type to represent thread id
- **pthread_attr:** data type to represent attributes of thread — (for configuration)

pthread_create:

```
int pthread_create(
    pthread_t* tidCreated,
    const pthread_attr_t* threadAttributes,
    void* (*startRoutine) (void*),
    void* argForStartRoutine);
```

NULL for default configuration

Return 0 if successful, !0 if errors

tidCreated: thread id for the created thread
threadAttributes: control the behavior of the new thread
startRoutine: function pointer to the function to be executed by thread
argForStartRoutine: arguments for the startRoutine function

pthread_exit: pthread terminates automatically at the end of the startRoutine with return value defined by return statement

```
void pthread_exit(void* exitValue);
```

exitValue: value to be returned to whoever synchronize with this thread

pthread_join: wait for the termination of another pthread

```
int pthread_join(pthread_t threadID, void **status);
```

Return 0 if successful, !0 if errors

threadID: TID of the pthread to wait for
status: exit value returned by the target pthread

Race Conditions

- Execution outcome depends on the order in which the shared resource is accessed/modified (non-deterministic)
- General modification flow: load memory value into register, update register value, load register to memory
  - Race conditions happen when loading happens at the wrong time — unpredictable preemption
- Caused by unsynchronized access to shared modifiable resources

Solution: designate code segment with race condition as critical section, only one process can execute in CS (synchronization)

Critical Section — A critical section is a section of code that modifies a shared resource.

```
// Normal code
Enter CS
// Critical work
Exit CS
// Normal code
```

critical section usually looks like this: x = x + 1000;

Properties of Correct CS Implementation

Properties of correct CS implementation:

Mutual Exclusion:
- If process P_i is executing in critical section, all other processes are prevented from entering the critical section.

Progress:
- If no process is in a critical section, one of the waiting processes should be granted access.

Bounded Wait:
- After process P_i request to enter critical section, there exists an upperbound of number of times other processes can enter the critical section before P_i.

Independence:
- Process **not** executing in critical section should never block other process.

Incorrect Synchronization

1. Deadlock: all processes blocked so no progress
2. Livelock: related to deadlock avoidance mechanism where processes keep changing state to avoid deadlock and make no other progress; processes are not blocked — cant be detected by OS
3. Starvation: some processes never get to make progress in their execution as it is perpetually denied necessary resources

1. Assembly-level implementation: Test and Set

TestAndSet(Register, MemoryLocation)

- Machine instruction provided by processors to aid synchronization
- Load the current content at MemoryLocation into Register and stores 1 into MemoryLocation (treated as lock)
- Performed as single machine operation (atomic)
- Used to create spinlocks

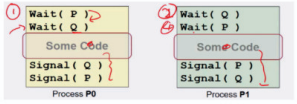
```
void EnterCS(int* Lock) {
    // Cannot enter if lock value is 1
    while (TestAndSet(Lock) == 1);
}
```

TestAndSet() takes a memory address M:
- Returns the current content at M
- Set content of M to 1

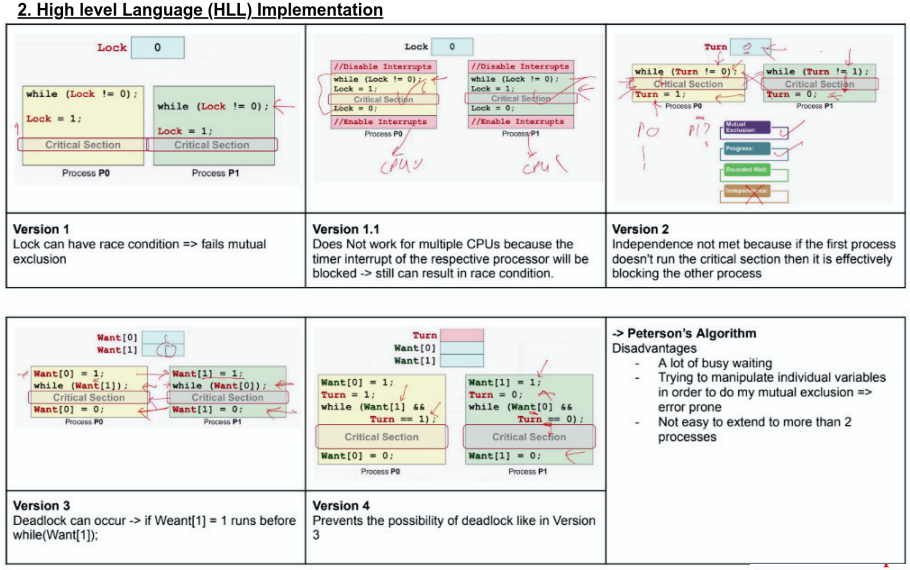
```
void ExitCS(int* Lock) {
    *Lock = 0;
}
```

In Test and Set, all 4 criterias are met but it is also very expensive, due to busy waiting

Busy waiting: keep checking the condition until it is safe to enter critical section which is wasteful use of processing power
Variants: compare and exchange, atomic swap, load link/store conditional

Mutual Exclusion	Proving mutual exclusion -> prove **N_cs = 0** **- N_cs = Number of processes in critical section = Process that completed wait() but not signal()** - N_cs = #Wait(S) - #Signal(S) - S_initial = 1 - S_current = 1 + N_cs = 1 + #Signal(S) - #Wait(S) ⇒ S_current + N_cs = 1 **- Since S_current >= 0 ⇒ N_cs <= 1**
No Deadlocks	**Deadlock** -> means all processes stuck at wait(S) ⇒ S_current = 0 and N_cs = 0 ⇒ S_current + N_cs = 0 =/= 1 (contradiction!!!) **Deadlock is still possible with incorrect use of semaphore eg.** 
No Starvation	Suppose P1 is blocked at wait(S) P2 is in CS, exits CS with signal(S) - If no other processes, P1 eventually wake up - If there are other processes, P1 eventually wakes up (assuming fair scheduling)

2. High level Language (HLL) Implementation



Version 1
Lock can have race condition => fails mutual exclusion

Version 1.1
Does Not work for multiple CPUs because the timer interrupt of the respective processor will be blocked -> still can result in race condition.

Version 2
Independence not met because if the first process doesn't run the critical section then it is effectively blocking the other process

Version 3
Deadlock can occur -> if Weant[1] = 1 runs before while(Want[1]);

Version 4
Prevents the possibility of deadlock like in Version 3

-> Peterson's Algorithm
Disadvantages
- A lot of busy waiting
- Trying to manipulate individual variables in order to do my mutual exclusion => error prone
- Not easy to extend to more than 2 processes

3. High level abstraction: Semaphores

Provide abstracted mechanisms which provide additional useful features

2 functionalities -> Wait(), Signal()
- generalised synchronisation mechanism
- only behaviours are defined -> can have different implementations

Provides
- A way to block a number of processes
- Known as sleeping processes
- A way to unblock/ wake up one or more sleeping process

Semaphore, S, is
- an integer variable
- Protected variable -> all operations on it are atomic
- Can be initialised to any non-negative value initially

Wait(S)
- If S <= 0, blocks (go to sleep)
- Decrement S
- Also known as P() or Down()

Signal(S)
- Increments S
- Wakes up one sleeping process if any
- This operation **never** blocks
- Also known as V() or Up()

Semaphore property:
Given S_initial >= 0, the following invariant must be true
S_current = S_initial + #signal(S) - #wait(S)
General Semaphore: S >= 0 (S = 0, 1, 2, 3, ...)
Binary Semaphore: S = 0 or 1

SYNCHRONISATION IMPLEMENTATIONS
Posix Semaphore
Header File:
- `#include <semaphore.h>`

Compilation Flag:
- `gcc something.c -lrt`
- Stand for "real time library"

Basi Usage:
- Initialize a semaphore
- Perform wait() or signal() on semaphore

Usually semaphores is used with threads because less complicated to set up the semaphore (memory shared etc)

pthread Mutex and Conditional Variables
Synchronisation mechanisms for pthreads
Mutex(pthread_mutex)
- Binary semaphore (i.e. equivalent Semaphore(1))
- Lock: pthread_mutex_lock()
- Unlock: pthread_mutex_unlock()

Conditional Variables(pthread_cond)
- Wait: pthread_cond_wait()
- Signal: pthread_cond_signal()
- Broadcast: pthread_cond_broadcast

The conditional variable HAS to be used with a mutex

Synchronization problems

1. Producer Consumer Problem

Producers: produce items to insert in buffer
- Only when buffer not full

Consumers: remove items from buffer
- Only when buffer not empty


```
while (TRUE) {
    Produce Item;
    wait( notFull );
    wait( mutex );
    buffer[in] = item;
    in = (in+1) % K;
    count++;
    signal( mutex );
    signal( notEmpty );
}
```
Producer

```
while (TRUE) {
    wait( notEmpty );
    wait( mutex );
    item = buffer[out];
    out = (out+1) % K;
    count--;
    signal( mutex );
    signal( notFull );

    Consume Item;
}
```
Consumer

2. Readers Writers problem

Processes share a data structure D
- **Readers**: Retrieves information from D
- **Writer**: Modifies information in D

Writer must have **exclusive** access to D
Readers can **access with other readers**

```
while (TRUE) {
    wait( roomEmpty );
    Modifies data
    signal( roomEmpty );
}
```
Writer Process

Blocks the room empty semaphore using a mutex to only allow one reader in at a time.
Important!
signal(mutex) must be outside the if statement because if you do wait(roomEmpty) and never come out, you will block everyone else.

Issues
- Not fair for the writer

- Initial Values:
  - roomEmpty = S(1)
  - mutex = S(1)
  - nReader = 0

```
while (TRUE) {
    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );
}
```
Reader Process

3. Dining Philosophers

3A: Tanenbaum Solution

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
```

```
void takeChpStcks( i )
{
    wait( mutex );
    state[i] = HUNGRY;
    safeToEat( i );
    signal( mutex );
    wait( s[i] );
}
```

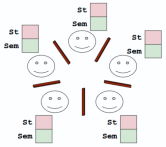
```
void putChpStcks( i )
{
    wait( mutex );

    state[i] = THINKING;
    safeToEat( LEFT );
    safeToEat( RIGHT );

    signal( mutex );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}
```

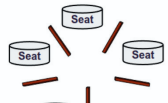


3B: Limited Eater

- If **at most 4 philosophers are allowed to sit at the table (leaving one empty seat)**

➔ **Deadlock is impossible!**

```
void philosopher( int i ){
    while (TRUE){
        Think( );
        wait( seats );
        wait( chpStk[LEFT] );
        wait( chpStk[RIGHT] );
        Eat( );
        signal( chpStk[LEFT] );
        signal( chpStk[RIGHT] );
        signal( seats );
    }
}
```



- Initial Values:
  - seats = S(4)
  - chpStk = S(1)[5]